ECE552 Lab 4 Report

Marko Ciric (1006723967), Raymond Huynh (1007058238)

Next Line Prefetcher

The microbenchmark uses two tests to verify that the next line prefetcher works as intended. The first test is the ideal case, where the program only accesses memory in intervals of one block size, in this case 64 bytes. A loop iterates over every 16 elements in a contiguous integer array. Since an integer is 4 bytes, 4 bytes x 16 elements = 64 bytes, representing the 64 bytes in between each consecutive prefetch from the next line prefetcher. The code iterates over this loop 4 times, leading to 250,000 cache accesses from the array accessing. As expected, the prefetcher correctly prefetches the next element and causes a cache miss rate of 0% for this piece of code.

The second test is the worst case, where the program only accesses memory in intervals of two blocks (128 bytes). Since the prefetcher only prefetches one block away from the current address, it will cause a cache miss for every iteration of the inner loop. Iterating the inner loop, which accesses every 32nd element in the contiguous array, 4 times, should cause approximately 125,000 cache misses. In testing, we successfully achieved this number, with 125,017 total cache misses in the program.

Stride Prefetcher

The microbenchmark uses two tests to verify that the stride prefetcher works as intended. The first test validates that the prefetcher can correctly learn patterns in accessing memory from different instructions. Similar to the next line prefetcher microbenchmark, the stride prefetcher microbenchmark involves accessing an array every n elements, representing 4 x n / 64 block intervals. The next line prefetcher failed with 32 element intervals (2 blocks), so the first test is for the stride prefetcher to try and successfully prefetch for intervals of 2 and 4 blocks. Since the loops are different instructions, the prefetcher is successfully able to prefetch at both 2 and 4 blocks respectively, leading to a 0% miss rate as expected.

The second test validates that the prefetcher fails when the stride keeps changing for the same instruction, causing it to constantly be in between initial and transient. To do so, in this test the inner loop switches between incrementing by 32 and 64 elements every iteration. In this configuration, the inner loop executes a total of 83,336 times, and as expected, we see 83,374 cache misses in testing, which is within a margin of error.

Average Memory Access Time

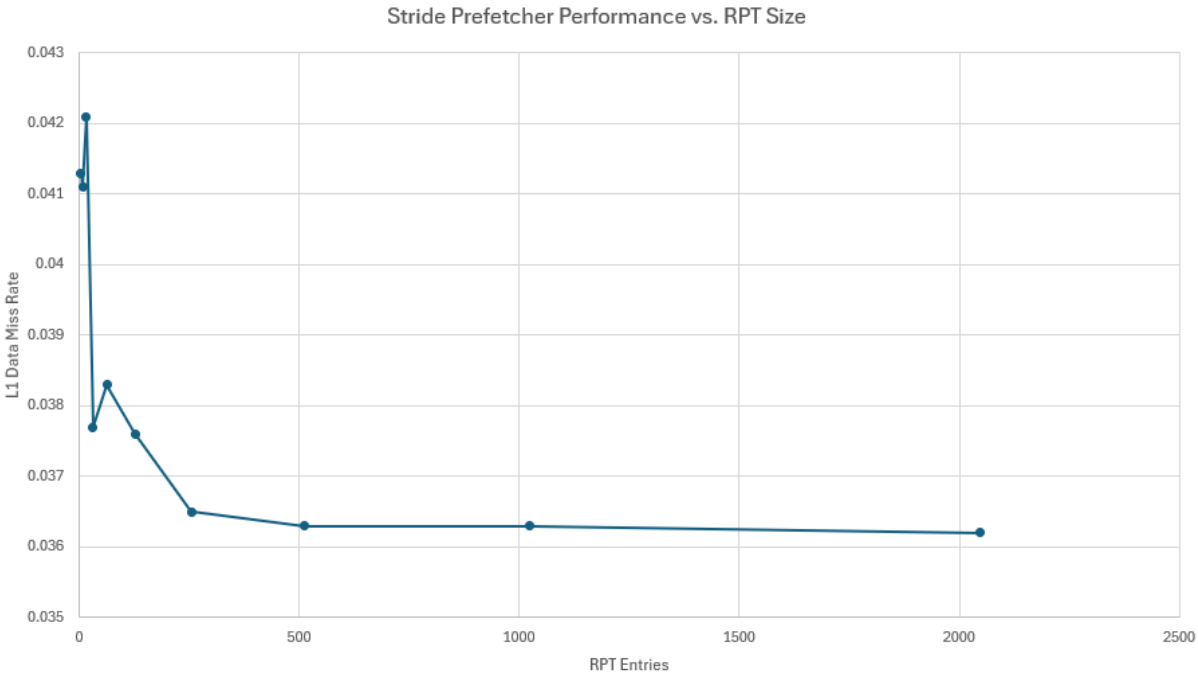
Knowing that accessing L1 data takes 1 unit, L2 data takes 10 units, and memory data takes 100 units. Inserting those hit times into the equation for average memory access time below:

