ARIZONA STATE UNIVERSITY

CSE 430 SLN 14814— **Operating Systems** — Spring 2015

Instructor: Dr. Violet R. Syrotiuk

Project #3

Available 03/31/2014; milestone due Wednesday, 04/15/2015; full project due Wednesday, 04/28/2015

Caching is one of the oldest and most fundamental concepts in modern computing. It is widely used in storage systems, databases, web servers, middleware, processors, file systems, disk drives, RAID controllers, operating systems, and in varied an numerous other applications.

In this project you will implement an LRU replacement algorithm and also the ARC replacement algorithm on real-life workloads, comparing the hit ratios of each algorithm for a given cache size.

1 Introduction

Consider a system consisting of two levels of memory: cache and auxiliary. The cache is assumed to be significantly faster than the auxiliary memory, but is also much more expensive. Hence, the size of the cache memory is usually only a fraction of the size of the auxiliary memory. Both memories are managed in units of uniformly sized items known as pages. We assume that the cache receives a continuous stream of requests for pages. We assume a demand paging scenario where a page of memory is paged in the cache from the auxiliary memory only when a request is made for the page and the page is not already present in the cache. If the cache is full, before a new page can be brought in one of the existing pages must be paged out. The victim page is selected using a cache replacement policy. In this project, you will implement the Least Recently Used (LRU), and the Adaptive Replacement Cache (ARC) cache replacement policies, computing their respective hit ratios for a given cache size on several real-life workloads.

1.1 Motivation

Real-life workloads possess a great deal of richness and variation, and do not admit a one-size-fits all characterization. They may contain long sequential I/Os or moving hot spots. The frequency and scale of temporal locality may also change with time. They may fluctuate between stable, repeating access patterns and access patterns with transient clustered references. No static, a priori fixed replacement policy works well over such access patterns. The ARC policy attempts to address this problem by adapting in an on-line, on-the-fly fashion to such dynamically evolving workloads.

1.2 Overview of ARC

The idea behind ARC is to maintain two LRU lists of pages. One list, L_1, contains pages that have been seen only once "recently," while the other list, L_2, contains pages that have been seen at least twice "recently." The items that have been seen twice within a short time have a low inter-arrival rate, and, hence, are thought of as "high-frequency." Hence, L_1 captures "recency" while L_2 as captures "frequency." We endeavour to keep these two lists to roughly the same size, namely, the cache size c. Together the two lists remember exactly twice the number of pages that would fit in the cache.

At any time, ARC chooses a variable number of most recent pages to keep from L_1 and from L_2. The precise number of pages drawn from the list L_1 is a parameter that is adaptively and continually tuned. Let FRC_p denote a fixed replacement policy that attempts to keep the p most recent pages in L_1 and $c-p$ most recent pages in L_2 in cache at all times. At any given time, the ARC policy behaves like FRC_p for some fixed p. However, ARC may behave like FRC_p at one time and like FRC_q, $p \neq q$, at some other time.

The key new idea in the ARC policy is to adaptively — in response to an evolving workload — decide how many top pages from each list to maintain in the cache at any given time. Such on-line, on-the-fly

adaptation is achieved by using a learning rule that allows ARC to track a workload quickly and effectively. Intuitively, by learning from the recent past, ARC attempts to keep those pages in the cache that have the greatest likelihood of being used in the near future. It acts as a filter to detect and track temporal locality. If during some part of the workload, recency (resp., frequency) becomes important, then ARC detects the change, and configures itself to exploit the opportunity. ARC dynamically, adaptively, and continually balances between recency and frequency — in an online and self-tuning fashion — in response to evolving and possibly changing access patterns.

2 A Class of Replacement Policies

Let c denote the cache size in number of pages. For a cache replacement policy π, we write $\pi(c)$ to emphasize the number of pages managed by the policy. We first introduce a policy DBL($2c$) that manages and remembers twice the number of pages present in the cache. With respect to the DBL policy, we then introduce a new class of cache replacement policies.

2.1 Double Cache and a Replacement Policy

Suppose we have a cache that can hold $2c$ pages. We now describe a cache replacement policy DBL($2c$) to manage such a cache. We use this construct to motivate the development of an adaptive cache replacement policy for a cache of size c.

