RJ 10285 (A0302-014) February 19, 2003
Computer Science

IBM Research Report

One Up on LRU

Nimrod Megiddo, Dharmendra S. Modha
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

One Up on LRU

Nimrod Megiddo and Dharmendra S. Modha
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120
Email: {megiddo,dmodha}@almaden.ibm.com

February 17, 2003

Abstract

Caching using LRU is an ubiquitous metaphor in modern computing. Is it possible to improve on LRU across a wide range of workloads and cache sizes without incurring excess overhead or requiring workload specific pre-tuning? We present Adaptive Replacement Cache (ARC) algorithm that seems to achieve this tantalizing target. This article is a "how-to" on converting a LRU implementation into ARC.

1 Introduction

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.

A cache is a fast, but small memory in front of a slow, but large auxiliary memory. Both memories handle uniformly sized items called *pages*. Requests for pages are first directed to the cache and, if the page is not in the cache, then to the auxiliary memory. If a new page not in the cache is requested, one of the pages currently in the cache must be flushed. A *replacement policy* determines which page is evicted. LRU is the most widely used replacement policy.

Until recently, attempts to outperform LRU in practice had not succeeded because of overhead issues and the need to pre-tune parameters. Adaptive Replacement Cache (ARC) is a new adaptive, self-tuning replacement policy with a high hit ratio and low overhead. It responds online to changing access

patterns, continually balances 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 and its running time per request is essentially independent of the cache size. Also, unlike LRU, ARC is "scan-resistant" 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.

In this article, we will demonstrate that it is extremely easy to convert an existing LRU implementation into an ARC implementation.

2 LRU

We first describe a simple implementation of LRU in C to motivate ARC. To manage a cache, usually, a cache directory consisting of cache directory blocks (CDB) is maintained.

```
struct CDB {
    long    page_number;
    struct  cache_page  *pointer;
    int     ARC_where, dirty;
    struct  CDB   *lrunext, *lruprev;
}
```

The field ARC_where is not used for LRU. The items lrunext and lruprev are used for creating a doubly-linked list. The field dirty is set to 1 for write data, and 0 for read. If the cache can hold c pages, then LRU needs c CDBs. The LRU list will be called L.

```
struct CDB *L;
```

We assume that the list L contains c pages already; in other words, we will not describe the code during the warm-up or start-up period of the cache. The following routine should be invoked upon every page request.

```
LRU (long page_number, int dirty) {
    struct CDB  *temp;
    temp = locate(page_number); /* use hash to locate the request page */
    if (temp != NULL) { /* hit */
        remove_from_list(temp); /* removes from current position */
    } else { /* miss */
```

```
        temp = lru_remove(L); /* removes the {\tt LRU} page */
        if (temp->dirty) destage(temp);
        temp->page_number = page_number; temp->dirty = dirty;
        fetch(page_number, temp->pointer);
    }
    mru_insert(temp, L); /* inserts at the {\tt MRU} position */
}
```

We leave the simple routines `locate, remove_from_list, mru_insert, lru_remove, destage`, and `fetch` to our smart readers as an exercise. Any existing `LRU` implementation already has these routines.

3 ARC

If the cache can support `c` pages, `ARC` will use `2*c` `CDB`s. In other words, `ARC` will maintain a cache directory that is twice the size of the cache. We will use the extra directory entries to maintain a history of certain recently evicted pages. The key new idea is to use this history to guide a certain adaptation process. The cache directory will consist of four disjoint doubly-linked `LRU` lists.

```
struct CDB *T1, *B1, *T2, *B2;
```

A `CDB` can be on only one of the four lists. The field `ARC_where` will be set to `0, 1, 2`, or `3`, respectively, if a `CDB` is contained in `T1, B1, T2`, or `B2`. The algorithm will maintain lengths of the lists `T1`, `B1`, and `B2`.

```
long T1Length, B1Length, B2Length;
```

The lists `T1` and `T2` will contain those `c` pages that are in the cache. The lists `B1` and `B2` will contain some `c` history pages that were recently evicted from the cache. For simplicity, we assume that the lists `T1` and `T2` contain `c` pages already and that the lists `B1` and `B2` contain `c` pages already. The lists `T1` and `B1` will contain those pages that have been seen only once recently, while the lists `T2` and `B2` will contain those pages that have been seen at least twice recently. The list `B1` contains those pages that are evicted from `T1`, and the list `B2` contains those pages that are evicted from `T2`. We think of the lists `T1` and `B1` as capturing "recency", while the lists `T2` and `B2` as capturing "frequency". The algorithm will adaptively–in a workload specific fashion–balance between the recency and frequency to achieve a high hit ratio. The algorithm tries to maintain the number of pages in the list `T1` to contain

```
long    target_T1;
```

pages. The parameter target_T1 will be 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.

```
struct CDB* replace() {
    struct CDB* temp;
    if (T1Length >= max(1,target_T1)) {
        temp = lru_remove(T1);
        mru_insert(temp, B1);
        T1Length--; B1Length++;
     } else {
        temp = lru_remove(T2);
        mru_insert(temp, B2);
        B2Length++;
    }
    if (temp->dirty) destage(temp);
    return temp;
}
```

