ECE552: Computer Architecture

University of Toronto
Faculty of Applied Science and Engineering
Lab 4

Prepared By	Danja Papajani Kathryn Tremblay

Q1

The next-line prefetcher we implemented is as described in the handout, where upon an access, it simply prefetches the next cache line (ADDR + cache_line_size), if that cache line has not already been inserted in the cache. To ensure our implementation had the correct functionality, we created a microbenchmark that accesses an array with a certain stride, and tested that for a given stride, the number of L1 data misses and the L1 data miss rate matched our expectations. The config file (q1.cfg) is the same as cache-lru-nextline.cfg, where each block is 64B. Since we used an array of integers (4B each), this means 16 elements of the array fit in one cache line. So to ensure our next-line prefetcher is able to correctly prefetch, we accessed every 16th element of the array, which is the first element of the next cache line (array[0]-array[15] are in one cache line, and array[16]-array[31] are in the next, and so on). When the next-line prefetcher is run with this access pattern, we saw very few misses, and a miss rate of 0%. We further tested by then accessing an element in the array that's in the cache line after the next, by incrementing 16 to 32. This resulted in a higher number of misses, which showed that once the program accesses elements that aren't in the next cache line, the performance decreases, as expected.

Q2

The microbenchmark created to test the stride prefetcher is similar to that explained in Q1. This prefetcher is able to perform well given a program that accesses memory with a steady stride. This is because of how the stride field in the RPT table transitions can associate the prefetch with a level of confidence, where a cache line is only prefetched as long as the RPT entry is not in the No Prediction state, and the cache line is not already in the cache. For q2.cfg, we used the same cache config as cache-lru-stride.cfg where each block has a size of 64B, so 16 integers fit on one cache line. We analyzed that the miss rate remained low as long as the access stride remained constant. We tested that the stride prefetcher was able to perform better than the next-line prefetcher by accessing an element in the array that's in every other cache line. We saw that the miss rate remained very low (0% miss rate) as long as the array was accessed with a consistent stride.

ECE552: Computer Architecture

Q3

Table 1. Collected statistics from benchmark compress

Config	L1 Miss Rate (%)	L2 Miss Rate (%)	Average Access Time
Baseline	4.16	11.4	1.89
Next-line	4.19	8.38	1.770
Stride	3.77	6.66	1.628

$t_{avg} = t_{L1} + \%miss_{L1}(t_{L2} + \%miss_{L2}(t_{mem}))$

Q4

The graph below shows the relation between the number of entries in an RPT and the L1 data miss rate (dl1.miss_rate). These statistics were collected by varying the range of RPT entries from 8 to 256. As one can see from Figure 1, there is a general correlation between increasing the numbers of RPT entries and decreasing the data misses that occur. These results confirm that adding more entries in the RPT allows more data to be available to the prefetcher, generating a lower miss rate.