$$1 \times (1 - L1_{Miss\ Rate}) + 10 \times (L1_{Miss\ Rate}) \times (1 - L2_{Miss\ Rate}) + 100 \times (L1_{Miss\ Rate}) \times (L2_{Miss\ Rate})$$

Config	L1 Miss Rate	L2 Miss Rate	Average access time
baseline	0.0416	0.1140	1.8012
next-line	0.0419	0.0838	1.6931
stride	0.0421	0.1109	1.7991

The formula is based on the possible hit/miss outcomes of accessing the memory hierarchy. Meaning if accessing L1 data was a hit, then the access time is equal to 1 unit of time multiplied by the hit rate ($1 - L1_{Miss\ Rate}$). But, if L1 misses and L2 is a hit then the access time is 10 units of time multiplied by ($L1_{Miss\ Rate}$) and the hit rate of L2 ($1 - L2_{Miss\ Rate}$). If both L1 and L2 caches result in a miss, then the time to access memory is 100 units of time multiplied by the miss rate of both L1 and L2.

Stride Prefetcher Performance in Relation to RPT Size



Generally, as the number of RPT entries increase, the L1 data miss rate decreases as well. Testing with numbers of RPT entries from 4 to 2048 on the compress benchmark, the miss rate drops from around 4.2% to around 3.6%. The L1 data miss rate can be considered a suitable metric for performance since a lower miss rate means that less time is spent retrieving data from the L2 cache or deeper, decreasing the overall latency of execution. At low numbers of entries, increasing the number of entries even slightly (e.g. 16 to 32) makes a massive improvement to the miss rate, shown by the steep curve on the right. However, there are clearly diminishing

returns at around 512 RPT entries and beyond, where the graph starts to even out. This implies that 512 RPT entries is enough to hold every memory-related instruction in the compress benchmark, therefore more entries only makes a marginal difference.

Additional Statistics

Some statistics in the sim-cache simulator that could be added to study the performance of prefetchers are the number of times a block was prefetched but not used, i.e. useless prefetches which can be estimated by (prefetch misses - prefetch hits). This would help track the number of prefetches polluting the cache before eviction. Another statistic is the average cache access time, similar to question 3, this can be used to determine whether the performance benefit of adding a prefetcher is consequential (higher than the baseline). Lastly, the number of times a prefetched block is accessed after the first hit can be used to quantify the prefetched data's usefulness over time.

