Bing Li, 1004191910
Tapasvi Patel, 1004176606

1.0 Next Line Prefetcher

Cache	Configuration	Test Hits	Test Misses
q1.cfg	Prefetch only for D1 cache **Nsets:** 64 **Blk size:** 64 bytes **Assoc:** 4 LRU Next-line	dl1.accesses: 706845 dl1.hits: 706830 dl1.misses: 15 dl1.prefetch_accesses: 158081 dl1.prefetch_hits: 0 dl1.prefetch_misses: 158081	dl1.accesses: 472467 dl1.hits: 420369 dl1.misses: 52098 dl1.prefetch_accesses: 105997 dl1.prefetch_hits: 0 dl1.prefetch_misses: 105997

Table 1: Results for mbq1.c

The next line prefetcher was validated in two ways. The first test was to test if hits were being generated (ie if the prefetcher was prefetching the correct data). This is shown in the Test Hits column of Table 1. A contiguous array of char characters (size very large) was generated and a for loop was run to access elements in the array. The first element accessed is at index 0, so assuming cache alignment, it would be found in the first block in the first set. Then, the prefetcher would prefetch the block on the second set, which is found at an address of the initial block + 64 (block size is 64 bytes). Thus, in the for loop, the next access would occur at array[initial + 64]. This should generate a cache hit cause the data has been prefetched. This access in turn will fetch the next line, and so on. Therefore, in the end, there should be very few cache misses, which is shown in the results in Table 1.

The second test was to test if only the next cache line was being prefetched. Therefore, the for loop was incremented every 3 cache lines. The first two access would generate prefetch to the second and third cache line, and the fourth cache line would cause a hit. Therefore, in the end, there should be a lot of cache misses, which is shown in the results in Table 1.

2.0 Stride Prefetcher

Cache	Configuration	Test Repeated Stride Pattern	Test Varying Stride Pattern
q2.cfg	Prefetch only for D1 cache **Nsets:** 64 **Blk size:** 64 bytes **Assoc:** 4 LRU Stride	dl1.accesses: 4021 dl1.hits: 3811 dl1.misses: 210 dl1.prefetch_accesses: 61 dl1.prefetch_hits: 0 dl1.prefetch_misses: 61	dl1.accesses: 4203 dl1.hits: 3977 dl1.misses: 226 dl1.prefetch_accesses: 62 dl1.prefetch_hits: 0 dl1.prefetch_misses: 62

Table 2: Results for mbq2.c

The stride prefetcher was evaluated in two ways. The first was testing if a repeated stride pattern was causing prefetches. The second was testing if a varying stride pattern was not causing prefetches. The results for testing a repeated stride pattern are shown in Table 2. In the benchmark, a for loop was lopped 50 times, and each loop a large array was indexed. The initial index was array[0], then array[64], then array[128] and so on. Thus, the stride pattern was 64. The non-benchmark code consisted of 12 prefetches. With the benchmark code and only the first for loop was 61. Therefore, 61-12 = 49 prefetches were done by the first for loop. This means the

strided pattern was working and the RPT was correctly updating and storing values. A more detailed table breakdown is given in mbq2.c.

The second test was checking if a varying strided pattern was not causing prefetches. This was done via another for loop, where the stride was increased by a factor of 2 every loop iteration. Thus, the PC entry in the RPT would not have been able to go to steady state, and no prefetches would be able to be done. The results are shown in table 2. An increase of prefetches from 61 to 62 was seen because in the second iteration of the second loop, a prefetch is done while the PC is in transient state. However, after that, no prefetches are done, and the calculations reflect this. A more detailed table breakdown is given in mbq2.c.