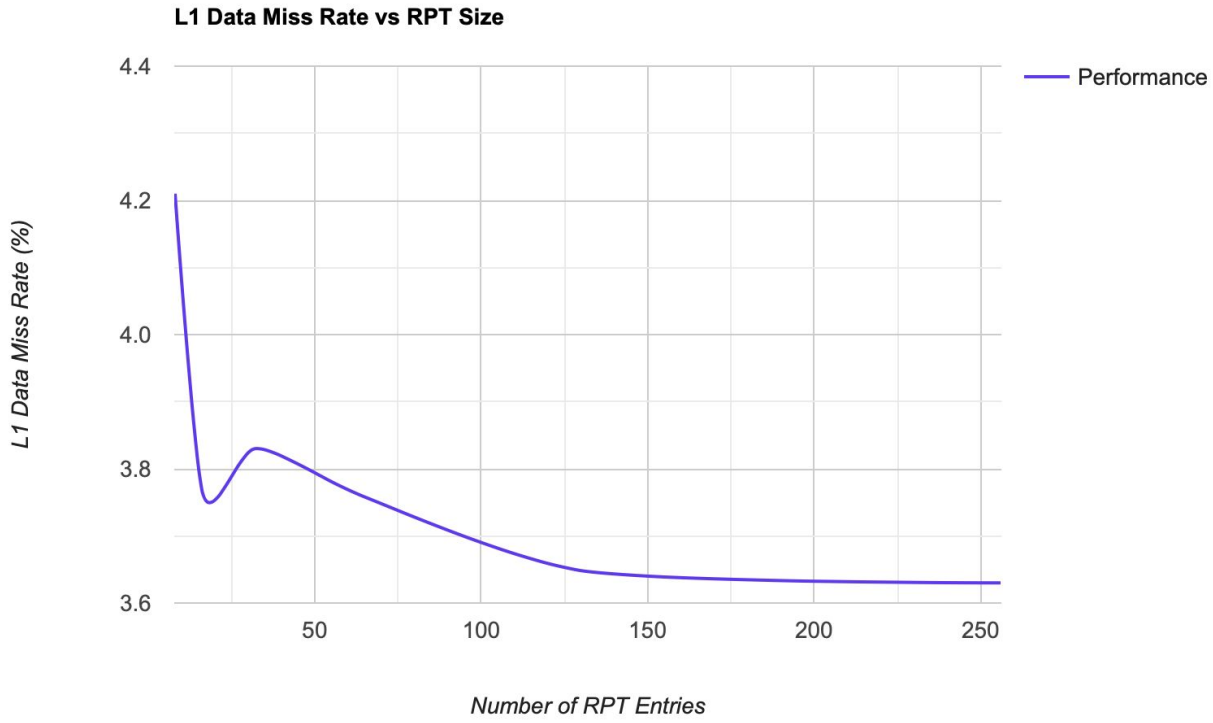

Figure 1. L1 Data Miss Rate for varying number of RPT Entries

Q5

One of the statistics I would include to measure the performance of a prefetcher include timeliness. By timeliness, I am referring to the clock cycles between how early a cache block is prefetched and when it is referenced. Let's say there is a loop where each iteration of the loop takes 5 cycles to complete. Let's also say that prefetching takes 20 cycles. In order to get useful data from the prefetch operation, we would need to start prefetching 20/5 = 4 iterations before it's use in order to maintain timeliness. In addition to timeliness, keeping track of the amount of useful data evicted due to a prefetch would help analyze the algorithm used to replace blocks. Ultimately, these two statistics would help us study the performance of any general prefetcher.

Q6

This microbenchmark was used to demonstrate how the open-ended prefetcher out-performs the stride prefetcher, since it accesses the array using multiple strides. The microbenchmark creates two strides, 9, and 100. Every other access is incremented by 9, and the other half is incremented by 100. We chose to modify q6.cfg from cache-lru-open.cfg by changing the block size from 64B to the minimum 8B, in order to simplify the visualization of the cache architecture. For this cache setup, 2 integers can fit in one cache line. This means that smaller strides can be used compared to the 64B block setup, because they won't be indexing into an already cached element of the array.

Since the open-ended prefetcher is able to keep track of various strides and only makes prefetches when it is confident, it was expected to perform better, and this was confirmed when running this microbenchmark against all 3 prefetchers. It is important to note that when running mbq6 with the configs q1.cfg and q2.cfg, they were slightly altered (we changed the block size from 64B to 8B) to match the cache setup of q6.cfg. The open-ended prefetcher had a miss rate of 0.0005 since it is capable of tracking 4 strides, so it could handle the alternating strides. The stride prefetcher had a miss rate of 0.0626, because it cannot handle more than a single consistent stride. The next-line prefetcher performed the worst, as expected, with a miss rate of 0.1249.

Explanation of the Open-Ended Prefetcher

The open-ended prefetcher implementation is a variation and extension of the stride prefetcher. This design is similar to the stride prefetcher in that it has an RPT and similar fields. However within the RPT, our implementation stores 4 strides and their respective state variable instead of just 1 stride and state. This allows for the prefetcher to retain some 'memory' of what strides it has seen in the past, instead of restarting from the beginning of the state transition diagram when it encounters a new stride. When the prefetcher sees a stride it has seen before, it 'upgrades' the state for that stride to be more confident and moves it closer to the Steady state.

Another optimization that this prefetcher has is that it only predicts when it is the most confident (the state is Steady), as opposed to the stride prefetcher which made predictions even when it

was in the Initial and Transient state, which makes it more likely to make false, unhelpful prefetches. The combination of multiple strides and their states being stored and only making a prediction when the prefetcher is confident results in lower miss rates when run against programs that exhibit multiple strides.

In this implementation, the size of the RPT was determined through experimentation to be 7000 entries, as this resulted in an average miss rate lower than 2.1% across the 3 benchmarks provided. CACTI results for this RPT structure show that for the open-ended prefetcher, the access time is 0.69252 ns and the size is 0.3735 mm^2, whereas the stride prefetcher has an access time of 0.14723 ns and the size is ~0.0006 mm^2. The size required is much larger and the access time is ~5x slower than that of the stride prefetcher, so if there is a memory constraint or a time constraint, the open-ended implementation is less attractive, but it may be worth the tradeoff given the lower miss rates observed.

Brief Statement of Work

KC Tremblay & Danja Papajani: worked on all deliverables (½ of workload each)