Computer Systems II

Project 3: Cache Replacement Policies: LRU, ARC

Introduction

Cache Replacement Policies have been an important area of research in storage technology from the advent of Caches in various hardware as well as software such as web servers, file system, operating system, data compression etc.

Today, we are discussing two of the cache replacement policies:

1. Least Recently Used(LRU)
2. Adaptive Replacement Cache(ARC)

Least Recently Used (LRU)

Least Recently Used (LRU) policy is based on replacing the least recently used with the new page in the cache. Here it can be thought as a FIFO buffer where the most recently used entry is at the top of the buffer and the least recently used one at the bottom of the buffer.

Whenever request for a new page arrives the policy checks whether the page is there in the cache or not. If it is then they pull out the entry from the buffer irrespective of their place. It could be at top, middle or at the end. Then it pushes at the top of the buffer. If it is not then it checks for the size of the buffer. If it is within the maximum allowed size i.e. the cache size then it pushes the page entry on the buffer. If it exceeds the maximum allowed size i.e. the cache size then we take out the last entry i.e. the least recently used entry, and push the new entry at the top of the buffer. This way the most recently used entries are always there in cache. It ensures temporal locality.

Adaptive Replacement Cache (ARC)

In Adaptive replacement cache we maintain two FIFO buffers (L1 and L2), one (L1) which contains the most recent entries whereas the other (L2) contain the most frequent entries. So, here the L1 maintains pages recently used pages and L2 maintains frequently used pages. We maintain the sizes for L1 and L2 for their sum to be equal to twice the cache size. The no of pages allocated for cache is determined by adapting and continually tuned parameter p. Then the no of entries in L2 in cache would be c-p. From past history the p parameter keeps on changing on-the-fly. It tries to keep track of temporal locality. It adapts itself, suppose at a given point the recency of pages becomes dominant then the ARC will adjust p parameter to exploit the opportunity.

Stages of Program development

Here we are going to discuss about the whole development process. We will discuss the various technics which were used and decisions made based on the observations. The data structures used are of mainly struct type for node in lists and int for various other bookkeeping parameters. The complete program was written along with the custom made libraries. No piece of code or library was used from other sources.

LRU

Firstly for LRU,

The Idea was to implement a FIFO buffer of exactly cache size. The buffer was implemented through doubly linked list as they were most efficient algorithmically when insertion from top and deletion from bottom is concerned.

The complete doubly linked list with feature of push at top, pop from bottom, remove from middle and display list where implemented in the program by self-designing the library which took quite some time.

The linked list are very flexible especially when we are removing an entry from the middle. Unlike array we don't have to shift parts of array during removal from middle, which could be time consuming. Also due to being a doubly linked list the removal from bottom or pop can be really easy, we don't have to traverse to the end to pop. The only disadvantage that we face is that it can only be traversed sequentially. This posed problem when we are searching for an element in the buffer. The search for element is of O(n) complexity where n is the size of the buffer. As the program used search in each step so we tried to optimize it. What was done is the search function was made to traverse from head as well as tail of FIFO buffer. This improved the worst case traversal time as we are now traversing from head and tail as well. So, in worst case the entry could be in middle i.e. n/2. It did improve the program performance (w.r.t time) by a factor of 10%. We went ahead with improving the search by using hash map.

Hash Map

Hash map is a data structure that could provide search in 0(1) complexity if its balanced. During the development of the program we developed and coded 3 different ways of implementing a hash map. We will start from the first version to the last version discussing pros and cons of each of the implementations:

Hash map (ver1)

As the address that is the tag for the pages were maximum 8 digits we tried to have array of size 10^8 dynamically allocated as in when needed. But due to compiler limitation of the array sizes we had to settle for 10^5. So in order to have complete coverage we implemented hashing with linked list. So we allocated a hash map which had a 2D array of pointers indexed by first 5 digits of the tag. Each entry in the array was pointer to a linked list where each entry have first 5 digits common. So, whenever we search for an entry we first resolve the address in the table using first 5 digits in the tag and then we traverse trough the linked list to get the entry if it is there. If it is there we return a 1 and then the buffer can be traversed to get the entry. As the chances of a tag to be not present in the buffer were high i.e. miss ratio, so it checks beforehand to check if the tag is there in the buffer or not. That gave a performance boost of 150% (w.r.t time) which was significant and it was due to high miss rates in low sized caches. But as the cache size increased (i.e. 1024->2048->…), the hash map started to become unbalanced resulting in long linked lists under certain tag groups. This deteriorated the performance and resulted in only 40-50% performance improvement in bigger cache sizes. We went ahead with evolving our hash map.