3.0 Average Memory Access Time

t_access_l1 = 1, t_access_l2 = 10, t_hit_mem = 100
From class notes: *t_avg = t_hit + %_miss*t_miss*
t_avg = t_access_l1 + l1_miss_rate(t_access_l2 + l2_miss_rate*t_hit_mem)*

- First get access to l1 data cache, and on miss, need to multiply l1 miss rate and average time it takes to get data further on
- Use same formula to get t_avg for l2 data cache, but this time use t_hit_mem for t_miss because next level after l2 is the main memory
- Combine both equations (tavg for l2 is t_miss for l1)

Config	**L1 Miss Rate**	**L2 Miss Rate**	**Average access time**
baseline	0.0416	0.1140	1.89024
next-line	0.0419	0.0496	1.626824
stride	0.0385	0.0616	1.62216

Table 3: Average access time for various prefetchers

4.0 Stride Prefetcher Performance

RPT sizes were chosen starting from 2, and then calculating powers of 2. At first glance, it would appear that increasing the RPT size should decrease the miss rate. This assumption makes sense because there will be fewer hashes that hash to the same location. The current hashing algorithm into the RPT takes the least significant bits of the PC (excluding the last 3 bits) and uses them to hash into the table. The number of bits taken depend on the table size (eg. 512 takes 9 bits). Initially, the miss rate starts high and then increases to a maximum at a table size of 8. Then the miss rate drops dramatically and levels out around a miss rate of 0.0387. With extremely low table sizes (2, 4), there are not many places to store unique address values in the RPT, thus each RPT entry contains state and stride information mixed from various addresses. The states transition in the transient state, thus there is some prefetching done (very little). With table size around 8, there is enough variation where some data can exhibit a stride pattern, and so more prefetching is done. However, the number of useful prefetches (data used later) is not that high, and it ends up evicting useful data from the cache, thus increasing the miss rate. After that, the miss rate drops considerably for table sizes greater than 8. This can be attributed to more addresses getting hashed on different RPT lines, thus being able to exhibit a better pattern. However, as the table size keeps increasing (512, 1024), the miss rate levels off. This can be

attributed to the fact that we are not fully using every entry in the RPT, as our hashing takes the same address bits every time. Thus, increasing the table size does not matter if we don't change the hashing algorithm to account for this. Therefore, the same addresses will still hash to the same lines, keeping the stride pattern the same and keeping the miss rate steady. A graph of RPT sizes not powers of 2 was also created, however, it produced zig zag results where every power of 2 size would have a low miss rate and non powers of 2 would have a miss rate around 0.042. RPT sizes not powers of 2 were not considered in the below graph.

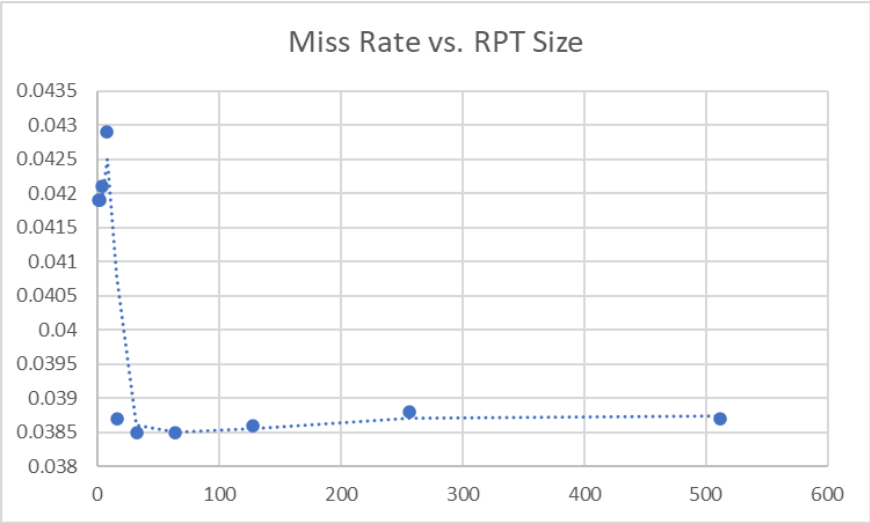

Figure 1: Miss Rate vs. RPT Size for Stride Prefetcher

5.0 Additional Data to Include in Simulator

Additional metrics to include:

- Whether a prefetch evicted data from the cache that was used in a following instruction
- Whether a prefetch was the reason for a cache hit to occur
- Number of cache misses that a prefetcher was able to eliminate
- What cycle a prefetch occurred and when that data was actually used by the program
- Whether a prefetch was actually used or evicted before it could be used