The cache replacement policy DBL($2c$) maintains two variable-sized lists L_1 and L_2, the first containing pages that have been seen *only once recently* and the second containing pages that have been seen *at least twice recently*. More precisely, a page is in L_1 if has been requested exactly once since the last time it was removed from $L_1 \cup L_2$, or if it was requested only once and was never removed from $L_1 \cup L_2$. Similarly, a page is in L_2 if it has been requested more than once since the last time it was removed from $L_1 \cup L_2$, or was requested more than once and was never removed from $L_1 \cup L_2$.

The replacement policy is:

- Replace the LRU page in L_1, if L_1 contains exactly c pages; otherwise, replace the LRU page in L_2.

The replacement policy attempts to equalize the sizes of two lists. The structure of the cache replacement policy is illustrated in Figure 1. The sizes of the two lists can fluctuate, but the algorithm ensures that following invariants always hold:

$$\begin{aligned} 0 &\leq |L_1| + |L_2| \leq 2c, \\ 0 &\leq |L_1| \leq c, \\ 0 &\leq |L_2| \leq 2c. \end{aligned}$$

2.2 A New Class of Policies

We now propose a new class of policies $\Pi(c)$. Intuitively, the proposed class contains demand paging policies that track all $2c$ items that would have been in a cache of size $2c$ managed by DBL, but physically keep only (at most) c of those in the cache at any given time.

Let L_1 and L_2 be the lists associated with DBL($2c$). Let $\Pi(c)$ denote a class of demand paging cache replacement policies such that for every policy $\pi(c) \in \Pi(c)$ there exists a dynamic partition of list L_1 into a top portion T_1^π and a bottom portion B_1^π, and a dynamic partition of list L_2 into a top portion T_2^π and a bottom portion B_2^π such that the following conditions hold:

1. The lists T_1^π and B_1^π are disjoint, the lists T_2^π and B_2^π are disjoint and $L_1 = T_1^\pi \cup B_1^\pi$ and $L_2 = T_2^\pi \cup B_2^\pi$.
2. If $|L_1 \cup L_2| < c$, then both B_1^π and B_2^π are empty.

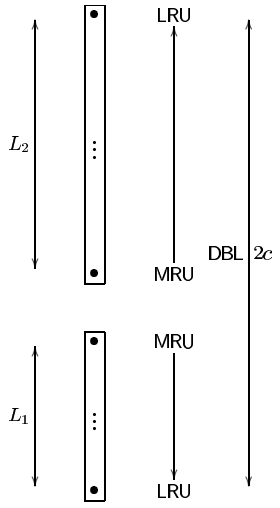

Figure 1: General structure of the cache replacement policy DBL. The cache is partitioned into two LRU lists: L_1 and L_2. The former contains pages that have been seen only once "recently" while the latter contains pages that have been seen at least twice "recently." Visually, the list L_2 is inverted when compared to L_1. This allows us to think of the more recent items in both the lists as closer in the diagram. The replacement policy deletes the LRU page in L_1, if L_1 contains exactly c pages; otherwise, it replaces the LRU page in L_2.

3. If $|L_1 \cup L_2| \geq c$, then, together T_1^π and T_2^π contain exactly c pages.

4. Either T_1^π (resp., T_2^π) or B_1^π (rest., B_2^π) is empty, or the LRU page in T_1^π (resp., T_2^π) is more recent than the MRU page in B_1^π (rest., B_2^π).

5. For all traces and at each time, $T_1^\pi \cup T_2^\pi$ contains exactly those pages that would be maintained in the cache by the policy $\pi(c)$.

The generic structure of the policy π is depicted in Figure 2. It follows from Conditions 1 and 5 that the pages contained in a cache of size c managed by $\pi(c)$ is always a subset of the pages managed by DBL($2c$).