Open-Ended Prefetcher

A delta-correlation data prefetcher was chosen for the open-ended prefetcher. The implementation was based on the pseudo-code and guidelines from a research paper [1]. From a high-level view, the delta-correlation prefetcher begins with a delta correlating prediction table, DCPT, where each entry of the table has a PC used for indexing into the table, a last address, last prefetch, a history of 16 deltas, and a pointer to a delta, this is present in the dcpt_entry struct in cache.h. The values of MAX_NUM_DELTAS and TABLE_SIZE were chosen small enough such that the access time of the cache is reasonably fast but large enough that the table can capture enough deltas and entries to have a low L1 data cache miss rate. A delta is the difference between consecutive memory addresses, so essentially this prefetcher leverages patterns in memory access deltas to predict future accesses. The DCPT is used to store observed delta patterns and their likelihood of recurrence. When a memory access does occur, the prefetcher finds the current delta and searches the DCPT for a matching pattern, it then generates prefetches based on the predicted deltas. Regarding the detailed implementation in code, first the DCPT entry is checked to check if it is a first access (meaning no match in the cache), if so, it is initialized. Then the delta is calculated and only if there is a difference between the address and the last address (if delta is not zero) will a prefetch be considered. Using a delta circular buffer where the buffer stores recent predicted addresses (strides), we then calculate the current delta and the previous delta in the sequence and generate a prediction, or candidate (a valid prediction), based on if there is a matched pattern. This check is iterated over a dynamic correlation window whose size is based on the amount of data in the buffer. Once a valid pattern is determined, we stop trying to find a candidate and the main delta-correlation algorithm is completed. Next in the DCPT flow, there is a prefetch filtering, limiting and issue process. One optimization that was added to the paper's algorithm was limiting the number of prefetches issued in one call where the intention is to reduce overhead and cache pollution. So, a prefetch request issues only if the candidate is a valid entry (meaning it was not recently fetched or

already in the cache) as this would be a useless prefetch and should be filtered out. After the prefetch issue then the batch_count would be incremented and checked such that a prefetch is only issued if the batch_count is below the MAX_PREFETCH, which was optimized to be 10, and if candidates are present. The L1 data cache miss rate results are detailed below:

L1 data cache miss rate	compress.eio	gcc.eio	go.eio	Average
dl1.miss_rate	3.69%	1.17%	1.1%	**1.987%**

Using the pureRAM.cfg and CACTI to confirm the feasibility of the delta-correlation prefetcher, the cache size was calculated to be 10,240B (delta table size * size of entry (dcpt_entry struct)), this is smaller than the 16KB L1 data cache provided, so it is realistic — a detailed calculation is provided as a comment in cache.c. A block size of 80B (size of dcpt_entry struct) and 4-way associativity (from cache-lru-open.cfg) led to an area overhead of 0.161563 mm^2 and access time of 0.481647 ns which are a reasonable increase compared to the baseline. The configuration file used for the microbenchmark, q6.cfg, differs from the CACTI configuration file since only values that are powers of 2 are allowed, rejecting 10,240B and 80B. However, these values are close enough to the cache-lru-open.cfg configuration, so that was used instead.

For the open-ended prefetcher microbenchmark, irregular strides (non-repeating patterns) and nested accesses were used to verify the correctness and performance of the delta-correlation prefetcher, as those are two patterns that the other prefetchers struggle with capturing. First, irregular accesses were implemented using an array of 1,000,000 elements and looping through the array using irregular indexes determined by a dynamic delta. Depending on whether the index value is divisible by 4 or 3, the delta can be either 17, 29, or -13; using both forward and backward strides also add to the irregularity. Then the index gets updated with the delta. The array is then accessed at that element assuming it is a valid entry. Next, to test the performance during a nested irregular access pattern, if the index is divisible by 5 then an inner loop executes with an irregular increment (j % 7 + 5), causing further irregular accesses to the array. Because of these irregular accesses (non-linear strides) which the other prefetchers are not designed well for, the stride prefetcher has a L1 data cache miss rate of 8.38%, while the delta-correlation prefetcher has a miss rate of 6.36%.

Statement of Work

Raymond oversaw the next line prefetcher and the stride prefetcher and their respective microbenchmarks and questions. Marko oversaw the open-ended prefetcher and its microbenchmark and questions. The report was written collaboratively.

Works Cited

[1] M. Grannaes, M. Jahre, and L. Natvig, "Storage Efficient Hardware Prefetching using Delta-Correlating Prediction Tables," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 4-6, Jan. 2011.