ECE552 - Lab 4 Report Pushkar Bettadpur | 999774971
Cache Prefetching Tirthak Patel | 999678969

SECTION 1: Provide succinct answers to all of the below questions. Make sure to include brief explanations for the mathematical derivations used to arrive at the answers.

Question 1: *Provide a micro-benchmark and a configuration file that you use to verify that your implementation of the next line prefetcher is correct. Explain your choice.*

In the file q1.c, we have implemented a microbenchmark that can be used to test our implementation of the next line prefetcher. It is a simple loop that accesses a large integer array (1 Million elements) with a stride of 1. The file q1.cfg defines our cache structure. With a cache line size of 8 bytes, we expect a cache miss for every 2 array accesses when pre-fetching is turned off. The data corresponds with our expectation as we get around 500,000 dl1 cache misses in this case. With our next-line prefetcher however, every access to element *i* means that elements *i+1, i+2 and i+3* should also be fetched into the cache. When we run our microbenchmark again with the prefetcher turned on, we see that the dl1 cache hits have gone up by around 500,000 and the corresponding misses have reduced by around 500,000 as well thereby proving the correctness of our next-line prefetcher. For detailed statistics and a look at our cache structure, please refer to mbq1.c and q1.cfg.

Question 2: *Provide a micro-benchmark and a configuration file that you use to verify that your implementation of the stride prefetcher is correct. Explain your choice.*

In order to test our stride prefetcher, we used the same array loop as above, except we access elements with a stride of 2. Given our cache-line size of 8 (as defined in q2.cfg), we expect all the 500,000 accesses to result in a cache-miss. With no prefetching, we get our expected value of around 500,000 dl1 cache misses. With stride prefetching turned on and the RPT having 16 entries, we expect our stride predictor to correctly record the fact that we are accessing elements with a fixed stride of 2. The results obtained show the number of dl1 cache misses reduce by around 500,000 thereby proving the validity of our stride prefetcher. For detailed statistics and a look at our cache structure, please refer to mbq2.c and q2.cfg.

Question 3: *Using the configuration files provided with the simulator and statistics collected from the simulator, estimate the average memory access time for data accesses for the benchmark Compress.*

Configuration	L1 Miss Rate	L2 Miss Rate	Average Access Time
Base-line	4.16%	11.40%	1.849
Next-line	4.19%	8.38%	1.728
Stride	4.13%	10.85%	1.820

Suppose:

```
Number of L1 Accesses = TA = 29063125        L1 Hit Time = L1T = 1
Memory Access Time = MT = 100 + L2T = 110    L1 Hit Time = L2T = 10
L1 Miss Rate / 100 = L1M                     L2 Miss Rate / 100 = L2M
```

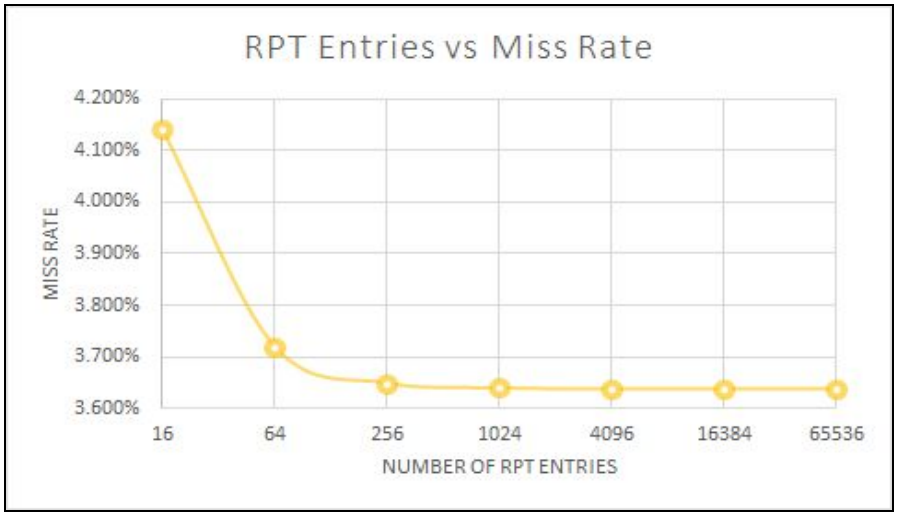
Then:

```
Total L1 Hits = L1H = TA*(1-L1M)
Total L2 Hits = L2H = (TA-L1H)*(1-L2M)
Total Memory Hits = MH = TA - L1H - L2H
∴ Average Access Time = (L1H*L1T + L2H*L2T + MH*MT) / TA
```

