ECE552 Lab 5 Data Caches

Freddy Chen 997363124 Anthony Alayo 997487401

Question 1:

mbq1.c:

- Allocate an array that's too big for the L1 data cache so there will be misses and prefetching
- Access elements with indices of increments BSIZE (right into the next block fetched) to see the next line prefetcher work well (0.21% miss rate)
- Access elements with indices of increments 2*BSIZE (right over the next block fetched) to see the next line prefetcher fail (53.38% miss rate)

```
 1 #include <stdio.h>
 2
 3 // Define variables based on the configuration of the l1 data cache config
 4 // <name>:<nsets>:<bsize>:<assoc>:<repl>:<pref>
 5 #define NSETS 64
 6 #define BSIZE 64
 7
 8 int main(void) {
 9   int arr_size = 10000 * NSETS;
10   char arr[arr_size];
11   int trash_sum = 0; // To ensure the compiler doesn't optimize the loop away
12
13   // If increment is BSIZE*2, we will try to access an element that's just over
14   // the block that got prefetched by the nextline prefetcher, and our miss
15   // rate becomes 53.38%.
16   // If increment is BSIZE, we will access an element in the prefetched block,
17   // and our miss rate becomes 0.21%.
18   int increment = BSIZE;
19   int i;
20   for (i = 0; i < arr_size; i = i + increment) {
21     trash_sum += arr[i];
22   }
23   printf("trash_sum %d\n", trash_sum);
24   return 0;
25 }
```

Question 2:

mbq2.c:

- Allocate an array that's too big for the L1 data cache so there will be misses and prefetching
- Access elements with indices of different strides (BSIZE or 2*BSIZE depending on the current index) each time to see the stride prefetcher fail (60.21% miss rate)
- Access elements with indices of constant strides (BSIZE) to see the stride prefetcher work well (0.45% miss rate)

```c
#include <stdio.h>

// Define variables based on the configuration of the l1 data cache config
// <name>:<nsets>:<bsize>:<assoc>:<repl>:<pref>
#define NSETS 64
#define BSIZE 64

int main(void) {
  int arr_size = 10000 * NSETS;
  char arr[arr_size];
  int trash_sum = 0; // To ensure the compiler doesn't optimize the loop away

  int i;
  for (i = 0; i < arr_size; i++) {
    trash_sum += arr[i];

    // If we offset i by a different stride each time, the stride prefetcher
    // won't do a good job predicting which block to fetch, and our miss rate
    // becomes 60.21%.
    // If we offset i by a constant stride, the stride prefetcher does well
    // and our miss rate becomes 0.45%.
    i = i % 2 ? i + BSIZE : i + 2*BSIZE;
    //i = i + 2*BSIZE;
  }
  printf("trash_sum %d\n", trash_sum);
  return 0;
}
```