Hash map (ver2)

Next we improved the hash map by having the same hash table but now the entries in the 2D array points to array of int (indexed by last 3 digits of tag) where we stored 0s and 1s in the hash map for entry to be not there and there in the FIFO buffer respectively.

This improved our performance by ~ 600% as there were no linked list to traverse while searching for a miss in cache i.e. FIFO buffer. The only drawback is the memory footprint it leaves. When an entry is not there in the hash map (i.e. the pointer of 2D hash table return NULL) we dynamically allocate a 1D array of int of size 999 to accommodate all possible tags. But the issue was the large memory allocation, suppose we have got only one entry for the 2D hash table. Now the rest 998 entries are waste of memory space. We also just can’t remove the array too as there could be an entry made to it in future. Such high memory allocation and de-allocation cycle could reduce the performance. Also even after this we can’t improve the performance of search algorithm which was deteriorating with the fact that the miss rate decreases with increase in cache size which resulted in more traversal through the FIFO buffer for the respective entry. So, we tried to make an improvement in the hash map again.

Hash map (ver3)

The last and the most complex hash map implementation was done with 2D array of double pointers. Each entry of the double pointer points to a dynamically allocated 1D array of pointers and each entry in 1D array points to the nodes in the FIFO buffer. If an entry is there we can check by resolving the address by tag as earlier and dereferencing the pointer to get directly to the node without traversing through it in the FIFO buffer. Else it return NULL. Now not only we can check for the existence of the entry in the FIFO buffer but also can directly jump to the node to perform removing from middle etc. This gave a tremendous amount of performance boost of 2500% (i.e. for LRU the runtime for any cache size was <2 sec for P3.lis) from the algorithm of search we started from i.e. ~ 56 sec. Now all the searches were made O(1) complexity as we don't have to traverse through the FIFO buffer. This boost in performance was significant but there was cost associated. The memory footprint now increased even more as now memory address were store which are type of long int. This left the worst possible memory footprint (i.e. close to 120MB for P3.lis).

As the last implementation was fast but now requires more memory due to long int for addresses. The program is guaranteed to work for P3.lis and works on P4.lis too. We are providing the stats for the P3.lis and P4.lis under this hash map implementation.

Table 1: Simulation results for LRU at general.asu.edu

LRU						
	P3.lis			P4.lis		
S.no	Cache Size	Run Time(in sec)	Hit rate (in %)	Cache Size	Run Time(in sec)	Hit rate (in %)
1	1024	0.99	1.05	1024	4.76	2.68
2	2048	1.02	1.12	2048	4.96	2.96
3	4096	1.04	1.34	4096	5	3.31
4	8192	1.05	1.67	8192	5.01	3.65
5	16384	1.06	2.17	16384	5.23	4.09
6	32768	1.09	3.75	32768	5.24	5.27
7	65536	1.16	13.14	65536	5.2	10.83
8	131072	1.17	45.37	131072	5.64	21.56
9	262144	1.18	65.67	262144	5.77	37.49
10	524288	1.08	80.08	524288	5.82	48.58