(Note: We considered the Memory Access Time to Include the time it takes for an L2 access to result in a miss. We also assumed the overall UL2 Miss Rate represents the data misses in the L2 cache)

Question 4: *For benchmark compress, study the performance of the stride prefetcher when varying the number of entries in the RPT.*

Number of RPT Entries	L1 Miss Rate
16	4.138%
64	3.718%
256	3.648%
1024	3.639%
4096	3.637%
16384	3.637%
65536	3.637%



The performance metric that we used is the miss rate: `(dl1.misses/dl1.accesses)*100%` . The results show that the performance improves exponentially as the number of entries increase. It saturates at around 4096 entries and increasing the number beyond that shows no improvement. Moreover, an RPT with 4096 entries shows only 0.5% performance improvement over 16 entries. So it might not be worth the space investment in low memory architectures with low performance requirements.

Question 5: *If you were asked to include more statistics in the sim-cache simulator to study the performance of prefetchers in general, which statistics you would consider adding? (No implementation necessary, only an explanation is sufficient.)*

It is very important to compute the latencies of the data prefetcher calculations. A complicated prefetcher with latencies greater than L2 access or Memory access times could negatively affect the performance instead of helping it. Hence, a statistic which calculates average latencies is needed.

Furthermore, it would also be useful to keep track of the data replaced by prefetched data and compute the miss-rate had that data been kept. This would help in assessing the efficiency of the replacement policy and check if harmful prefetches, which overwrite useful data, are being issued.

Question 6: *Provide a micro-benchmark and a configuration file that you use to demonstrate the performance of your open-ended prefetcher. Explain your choice.*

Our open ended prefetcher, as will be described below, builds on certain principles of the stride prefetcher. Primarily, it stores a historical pattern of *deltas* in order to correctly prefetch for complex access patterns (in cases where the stride predictor would fail). Subsequently, we used the array loop structure of the previous 2 benchmarks, but we reduce the number of iterations for the sake of simpler numbers. Our loop body is modified such that for every element *i*, the next element accessed is either *i+5* or *i+11* for every alternating iteration. This creates a complex stride pattern that the stride predictor shouldn't be able to predict, but our open ended predictor should. Our results reflect the same as when we provide q6.cfg as our cache-config. with prefetch type to the stride predictor, we get around 12614 dl1 misses but when we switch to the open-ended predictor, the dl1 misses reduce drastically to 140 thereby demonstrating the efficacy of our predictor. For detailed statistics and a look at our cache structure, please refer to mbq6.c and q6.cfg.

<u>SECTION 2</u>: Describe your open-ended data prefetcher implementation. Reason about how realistic your data prefetcher is in terms of area overhead and access time. Feel free to use CACTI from Lab assignment 2, to get real area and access time numbers.

For the open-ended data prefetcher, we have implemented the Data Correlating Prediction Tables (DCPT) prefetcher as proposed in [1]. The prefetcher builds on the traditional RPT and PC/DC

prefetchers to improve performance. It uses a table which stores the tag, last address, last prefetch and precomputed deltas, and thus, avoids the need to maintain a GHR and recompute deltas. Every time there is a cache access, the heuristic compares the most recent delta pair against previous delta pairs and requests prefetches upon matches. We are using a table with 512 entries and 6 deltas per table, as we empirically determined that this configuration gives the best performance.

The DCPT table takes about: 516*(9*(32)) bits = 18 KB of space. Intel's latest conventionally used i7 and Xeon processors have L1 cache sizes of 256 KB and 512 KB respectively [2]. The DCPT table would produce an overhead of 7.0% and 3.5% respectively for these caches. This is a reasonable overhead considering the performance gains achieved.

The DCPT prefetcher gives L1 Miss Rate of 3.63% and L2 Miss Rate of 4.67%, yielding an average access time of 1.496 for compress benchmark. This result is much better than the results we saw with the other two prefetchers owing to the lower L1 and L2 data cache miss rates. It produces an average miss rate of 2.0% across the three benchmarks provided.

SECTION 3: Include a brief statement of work completed by each partner.

Tirthak and Pushkar worked together on all the components of the lab.

REFERENCES

[1] M. Grannes, M. Jahre, and L. Natvig, "Storage Efficient Hardware Prefetching using Delta-Correlating Prediction Tables," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–16, Jan. 2011. [Online]. Available: http://www.jilp.org/vol13/v13paper2.pdf. Accessed: Nov. 17, 2016.

[2] "Intel core i7-4770K vs Intel Xeon E5-2690: 21 facts in comparison," in *VERSUS*, VERSUS, 2016. [Online]. Available: https://versus.com/en/intel-core-i7-4770k-vs-intel-xeon-e5-2690. Accessed: Nov. 19, 2016.