ECE552 Lab 4 Report

Viet Minh Nguyen – 1007503989

Question 1:

The micro-benchmark I created for the next line prefetcher consists of 2 programs:

1) Array of char (similar size to byte_t), and a for loop that access elements sequentially

```
int main() {
        // 1 MB array
        char* arr = (char*)malloc(sizeof(char) * 1000000);
        int i;
        for (i = 0; i < 1000000; i += 1) {
                arr[i] = (i % 128);
        }
        return 0;
}
```

This should have a very small missrate. Each time we have a miss, we load the entire block (64 bytes), along with the next line (another 64 bytes). Therefore, if we simply access the data sequentially, there should be small number of misses. **d1.miss_rate = 0**

2) For loop access array with step of 128.

```
int main() {
        // 1 MB array
        char* arr = (char*)malloc(sizeof(char) * 1000000);

        int i;
        for (i = 0; i < 1000000; i += 128) {
                arr[i] = (i % 128);
        }

        return 0;
}
```

This should have very high missrate. As mentioned above, if we then make the steps be 128 (64 + 64) to intentionally avoid the 2 blocks that are prefetched, there should be a lot of misses in the program. **dl1.miss_rate = 0.6723**

Question 2:

The stride prefetcher relies on the idea that memory accesses usually follow a regular and predictable pattern (strides), especially nested loops. Therefore, we created 2 programs, 1 with a simple traversal pattern (left, **d1.miss_rate = 0**), and 1 with irregular traversal pattern to check the behaviour of the stride prefetcher (right, **d1.miss_rate = 0.3109**)

```
int main () {
        int a = 74;
        int b = 29;
        int i; int j;

        char* arr = (char*)malloc(sizeof(char) * 1000000);
        for (i = 0; i < 100000; i++) {
                for (j = 0; j < 800; j += a) {
                        arr[j] = (j % 128);

                        if (a == b) a = b/2;
                        else a = b;
                }
        }
}
```

```
int main () {
        int a = 74;
        int b = 29;
        int i;

        char* arr = (char*)malloc(sizeof(char) * 1000000);
        for (i = 0; i < 1000000; i += a) {
                arr[i] = (i % 128);

                if (a == b) a = b/2;
                else a = b;
        }
}
```

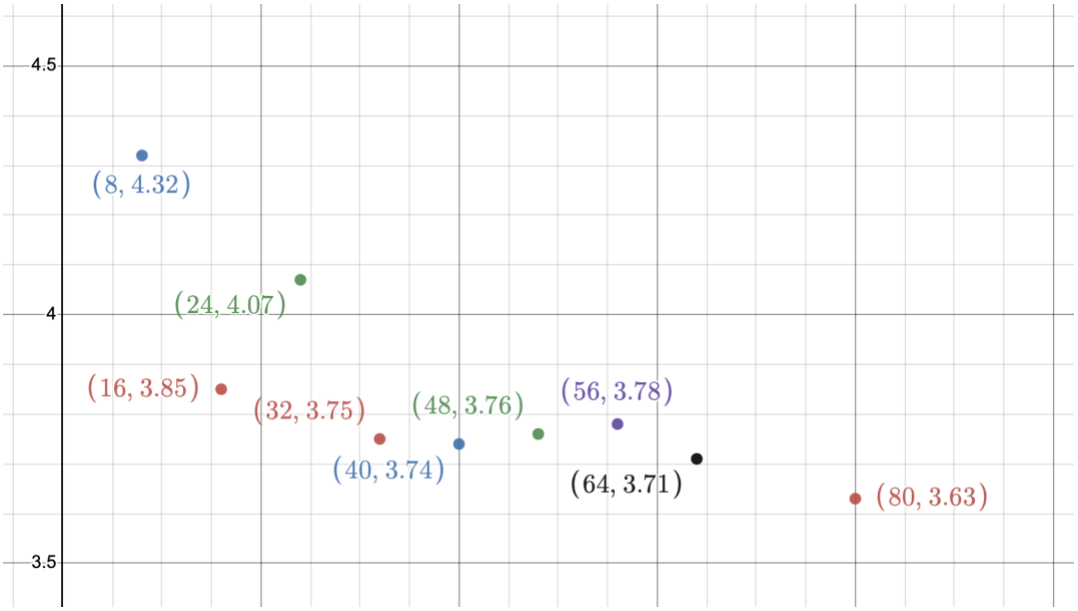
Question 3:

$$T_{avg} = T_{L1} + \ miss\ rate_{L1} * \left(T_{L2} + \left(miss\ rate_{L2} * T_{memory}\right)\right)$$

