Cache Optimization

I-JUNG, HSU(徐宜蓉), 0713309

***Abstract*—In this work, several cache optimization methods are explained and experimented. Among all methods, the least-frequently used (LFU) performs the best when doing the task “computing π to 5000th and 10000th decimal places”.**

Keywords—cache optimization methods; N-way cache; cache replacement policy

I. Introduction

The following sections discuss about 2 different aspects that we can try to do cache optimization. The first one is changing the number of cache ways. The second one is changing the cache replacement policy. The last one is prefetching.

All experiments are done without increasing the cache size. The cache size is the same as the original one, that is, 2KB.

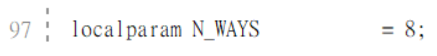
The software for testing is a program which calculate π using Machin's formula. The preset NDIGITS is 1000, meaning that we want to calculate π to 1000th decimal places. In order to observe the difference, here use NDIGITS = 5000 and NDIGITS = 10000.

Codes, ILA waveforms and some other files are available on https://github.com/TeeNsinTimes/MPD/tree/main/HW3.

II. Experiments On Different Number of Cache Ways

A. 4-Way (Original)

The original design of the data cache is 4-way.

It is defined in 

```
localparam N_WAYS          = 8;
```

, and is related to “way_hit”, “cache_hit”, “hit_index”, “cache_write”, “DIRTY_”. Therefore, if we change N_WAYS, we should also change the these related codes.

B. 2-Way and 8-way

Intuitively, I tried 8-way first because it should have a higher hit rate than 4-way. However, the results are identical to 4-way.

Therefore, I tried 2-way to see if the performance will get worse. Indeed, it is a little bit worse than 4-way.

C. 16-Way

Since we can not observe execution time change when increasing to 8-way, I wanted to try 16-way. However, the hardware resources are not enough. Although I removed all registers (counters) using for debug, the implementation error still happens.

D. Results and Discussion

5000	Time (msec)	Hit cnt	Miss cnt	Hit latency sum (cycle)	Miss latency sum (cycle)
2-Way	37381	2f499df	c2b349	2bd8631a	5a06db77
4-Way	**37366**	**2f499e6**	**c2b342**	**2bce98e2**	**59fcf7a6**
8-Way	**37366**	**2f499e6**	**c2b342**	**2bce98e2**	**59fcf7a6**

10000	Time (msec)	Hit cnt	Miss cnt
2-Way	148646	bc3f5be	3131730
4-Way	**148563**	**bc3f4c0**	**313172e**
8-Way	**148563**	**bc3f5c0**	**313172e**

The results of 4-way and 8-way are same. This indicates that the “victims” in 4-way are entered into the table at least 8 entries ago. So that even though the cache has 8 ways now, the victims are still replaced.

As for comparing 2-way and 4-way, the difference of execution time is 15 msecs. Although the hit rate of 4-way is a little bit higher than 2-way, the main reason should be hit/miss latency.

The average hit latency for 2-way is 14.835 cycles and for 4-way is 14.822; the average miss latency for 2-way is 118.371 and for 4-way is 118.320. It’s opposite to my prediction. I think the latency of 4-way should be greater than 2-way since it has more overheads (ex. When checking cache hit or not, it has to do 4 times instead of 2).

III. Experiments On Different Cache Replacement Policies

I consider that if there are more ways in the cache, the change that can bring by modifications will increase. Therefore, I decide to use as more cache ways as possible, that is, 8-way.

A. FIFO (Original)

The original design of data cache replacement policy is First-in-first-out (FIFO). It uses a counter to record which way should be replaced next. The counter only adds 1 when cache miss, it does not do anything when cache hit.

For 8-way cache, if one way is hit continuously 7 times after it is written into a cache, and the next cache misses, then that way will still be replaced.

B. LIFO

Last-in-first-out (LIFO) always take the block added most recently as victim.

It should perform bad because the first 7 items will never be replaced, even though they're no longer used.