6.0 Openend Prefetcher

Our team implemented a CZome Delta Correlation prefetcher (C/DC)[1], with a few tweaks for simplicity. The C/DC splits the main memory into equal size segments, and tries to find a data access pattern within each segment, each having its own History Delta Table (HDT) which indicates the historical stride patterns and each segment has a corresponding previous miss address. The HDT is implemented as a circular buffer but operates more like a shift register, meaning that upon entry of a new delta, the oldest delta is removed. Upon a cache miss, the new delta is calculated and previous miss address is updated. The 2 most recent deltas become the correlation keys, and the HDT is traversed backwards until the correlation keys match to a pair of entries, in which case the next 2 deltas (a,b) are used as the prefetch deltas (i.e. the prefetch addresses are $addr+a$ and $addr+b$). If no pairs are found, no prediction is generated. If the 2 most recent deltas are the same, the prefetcher becomes a stride prefetcher and the 2 prefetch

[1] Kyle J. Nesbit, Ashutosh S. Dhodapkar, James E Smith, AC/DC: An Adaptive Data Cache Prefetcher, University of Wisconsin - Madison. Accessed: Nov 24, 2021 [Online]. Available: https://www.eecg.utoronto.ca/~moshovos/ACA05/read/AC_DC.pdf

addresses are *addr+s* and *addr+2s*. The size of the table is 256 entries. Therefore, the main memory is split into 256 regions. Each entry has a delta table that is 64 entries long along with some metadata (head and tail pointers and prev address stored). Each delta in the delta table is an int. Therefore, the entire size of the table (without metadata) is 256 * 64 * 4 = 65,536 bytes. This is the software implementation, and the hardware can be optimized to account for minimum delta length. For access time, each entry in the table can be hashed by an address that only needs 8 bits (2^8 = 256) and each delta table is sequential in memory. Therefore, this design is feasible for hardware as the memory layout is minimum and the access time is also optimized.

6.1 Openend Microbenchmark

Cache	**Configuration**	**No-prefetcher**	**Establish delta pattern**	**Check delta pattern**
q6.cfg	Prefetch for D1 **Nsets:** 64 **Blk size:** 64 bytes **Assoc:** 4	miss_rate: 0.0010 misses: 31467	num_accesses: 15003720 num_misses: 6803 miss_rate: 0.0005	num_accesses: 30003722 num_misses: 13481 miss_rate: 0.0004

Table 4: Results for mbq6.c

The openend prefetcher was tested using the microbenchmark found in mbq6.c. There were a couple of tests done to check the validity of the prefetcher. Our openend solution works on the principle that a delta pattern occurs when accessing a segment of the main memory. The main memory is divided into chunks, and an access pattern is sought to predict the next address to prefetch address given a current pattern. In the microbenchmark three tests were run. The first was without a prefetcher, to determine the number of misses and the miss rate. This result is shown in table 4. The next test was a for loop accessing a large array and establishing an access pattern. The pattern chosen was 2, 3, 4. Thus, when the for loop runs, it should store this pattern in an entry in the table (or multiple entries since the array is very large) and when an access pattern of 2, 3 is seen, it should use 4 to predict the next prefetch address. The last test was run using two for loops, the first one setting the access pattern, the second using the access pattern to predict. The openend was working because for very large access, there were a small number of misses compared to the no prefetcher, thus the openend was storing the correct access pattern and predicting the correct next prefetch address. The results are shown in table 4. A more detailed breakdown is given in mbq6.c

6.2 Openend Results on Benchmarks

Cache	**compress (d1 miss rate)**	**gcc (d1 miss rate)**	**go (d1 miss rate)**	**average (d1 miss rate)**
q6.cfg	0.0391	0.0122	0.0116	0.0209666

Table 5: Results for Openend Prefetcher

7.0 Statement of Work

Bing: opened prefetcher, mbq6.c, q6.cfg, report
Tapasvi: nextline and stride prefetcher, mbq1.c, mbq2.c, q1.cfg, q2.cfg, report