

Lab4 report

Question 1: next line prefetcher microbenchmark

```
#define STRIDE 128

int main(){
    char array[ARR_SIZE]; //each element has a size of 1 byte
    char a;
    int i;
    for(i = 0; i < 1000000; i += STRIDE) // 3 accesses to i
        a = array[i]; // 1 access to i, 1 access to a, 1 access to array[i]
}
```

The cache configuration we used is a L1 data cache that has a block size of 64, a set size of 64, an associativity of 1, and a replacement policy of LRU. There are 6 memory accesses for each iteration of the for loop: 4 accesses to get the value of “i”, 1 access to get “a”; 1 access to the value of “array[i]”. Since the stride is 128 and the block size is 64, the memory accesses to the array are always 2 blocks apart, which means there is a miss to every array access. So, the overall miss rate for L1 data cache will be $1/6=16.7\%$. This is the same as the design with no prefetcher used. However, if we changed the STRIDE to be 64, then the miss rate is close to zero, which is better than no prefetching. The actual miss rates for STRIDE = 64 and STRIDE = 128 are and 15.96%, 0.8%, which matches our expectations.

Question 2: stride prefetcher microbenchmark

```
#define STRIDE 128
#define STEP 2
#define ITER 1000000
#define ARR_SIZE STRIDE * STEP * ITER
int main(){
    char array[ARR_SIZE];
    char a;
    int i;
    int arr_idx = 0;
    for(i = 0; i < ITER; i ++){ // 3 accesses to i
        if(i % 2 == 0) // 1 access to i
            arr_idx += STRIDE; // 2 accesses to arr_idx
        else
            arr_idx += STRIDE * STEP; // same as above
        a = array[arr_idx]; // 1 access to a, 1 access to arr_idx, 1 access to array[arr_idx]
    }
}
```

The cache configuration has a L1 data cache with a block size of 64, a set size of 64, an associativity of 1, replacement policy of LRU, and a rpt size of 64. There are 9 memory accesses for every iteration of the for loop: 4 accesses to i, 3 accesses to arr_idx; 1 access to a; 1 access to array[arr_idx]. If STEP is not 1. The arr_idx is incremented differently for every other iteration. The reference prediction table will not be able to detect such pattern thus having a miss prediction in every iteration, giving a miss rate of $1/9 = 11.11\%$. If STEP is 1, the arr_idx is incremented by STRIDE for every iteration. The stride prefetcher will be able to detect the pattern, thus giving a miss rate that is close to zero. The actual miss rate for STEP = 1 and STEP = 2 is 1.04% and 11.45%, which matches our expectation. Note that for the next-line prefetcher, even if the step size is 1. It will not be able to predict the correct next block to fetch since the array elements are two blocks apart.

Question 3: average memory access time

The average access time is calculated by $T_{ave} = T_{access-L1Data} + \%miss_{L1} * (T_{access-L2} + \%miss_{L2} * T_{hit-Memory})$ Assuming the following hit times: $T_{access-L1Data} = 1, T_{access-L2} = 10, T_{hit-Memory} = 100$.

$$T_{ave}(\text{baseline}) = 1 + 0.0416 * (10 + 0.1140 * 100) = 1.89024$$

$$T_{ave}(\text{next-line}) = 1 + 0.0419 * (10 + 0.0838 * 100) = 1.770122$$

$$T_{ave}(\text{stride}) = 1 + 0.0385 * (10 + 0.0578 * 100) = 1.60753$$

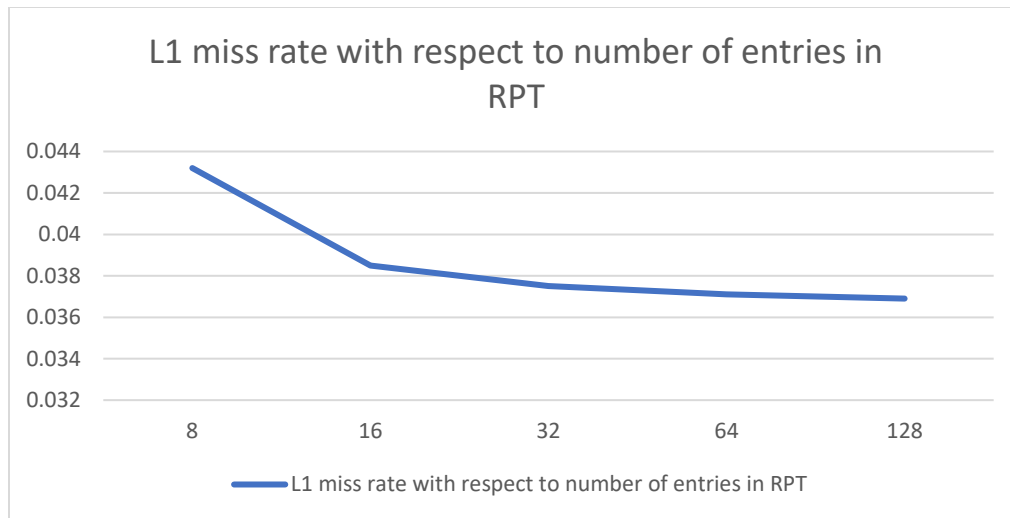
The L1, L2 miss rate and the average access time for designs with no prefetcher, next-line prefetcher and stride prefetcher is given by the following table.