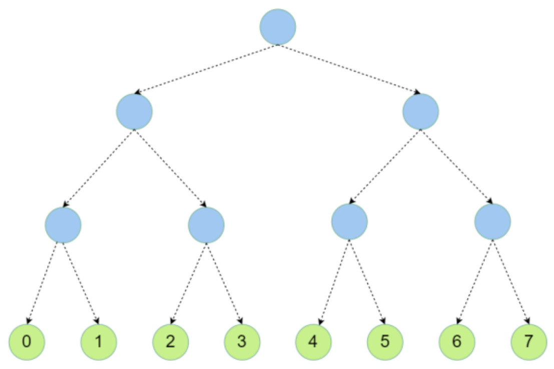
C. *LRU*

Least recently used (LRU) discards the least recently used items first. It requires an array to record the order of ways. The most recent used is 0, the least recently used is 6.

The array should be updated every time cache request happens. (not only when cache miss) Therefore, it's overhead is larger than FIFO and LIFO.

D. *PLRU*

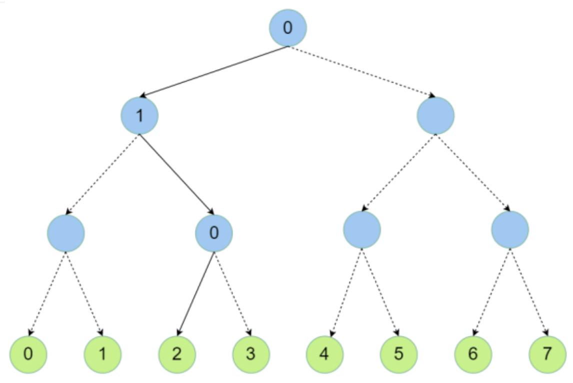
The concept of Pseudo-LRU is similar to LRU. It uses a binary tree to record which item should be replaced.



I found there are 2 different implementations of this method. One is "reverse the path direction of the way taken (hit or replaced)", the other one is "change the path direction to the reverse direction of the way taken".

I think the latter is more reasonable, so I implemented it.

Consider "way 2" is requested:



When we want to find the next victim, the method tells us to follow the "reverse" directions of the recorded. So, in the worst case, if the following requests are [3], [0/1], [4/5/6/7], then [2] will be the next victim. It is not the "real" least recently used item, but at least we can guarantee that it won't be replaced within 3 requests.

As for the former method, consider another [2] is requested. It will reverse the path again, and make [2] be the next victim.

E. *LFU*

Least frequently used (LFU) is similar to LRU, but it uses the array to store how many times an item is requested.

F. *LFUDA*

LFU with dynamic aging (LFUDA) is similar to LFU, but is adds a cache age factor to the reference count when a new object is added to the cache or when an existing object is re-referenced. Here, I use "1" as the factor.

G. *Results and Discussion*

	NDIGITS = 5000 Time (msec)	NDIGITS = 10000 Time (msec)
FIFO	37366	148563
LIFO	37723	149335
LRU	37380	148647
PLRU	37380	148563
LFU	**36343**	**146740**
LFUDA	37083	148387

LIFO is the worst as expected.

LRU performs worse than FIFO, although it's hit rate is a little bit higher. I think that's because of the overhead.

The PLRU performs much better than LRU because it has less overhead.

LFU and LFUDA both perform better than other policies, which indicates that the "pi" program relies on frequently used items more.

But why LFU is doing better than LFUDA? I guess it's because the "frequently used" items are needed over the whole program. It's importance should not be minused.

IV. Conclusion

After the above experiments, we can find that 8-way LFU cache is the best design for the computing pi program.

References

[1] https://en.wikipedia.org/wiki/Cache_replacement_policies
[2] Omran Safaa & Amory Ibrahim, "Implementation of LRU Replacement Policy for Reconfigurable Cache Memory Using FPGA.", 2018.
[3] Nimrod Megiddo & Dharmendra S. Modha, "ARC: A SELF-TUNING, LOW OVERHEAD REPLACEMENT CACHE", 2003.
[4] https://www.youtube.com/watch?v=0qBrbVAJbfc.
[5] https://www.csie.ntu.edu.tw/~r89004/hive/cache/page_2.html