# one up on LRU

by Nimrod Megiddo

Nimrod Megiddo has published in the areas of optimization, algorithms and complexity, game theory, and learning, and has taught and lectured in several universities.

megiddo@almaden.ibm.com

and Dharmendra S. Modha

D. S. Modha has published on caching algorithms, information and coding theory, data mining, learning theory, signal processing, and data visualization. He holds 6 patents.
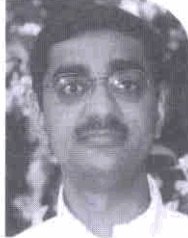
dmodha@almaden.ibm.com

Introduction

The concept of caching dates back (at least) to von Neumann's classic 1946 paper that laid the foundation for modern practical computing. Today, caching is used widely in storage systems, databases, Web servers, middleware, processors, file systems, disk drives, RAID controllers, operating systems, and in varied and numerous other applications.

Generically, a cache is a fast, usually small, memory in front of a presumably slower but larger auxiliary memory. For our purposes, both memories handle uniformly sized items called *pages*. We also assume demand paging. Host requests for pages are first directed to the cache for quick retrieval and, if the page is not in the cache, then to the auxiliary memory. If an uncached page is requested, one of the pages currently in the cache must be replaced (often requiring that the page be flushed back to the auxiliary memory, if it was written to by the host). A *replacement policy* determines which page is evicted. LRU is the most widely used replacement policy.

Until recently, attempts to outperform LRU in practice have not fared well because of overhead issues and the need to pre-tune various parameters. Adaptive Replacement Cache (ARC) is a new adaptive, self-tuning replacement policy with a high hit ratio and low overhead. It responds in real time to changing access patterns, continually balancing between the recency and frequency features of the workload, and demonstrates that adaptation eliminates the need for workload-specific pre-tuning. Like LRU, ARC can be easily implemented. Even better, its per-request running time is essentially independent of the cache size. Unlike LRU, ARC is "scan-tolerant" in that it allows one-time sequential requests to pass through without polluting the cache. ARC leads to substantial performance gains over LRU for a wide range of workloads and cache sizes.

ARC's Paradigm

Suppose that a cache can hold c pages. The ARC scheduler maintains a cache directory that contains $2c$ pages, c pages in the cache and c history pages. ARC's cache directory, referred to as DBL, maintains two lists: L1 and L2. The first list contains pages that have been seen only once recently, while L2 contains pages that have been seen at least twice recently. The replacement policy for managing DBL is: Replace the LRU page in L1 if it contains exactly c pages; otherwise, replace the LRU page in L2.

The ARC policy builds on DBL by carefully selecting c pages from the $2c$ pages in DBL. The basic idea is to divide L1 into a top T1 and bottom B1 and to divide L2 into top T2 and bottom B2. The pages in T1 are more recent than those in B1; likewise for T2 and B2. The algorithm includes a target size target_T1 for the T1 list. The replacement policy is simple: Replace the LRU page in T1, if T1 contains at least target_T1 pages; otherwise, replace the LRU page in T2.

The adaptation comes from the fact that the target size target_T1 is continuously varied in response to an observed workload. The adaptation rule is also simple: Increase target_T1, if a hit in the history B1 is observed; similarly, decrease target_T1, if a hit in the history B2 is observed.

LRU

Consider a very simple implementation of an LRU cache to motivate ARC. A typical implementation maintains a cache directory comprising cache directory blocks (CDB), often with a structure like this:

```
struct CDB {
    long                page_number;   /* page's ID number */
    struct cache_page   *pointer;      /* page's location in cache */
    int                 ARC_where;     /* not used for LRU */
    int                 dirty;         /* if 'dirty', write before replacing */
    struct CDB          *lrunext;      /* for doubly linked list */
    struct CDB          *lruprev;      /* for doubly linked list */
};

struct  CDB *L;                        /* the LRU list */

long    LLength;                       /* length of list L */
```

LRU caches with c pages require c CDBs. The following code manages the LRU list and is invoked for each page request:

```
/* keep the cache descriptor list, L, in LRU order */

struct CDB *
LRU (long page_number, int dirty) {
    struct CDB  *temp;
    temp = locate(page_number);                  /* search L for page #page_number */
    if (temp != NULL)                            /* page in cache? */
        remove_from_list(temp);                  /* page in cache: remove now, reinsert later */
    else {                                       /* page is not in cache ... */
        if (LLength == c) {                      /* cache full? */
            temp = lru_remove(L);                /* cache full: remove the LRU page at end of list */
            if (temp->dirty)                     /* dirty -> page out changed pages */
                destage(temp);
        } else {                                 /* cache not yet full */
            temp = get_new_CDB();                /* populate & bookkeep */
            temp->pointer = get_new_page();
            LLength++;
        }
        temp->page_number = page_number;    .    /* bookkeep */
        temp->dirty = dirty;                     /* bookkeep */
        fetch(page_number, temp->pointer, dirty);/* put new page in place */
    }
    mru_insert(temp, L);                         /* this page now goes to head of LRU queue */
    return temp;
}
```

We leave the simple routines locate, remove_from_list, mru_insert, lru_remove, destage, get_new_CDB, get_new_page, and fetch as an exercise. If the page is dirty, that is, a write request, then the fetch routine simply uses the changed page supplied by the host if the page is a read request, then the fetch routine reads the page from the auxiliary memory. Any existing LRU implementation already has these routines.

