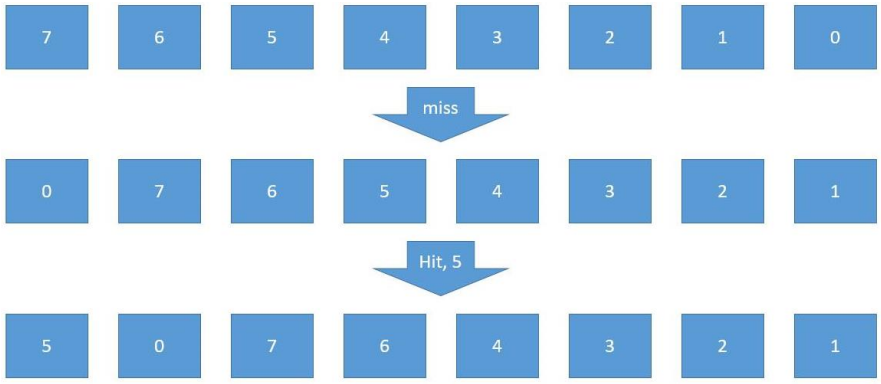
LRU Cache Replacement Policy and Some Improvements

1. Background

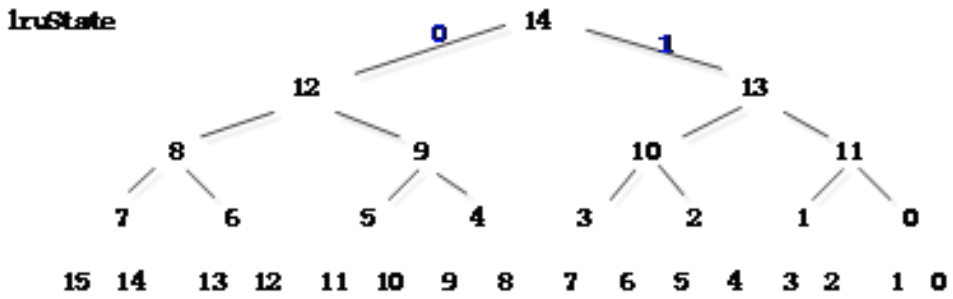
In this course, a basic fact is taught that microprocessors need sufficiently high speed cache memory since change data with main memory is very time-consuming, and that happens at least once in each instruction. The performance of cache memory depends on cache access time and hit ratio. Higher hit ratio will lower the number of miss penalty. When cache is full, overwriting a block to be used is terrible since this block will be accessed and transferred from lower level again. Therefore, a good cache replacement policy is important. There are two kinds of principles: spatial locality, which refers to the location near a requested address is likely to be used, and temporal locality, which refers to the same address is likely to be requested again. Based on these principles, there are some replacement policies including random choosing, FIFO, LFU, and LRU, among which LRU is the most frequently used one.

2. Basic concepts and algorithm of LRU (Least-Recently-Used)

There are two kinds of LRU, a traditional one and a tree-based pseudo LRU [1]. The convolutional LRU algorithm is to firstly initialize blocks in a set with an increasing sequence, i.e. set a queue. When a miss occurs, replace the block of least order and set it to the back of the queue, which means this blocks becomes the backmost and the others moves forward one position. When a hit occurs, put the block to the back of the queue and all the blocks behind the original hit block moves forward one position. A diagram illustrating these procedures is shown below:



The second one, tree-based pseudo LRU, is also widely used in replacement policy. Its basic structure for a 16-way set is shown below:



In this graph, all the number in the bottom line indicates a block, the number in the binary tree is lruState[14:0], where 0 indicates pointing left and 1 indicates pointing right. At first, we set all the 15 lruStates to be one. When a new miss occurs,

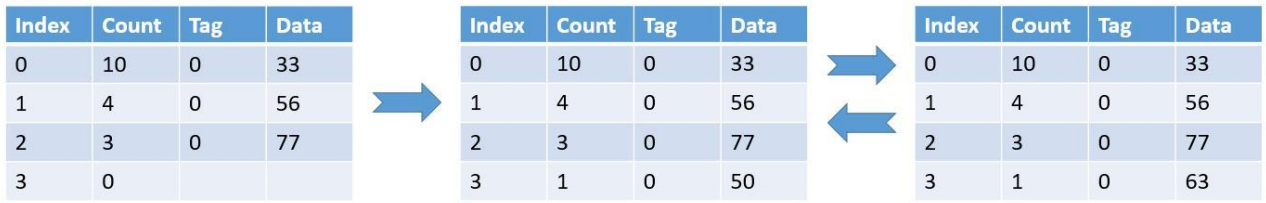
the data start from lruState[14] and goes iteratively to where it points to, and change all the lruState bits in its path (negation). When a new hit occurs, the data also goes through the tree but set all the lruState bits in its path to the opposite direction of its path, i.e. try not point to this block. This is an example written by VHDL language:

```
1. lruState[14:0]=15'b001 0100 0010 0000 //initial state
2. Block[10]=new_block //when miss happens, new block goes to 10th block
3. //Now lruState[14:0]=15'b100 0110 0000 0000 //14,12,9,5 change
4. //Block[4] hit
5. //Now lruState[14:0]=15'b010 0010 0000 0000 //14,13,10 change, since 2 alrea
   dy points to the opposite direction, it doesn't change.
```

For smaller caches, traditional LRU is better but for larger ones, it's computationally expensive. Compared with the traditional LRU method, the tree-based pseudo one is more timely economic, and has lower spatial expenses in spite of slightly lower hit ratio [2].

