ECE552 Lab4 Report

Lei, Mei Siu 1000302718

Lam, Carl 1000340516

Question 1:

q1.cfg: dl1:64:8:1:l:1

Our next line micro-benchmark instantiates an array of structs that is defined to have two integers, and then iterate over each element of the array sequentially. Each struct fits into one cache-line perfectly. For this micro-benchmark and cache configuration, we can see that without a prefetcher, there would be constant misses because each iteration accesses a new cache line. However, the next-line prefetcher is able to perfectly capture the spatial locality of this program: the next iteration will access the next cache line. Thus, next-line prefetcher will have almost 100% accuracy and coverage.

We verified the correctness of our next-line prefetcher by confirming that the micro-benchmark and configuration resulted in an almost 0% miss rate, and 100% miss rate when we changed the prefetcher type to be no-prefetching.

Question 2:

q2.cfg: dl1:64:8:1:l:16

We use the same configuration as the one of the next-line prefetcher to better contrast the difference between the two prefetchers. The control variable we changed is the size of the struct in the microbenchmark. We increased the size of the struct by 2x, and we still only access the first integer of the struct in each iteration. The effect now is that every iteration accesses data that is two cache-lines ahead of the previous iteration. The next-line prefetcher (q1.cfg) would fail because it is designed to only prefetch the next cache-line of the current cache-line on demand, which is never accessed in this program. In contrast, the stride prefetcher (q2.cfg) is more flexible in a sense that it uses past stride pattern to predict future memory access. In this case, since our stride is always four integers, 16 bytes, the stride prefetcher is able to prefetch the required data every time.

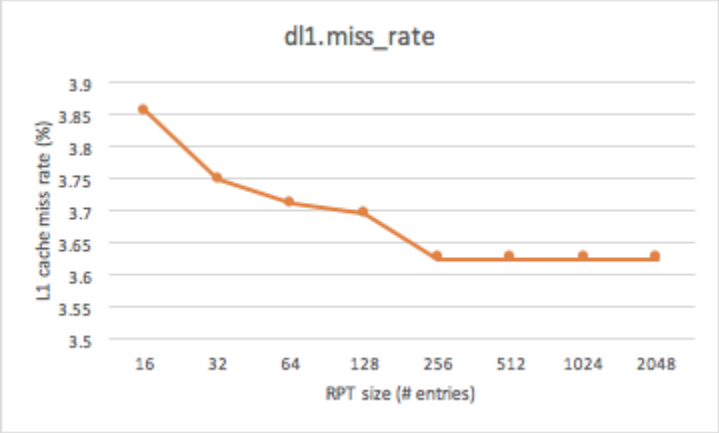
We verified the correctness of our stride prefetcher by confirming that the micro-benchmark and configuration resulted in an almost 0% miss rate, and 100% miss rate when we changed the prefetcher type to be next-line prefetcher or no-prefetching.

Question 3:

Config	L1 Miss Rate	L2 Miss Rate	Average access time
baseline	4.16	11.4	1.801
next-line	4.19	8.38	1.693
stride	3.85	5.78	1.547

Average access time = 1*(1-L1_Miss_Rate) + (L1_Miss_Rate)*(10*(1-L2_Miss_Rate) + 100*L2_Miss_Rate)

Question 4:



The miss rate steadily drops as we increase RPT size until it hits about 256 entries at which point miss rate levels out. Increasing RPT size helps because it reduces the chance of useful entries being rewritten by future instructions of a loop. However once the table reaches a large enough size (for compress, this is 256), then the table is large enough to store an entry for each instruction in a loop and any more entry space that gets added becomes unused.

Question 5:

The performance of a prefetcher is measured based on three criteria: coverage, accuracy and timeliness. Currently, the statistics provided by the simulator allow us to have a good sense of the coverage. We can calculate the coverage by comparing the number of misses with and without a prefetcher. However, we do not have enough information for us to analyze accuracy and timeliness.

To understand the accuracy of our prefetcher, we need to know the total number cache-lines we prefetched. Currently we only know the total number of accesses but we might or might not prefetch per access, and we might prefetch more than one cache-lines per perfetch. Therefore, the simulator would help us better analyze the accuracy of a prefetcher if it gives the total number of cache-lines prefetched.

Timeliness says that the prefetcher needs to figure out the data to prefetch and issue the request before the program itself issues the request. However, the currently simulator does not consider the latency of the prefetcher algorithm. Thus, if the simulator can simulate the program latency and the prefetcher latency, and calculate a hit rate that captures these two factors, we will be able to better analyze the timeliness of the prefetcher. For example, if a prefetcher has a much higher miss rate with the consideration of prefetcher latency than without, it means that the prefetcher has poor timeliness. Similarly, if a prefetcher's miss rate is about the same with or without the consideration of latency, that means that it has good timeliness.

Question 6:

Before reading the micro-benchmarks and configurations, please see the next section for a description of our open-ended implementation.

Mbq6.c and q6.cfg

q6.cfg: dl1:64:8:1:l:2

mbq6.c: We have two arrays of structs, **a** and **b**. Each struct contains four integers. The program has an outer loop and an inner loop. The inner loop accesses over three elements of **a**

sequentially, to make sure its corresponding entry in the RPT becomes steady. After the inner loop finishes, there is another memory access instruction that writes to **b**. The index of outer loop increments by 3 for every iteration.