ARC

ARC requires 2*c CDBs. The extra directory entries maintain a history of certain recently evicted pages. The key new idea is the use of this history to guide an adaptation process. The cache directory consists of four disjoint doubly linked LRU lists along with their lengths:

```
struct CDB *T1, *B1, *T2, *B2;
long T1Length, T2Length, B1Length, B2Length;
```

Any given CDB will occupy a spot on one of the four lists. The field ARC_where will be set to 0, 1, 2, or 3, depending on the list in which it appears (T1, B1, T2, or B2, respectively).

The T1 and T2 lists describe c pages currently resident in the cache. The B1 and B2 lists contain c, a "history" of pages that were very recently evicted from the cache.

Furthermore, the T1 and B1 lists contain those pages that have been seen only once recently, while the T2 and B2 lists contain those pages that have been seen at least twice recently. The B1 list contains those pages evicted from T1, while B2 contains those pages that are evicted from T2.

The T1 and B1 lists capture "recency" information, while the T2 and B2 lists capture "frequency" information.

The algorithm adaptively – in a workload-specific fashion – balances between the recency and frequency lists to achieve a high hit ratio. It tries to maintain the number of pages in the T1 list to contain target_T1 pages. This parameter is adapted on virtually every request.

When the cache is full, the page to be evicted will be either the LRU page in T1 or the LRU page in T2.

This code demonstrates the page replacement procedure:

```
#define _T1_   0
#define _T2_   2
#define _B1_   1
#define _B2_   3

struct cache_page *
replace() {
    struct CDB* temp;
    if (T1Length >= max(1,target_T1)) {    /* T1's size exceeds target? */
                                           /* yes: T1 is too big */
        temp = lru_remove(T1);             /* grab LRU from T1 */
        mru_insert(temp, B1);              /* put it on B1 */
        temp->ARC_where = _B1_;            /* note that fact */
        T1Length—; B1Length++;             /* bookkeep */
    } else {
                                           /* no: T1 is not too big */
        temp = lru_remove(T2);             /* grab LRU page of T2 */
        mru_insert(temp, B2);              /* put it on B2 */
        temp->ARC_where = _B2_;            /* note that fact */
        T2Length—; B2Length++;             /* bookkeep */
    }
    if (temp->dirty) destage(temp);        /* if dirty, evict before overwrite */
    return temp->pointer;
}
```

The main algorithm comprises five cases which correspond to whether a page request is found in one of the four lists or in none of them. Only hits in T1 and T2 are actual cache hits. Hits in B1 and B2 are "phantom" hits that affect adaptation.

In particular, the cache parameter target_T1 is incremented for a hit in B1 and decremented for a hit in B2. This means that B1 hits favor recency while B2 hits favor frequency. The cumulative effect of the continual adaptation leads to an algorithm that adapts quickly to evolving workloads.

If a page is not in any of the four lists, then it is put at the MRU position in T1. From there it ultimately makes its way to the LRU position in T1 and eventually B1, unless requested once again prior to being evicted from B1, so it never enters T2 or B2. Hence, a long sequence of read-once requests passes through T1 and B1 without flushing out possibly important pages in T2. In

this sense, ARC is "scan-tolerant." Arguably, when a scan begins, fewer hits occur in B1 compared to B2. Hence, by the effect of the adaptation of target_T1, the list T2 will grow at the expense of the list T1. This further accentuates the tolerance of ARC to scans.

If the list B1 produces a lot of hits, then ARC grows T1 to make room for what appears to be localized requests, and, hence, favors recency. If the list B2 produces a lot of hits, then ARC grows T2 to favor frequency. ARC continually balances between recency and frequency in a dynamic, real-time, and self-tuning fashion, making it very suitable for workloads with a priori unknown characteristics or workloads that fluctuate from recency to frequency. ARC requires no magic parameters that need to be manually tuned or reset.

