A New Web Cache Replacement Algorithm

Anupam Bhattacharjee, Biplob Kumar Debnath†
Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology,
Dhaka-1000, Bangladesh.
† Department of Electrical and Computer Engineering,
University of Illinois at Chicago
Chicago, IL 60607, USA.
abrubuet@yahoo.com, bkdebnath@yahoo.com

Abstract
In recent research, the problem of document replacement in web caches has received much interest. Web caches are different from system/processor caches because web caches have several additional criteria (frequency and recentness of pages, size of a document, cost of fetching a document etc.). It has been shown that, the classical LRU replacement policy performs poorly in web caches because the above criteria decrease hit rate and increase eviction latency and access latency of web request. Moreover, in order to implement the classical and novel schemes, one needs to maintain complex data structures and the operations of the data structures lead net traffic jam. The current randomized algorithms don’t perform constant well. We propose a randomized algorithm that avoids the need for any data structure and performs constantly well.

In this paper, a new randomized algorithm approximating Random Replacement (RR) and Least Recently Used (LRU) scheme is introduced for page replacement in caches. The basic version of the new proposed algorithm performs as follows: bins are logical division in cache and number of bins depends on percentage of error acceptable for approximation. When a new page is to be evicted from the cache, the algorithm randomly selects a bin containing pages (old, new, or both type of pages). After searching for an old page randomly in the selected bin, it replaces the old page, if there any, with the new page. Otherwise it selects another bin randomly and continues in the same manner. The algorithm can easily be implemented without using any complex data structure. It also reduces the response time effectively. Experiment shows that the algorithm performs better while being used in web caches.

Keywords: LFU, LRU, Randomized Algorithms, RR, Marker Rule, Web caching.

I. INTRODUCTION

The enormous popularity of the World Wide Web in recent years has caused a tremendous and exponential increase in HTTP requests increasing network traffic and HTTP page access latency. So, to reduce download latency, web server load and network traffic, web caching is a very important feature. A key component of a cache is its replacement policy, which is a decision rule for evicting a page currently in the cache to make room for a new page. The rule uses a utility function which maps pages according to their suitability for eviction. For example, the utility function of LRU assigns to each page a value which is the time since the page’s last use. The replacement scheme would then replace that page which is the most suitable for eviction.

Whereas for processor caches LRU and its variants have worked very well [1], it has recently been found [2] that LRU is not suitable for web caches. First, the sizes of the documents are not same in web caches. Secondly, the costs of fetching different documents in processor caches are same but it is not true for web cache (one reason is the size variation). So, a good utility function should combine all the criteria stated above. Some such schemes are LRU-Threshold [3], GD-Size [2], GD*[4], LRV [5], SIZE [6], HYBRID [7]. However, data structures (hash tables, priority queues etc.) that are needed for implementing these new utility functions turn out to be complicated. Many of them require priority queues in order to reduce the time to find a replacement from \(O(K)\) to \(O(\log K)\) where \(K\) be the number of documents in the cache. Further, these data structures need to be constantly updated (even there is no eviction) although they are solely used for eviction.

Algorithms that assume no knowledge of future requests and make their decisions only on past requests are termed as online algorithms [8]. The optimum online algorithm is known as Least Recently Used (LRU) algorithm [1, 12]. It works by replacing the page in the cache whose most recent request occurred farthest in the past [8, 9]. Another algorithm in this case is Least Frequently Used (LFU) algorithm [10, 11]. It works by replacing the page that has least frequency of occurring in the client requests. The implementation of LRU requires keeping track of the pages of all pages requested. In general, this is implemented by a stack or a linked list [10]. For “constantly update” operations in the schemes, up to six pointers need to be updated [10].

---

Cache replacement schemes in web-caching demand some algorithms that don’t use any very complicated data structure. For this complexity and need for hardware support LRU scheme is not implemented in most of today’s systems. Instead of using basic version of LRU, randomized algorithms that approximate LRU are used to reduce the cost of performance [11]. If randomized algorithms are used in caching the eviction rule is simply based on some sample of the cache instead of a thorough search. This is why; these algorithms are easier to be implemented.

