HW 3 – Cache Optimization

沈昱宏, 110705013

*Abstract*—In this homework assignment, I analyze the cache behavior of the pi program using 5000 digits. I fixed the data cache size to 4KB and tried different implementations to improve the cache performance on the pi program. I tried two ways to improve the cache performance. Firstly, I modified the cache ways to try to improve the performance. Secondly, I tried changing the FIFO cache into LRU cache. The result shows that the LRU cache did improve the cache performance.

Keywords—Aquila, Pi, Cache

# The PI Program

To design a cache that is suitable for the PI program, it is crucial to know how the PI program runs. There are 5 main variables in the program, and all of them are arrays of integers. All the integer arrays are accessed sequentially, some of them will only be accessed one by one, some of them will be used with another integer array.

# Measuring D$ Performance

This experiment is done on ARTY board using the original implementation and NDIGITS = 5000 in the PI program. I counted the cache hit / miss rate by counting the total number of cache hits for r/w and the total number of p\_strobe\_i for r/w. Additionally, I computed the total number of cycles between p\_strobe\_i and p\_ready\_o to find the cache latencies of r/w. Please note that I do not count the latency of the cache when the request is hit. To separate the hit cycles, I use the following formulat to compute the cache latency (#cycle\_for\_read – #hit\_read) / (#read\_requests - #hit\_read). The result shows that both reading and writing have a good hit rate and the latency of reading & writing on cache miss is

approximately the same.

1. Cache Statistics (FIFO)

|  |  |  |
| --- | --- | --- |
| Category | r/w | Percentage / #Cycle |
| Cache Hit Rate | read | 77.5 % |
| write | 83.3 % |
| Average Cache Latency | read | 55.7 cycles |
| write | 55.6 cycles |

# N-way Associative Caches

I changed the number of cache ways and observed the performance of different cache ways under N\_DIGITS = 5000. The result shows that the cache performance of different cache ways is quite similar.

1. MSEC For Each Cache Ways (Ndigits = 5000)

| N\_WAYS | Execution time (msec) |
| --- | --- |
| 2 | 30522 |
| 4 | 30522 |
| 8 | 30516 |

Additionlly, I tried using different N\_WAYS for smaller digits. I tested the execution time for 2500 digits and the result shows that a 2\_way set associative cache performs best. This showed that the performance of higher way cache decreased slower as the NDIGITS (memory usage) grows.

1. MSEC For Each Cache Ways (Ndigits = 2500)

| N\_WAYS | Execution time (msec) |
| --- | --- |
| 2 | 6377 |
| 4 | 7294 |
| 8 | 7474 |

# Replacement Policy

## LRU

The current implementation of the replacement policy is FIFO, which is simple but not effective. I implemented a LRU replacement policy and modify the victim\_sel signal to choose the address to replace. I place the hit record in a table, the table size is N\_LINES \* N\_WAYS \* WAY\_BITS. When an address hits, I will reorder the line in the table so that the curretly used element is placed at the last register in N\_WAYS. The first element in the line (least recently used) will be selected as the victim.

## Result

The evaluation metric used is the same as part 2. The result shows that there are some improvements when changing the FIFO replacement policy into LRU cache, especially on the cache hit rate on read requests.

1. Cache Statistics (LRU)

|  |  |  |  |
| --- | --- | --- | --- |
| Category | r/w | Percentage / #Cycle | Improvement |
| Cache Hit Rate | read | 80.4 % | 3.8 % |
| write | 84.6 % | 1.6 % |
| Average Cache Latency | read | 55.5 cycles | 0 % |
| write | 55.6 cycles | 0 % |

1. Improvement of LRU Cache (N\_ways = 4, Ndigits = 5000)

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Time (msec) | Increase % | Total Hit Rate |
| FIFO | 30522 | 0 % |  |
| LRU | 28987 | 5.3 % |  |

# Anaysis

With buffer size = 32, I did some experiment on the entry num of the distr\_RAM. When setting entry\_num to 2, the increase pertage of the score of CoreMark reached 0.93 percent. I speculate that the CoreMark program tends to use the same return address consecutively, making the RAP able to hit the return address even if there are only two addresses being stored in the table. I computed the hit rate when entry\_num = 2, and the result shows that the hit rate is 87.7%, which shows that my guess is correct.

# Discusion

I implemented the RAP to get the address deterministicly instead of predicting the target address. Since I assume the LIFO buffer will always give the correct return address, I do not include any check on the misprediction (like the one in BPU). This is a little dangerous when implementing because if the push and pop of the stack is not controlled properly, the whole process will crash (this happened a lot during debugging). While it might cause a lot of errors during implementation, it will increase a lot compared to predicting the target address and verifying after the execution stage (like the one in BPU). If RAP is done in the similar implementation of BPU, there will be some mispredictions becaues the same function might be called by different functions, making the predictions to be wrong.