ARC (Adaptive Replacement Cache)

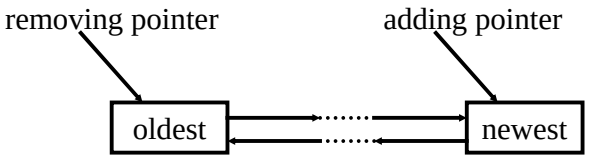
- "We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has great capacity than the preceding but which is less quickly accessible." Von-Neumann, 1946.
- The selecting of the "victim" to be taken out of the faster memory has been traditionally done for decades by the LRU algorithm.
- The LRU is fast and easy for implementation, but can there be a better algorithm?

Motivation

- The LRU is employed by:
  - RAM/Cache Management
  - Paging/Segmenting systems
  - Web browsers and Web Proxies
  - Middleware
  - RAID Controller and regular Disk Drivers
  - Databases
  - Data Compression (e.g. LZW)
  - Many other applications
- ARC was suggested by N. Megiddo and D. Modha of IBM Almaden Research center, San Jose, CA on 2003 and is better than LRU.

LRU complexity

- LRU is implemented by a double linked list.
- When a page in the list is accessed, it will be moved from its place to the end of the list.
- When a new page is accessed, it will be put in the end of the list and the oldest page is taken out of the list.
- Both of the operation are of an O(1) complexity.



LRU disadvantages

- The fundamental locality principle claims that if a process visits a location in the memory, it will probably revisit the location and its neighborhood soon.
- The advanced locality principle claims that the probability of revisiting will increased if the number of the visits is bigger.
- LRU supports just the fundamental locality principle.
- What will happen if a process scans a huge database?

LFU

- LFU replaces the least frequently used pages.
- LFU takes in account the advanced locality principle and a scan of a huge database will not be a trouble, **HOWEVER**…
- LFU is implemented by a heap; hence it has a logarithmic complexity for adding or removing a page from the heap and also for updating a place of a page in the heap.
- Stale pages can remain a long time in the memory, while "hot" pages can be mistakenly taken out.

LRU vs. LFU

- LRU was suggested on 1965 and LFU on 1971.
- The logarithmic complexity of LFU was the main reason for its impracticability.
- LFU had a periodic check for the stale pages.
  - This solved the stale pages problem.
  - But, it made the performance even worse.
- LRU beat LFU; thus, LFU has been pushed into a corner, until 1993, when O'Neil revisits LFU in order to develop the LRU-K technique.

LRU-K

- LRU-K memorizes the times for each cache page's k most recent references and replaces the page with the least k^{th} most recent references.
- If there is no k^{th} reference, LRU-K will consider the reference to be infinite (The oldest).
- LRU-K retains a history of references for pages that are not currently present in the memory.

LRU-K Pro. & Con.

- When a page is referenced, it will typically not be moved to the end of the list; hence a linked list cannot be good for the implementation and a heap is needed $\Rightarrow$ logarithmic complexity.
- Scanning a huge database will not be a trouble like with LFU, but LRU-K outperforms LFU, because stale pages are handled better.
- LRU-K maintains the advanced locality model, but did not succeed to beat LRU because of the complexity.

2Q

- On 1994 Johnson & Shasha suggested an improvement to LRU-2.
- 2Q has two queues A1 and Am:
  - On the first reference to a page, 2Q places the page in the A1 queue.
  - If a page is accessed again when it is in A1 or Am queues, the page will be moved to the end of Am queue.
  - The sizes of A1 and Am are constants (e.g. 20% of the cache for A1 and 80% for Am). When a new page is added to one of the queues, an old page from the same queue will be removed if need.

2Q Pro. & Con.

- 2Q is implemented by linked lists $\Rightarrow$ O(1) complexity.
- 2Q has just two queues; thus it just partially adapts the advanced locality model.
- The execution time of 2Q is about 25% longer than LRU, but it gives 5%-10% better hit ratio. These results were not convincing enough.

LRFU

- On 2001 Lee at el. suggested a combined algorithm of LRU and LFU named LRFU.
- LRFU can be tuned to be close to LRU or be close to LFU.
- LRFU's clock is expressed by the number of the page that have been accessed. I.e. the time is incremented by one on each page reference.