Here's the straightforward code that implements this procedure:

```
ARC(long page_number, int dirty) {
    struct CDB *temp, *temp2;
    temp = locate(page_number);                                        /* find the requested page */
    if (temp != NULL) {                                                /* found in cache directory? */
        switch (temp->ARC_where) {                                     /* yes, which list? */
            case _T1_:
                T1Length—; T2Length++;
                /* fall through */
            case _T2_:
                remove_from_list(temp);                                /* take off whichever list */
                mru_insert(temp, T2);                                  /* seen twice recently, put on T2 */
                temp->ARC_where = _T2_;                                /* note that fact */
                if (dirty) temp->dirty = dirty;                        /* bookkeep dirty */
                break;

            case _B1_:
            case _B2_:
                if (temp->ARC_where == _B1_) {                         /* B1 hit: favor recency */
                    target_T1 = min(target_T1 + max(B2Length/B1Length, 1), c);
                                                                       /* adapt the target size */
                    B1Length—;                                         /* bookkeep */
                } else {                                               /* B2 hit: favor frequency */
                    target_T1 = max(target_T1 - max(B1Length/B2Length, 1), 0);
                                                                       /* adapt the target size */
                    B2Length—;                                         /* bookkeep*/
                }
                remove_from_list(temp);                                /* take off whichever list */
                temp->pointer = replace();                             /* find a place to put new page */
                temp->page_number = page_number;                       /* bookkeep */
                temp->dirty = dirty;                                   /* bookkeep */
                mru_insert(temp, T2);                                  /* seen twice recently, put on T2 */
                temp->ARC_where = _T2_;                                /* note that fact */
                fetch(page_number, temp->pointer, dirty);             /* load page into cache */
                break;
        }
    } else {                                                           /* page is not in cache directory */
        if (T1Length + B1Length == c) {                                /* B1 + T1 full? */
            if (T1Length < c) {                                        /* Still room in T1? */
                temp = lru_remove(B1);                                 /* yes: take page off B1 */
                B1Length—;                                             /* bookkeep that */
                temp->pointer = replace();                             /* find new place to put page */
            } else {                                                   /* no: B1 must be  empty */
                temp = lru_remove(T1);                                 /* take page off T1 */
                if (temp->dirty) destage(temp);                        /* if dirty, evict before overwrite */
                T1Length—;                                             /* bookkeep that */
            }
        } else {                                                       /* B1 + T1 have less than c pages */
            if (T1Length + T2Length + B1Length + B2Length >= c) {      /* cache full? */
                /* Yes, cache full: */
                if (T1Length + T2Length + B1Length + B2Length == 2*c) {
                    /* directory is full: */
                    B2Length—;                                         /* find and reuse B2's LRU */
                    temp = lru_remove(B2);
                } else                                                 /* cache directory not full, easy case */
                    temp = get_new_CDB();
                temp->pointer = replace();                             /* new place for page */
            } else {                                                   /* cache not full, easy case */
                temp = get_new_CDB();
                temp->pointer = get_new_page();
            }
        }
        mru_insert(temp, T1);                                          /* seen once recently, put on T1 */
        T1Length++;                                                    /* bookkeep: */
        temp->ARC_where = _T1_;
        temp->page_number = page_number;
        temp->dirty = dirty;
        fetch(page_number, temp->pointer, dirty);                     /* load page into cache */
    }
}
```

The Proof Is in the Pudding

Although ARC uses four lists, the total amount of movement between lists is comparable to LRU. The space overhead of ARC due to extra cache directory entries is only marginally higher – typically less than 1%. Hence, we say that ARC is low-overhead.

To assess ARC's performance, we conducted trace-driven simulations, results of which populate Table 1. ARC outperforms LRU for a wide range of real-life workloads – sometimes quite dramatically. For brevity, we have shown only one typical cache size for each workload. In fact, ARC outperforms LRU across the entire range of cache sizes for every workload in our test!

Traces P1–P14 were collected by using VTrace over several months from Windows NT workstations running real-life applications. ConCat was obtained by concatenating the traces P1–P14, while Merge(P) was obtained by merging them. DS1 is a seven-day trace taken from a database server at a major insurance company. The page size for all these (slightly older) traces was 512 bytes. We captured a trace of the SPC1 (Storage Performance Council) synthetic benchmark, which is designed to contain long sequential scans in addition to random accesses. The page size for this trace was 4KB. Finally, we considered three traces – S1, S2, and S3 – that were disk-read accesses initiated by a large commercial search engine in response to various Web search requests over several hours. The page size for these traces was also 4KB. The trace Merge(S) was obtained by merging the traces S1–S3 using timestamps on each of the requests.

Conclusion

ARC is an easily implemented, new, self-tuning, low-overhead, scan-tolerant cache replacement policy that seems to outperform LRU on a wide range of real-life workloads. We have outlined a simple implementation that may be adapted to a variety of applications. The reader interested in a formal presentation of ARC, a detailed literature review, and extensive simulation results can consult the full paper, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in USENIX Conference on File and Storage Technologies (FAST '03), March 31–April 2, 2003, San Francisco, CA (*http://www.usenix.org/events/fast03/*).

WORKLOAD	SIZE (MB)	LRU (% HITS)	ARC (% HITS)
P1	16	16.55	28.26
P2	16	18.47	27.38
P3	16	3.57	17.12
P4	16	5.24	11.24
P5	16	6.73	14.27
P6	16	4.24	23.84
P7	16	3.45	13.77
P8	16	17.18	27.51
P9	16	8.28	19.73
P10	16	2.48	9.46
P11	16	20.92	26.48
P12	16	8.93	15.94
P13	16	7.83	16.60
P14	16	15.73	20.52
ConCat	16	14.38	21.67
Merge(P)	128	38.05	39.91
DS1	1024	11.65	22.52
SPC1	4096	9.19	20.00
S1	2048	23.71	33.43
S2	2048	25.91	40.68
S3	2048	25.26	40.44
Merge(S)	4096	27.62	40.44

Table 1. At-a-glance comparison of LRU and ARC for various workloads. It can be seen that ARC outperforms LRU, sometimes quite dramatically.