Researches are going on finding algorithms that approximate LRU. Some algorithms having better performance and simplicity are found. The very basic randomized algorithm is the Randomized Replacement (RR) algorithm, which works by replacing a randomly drawn page from cache [8]. Marker algorithm which is better than RR associates a marker bit to each item in cache and initially resets them to zero. When an item from the cache is accessed the marker of the page is set. The replacement algorithm then chooses one page whose marker is not set and evicts it [8, 10].

In the new algorithm, the combination of LRU, RR, and Marker Algorithm is used. For the algorithm, we treat some logical divisions/distributions in a web cache. Without loosing generality we assume that there all the bins have same numbers of pages. RR is used to select bins and pages, LRU is used to find pages from the bins, and Marker Algorithm is used to distinguish between old and new pages. So, the utility function of the algorithm is based not only on the recentness of use but also on the size, eviction time, the cost of fetching and traffic latency.

The rest of the paper is organized as follows. In section (II) we present preliminaries for the algorithm. We discuss some current randomized algorithms in section (III). In section (IV) the new algorithm is presented. In section (V) and section (VI) an analysis of the algorithm is presented and its performance measurement is determined. In section (VII), an algorithm to check the states of the pages is discussed. In section (VIII) the experimental result is shown and at last, section (IX) concludes the paper.

II. PRELIMINARIES

For simplicity and better performance along with ease of discussion, we treat the web cache divided into some logical partitions. Each logical partition is termed as a bin.

For example, in Fig. 1 the web cache is divided logically into three divisions i.e. there are three bins in the web cache. As we are using randomized algorithms, our goal is to minimize errors. So, we can accept errors to a certain extent. We will express this limit of errors by \( n \) in the rest of the paper. So, a page is treated as useless if it belongs to the oldest \( n \)th percentile. An error is said to have occurred if the evicted page does not belong to the oldest \( n \)th percentile of all the pages in cache for some desirable value of \( n \). The ultimate target or goal of the algorithm is to minimize total error and response time. For the new algorithm, it is supposed that all the pages are divided into \( n \) bins. For example, if there are 30 pages and we accept 3% error, then, page #1-10 belongs to bin #1, page #11-20 to bin #2 and page #21-30 to bin #3 (for clarification check Fig. 1). If \( x \) is the number of pages drawn for sampling, the probability of errors equals to \( \left(1 - \frac{n}{100}\right)^x \), approximately \( e^{-n/100} \). With the increase of \( x \), the probability of errors decreases exponentially but at the same time the response time increases. So, clients will get responses much later and thus it increases traffic jam in the web. There should be a trade-off between the two benefits.

III. SOME RANDOMIZED ALGORITHMS

Before introducing the new algorithm, we discuss some randomized algorithms that are more efficient than the classical LRU rule. RR algorithm draws pages randomly from the web cache and evicts it with the new page. It is the simplest algorithm but inefficient as it often replaces new pages keeping old pages in the web cache. Marker algorithm associates a marker bit to each item in cache and initially resets them to zero. When an item from the cache is accessed the marker of the page is set. The replacement algorithm then chooses one page whose marker is not set and evicts it. It is more efficient than RR algorithm but its performance is not exponentially better. The latest and most successful scheme in this manner is developed by Psounis, Prabhakar and Engler [8]. Pseudo-code of the scheme:

1. if (eviction) {
2.  if (first_iteration) {

It works as follows: it draws \(N\) pages from the cache randomly and keeps \(M\) oldest pages from them. In the next iteration it draws \(N - M\) pages from the cache. Now, decision is given based on previous \(M\) saved pages and current \(N - M\) pages.

The algorithm minimizes error exponentially for small values of \(M \geq 1\). But, it increases response time. The algorithm performs well for some fixed values. Another disadvantage of the scheme is that it allows more randomization and for this reason, it lacks perfection. Our algorithm reduces the randomization part. The response time of the new algorithm is linear and it is more efficient. That is why; it doesn’t increase traffic jam in HTTP world.