Config	L1 Miss Rate	L2 Miss Rate	Average access time
Baseline	0.0416	0.1140	1.89024
next-line	0.0419	0.0838	1.770122
stride	0.0385	0.0578	1.60753

Question 4: performance of the stride prefetcher with varying RPT entry number

The graph depicts the change of L1 miss rate as the number of entries in RPT varies. As the number of entries in RPT increases the L1 miss rate decreases. However, the benefit diminishes as the number of entries exceeds a certain point.

Increasing the number of entries in RPT reduces the amount of collisions in RPT since there are more bits used to index into the table. The benefit diminishes after a certain point since the program has limited amount of load and store instructions that execute in a loop.



Question 5: additional metrics

I would add the number of useful prefetches as a metric to calculate the performance of prefetching. This metric can be obtained by recording the number prefetched blocks that are used by later instructions. By adding this metric, we can have a better idea about the accuracy of prefetching and whether the prefetching pollute the cache with unnecessary data. I would also consider adding metrics to distinguish between the miss types, which are compulsory, capacity, and conflict misses. By adding this metric, we can understand what types of misses different prefetchers affects and how much the prefetchers affect them. For example, one prefetcher that reduces 4 compulsory misses and introduces 3 capacity misses is different from a prefetcher that reduces 10 compulsory misses and introduces 9 capacity misses. But they have the same number of misses. By having this metric, we can understand better how well the prefetcher is at reducing compulsory misses and how much it pollutes the cache.

Question 6: open-ended prefetcher microbenchmark

```
#define BLKSIZE    128
int main(){
    char array[10000000];
    char b;
    int i = 0;
    int j = 0;
    int pattern[] = {1,2,3,4,5,6};
    while(i < 10000000){
        b = array[i];
        i += pattern[j]*BLKSIZE;
        j = (j+1)%6;
    }
}
```

The performance of the open-ended prefetcher is demonstrated by comparing against the stride prefetcher. The stride prefetcher is not able to remember varying pattern, meaning it can only remember constant stride. However, a delta prefetcher can remember patterns since it has a delta buffer recording stride history. In this example, we use a fixed pattern for the array index. The

stride prefetcher gives a miss rate of 10.18% while the delta correlation prefetcher gives a miss rate of 2.69% which is way better than the stride prefetcher. The cache configuration of the open-ended prefetcher is the same as the stride prefetcher except for the prefetch type.

Open-ended prefetcher

The open-ended prefetcher is a delta-correlating prefetcher. The heuristic combines both reference prediction table and PC/DC prefetching (calculating the deltas between the successive cache misses and stores them in a delta-buffer) by using a table-based approach. In DCPT (delta-correlating prediction table), each entry has a “last address” field, a “last prefetch” field, a “delta buffer”, and a “delta pointer” pointing to the head of the circular buffer. Each entry is indexed by the PC. Initially, the PC is used to look up into a table of entries. If an entry with the corresponding PC is not found, then a replacement entry is initialized. If an entry is found, the delta between the current address and the previous address is computed. If the delta is not zero, the new delta is inserted into the delta buffer and the last address field is update. Delta correlation begins after updating the entry. The deltas are traversed in the reverse order, looking for a match to the two most recently inserted deltas. If a match is found, the first prefetch candidate is generated by adding the delta after the match to the value found in last address. The next prefetch candidate is generated by adding the next delta to the precious prefetch candidate. The process is repeated for each of the deltas after the matched pair including the newly inserted deltas. The next step is prefetch filtering. If a prefetch candidate matches the value stored in last prefetch, the content of the prefetch candidate buffer up to this point is discarded. Every prefetch candidate is looked up in the cache to see if it is already present. If it is, the prefetch is discarded. Third, the candidate is checked against a buffer holding other prefetch requests that have not been completed. This buffer can hold 32 prefetches. If it is full, the prefetch is discarded in FIFO order. Finally, the last prefetch field is updated with the address of the issued prefetch.

The area overhead and access time for DCPT and L1 data cache is given as follows:

	Configuration	Area (mm ²)	Access latency (ns)
DCPT (64 entries, each entry is 48 bytes)	Type: RAM Size: 3072 bytes Block size: 48 bytes	0.00509838	0.234512
L1 Data cache	Type: Cache Size: 16384 bytes Block size: 64 bytes Associativity: 4 Tag: 20 bits	0.0244037	0.338137 (data side) 0.146367 (tag side)

It is realistic to implement the prefetcher since the area of DCPT is negligible compared to that of L1 data cache. The access latency is smaller than that of L1 data cache which does not introduce too much access time overhead.

Work distribution

Tianyi Xu: implemented next-line, stride prefetcher, wrote microbenchmark and configuration files, wrote report.

Hao Wang: implemented open-ended prefetcher