Given: $T_{L1} = 1, T_{L2} = 10, T_{memory} = 100$

Config	L1 Miss Rate	L2 Miss Rate	Average Access Time
Baseline	0.0416	0.1140	1.89024
Next-line	0.0419	0.0838	1.77012
Stride	0.0385	0.0578	1.60753

Question 4:



Number of RPT entries (x axis) vs L1 data miss rate (y axis)

As we can observed from the graph, the trend is: as the number of RPT entry increases, the miss rate decreases (except for some cases where this may not be exactly correct, but it heavily depends on the micro-benchmark itself). The reason is that as the size of the RPT increases, the stride can simply more easily capture the pattern (like the history register), and avoid evicting useful predictions.

Question 5: I can add the average access time, which can include both the theoretical value similar to the calculation in Question 3, and the actual average access time for each cache level (and all memory access operations), since the given program in sim-cache is already recording access time for each cache_access operations (return value of cache_access). More details on the cache operations can also be expanded from this, like access time between load/store

operations, longest/shortest access time, etc., all can give us insights into the behaviour of each prefetcher and can improve its design.

Question 6 + Open-ended description:

The open-ended prefetcher design I chose expands on the stride prefetcher with a RPT, and added a global address history buffer (of 16 entries) that records the previous memory access addresses. The open-ended design turns out to improve the cache performance for all 3 benchmarks compared to the basic stride prefetcher, even with the same number of RPT entries.

Since the GHB records addresses of previous memory access, we the new design only does prefetching based on (current address + entry stride) if the entry state is STEADY. Otherwise, the prediction will be, directly, the memory access address that followed the current address in the past (searching through GHB records). Therefore, intuitively, this design is trying to give a more guaranteed prediction of the next memory access address in every scenario.

This design is easy to be implemented because it simply adds a GHB compared to the basic one.

With 16 RPT entries (similar to the default in basic stride – miss rate decreases if add more)

- compress: dl1.miss_rate = 0.0370 (0.0385 for basic stride)
- gcc: dl1.miss_rate = 0.0123 (0.0126 for basic)
- go: dl1.miss_rate = 0.0115 (0.0163 for basic)

→ Average miss rate = **0.0203** (< 0.21).

```c
// Matrix multiplication (stride prefetching should be good here (easy pattern)
// Create 2D array of size 1000 x 1000, initialize to random values
// Also, have an additional miss queue
int main () {
        int i; int j; int k;
        int epoch;
        int** A = (int**)malloc(sizeof(int*) * 1000);
        int** B = (int**)malloc(sizeof(int*) * 1000);
        int** C = (int**)malloc(sizeof(int*) * 1000);
        for (i = 0; i < 1000; i++) {
                A[i] = (int*)malloc(sizeof(int) * 1000);
                B[i] = (int*)malloc(sizeof(int) * 1000);
                C[i] = (int*)malloc(sizeof(int) * 1000);
        }
        // Initialize matrices
        for (i = 0; i < 1000; i++) {
                for (j = 0; j < 1000; j++) {
                        A[i][j] = i * 13 + j;
                        B[i][j] = j * 47 - i;
                        C[i][j] = 0;
                }
        }

        // Matrix multiplication
        for (i = 0; i < 1000; i++) {
                for (j = 0; j < 1000; j++) {
                        for (k = 0; k < 1000; k++) {
                                C[i][j] = A[i][k] * B[k][j];
                        }
                }
        }
}
```

Given that this expands on the functionalities of the basic stride design, we can have the same arguments about how this design is good for detecting patterns in the program. I added a program that does 2D matrix access (for initialization) and 2D matrix multiplications. For a easy-to-predict pattern, the miss rate is indeed low: **dl1.miss_rate = 0.0186**