Remark. Condition 4 implies that if a page in L_1 (resp., L_2) is kept, then all pages in L_1 (resp., L_2) that are more recent than it must all be kept in the cache. Hence, the policy $\pi(c)$ can be thought of as "skimming the top (or most recent) few pages" in L_1 and L_2. Suppose that we are managing a cache with $\pi(c) \in \Pi(c)$, and also let us suppose that the cache is full, that is, $|T_1^\pi \cup T_2^\pi| = c$, then it follows from Condition 4 that, from now on, for any trace, on a cache miss, only two actions are available to the policy $\pi(c)$: (i) replace the LRU page in T_1^π or (ii) replace the LRU page in T_2^π.

2.3 LRU

We now show that the policy LRU(c) is contained in the class $\Pi(c)$. In particular, we show that the most recent c pages are always contained in DBL($2c$). To see this fact observe that DBL($2c$) deletes either the LRU item in L_1 or the LRU item in L_2. In the first case, L_1 must contain exactly c items. In the second case, L_2 must contain at least c items. Hence, DBL never deletes any of the most recently seen c pages and always contains all pages contained in a LRU cache with c items. It now follows that there must exist a dynamic partition of lists L_1 and L_2 into lists T_1^{LRU}, B_1^{LRU}, T_2^{LRU}, and B_2^{LRU} such that the conditions 1–5 hold. Hence, the policy LRU(c) is contained in the class $\Pi(c)$ as claimed.

Conversely, if we consider DBL($2c'$) for some positive integer $c' < c$, then the most recent c pages need not always be in DBL($2c'$). For example, consider the trace $1, 2, \ldots, c, 1, 2, \ldots, c, \ldots, 1, 2, \ldots, c, \ldots$. For this

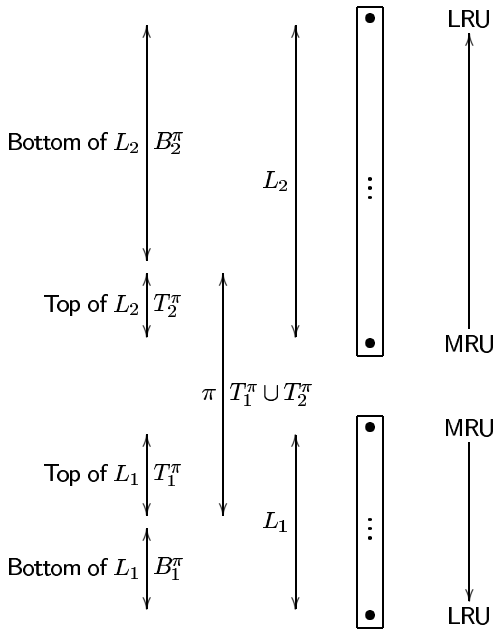

Figure 2: General structure of the cache replacement policy $\pi(c) \in \Pi(c)$. The lists L_1 and L_2 are exactly as in Figure 1. The list L_1 is partitioned into a top portion T_1^π and a bottom portion B_1^π. Similarly, the list L_2 is partitioned into a top portion T_2^π and a bottom portion B_2^π. The policy $\pi(c)$ maintains pages in $T_1^\pi \cup T_2^\pi$ in cache.

trace, hit ratio of LRU(c) approaches 1 as the size of the trace increases, but the hit ratio of DBL($2c'$), for any $c' < c$, is zero.

3 Adaptive Replacement Cache (ARC)

3.1 Fixed Replacement Cache

We now describe a *fixed replacement cache*, $\text{FRC}_p(c)$, in the class $\Pi(c)$, where p is a unable parameter, $0 \leq p \leq c$. The policy $\text{FRC}_p(c)$ satisfies conditions 1–5. For brevity, we write $T_{1,p} \equiv T_1^{FRC_p(c)}$, $T_{2,p} \equiv T_2^{FRC_p(c)}$, $B_{1,p} \equiv B_1^{FRC_p(c)}$, and $B_{2,p} \equiv B_2^{FRC_p(c)}$.

The crucial additional condition that $\text{FRC}_p(c)$ satisfies is that it attempts to keep exactly p pages in list $T_{1,p}$ and exactly $c - p$ pages in the list $T_{2,p}$. In other words, the policy $\text{FRC}_p(c)$ attempts to keep exactly p top (most recent) pages from the list L_1 and $c - p$ top (most recent) pages from the list L_2 in the cache. Intuitively, the parameter p is the *target size* for the list $T_{1,p}$. The replacement policy is:

1. If $|T_{1,p}| > p$, replace the LRU page in $T_{1,p}$.
2. If $|T_{1,p}| < p$, replace the LRU page in $T_{2,p}$.
3. If $|T_{1,p}| = p$, and the missed page is in $B_{1,p}$ (resp., $B_{2,p}$), replace the LRU page in $T_{2,p}$ (resp., $B_{2,p}$).

