letters to the editor

Dear Editor,

I found the "one up on LRU" article in the August 2003 *;login:* issue (Vol. 28, No. 4) to be difficult to understand, even after several readings and consulting the original paper in the FAST proceedings. I sensed on my first reading that Megiddo and Modha are presenting an important result that will be widely implemented and eventually be incorporated into every undergraduate CS program.

This caused me to reread and study the paper until I got a better understanding. For my own benefit, I decided to write my own interpretation of the paper and am providing it to *;login:* in the hope that it will be of benefit to other readers (provided that it is an accurate interpretation).

Here is my interpretation:

The primary objective of a cache management system is to hold those pages in cache that are most likely to be reused in the near future. In an ideal system, pages that will be used only once in a while will never be cached, since they will not be reused in the near future. The focus of the system now becomes one of managing those pages that will be seen again in the near future. Although not quite ideal, the Least Recently Used (LRU) algorithm is a simple and effective solution for handling those pages that will be used twice or more, since any page that is used twice will likely be used again (from empirical observation and captured in the Principle of Locality). However, it is extremely difficult if not impossible to determine in advance that a page will be used only once.

The basic idea of the ARC system is to divide the cache into two parts: one part for pages that have been used only once and one part for pages that have been used twice or more. Any request for a page that previously had been used only once causes that page to be reassigned to the other part of the cache, the part with pages that have been used twice or more. Separate LRU lists are used to track the pages in the cache: T1 (using the nomenclature from the ARC paper) for the pages that have been used only once recently and T2 for those pages that have been used twice or more.

The challenge is to determine how big to make each of the two parts. If the target size for the part managed by the T1 list (called target_T1 in the paper) is too large, then pages that deserve to stay in the cache, because they have been used frequently recently, have been prematurely ejected. If target_T1 is too small, then a page may be ejected before its second request. The solution in ARC is to keep a list of recently ejected pages for each of T1 and T2 (called B1 and B2, respectively). If the requested page is in B1, then it is likely that target_T1 is too small and so ARC increments target_T1 by one. If the requested page is in B2, then it is likely that target_T1 is too large and so ARC decrements target_T1. That is, ARC dynamically adjusts target_T1 based on recent access patterns before loading the requested page into cache and putting it on the T2 list.

Note that a request for a page on B2 (which causes target_T1 to decrease) does not necessarily result in a page on the T1 list being ejected from cache. Consider the case where there were several requests in a row for pages on the T1 list, which would have resulted in those pages being moved from the T1 list to the T2 list without any change in target_T1. Even with the decrement of target_T1 due to the request for a page on B2, the T1 part of the cache is still below the target, so a page from the T2 list is ejected from cache.

There is one more case that has not yet been discussed: the case where the requested page has not been used at all recently. That is, the requested page is neither in the cache (i.e., not in T1 or T2) nor on the history lists (i.e., not in B1 or B2). Before ARC can load the requested page into the T1 part of the cache, it must decide which page to eject from the cache and which page to eject from history. If adding the requested page to T1 will cause T1 to exceed the target_T1, then ARC ejects the LRU page of T1 from the cache; otherwise, it ejects the LRU page of T2 from the cache. In other words, ARC allows the T1 part of the cache to grow until it reaches the target_T1 size. Similarly, if adding the requested page causes the number of pages that have been seen only once (i.e., in T1 and B1) to exceed the size of the cache (c in the paper), then the LRU from B1 is deleted; otherwise the LRU of B2 is deleted. Note that the reason c was selected as limit for the number of pages used only once recently (i.e., T1 + B2) is that tracking more pages would imply that, even if the whole cache were being used for pages only seen once recently, the pages being requested for the second time would already have been cycled out of the cache (this is my conjecture).

This raises the question of how much history should be kept. (The following is my conjecture.) As discussed in the previous paragraph, the number of pages that have been seen only once that are being tracked (i.e., T1 + B1) is limited to c (the size of the cache). Since there is a balance being sought between tracking of pages that have been used only once recently and those that have been used twice or more, it can be concluded that no more than 2c pages should be tracked in total. That is, T1 + B1 + T2 + B2 should be less than or equal to 2c.

Note that T2 + B2 can exceed c, even though T1 + B1 cannot exceed c. This is because pages can move from T1 to T2, but not back again (without being completely recycled). That is, the ARC algorithm is never going to eject a page in cache unless it needs room to load

another page (since a request for a page from T1 results in it being reassigned to T2 without any cache movement).