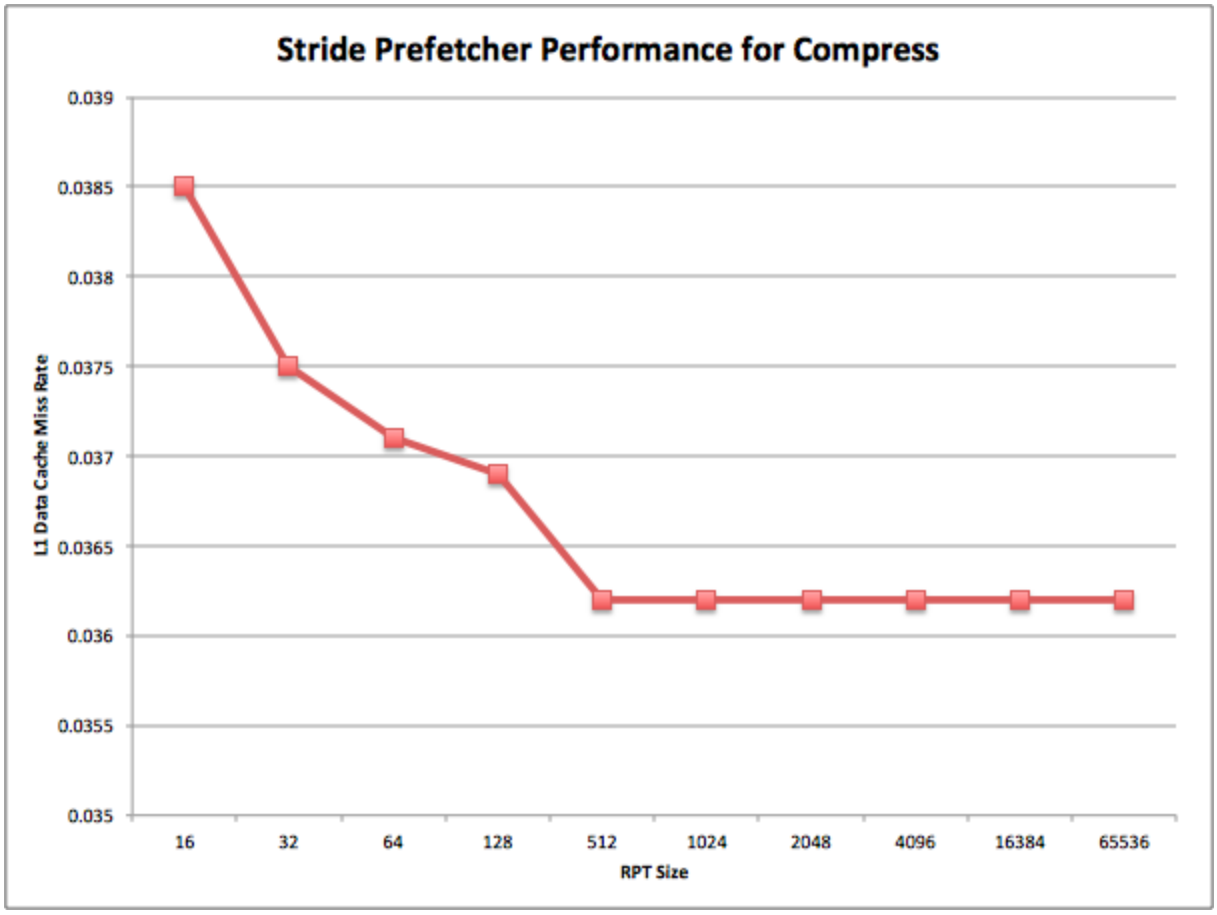
Question 3:

Config	**L1 Miss Rate**	**L2 Miss Rate**	**Average access time ***
no prefetcher	0.0416	0.1140	1.89024
next line	0.0419	0.0838	1.770122
stride	0.0385	0.0578	1.60753

* Average access time calculated using the following formula:
TAVG = Taccess-L1Data + L1 miss rate * (Taccess-L2 + L2 miss rate * Thit-Memory)

This formula is derived in class. We always have to look in the L1 data cache, and of those accesses that miss, we always have to look the L2 cache, and of those accesses that miss, we look to memory.

Question 4:



The miss rate for the stride prefetcher decreases monotonically as we increase the size of the RPT for the stride prefetcher. This makes sense because we have more entries in the RPT for making stride predictions and we evict less entries as we increase our size (less aliasing). As a result, our stride prefetcher keeps its states for more instructions, making it a better prefetcher.

We think we don't see any performance gains after an RPT size of 512 because it is enough to fit the working set for the entire duration of the compress benchmark. In other words, the additional space is not being used.

Question 5:
It would be useful to include statistics about the average accuracy and coverage of each prefetch. The accuracy statistics will let us know whether or not our prefetcher is bringing in a lot of useless blocks. The coverage statistics will let us know how many misses we saved as a direct result of prefetching a block.

Question 6:
Our open prefetcher is a hybrid prefetcher that uses the stride prefetcher first to see if we can make a prefetch, and if not, we fall back to looking into a data address miss queue for any history

of the address we missed on.