The last replacement decision is somewhat arbitrary, and can be made differently if desired. The subroutine REPLACE in Algorithm 1 is consistent with 1–3.

3.2 The ARC Policy

We now introduce an adaptive replacement policy ARC(c) in the class $\Pi(c)$. At any time, the behaviour of the ARC policy is completely described once a certain adaptation parameter $p \in [0, c]$ is known. For a given value of p, ARC behaves exactly like FRC$_p$. But, ARC differs from FRC$_p$ in that it does not use a single fixed value for the parameter p over the entire workload. The ARC policy continuously adapts and tunes p in response to an observed workload.

Let T_1^{ARC}, T_2^{ARC}, B_1^{ARC}, and B_2^{ARC} denote a dynamic partition of L_1 and L_2 corresponding to ARC. For brevity we write $T_1 \equiv T_1^{ARC}$, $T_2 \equiv T_2^{ARC}$, $B_1 \equiv B_1^{ARC}$, and $B_2 \equiv B_2^{ARC}$.

The policy ARC dynamically decides, in response to an observed workload, which item to replace at any given time. Specifically, on a cache miss, ARC adaptively decides whether to replace the LRU page in T_1 or to replace the LRU page in T_2 depending upon the value of the adaptation parameter p at that time. The intuition behind the parameter p is that it is the target size for the list L_1. When p is close to 0 (resp., c), the algorithm can be thought of as favouring L_2 (resp., L_1).

We now provide the complete ARC policy in Algorithm 1. If the two "ADAPTATION" steps are removed, and the parameter p is a priori fixed to a given value, then the resulting algorithm is exactly FRC$_p$. The algorithm also implicitly simulates the lists $L_1 = T_1 \cup B_1$ and $L_2 = T_2 \cup B_2$.

3.3 Learning

The policy continually revises the parameter p. The fundamental intuition behind learning is the following: if there is a hit in B_1 then we should increase the size of T_1, and if there is a hit in B_2 then we should increase the size of T_2. Hence, on a hit in B_1, we increase p which is the target size of T_1, and on a hit in B_2, we decrease p. When we increase (resp., decrease) p, we implicitly decrease (resp., increase) $c - p$ which is the target size of T_2. The precise magnitude of the revision in p is also very important. The quantities δ_1 and δ_2 control the magnitude of revision. These quantities are termed the *learning rates*. The learning rates depend upon the sizes of the lists B_1 and B_2. On a hit in B_1, we increment p by 1, if the size of B_1 is at least the size of B_2; otherwise, we increment p by $|B_2|/|B_1|$. All increments to p are subject to a cap of c. Thus, the smaller the size of B_1, the larger the increment.

Similarly, on a hit in B_2, we decrement p by 1, if the size of B_2 is at least the size of B_1; otherwise, we decrement by $|B_1|/|B_2|$. All decrements to p are subject to a minimum of 0. Thus, the smaller the size of B_2, the larger the decrement.

The idea is to "invest" in the list that is performing the best. The compound effect of a number of such small increments and decrements to p is quite profound. We can roughly think of the learning rule as inducing a "random walk" on the parameter p.

Observe that ARC never becomes complacent and never stops adapting. Thus, if the workload were to suddenly change, then ARC tracks this change and adapts itself accordingly to exploit the new opportunity. The algorithm continuously revises how to invest in L_1 and L_2 according to the recent past.

4 Input Data from Real-Life Workloads

Table 1 summarizes various traces. These traces capture disk accesses by databases, web servers, NT workstations, and a synthetic benchmark for storage controllers. Some of these traces have been provided as input files to your program; each is a text file with the name `TraceName.lis`.

The format of each file is as follows: Every line in every file has four fields.