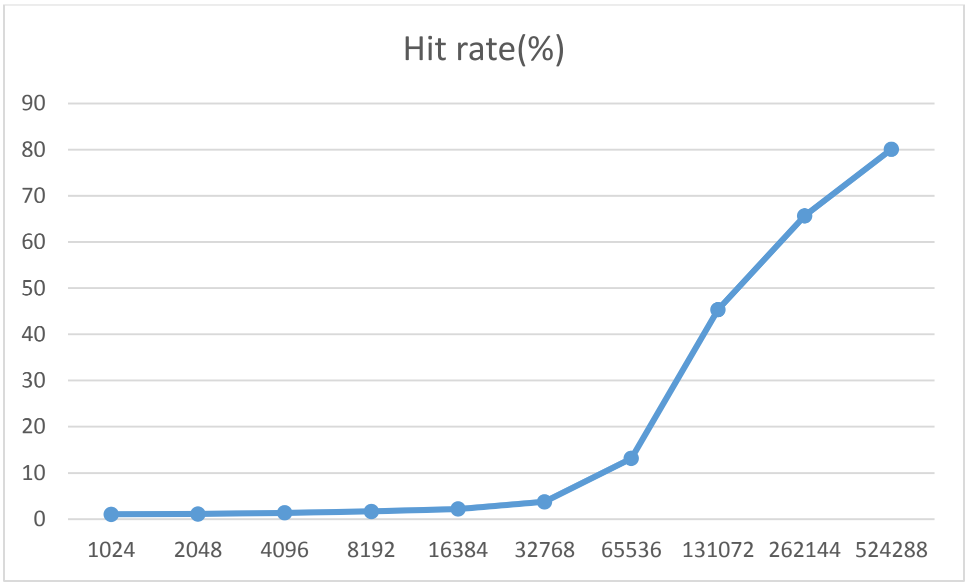

Figure 1: Hit rate (%) in LRU for P3.lis

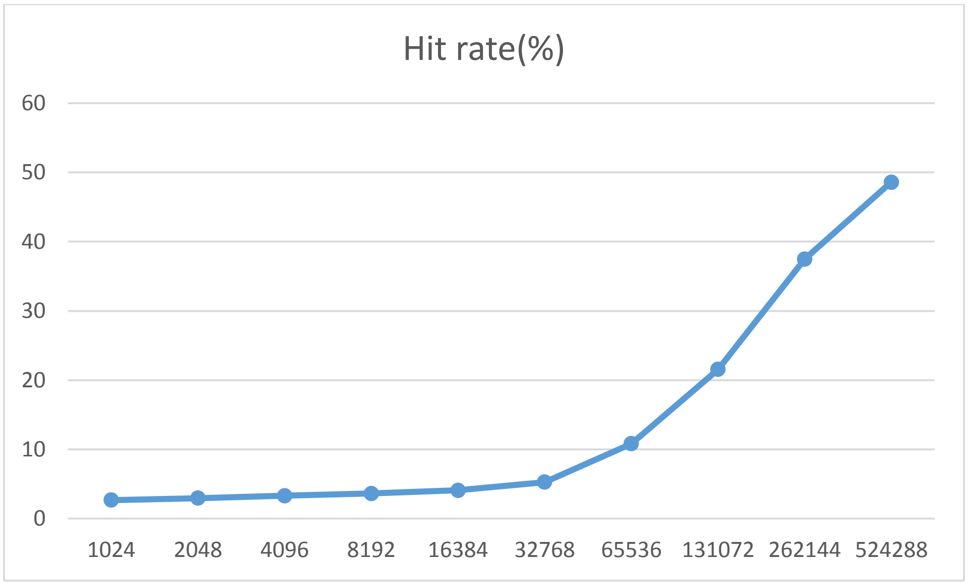

Figure 2: Hit rate (%) in LRU for P4.lis

ARC

The ARC code consists of four FIFO buffers i.e. L1, L2, B1, and B2. The algorithm is same as the given in research paper. The program was implemented with P3.lis in mind. So, it will guarantee to work on P3.lis but after implementation of the Hash map (version 3) the memory footprint becomes huge as there are four hash map tables now. Therefore we request not to use any other file except P3.lis. Also it has been stated as program requirement that program should run on P3.lis with cache size 2048. It will run on any cache size for P3.lis. The program is not tested on P4.lis or P6.lis and they might fail due to memory requirements. The program works surprisingly good as compared to the first implementation of ARC that was without hash map i.e. Performance boost of 765% (w.r.t time) for cache size 1024 in P3.lis.

As the program was developed after LRU only the hash map (version 3) was implemented. The following are the simulation results on general.asu.edu.

Table 2:Simulation results for ARC at general.asu.edu

ARC			
P3.lis			
S.no	Cache Size	Run Time(in sec)	Hit rate (in %)
1	1024	73.15	2.63
2	2048	131.19	3.02
3	4096	312.57	3.66
4	8192	807.87	5.34
5	16384	1502.13	8.48

