ECE552: Computer Architecture
Assignment 3: Data Caches
Due December 2, 2011

Canna Wen
996117205
wencanna@ecf.utoronto.ca

## 1. Introduction

This lab uses sim-cache.c to simulate the cache behaviour of programs. We experimented with data prefetchers (next line and stride) to see their effects on cache misses. Data prefetching is a useful way to speed up programs, and to hide the latency of memory accesses.

## 2. Design of Open-Ended Prefetcher

The open-ended prefetcher acts much like the stride prefetcher, except for two main differences:

1. The open prefetcher has a bigger RPT
   The RPT for the open-ended prefetcher is set in the configuration file (with the default configuration being 16) and this tends to cause a lot of collisions, so the prefetcher results actually turn out to be worse than the no prefetch results. In the open-ended prefetcher, it was experimentally determined that was the optimal number of RPT entries was 512.

2. The prediction is done on different states
   In the stride prefetcher, the simulator prefetched on 3 states: steady, init, and transitory. In the open prefetcher, it only prefetched on 2 states: steady and init. This causes less prefetches to happen, and it was shown to have better results. This means when we were prefetching in the "transitory" state, we were kicking out useful cache blocks and prefetching useless ones.

Unfortunately, a dl1.miss_rate of 1.72% was not achieved. The average miss rate for the L1 data cache was 1.9669%.

## 3. Methodology

The go, gcc, and compress benchmarks were run with sim-cache's simulations. Three different prefetchers tested: a next-line prefetcher, a stride prefetcher, and the open prefetcher, as described above.

The counters were incremented automagically in another section of code, so we did not to have to increase any counters in the code we wrote.

## 4. Results

Question 1: What was the average memory access time?

Assuming $T_{hit} = T_{miss}$ for any given level of cache

$$T_{average} = ( 1 - dl1.miss\_rate ) * ( T_{dl1\_hit} ) + (dl1.miss\_rate) * ( 1 - dl2.miss\_rate ) * ( T_{l2\_hit} + T_{dl1\_hit} ) + (dl1.miss\_rate) * ( dl2.miss\_rate ) * ( 1 ) * ( T_{mem} + T_{l2\_hit} + T_{dl1\_hit} )$$

Otherwise, assuming that $T_{l2\_hit}$ takes into account $T_{dl1\_miss}$ and that $T_{mem}$ takes into account both $T_{dl1\_miss}$ and $T_{l2\_miss}$

$$T_{average} = ( 1 - dl1.miss\_rate ) * ( T_{dl1\_hit} ) + (dl1.miss\_rate) * ( 1 - dl2.miss\_rate ) * ( T_{l2\_hit} ) + (dl1.miss\_rate) * ( dl2.miss\_rate ) * ( 1 ) * ( T_{mem} )$$

Question 2: Microbenchmark for next-line prefetcher

mb1q.c

```
for(i=0;i<100000;i++)//for a large number
      A[i]=0;//access linearly
```

Since we are accessing the array A linearly (and each element is smaller than the size of a block), we should only have a cache miss occur once on the first access to A and subsequent accesses should all be prefetched. We can see that the next line prefetcher behaves properly if there is a very low (near 0%) rate of cache misses for the L1 cache.

Question 3: Microbenchmark for stride prefetcher

mb2q.c

```
for(i=0;i<100000;i++)//for a very large number
{
      if(i%10==0 && (i+1000)<10000)//every so often
            A[i+1000]++;//modify something far away
      A[i]++;//but usually, just modify linearly
}
```