1. First field: starting_block
2. Second field: number_of_blocks (each block is 512 bytes)
3. Third field: ignore
4. Fourth field: request_number (starts at 0)

Algorithm 1 Algorithm for Adaptive Replacement Cache (ARC). This algorithm is completely self-contained, and can directly be used as a basis for an implementation. No tunable parameters are needed as input to the algorithm. We start from an empty cache and an empty cache directory. ARC corresponds to $T_1 \cup T_2$ and DBL corresponds to $T_1 \cup T_2 \cup B_1 \cup B_2$.

ARC(c)

INPUT: The request stream $x_1, x_2, \ldots, x_t, \ldots$.
INITIALIZATION: Set $p = 0$ and set the LRU lists T_1, B_1, T_2, and B_2 to empty.

For every $t \geq 1$ and any x_t, one and only one of the following four cases must occur.

Case I: x_t is in T_1 or T_2. A cache hit has occurred in ARC(c) and DBL($2c$).
Move x_t to MRU position in T_2.

Case II: x_t is in B_1. A cache miss (resp. hit) has occurred in ARC(c) (resp. DBL($2c$)).

ADAPTATION: Update $p = \min\{p + \delta_1, c\}$ where $\delta_1 = \begin{cases} 1 & \text{if } |B_1| \geq |B_2| \\ |B_2|/|B_1| & \text{otherwise.} \end{cases}$

REPLACE(x_t, p). Move x_t from B_1 to the MRU position in T_2 (also fetch x_t to the cache).

Case III: x_t is in B_2. A cache miss (resp. hit) has occurred in ARC(c) (resp. DBL($2c$)).

ADAPTATION: Update $p = \max\{p - \delta_2, 0\}$ where $\delta_2 = \begin{cases} 1 & \text{if } |B_2| \geq |B_1| \\ |B_1|/|B_2| & \text{otherwise.} \end{cases}$

REPLACE(x_t, p). Move x_t from B_2 to the MRU position in T_2 (also fetch x_t to the cache).

Case IV: x_t is not in $T_1 \cup B_1 \cup T_2 \cup B_2$. A cache miss has occurred in ARC(c) and DBL($2c$).

Case A: $L_1 = T_1 \cup B_1$ has exactly c pages.
If ($|T_1| < c$)
Delete LRU page in B_1. REPLACE(x_t, p).
else
Here B_1 is empty. Delete LRU page in T_1 (also remove it from the cache).
endif

Case B: $L_1 = T_1 \cup B_1$ has less than c pages.
If ($|T_1| + |T_2| + |B_1| + |B_2| \geq c$)
Delete LRU page in B_2, if ($|T_1| + |T_2| + |B_1| + |B_2| = 2c$).
REPLACE(x_t, p).
endif

Finally, fetch x_t to the cache and move it to MRU position in T_1.

Subroutine REPLACE(x_t, p)
If (($|T_1|$ is not empty) and (($|T_1|$ exceeds the target p) or (x_t is in B_2 and $|T_1| = p$)))
Delete the LRU page in T_1 (also remove it from the cache), and move it to MRU position in B_1.
else
Delete the LRU page in T_2 (also remove it from the cache), and move it to MRU position in B_2.
endif

As an example: first line in `P6.lis` is: 110765 64 0 0. Here, 110765 is the starting block, there are 64 blocks each of 512 bytes so this represents 64 requests (each of a 512 byte page) from 110765 to 110828, the first zero is ignored, and the second zero is a request number (that goes from 0 to 63).

Table 1: A summary of various traces.

Trace Name	Number of Requests	Unique Pages
OLTP	914145	186880
P1	32055473	2311485
P2	12729495	913347
P3	3912296	762543
P4	19776090	5146832
P5	22937097	3403835
P6	12672123	773770
P7	14521148	1619941
P8	42243785	977545
P9	10533489	1369543
P10	33400528	5679543
P11	141528425	4579339
P12	13208930	3153310
P13	15629738	2497353
P14	114990968	13814927
ConCat	490139585	47003313
Merge(P)	490139585	47003313
DS1	43704979	10516352
SPC1 like	41351279	6050363
S1	3995316	1309698
S2	17253074	1693344
S3	16407702	1689882
Merge (S)	37656092	4692924

Hit ratios for each the various trace files can be found in the original research paper describing the ARC policy [1], and also in the presentation `arc-fast.pdf` in the folder for Project #3. You may compare your results to these to check correctness of your implementation.