3. An improvement - LFU (least frequently used)

For LRU, it doesn't take the frequency of using into account, but another algorithm does. The basic idea of LFU is to remain counting the number of calling for each block, and always replace the least frequently used one. If there is more than one choice, it obeys LRU principle. For most cases, LFU seems to have higher hit ratio than LRU, but also has huge expenses. The more terrible thing is, for some lines with large frequency but not to be used (for example, a just-closed file), it will remain entrenched in cache [3]. Moreover, the wasted occupation will stop newer additions from the cache so that it's hard to get sufficiently large counts to stay in the cache. This is so-called cache pollution, the reason why sometimes the performance of LFU is getting slower and slower when close more files. I think it can be improved by a dying strategy that the count of inactive blocks will be decreased faster and faster if not used for a while. The following pictures show my idea:



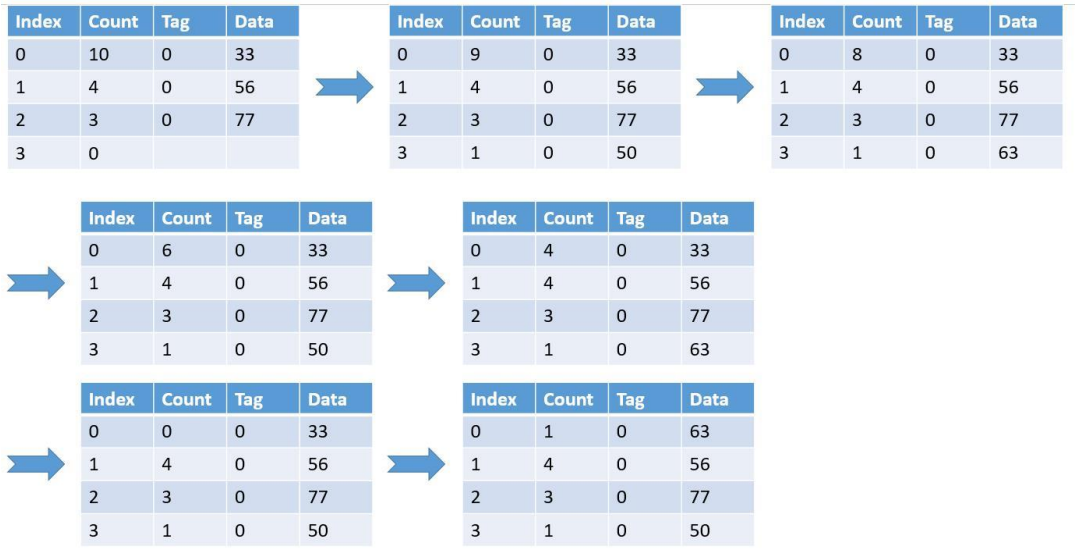
Index	Count	Tag	Data
0	10	0	33
1	4	0	56
2	3	0	77
3	0		

Index	Count	Tag	Data
0	10	0	33
1	4	0	56
2	3	0	77
3	1	0	50

Index	Count	Tag	Data
0	10	0	33
1	4	0	56
2	3	0	77
3	1	0	63

In the figure above, if the data 33 is just closed and will not be used, it will pollute cache since it's hard to remove a 10 counts block. If the program will use 56, 77, 50, and 63 for quite a lot times. Since 33 is not removed, 50 and 63 will always miss and replace each other, which will cause huge miss rate.

However, when a dying policy is conducted in the figure below, the count of 33 will decrease and the decreasing speed is getting faster. After several replacements, the count of 33 will goes to 0 and will be replaced (data 56, 77 is also being used so their counts don't vanish).



4. SLRU (segmented LRU) and other more intelligent improvements

In SLRU replacement, it defines a notion N:p, where N denotes the number of protected segments, and p denotes the number of probationary ones. The ratio between protected and probationary segments affects the cache performance. So it is very complex to optimize the performance of SLRU policy [4]. There are some other intelligent improvements, including some imitation learning approaches to automatically learn cache access patterns, for example, an oracle policy by leveraging Belady's [5]. Despite the fact that there are many improvement policies better than SLRU, many of them come at the expenses of large hardware requirements

To sum up, the LRU and some basic extension replacement policy are very important in exploring better method to improve cache performance.

Bibliography

[1]S. S. Omran and I. A. Amory, “Implementation of LRU Replacement Policy for Reconfigurable Cache Memory Using FPGA,” *IEEE Xplore*, Oct. 01, 2018. https://ieeexplore.ieee.org/document/8548892 (accessed Dec. 10, 2022).
[2]K. Morales and B. K. Lee, “Fixed Segmented LRU cache replacement scheme with selective caching,” *IEEE Xplore*, Dec. 01, 2012. https://ieeexplore.ieee.org/document/6407712 (accessed Dec. 10, 2022).
[3]R. Karedla, J. S. Love, and B. G. Wherry, “Caching strategies to improve disk system performance,” *Computer*, vol. 27, no. 3, pp. 38–46, Mar. 1994, doi: 10.1109/2.268884.
[4]K. Hurt and B. K. Lee, “Proposed enhancements to fixed segmented LRU cache replacement policy,” *IEEE Xplore*, Dec. 01, 2013. https://ieeexplore.ieee.org/document/6742754
[5]“Papers with Code - An Imitation Learning Approach for Cache Replacement,” *paperswithcode.com*. https://paperswithcode.com/paper/an-imitation-learning-approach-for-cache (accessed Dec. 10, 2022).