Explanation: We recompiled our cache.c with the number of **RPT_SIZE=1**. In the above microbenchmark, the instruction that accesses **a** always has a stride of one. However, there is another instruction that accesses **b**. Since there is only one RPT entry, the stride predictor constantly swaps these two instructions in and out of the RPT. Therefore, next time around when the inner loop gets executed, it needs to re-establish the stride pattern. However, in our open-ended prefetcher, we do not evict the entry, so it keeps prefetching. In our microbenchmark, the stride pattern does not change so the prefetching accuracy is always 100%, but our open-ended prefetcher is able to achieve a lower miss rate by having a higher coverage compared to the stride predictor.

The microbenchmark and the configuration has a miss rate of 25% with our open-ended predictor; 1 in every 4 memory accesses is a miss in **b**, but no misses in accesses to a. The same when applied to the stride prefetcher (modified cache.c so that stride would have RPT_SIZE=1) resulted in a 50% miss rate, **1** for **b** and 1 for **a**.

Mbq6lru.c and q6lru.cfg

q6lru.cfg: dl1:1:16:4:l:2

mbq6lru.c: Three arrays of integers, a, b, c. The outer loop executes for a million times. The inner loop has the following access pattern and i goes from 0 to 3.

a[i]->b[i]->c[i]

Explanation:

In this benchmark, only 3 cachelines are ever needed (a[0-3], b[0-3], c[0-3]). However, once we access the 4^{th} index (e.g. a[3]), it will begin prefetching the next cacheline [4-7] for each variable. In a stride prefetcher, each of these prefetches would be placed at the head of the LRU queue, eventually pushing out the [0-3] cachelines. However, in our open-ended implementation, newly prefetched blocks are placed at the tail, and the prefetched blocks only push each other out, which is fine because the [4-7] blocks are never read.

We verified the performance of our LRU policy by comparing the miss rate of the stride prefetcher and that of our open-ended prefetcher. Our open-ended prefetcher had a miss rate of 0% while the strider prefetcher had a miss rate of 25%.

Open-Ended Prefetcher

Our open-ended prefetcher is a modification of the stride-prefetcher that has a 16-entry RPT. The goal was to reduce the number of harmful evictions done to the cache and RPT. There are two differences:

1. RPT Eviction policy: While the stride-prefetcher always evicts an entry when a new PC is mapped to an in-use entry, our open-ended prefetcher will only change the state of the current entry from steady->init->transient->no-prediction and evict the current entry if the current state is no-prediction.
    a. Intuition: A prefetcher is only useful if first, it does get to prefetch frequently enough, and second, it is accurate. We noticed that the prefetcher only starts to prefetch when

the same execution is executed again before it gets evicted. Another note is that we are generally more confident with the entries that are in the steady state, assuming that those instructions did get executed many times. Therefore, blindly evicting an RPT entry might decrease the prefetch rate and is potentially punishing entries that have high accuracy by replacing them with unknown entries. Therefore, the eviction policy is more lenient toward the existing entries and do not evict them if they continue to prove themselves to be useful and accurate.

2. Replacement policy: While prefetched data has the same level of priority of on-demand data in the stride-prefetcher, i.e. it is inserted to the head of the LRU list, prefetched data has a lower level of priority in our open-ended prefetcher, i.e. it is inserted to the tail of the LRU list.
    a. Intuition: Programs tend to have temporal locality, meaning that data that is already in the cache is likely to be reused. On the other hand, prefetched data has neither temporal or spatial locality (although they might be predicted based on temporal or spatial locality of current data). Thus, we might want to consider prefetched cache-line as a better victim for replacement.

Since we have two modifications, we have two sets of microbenchmark and configuration. Mbq6.c and q6.cfg are used to demonstrate the performance of our new eviction policy. Mbq6lru.c and q6lru.cfg are used to demonstrated the performance of our new replacement policy.

Total Memory Size

Our modified RPT prefetcher is a 16-entry table that consists of a tag (32bits – useless bits – 4 bits for table index = 25 bits), a prev_addr (32bits based on sizeof(md_addr_t)), a stride value (32bits), and a 2-bit state. So the cache size calculations are as follows.

RPT_SIZE = 16

Entry_size = 25 (tag) + 32 (prev_addr) + 32 (stride) + 2 (state) = 91 bits ~ 12 bytes

Total size = 16 * 12 = 192 bytes

Cacti Simulation

For the prefetcher, we modelled it as a cache configuration with tag size = 25, block size = 9 bytes, and cache size = 144 bytes (16*9)

Open Ended Prefetcher				
	Area (mm2)	Access Latency (ns)	Power (energy/access)(nJ)	Power (leakage read/write power) (mW)
Data array	0.00041349	0.13338	0.000276658	0.0562954
Tag array	0.00017050	0.117278	0.00014375	0.0301956

These numbers, along with the total cache size, all seem very reasonable.

Work Completed by Each Partner

Mei Siu Lei: Co-implemented nextline/stride, implemented open-ended cache replacement policy, wrote benchmarks for nextline/stride, relevant sections of report

Carl Lam: Co-implemented nextline/stride, implemented RPT table eviction policy, wrote benchmarks for open-ended, relevant sections of report