For example, for the trace file `P4.lis`, Table 2 shows a comparison of ARC hit ratio to LRU and MQ (which you are not required to implement) for $c = 1024, 2048, \ldots, 524288$. Hit ratios are reported as percentages.

5 Project #3 Requirements

In this project you are to implement two cache replacement policies, LRU and ARC. The program for each policy takes two command line parameters:

1. c, the cache size in number of pages, and
2. `TraceName` indicating that the trace file to use is `TraceName.lis`.

Your program should process the trace file (see §4 for a description of the trace file format) and compute the hit ratio for the policy.

While the pseudocode for the ARC policy in Algorithm 1 is quite detailed, be sure to check boundary conditions, e.g., removing from empty queues, and ensuring the cache does not become oversized. In addition, the calculations for the adaptive parameter p often uses division of queue sizes. In some cases, the queue in the denominator may be empty, leading to a divide-by-zero exception.

Table 2: A comparison of hit ratios of LRU, MQ, and ARC on the trace `P4.lis` for increasing c.

	P4		
c	LRU	MQ ONLINE	ARC
1024	2.68	2.67	2.69
2048	2.97	2.96	2.98
4096	3.32	3.31	3.50
8192	3.65	3.65	4.17
16384	4.07	4.08	5.77
32768	5.24	5.21	11.24
65536	10.76	12.24	18.53
131072	21.43	24.54	27.42
262144	37.28	38.97	40.18
524288	48.19	49.65	53.37

Many of the trace files are large and you should expect that your programs may run for a long time! It is suggested to start with the smaller traces, e.g., `P3.lis` (or even a portion of it), and work with small values of c initially.

6 Submission Instructions

Project #3 has two submission deadlines.

1. The milestone deadline is before noon on Wednesday, 04/18/2015. The milestone for Project #3 is an implementation of the LRU replacement policy.

2. The full project deadline is before noon on Wednesday, 04/28/2015. The full project includes an implementation of the LRU and ARC replacement policies and a output from a set of experiments on trace files to compare their hit ratios.

6.1 Requirements for the Milestone Deadline

Submit electronically, before class time (noon) on Wednesday, 04/18/2015 using the submission link on Blackboard for Project #3 Milestone, a zip[1] file named `yourFirstName-yourLastName.zip` containing the following items:

Design and Analysis (30%): Briefly describe your implementation of the LRU replacement policy (e.g., what data structure(s) did you use?, what representation (e.g., float or integer) did you use for important variables?, did you use rounding or truncation?) and how you computed the hit ratio. If your implementation has special features to improve time and/or space complexity, describe them.

Implementation (50%): Provide your documented C/C++ source code for your implementation of the LRU cache replacement policy, and its hit ratio. Also provide a script file to compile your code and run it on a series of instances of input.

Correctness (20%): Your program **must** run on `general.asu.edu` using a selection of cache sizes and sample input files. Hit ratios will be compared to those for LRU published in [1].

The milestone is worth 30% of the total final project.

[1] **Do not** use any other archiving program except `zip`.

6.2 Requirements for the Full Deadline

Submit electronically, before class time (noon) on Wednesday, 04/28/2015 using the submission link on Blackboard for Project #3, a zip file named `yourFirstName-yourLastName.zip` containing the following items:

Design and Analysis (30%): Now add a brief description of your implementation of the ARC replacement policy, answering questions such as those asked in §6.1. In addition describe problems, if any, in implementing the protocol.

Implementation (50%): Provide both your documented C/C++ source code for your implementation of the LRU and ARC cache replacement policies. Also provide a script file to compile your code and run it on a series of instances of input.

Correctness (20%) Your programs **must** run on `general.asu.edu` using a selection of cache sizes and sample input files. Hit ratios will be compared to those for LRU and ARC published in [1].

References

[1] Nimrod Megiddo and Dharmendra S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," USENIX Conference on File and Storage Technologies (FAST'03), San Francisco, CA, pp. 115-130, March 31-April 2, 2003.