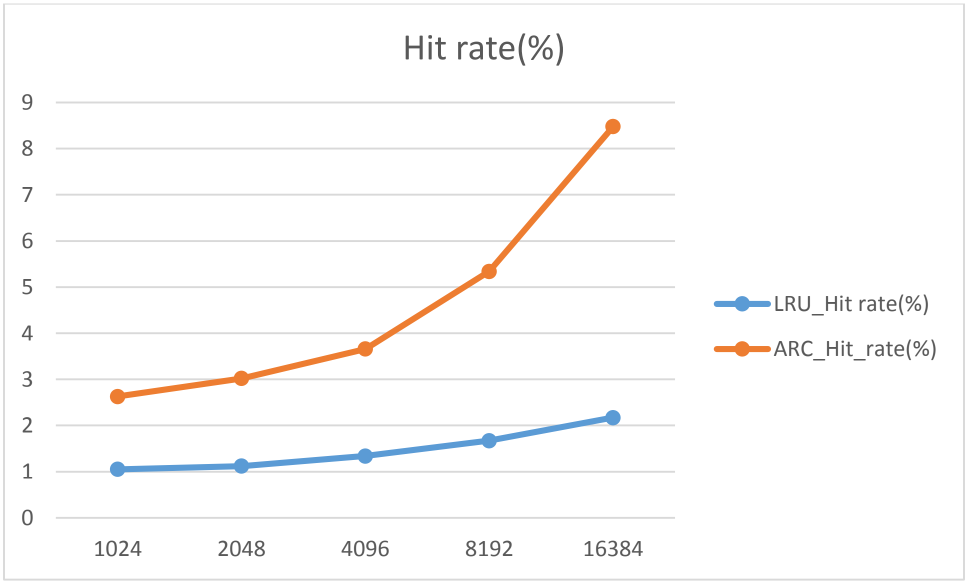

Figure 3: Comparison of Hit rate for LRU and ARC

Algorithm

1. LRU

Let current request page be x_t

```
If x_t in present in hash_map (represents cache entries) then
        Report a hit
        Get the address from hash map by dereferencing. Jump to the address
        Remove that entry from the buffer and push it at the top of FIFO buffer
Else
        Report a Miss
        If size of FIFO buffer < cache_size
                Push it at the top of FIFO buffer
                Create entry in hash_map
        Else
                Remove last entry from bottom
                Delete last entry from hash_map
                Push the current entry at top of FIFO buffer
                Push entry in hash_map
```

2. ARC

```
INPUT: The request stream x1,x2,…..
INITIALIZATION: Set p=0 and set the LRU lists T1, T2, B1, B2 and to empty.
For every t > 1 and any $x_t$, one and only one of the following four cases must occur.
Case I: $x_t$ is in or T1 or T2. A cache hit has occurred in ARC(c)and DBL(2c) .
Move $x_t$ to MRU position in T2.
Case II: $x_t$ is in B2. A cache miss (resp. hit) has occurred in ARC(c)(resp. DBL(2c) ).
    ADAPTATION: Update p=min(p+delta1,c) where delta1 ={1 if |B1|>|B2|,|B2|/|B1| otherwise}
    REPLACE (xt,p). Move xt from B1 to the MRU position in T2 (also fetch xt to the cache).
Case III: is in . A cache miss (resp. hit) has occurred in ARC (resp. DBL ).
    ADAPTATION: Update p=max(p-delta1,0) where delta2 ={1 if |B2|>|B1|,|B1|/|B2| otherwise}
    REPLACE (xt,p). Move xt from B2 to the MRU position in T2 (also fetch xt to the cache).
Case IV: xt is not in T1 U B1 U T2 U B2. A cache miss has occurred in ARC(c) and DBL(2c) .
    Case A: L1=T1 U B1 has exactly c pages.

            If (|T1|<c)
                Delete LRU page in B1. REPLACE(xt,p) .
            else
                Here B1 is empty. Delete LRU page in T1 (also remove it from the cache).
            endif
    Case B: L1=T1 U B1 has less than c pages.
        If (|T1|+|T2|+|B1|+|B2|>=c)
            Delete LRU page in B2, if (|T1|+|T2|+|B1|+|B2|==2c)
            REPLACE(xt,p)
        endif
    Finally, fetch xt to the cache and move it to MRU position in T1.

Subroutine REPLACE(xt,p)
    If ( (|T1| is not empty) and ( (|T1| exceeds the target p) or (xt is in B2 and  |T1|==p)) )
        Delete the LRU page in T1 (also remove it from the cache), and move it to MRU position in B1.
    else
        Delete the LRU page in T2(also remove it from the cache), and move it to MRU position in B2.
    endif
```

Bibliography

ARC: A SELF-TUNING, LOWOVERHEAD REPLACEMENT CACHE by Nimrod Megiddo and Dharmendra S. Modha