Combined Recency and Frequency

- Let us define Combined Recency and Frequency (CRF) as: $CRF_{t_c}(b) = \sum_{i=1}^{k} F(t_c - t_{b_i})$
- where
  - t_c is the current time.
  - b is the block id.
  - t_{bi} is the i^{th} time that block b has been referenced.
  - F(x) is the function $2^{-\lambda x}$.
  - k is the number of the times that the block has been referenced.

The F(x) function

- When LRFU has to take a page out of the memory, it will choose the page with the minimal CRF.
- If F(x) is a constant for any x, LRFU will exactly act like LFU; hence if λ is set to 0, LRFU will be LFU.
- If F(x) holds $F(i) > \sum_{j=i+1}^{k} F(j)$ for any i and any k where k≥i+1, LRFU will exactly act like LRU; hence if λ is set to 1, LRFU will be LRU.

LRFU Performance

- According to experimental results, λ is usually set to a very small numbers less than 0.001.
  - This means LRFU is LFU with a slight touch of LRU.
- LRFU outperforms LRU, LFU, LRU-K and 2Q in hit rate.
- The pages are kept in a heap; hence the complexity of LRFU is O(log(n)).

ARC Definitions

- Let *c* be the number of pages in the memory.
- L_1 and L_2 are two linked lists. L_1 contains the pages that have been accessed just once, while L_2 contains the pages that have been accessed at least twice.
- The allowed operations on L_1 and L_2 are the same operations that are allowed on an LRU linked list.

ARC Policy

- $0 \le |L_1| + |L_2| \le 2c$
- $0 \le |L_1| \le c$, L_2 can be bigger than c.
- When a page is accessed; if the page is in $L_1 \cup L_2$, move it to the MRU of L_2; otherwise move it to the MRU of L_1.
- If adding the new page makes $|L_1| + |L_2| > 2c$ or $|L_1| > c$; if L_1 (before the addition) contains less than *c* pages, take out the LRU of L_2; otherwise take out the LRU of L_1.

Lists' Partitions

- $|L_1|+|L_2| \leq 2c$, but the size of the memory is just c.
- Let T_1 be the most recent pages in the memory and B_1 be the least recent pages in the memory.
- Similarly, L_2 is partitioned into T_2 and B_2.
- $T_1 \cup T_2$ contains the pages that are actually in the memory.

Which page is taken out

- When a page is moved from a "T" list to a "B" list, it will be taken out of the memory.
- Let p be the current target size for the list T_1.
- If $|T_1|>p$, move the LRU of T_1 to be the MRU of B_1.
- If $|T_1|<p$, move the LRU of T_2 to be the MRU of B_2 .
- If $|T_1|=p$,
  - If the accessed page has been in B_2, move the LRU of T_1 to be the MRU of B_1. (Because p is going to be decremented).
  - If the accessed page has been in B_1 or has not been in the memory, move the LRU of T_2 to be the MRU of B_2.

Setting *P*

- If there is a hit in T_1 or T_2, do nothing.
- If there is a hit in B_1
  - If the size of B_1 is at least the size of B_2, increment p by 1; otherwise, increment p by $|B_2|-|B_1|$.
- If there is a hit in B_2
  - If the size of B_2 is at least the size of B_1, decrement p by 1; otherwise, decrement p by $|B_1|-|B_2|$.
- The increments and the decrements are subject to the stipulation $0 \leq p \leq c$
- The idea is to "invest" in the list that is performing the best.

ARC Advantages

- When scanning a huge database, there are no hits; hence p will not be modified and the pages in T_2, will remain in the memory $\Rightarrow$ Better than LRU.
- Stale pages do not remain in the memory $\Rightarrow$ Better than LFU.
- ARC is about 10%-15% more time consuming than LRU, but the hit ratio is in average about as twice as LRU.
- Low space overhead for the "B" lists.

Conclusions

- ARC captures both "recency" and "frequency".
- ARC was first introduced on 2003, but the journal paper was published on 2004.
- Folklore holds that Operating Systems has 3 principles:
  - Hash, Cache and Trash.
    - ARC improves the caching; hence so significant.