Each time data access is missed in the cache, the address gets enqueued in the data address miss queue, effectively forming a history of data addresses missed.

There is a slight modification for the stride prefetcher in that we only make a stride prefetch if the entry is in steady state. Otherwise, we go straight away into looking through our history of data address misses.

If we find our address in the data address miss queue, we prefetch the next address in the queue and hope that that will be the next data address accessed.

For our open ended prefetcher, we use the following parameters (as seen in cache.h):

- entries in RPT: 1024
- spots in data address miss queue: 2048

From those parameters, we can calculate how many bits are required for those two structures:

- RPT: 1024 entries x ((32 - 10 - 3) tag bits + 32 prev-addr bits + 32 stride bits + 2 state bits) = 87040 bits = 10880 bytes = 10.625kB (smaller than our 16kB L1 data cache)
- data address miss queue: 2048 * 32 addr bits = 65536 bits = 8192 bytes = 8kB (half the size of our 16kB L1 data cache)

Here is a comparison of our new structures with the L1 data cache (estimated with CACTI):

Structure	Access time (ns)	Cache height x width (mm):
L1 Data Cache	0.499477	0.134653 x 0.210113
RPT	0.259995	0.26398 x 0.0682748
Data address miss queue	0.271737	0.135717 x 0.0978843

From the table above, we believe these parameters make this open prefetcher feasible.

mbq6.c:

- Allocate an array that's too big for the L1 data cache so there will be misses and prefetching
- Access elements with indices of different strides (BSIZE or 2*BSIZE depending on the current index) each time to see the stride prefetcher fail
- Based on our strides, we will access a lower bound total of arr_size / BSIZE data addresses. Or:
  total data address accesses = arr_size / BSIZE
- If total data address accesses < size of our miss queue, then our open prefetcher performs wonderfully with a miss rate of 0.01%
- If total data address accesses > size of our miss queue, then our open prefetcher (and

stride prefetcher) will fail and get a miss rate of 100%.

```c
#include <stdio.h>

// Define variables based on the configuration of the l1 data cache config
// <name>:<nsets>:<bsize>:<assoc>:<repl>:<pref>
#define NSETS 64
#define BSIZE 64

// The open prefetcher is a hybrid of the stride prefetcher and a data address
// miss queue. Each time a data access is missed in the cache, it gets enqueued
// in the data address miss queue.
// The open prefetcher first checks whether or not we can do a stride prefetch.
// We only do a stride prefetch if the entry is in the steady state.
// Otherwise, we check the data address miss queue and look for the address
// that we missed. If we find it, we return the next address in the queue and
// hope that that's the next data address being accessed.
int main(void) {
  // Depending on how big arr_size is, the data that we access might not
  // fit entirely into the data address miss queue used by our open predictor.
  // If the size is 1000 * NSETS, then all of the data that we access will fit
  // into our miss queue ((1000 * NSETS) / BSIZE = 1000 < 2048), and our miss
  // rate will be 0.01%!
  // If the size is 10000 * NSETS, then the data that we access won't fit into
  // our miss queue ((10000 * NSETS) / BSIZE = 10000 > 2048), and our miss rate
  // will be 100%.
  int arr_size = 1000 * NSETS;
  char arr[arr_size];
  int trash_sum = 0; // To ensure the compiler doesn't optimize the loop away

  // We have an outer loop here so that the open prefetcher is able to take
  // advantage of the data address miss queue
  int i, j;
  for (j = 0; j < 100000; j++) {
    for (i = 0; i < arr_size; i++) {
      trash_sum += arr[i];

      // If we offset i by a different stride each time, the stride prefetcher
      // won't do a good job predicting which block to fetch. We will fall back
      // to looking into the data address miss queue in this case.
      i = i % 2 ? i + BSIZE : i + 2*BSIZE;
    }
  }
  printf("trash_sum %d\n", trash_sum);
  return 0;
}
```

Statement of work:

Freddy Chen - *code, microbenchmarks, lab report* Anthony Alayo - *code, simulations, lab report*