ECE552, Fall 2014

Lab 4 Report

TIMMY RONG TIAN TSE (998182657)
BRIAN AGUIRRE (998528213)

Question 1

For this microbenchmark, we have two parts to test the validity of the next line prefetcher. First, we have a piece of code that performs positive testing. This piece of code performs sequential access to an array. We would expect the next line prefetcher to do well in this scenario since data is accessed sequentially and essentially all data within the prefetched block will be access (with exception to cold misses). Indeed, the next line prefetcher did well in this microbenchmark with a miss rate of 0.0221.

The next `for` loop performs negative testing on the next line prefetcher. Here, we access the array every `2*BSIZE` where `BSIZE` is the block size of the cache. The idea is that every time a block is fetched, we introduce a cache miss by skipping over the entire block that was brought in. Indeed, the next line prefetcher performed poorly in this case, with only a miss rate of 0.1441. For the configuration file, we used the provided `cache-lru-nextline.cfg`.

Question 2

We perform a positive and negative test benchmark again for the stride prefetcher. For the positive test, we used a `for` loop which would always have a constant stride of `2*BSIZE`. We expect the stride prefetcher to perform well here because there is always a constant stride and so it would be likely that the RPT table will have steady entries. As predicted, the miss rate was at a very low of 0.0004.

For the negative test, we tried to randomize the accesses to array elements. We expect the stride prefetcher to perform more poorly here because since the strides are randomized, the RPT table will have a less likely chance of being in a stable state than if strides were more constant. The miss rate was 0.0066—higher than when the strides were constant. We used the provided `cache-lru-stride.cfg` as the configuration file.

Question 3

Table 1: Miss Rates and Average access times

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

The average access time is calculated as follows,

$$Average\,access\,time = T_{access-L1Data} + T_{access-L2Data} \times MR_{L1} + T_{hit-Memory} \times MR_{L1} \times MR_{L2} \tag{1}$$

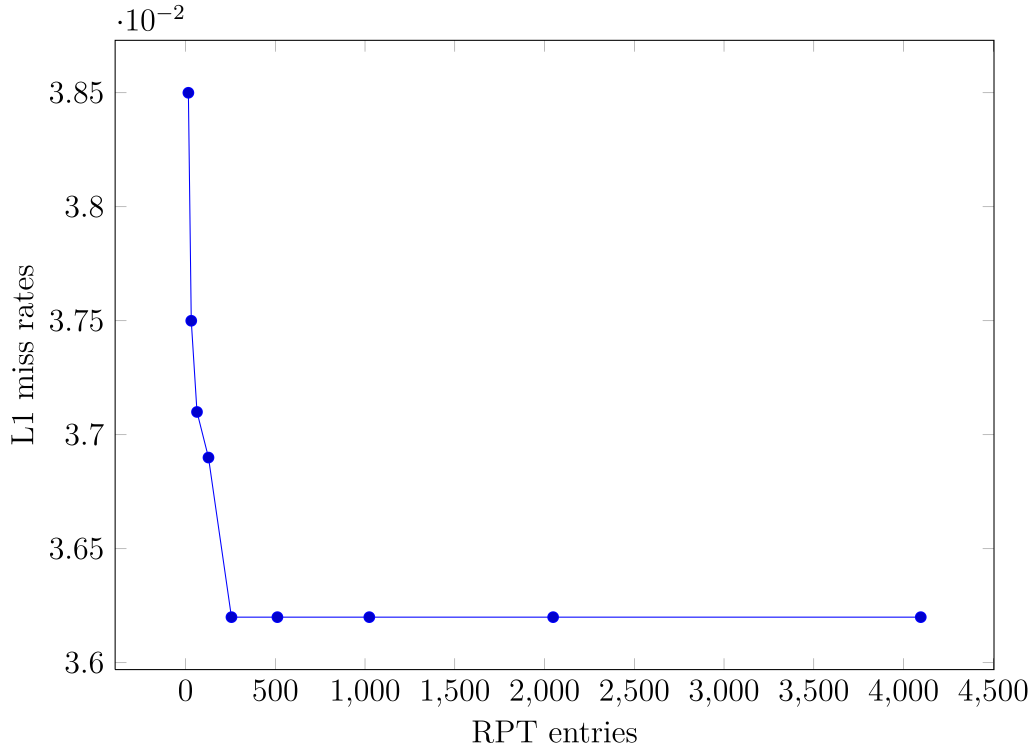
where MR is the miss rate at the specified subscripted data level.