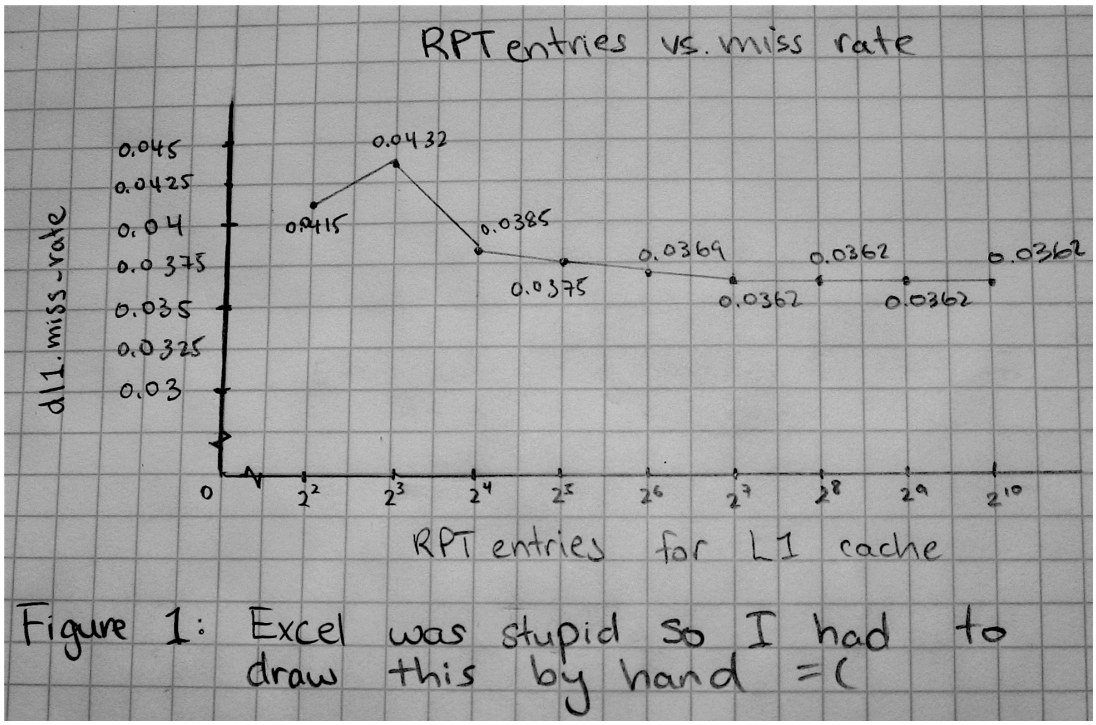
The stride prefetcher modifies A linearly, but on occasion (every 10 cycles), it modifies an address far away. The stride prefetcher will look at the PC tag of the A[i+1000]++ instruction, and see that its stride is a large number. Since this stride happens every time the PC is seen, the state should be “steady” and the prefetcher will obtain the correct block every time (except for the first time). The stride of the A[i]++ instruction will also

be correctly guessed every time (again, with exception of the warmup time). This should result in a very low amount of cache misses in total, nearing 0%.

Question 4: Access time comparisons for compress (using first assumption from question 1)

| | dl1.miss_rate | ul2.miss_rate | access time |
|---|---|---|---|
| no prefetch | 0.0416 | 0.1140 | 1.890s |
| next line | 0.0419 | 0.0838 | 1.770s |
| stride | 0.0385 | 0.0578 | 1.608s |

Question 5: Changing number of RPT entries and cache misses




In Figure 1, we see the L1 data cache miss rates graphed against the number of RPT entries. The x-axis is drawn in a logarithmic scale, from 4 to 1028 entries. The graph stabilizes at a 0.362 miss rate at around 128 entries. This may be because from that point on, most collisions are avoided and no significant performance gains are to be made by increasing the RPT size.

Question 6: More statistics

It would be useful to know how many of the prefetches were useful, and the accuracy of the predictor (does it pull in a bunch of blocks it doesn't need?).

It would also be useful to know if we had a prefetch that evicted a block that was at the time the least recently used block, but it was to be used soon after.

Question 7: Microbenchmark for open predictor

Since not much is changed from the stride prefetcher, the same microbenchmark (mb2q.c) can be reused on the open prefetcher.

## 5. Conclusions

The stride prefetcher performs better than the next line prefetcher, since it is more flexible. The stride prefetcher can emulate the next line prefetcher if 1 block is the optimal stride. Adding more RPTs to the stride prefetcher increases the number of useful prefetches, to a point. At some point, there are no more harmful collisions that occur, and the miss rate plateaus (as seen in Figure 1, in Question 5)

It may seen logical that the next line prefetcher does worse than no prefetcher at all, as data structures can often span more than one cache block, but it seems counterintuitive that the stride prefetcher would have a worse d1.miss_rate than “no prefetch” as well. This is because the tested size of the RPT (16) is too small, so there are conflicts that occur kicked out useful values. Once the RPT size was expanded, the results were much improved, as seen in the open-ended prefetcher.

It should be noted, that the L1 cache miss rate may not be the best way to judge performance. As seen in question 4, the second level cache also affects the access time (which is what we ultimately want to minimize)