This provides us with all the information required to construct the ARC algorithm, which manipulates the four lists (T1, T2, B1, and B2) and the cache when a page p is requested using one of the following five cases:

```
p is on T2:
    # Use the page again
    move p to MRU(T2)
p is on T1:
    # This is an OK cache hit
    move p from T1 to MRU(T2)
p is on B1:
    # Need to get it in the cache
    # so allocate more space for T1
    increment target_T1
    if T1's part is full
        # i.e., size T1 >= target_T1
        move LRU(T1) to MRU(B1)
        eject MRU(B1) from cache
    else
        move LRU(T2) to MRU(B2)
        eject MRU(B2) from cache
    endif
    move p from B1 to MRU(T2)
    load p into cache
p is on B2:# (similar to B1)
    # Darn, wish it was in the cache
    # ... so deallocate space for T1
    decrement target_T1
    if T1's part is full
        # i.e., size T1 >= target_T1
        move LRU(T1) to MRU(B1)
        eject MRU(B1) from cache
    else
        move LRU(T2) to MRU(B2)
        eject MRU(B2) from cache
    endif
    move p from B2 to MRU(T2)
    load p into cache
p has not been used recently:
    # i.e., it is not on T1, T2, B1 nor B2
    if seen once list is full
        # i.e., T1 + B1 = c
        if B1 has entries
            delete LRU(B1)
            if T1's part is full
                # i.e., size T1 >=
                # target_T1
                move LRU(T1) to
                        MRU(B1)
                eject MRU(B1) from
                        cache
            else
                move LRU(T2) to
                        MRU(B2)
                eject MRU(B2) from
                        cache
            endif
        else# B1 is empty
            eject LRU(T1) from cache
            delete LRU(T1)
        endif
    else
        if cache is full
            if too much history being
                    kept
                delete LRU(B2)
            endif
            if T1's part is full
                # i.e., size T1 >=
                #target_T1
                move LRU(T1) to
                        MRU(B1)
                eject MRU(B1) from
                        cache
            else
                move LRU(T2) to
                        MRU(B2)
                eject MRU(B2) from
                        cache
            endif
        endif
    endif
    insert p into MRU(T1)
    load p into T1
```

Note that I had to inline a "replace" subroutine here in order to see all the list and cache manipulations together (they are divided between two routines in the paper [and are on backing pages in *;login:*!]). I also relegated the handling of dirty pages to the eject function, since it is not really germane to the new concepts introduced by ARC.

Regards

Henry Baragar
henry.baragar@instantiated.ca

Principal, Technical Architecture
Instantiated Software Inc.

The authors respond:

This letter will be of value and interest to readers of *;login:*. However, there is one subtle point of ARC that the author has not captured. Inclusion of this will make the exposition complete. The following is known as the "learning rule" and is a very important part of making the algorithm work:

Upon a hit in B1, the parameter target_T1 is incremented by a maximum of 1 or B2Length/B1Length. But target_T1 can never exceed the cache size. Similarly, upon a hit in B2, the parameter target_T1 is decremented by a maximum of 1 or B1Length/B2Length. But target_T1 must be nonnegative.

Nimrod Meggiddo and Dharmendra S. Modha

###

Tina,

I got my copy of *;login:* this morning and read your "Value Added" article (*;login:* Vol. 28, No. 5, p. 4). One would think what you wrote goes without saying.

However, imagine a world that was invaded by idiots in the desperate belief by some that a warm body was better than no body, and a day when candidates and employees could name their prices (often petty) regardless of skill and what-not, which has now evolved into a world where many people are working harder than ever, are fearful of losing their job, and are, at times, rewarded with management that believes employees have nowhere to go and that there are plenty of qualified people to replace them.

Oh, that's right. That happened to our world.

So, yes, it needed to be said. Partly to acknowledge the mistakes of the past and present, but also to remind everyone

of the value the people-who-answer-questions have. And that they help make the world something you can live with and even enjoy – fighting off that nasty stuff.

It's funny to me – to one of your other points – that there is no specific functional classification for these values-oriented individuals. Some are administrative assistants; some are system administrators; others are programmers; and some are even managers, directors, and VPs. By "function" anyway. To me, these people are really generalists of the human race, and I know because I'm one of them. I don't think I'm at the top of the ladder. But I'm lucky. I'm appreciated very much for what I do and am.

Being one, and being a manager and benefactor of these Beings with Values, I first of all wanted to say all of this. Then I had to be a nit picker and tell you that I saw two other attributes that deserve mention as well (after all, you had to cram this into a single column):

1. Teamwork. There are lots of people who can answer a variety of questions competently, but they don't play well with others. Those who do this task willingly and even cheerfully and with passion are the most valued and enjoyed of all.

2. Diplomacy & honesty. The diplomacy is a given and relates to teamwork: if people don't like the way you talk to them, they won't come and ask you questions until they are damned desperate. The honesty is twofold and the premise is credibility. First, self-honesty: knowing what your value is and not over- or under-stating it (which is difficult). Second, honesty toward others but with a touch of diplomacy when required.

Debby Hungerford
debby-h@pacbell.net