Question 4

Table 2: RPT entries and L1 miss rates

RPT entries	L1 miss rates
16	0.0385
32	0.0375
64	0.0371
128	0.0369
256	0.0362
512	0.0362
1024	0.0362
2048	0.0362
4096	0.0362

Figure 1: Relation between RPT entries and L1 miss rates



In the graph, we varied the RPT entries in the x-axis and the resulting L1 miss rates are plotted on the y-axis. The trend shows that as we increase the RPT entries, the miss rate lowers very quickly but then it tapers off at around 3.62. In other words, for `compress`, it is a waste of memory to have RPT entries beyond around 512.

Question 5

One statistic that we would consider is the miss rate of the L2 cache. Since the difference in access time of the L2 cache and the memory is very large, we would consider how often there is miss in the L2 cache. Even if a cache performs very well in L1 but always misses in the L2, the performance of L1 will be hindered by the large latency needed to access memory when it misses. In comparison, a cache that has a mediocre hit rate in L1 but a great hit rate in L2 would arguably be just as good as the previous cache. This is because the great L2 hit rate brings the performance up by eliminating the need to access main memory too often.

Other statistic that would help is perhaps the rate with which load and store operations are performed. This statistic would be nice to gauge whether prefetching is contending with the CPU over the bandwidth of accessing main memory.

Last, if we were able to access the statistic of how often blocks are being hit and evicted, then we can gauge whether the cache is being polluted or whether we are experiencing the ping-pong effect wherein a block gets evicted and fetched back in continuously.

Question 6

For the microbenchmark, we used a combination of code from both the microbenchmarks for the next line prefetcher and the stride prefetcher. We would expect that in general, the prefetcher would do decent since it has a stride prefetcher component and a history prediction component to cover for the stride predictor when it reaches the "no prediction" state. With the combination of these two, we hoped the prefetcher would do well both when there is a constant stride and when there is not. Please see the next section for more details. We used the provided `cache-lru-open.cfg` file for the configuration.

Open-Ended Prefetcher

For the open-ended, we implemented a stride prefetcher with a local history table (LHT). The stride prefetcher has priority and will be the first to make a prediction. The LHT will only attempt to make a prefetch when the stride prefetcher offers "no prediction". The LHT has columns that can be indexed by the current PC. Each row is essentially a FIFO that records the sequence of addresses that have be witnessed thus far. When it attempts to make a prediction, it indexes into a specific row with the current PC. Then it scans from the youngest address to the oldest and try to find the addresses that is currently being accessed. If it finds this address in the FIFO, and this address is not the most recent address, then it makes a prefetch to one younger instruction in the FIFO than the instruction that was just found in the FIFO. This table gets updated every time there is a cache miss: the missed address gets pushed onto the FIFO indexed by the current PC. The rationality by this predictor is that when we see a certain address, we try to look through history to find the last time this address occurred in history and predict future access by prefetching the address immediate of this address in our history.

The idea was roughly based on the concept of Markov prefetching. This idea is loosely described as follows: based on where we are in the current state, we can predict the probability of which state we are going to transition to next, based on what we have witness with the transition of our current state in history. In a closer model of Markov prefetching we would have each address have a transition to another address with a probability associated with each transition. We opted for the more simpler approach by having PC indexing to a FIFO and prefetching the one next address to the address we are currently witnessing.

The design is feasible in hardware through a table that records the history of the next accesses of a given address. When the entries are full, we will have a policy to evict entries to make room for new entries. For example, we could have a FIFO policy like the one that we implemented. However, we could also have a counter on each entry to record the frequency of access or perhaps how recent an entry was accessed. Then we can evict the least accessed entry and least frequently used entry, respectively. Of course in real hardware, we will be unable to have a structure as large as the one we have instantiated in software (since there were no hardware constraints, we tried to allocate a very large array to get a high ranking ☺). A real prefetcher would conceivably be less than approximately 64KB (definitely less than your cache size).

Statement of Work

Tim worked on the open-ended prefetcher, stride prefetcher and the document. Brian worked on the next line prefetcher and the microbenchmarks.