We are now ready to present the main algorithm that should be invoked upon every new input request. It really consists of five cases depending upon whether a hit in one of the four lists happened or in none of them happened. Remember that only hits in T1 and T2 are really cache hits. The hits in B1 and B2 are really phantom hits that are used to effect an adaptation. In particular, we increment target_T1 on a hit in B1, and we decrement target_T1 on a hit in B2. In words, on a hit in B1 (resp. B2), we favor recency (resp. frequency).

```
ARC(long page_number, int dirty) {
    struct CDB *temp, *temp2;
    temp = locate(page_number); /* use hash to locate the request page */
    if (temp != NULL) {
        switch (temp->ARC_where) {
            case 0: /* the page is in T1 */
                T1Length--;
            case 2: /* the page is in T2 */
                remove_from_list(temp);
                temp->ARC_where = 2;
                if (dirty) temp->dirty = dirty;
```

```
            T2->mru_insert(temp);
            break;
        case 1: /* the page is in B1 */
        case 3: /* the page is in B2 */
            if (temp->ARC_where == 1) {
                /* adaptation to favor recency */
                target_T1 = min( target_T1 +
                    max(B2Length/B1Length, 1), c);
                B1Length--;
            } else {
                /* adaptation to favor frequency */
                target_T1 = max( target_T1 -
                    max(B1Length/B2Length, 1), 0);
                B2Length--;
            }
            remove_from_list(temp);
            temp2 = replace();
            temp->pointer = temp2->pointer; temp2->pointer = NULL;
            temp->page_number = page_number; temp->dirty = dirty;
            temp->ARC_where = 2;
            mru_insert(temp, T2);
            fetch(page_number, temp->pointer);
            break;
    }
} else { /* the requested page is not in T1, T2, B1, or B2 */
    if (T1Length + B1Length == c) {
        if (T1Length < c) {
            B1Length--; temp = lru_remove(B1);
            temp2 = replace();
            temp->pointer = temp2->pointer; temp2->pointer = NULL;
        } else {
            T1Length--; temp = lru_remove(T1);
        }
    } else {
        B2Length--; temp = lru_remove(B2);
        temp2 = replace();
        temp->pointer = temp2->pointer; temp2->pointer = NULL;
    }
    T1Length++;
    temp->ARC_where = 0;
```

```
        temp->page_number = page_number; temp->dirty = dirty;
        mru_insert(temp, T1);
        fetch(page_number, temp->pointer);
    }
}
```

4 Discussion

Although ARC uses four lists, the total amount of movement between lists is comparable to LRU. Also, 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.

If a page is not in any of the four lists, then it is put at the MRU position in T1. From there it 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-resistant." 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 resistance of ARC to scans.

If the list B1 produces a lot of hits, then ARC grows T1, and, hence, favors recency. Whereas if the list B2 produces a lot of hits, then ARC grows T2, and, hence, favors frequency. The algorithm ARC continually balances between recency and frequency in a dynamic, online, and self-tuning fashion making it very suitable for workloads with *a priori* unknown characteristics or workloads that fluctuate from recency to frequency. The algorithm ARC requires no magic parameters that need to be manually pre-set.

5 The Proof is in the Pudding

To assess the performance improvement due to ARC, we ran trace-driven simulations. We demonstrate in Table 1 that ARC ourperforms LRU for a wide range of real-life workloads–sometimes quite dramatically. Although, 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 each workload! We briefly describe various traces. P1-P14 were each collected over several months from Windows NT workstations running real-life applications by using VTrace. ConCat was obtained by concatenating the traces P1-P14,

Workload	space MB	LRU	ARC
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.

while Merge(P) was obtained by merging them. DS1 is a 7-day trace taken off a database server at a major Insurance company. The page size for all these 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 4 KBytes. Finally, we consider 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 serveral hours. The page size for these traces was 4 KBytes. The trace Merge(S) was obtained by merging the traces S1–S3 using time stamps on each of the requests.

6 Conclusions

We have presented a new self-tuning, low overhead, scan-resistant cache replacement policy `ARC` 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 more details can the 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/).