IV. PSEUDO-CODE AND EXPLANATION

The pseudo-code of the proposed algorithm is as follows:

```plaintext
1. total_bins := n ;
2. while (true) {
3.   bin_selected := random (total_bins) ;
4.   if (there is an empty slot) {
5.     insert the new page there ;
6.     return ;
7.   }
8.   iteration := 0 ;
9.   while (iteration \leq \text{no of pages in the selected bin}) {
10.      select a page randomly ;
11.      if (marker bit is zero) {
12.         replace it ;
13.         return ;
14.      }
15.      iteration := iteration +1 ;
16.   }
17.   if (iteration \leq \text{no of pages in the selected bin})
18.     return ;
19. }
```

In line #1, number of bins is determined from the acceptable percentage of errors. If we want to accept up to 3% error (i.e. for 100 evictions 97 times the eviction rule evicts useless pages and 3 times new pages), 3 bins are considered. For each logical division i.e. bin, a corresponding counter is kept. Line #3 of the algorithm selects a bin randomly. If there is enough space for the new page (checked by line #4) then line #5 of the algorithm inserts the new page there, updates the corresponding counter and the task of eviction ends (line #6). If there is not enough space for the new page, the algorithm should evict an old page. It searches for an old page for a certain time (iteration \(\leq \text{number of pages in the selected bin}\)). It selects a page randomly (line #10) and checks whether the marker bit of the page is not set (line #11 i.e. it is an old page. If such a page is found it replaces the old page with the new one (line #12) and finishes the search operation. Otherwise it continues searching increasing iteration counter (line #15). Line #17 checks whether an old page has been found and evicted (line #11- line #13). If it has evicted one, the task is finished (line #18) and the algorithm terminates its operation. Otherwise, it jumps to line #3 to select another bin randomly and continues similarly.

Another point is that, when starting a web cache, all pages are new. But, the algorithm doesn’t get stacked because there are enough spaces for the new page (line #4). So, the algorithm is sound and complete.

V. ANALYZING THE ALGORITHM

Let \(B\), \(N\) and \(P\) respectively denote total number of bins treated, number of pages in any bin, and probability that a page is useless.

Probability of selecting a bin = \(\frac{1}{B}\).

Probability of selecting a page = \(\frac{1}{N}\).

So, Probability of selecting a page from a selected bin = \(\frac{1}{NB}\).

After \(k\) iterations,

Probability of an old page to be selected is

\[= \sum_{i=1}^{k} \frac{1}{BN}[(1-P)^{i-1}]P.\]

So, probability of error = probability of a new page to be selected = \(1 - \sum_{i=1}^{k} \frac{1}{BN}[(1-P)^{i-1}]P\).

It is obvious from the equations that probability of selecting old pages increases as \(k\) (iteration) increases. With the increase of iterations the number of pages feasible to be useless also decreases. So, probability of finding an old page increases too. Thus, agreeing with
the previous observations, probability of error decreases with increasing number of iterations.

VI. RESPONSE TIME

The algorithm reduces the response time very sharply. Selecting a bin needs constant time. Selecting page from the selected bin needs $O(1)$ time. So, after $k$ iterations, time to response = $O(1) + \{k \times O(1)\} = O(k)$.

Average time to respond

$$\frac{1}{N} \times \sum O(k) = O(N + 1) = O(N)$$

So, response time (time to select a page with marker bit zero) = $\frac{1}{B} \times \sum \frac{1}{N} \sum O(k)$.

For worst case, it is $O(NB)$.

For best case, it is $O(1)$.

For average case, it is $O(N)$.

So, the algorithm needs linear time to respond. But, if the number of iteration increases, it tends to best cases i.e. it needs constant time to respond. Generally, as the number of pages increases, number of old pages also increases, so it becomes easier to select an old page with less number of iterations. Hence, the time to respond decreases increasing the performance in case of web cache replacement scheme.

VII. ALGORITHM FOR STATES OF PAGES

The algorithm used to determine freshness/recentness of pages is called the Squid Proxy Cache algorithm. The algorithm is as follows:

1. if (age > client-max-age) return “STALE”
2. else if (age ≤ min-age) return “FRESH”
3. else if (expires) {
4. if (expires ≤ now) return “STALE”
5. else return “FRESH”
6. }
7. else if (age > max-age) return “STALE”
8. else if (lm-factor< percent) return “FRESH”
9. else return “STALE”

Checking of freshness is duly performed when objects are requested. There are some parameters associated with each page. The refreshment parameters that are used in the algorithm are identified as client-max-age, min-age, percent, lm-factor and max-age. client-max-age is the maximum age the client will accept as taken from http cache control request header. age is how much the object has aged since it was retrieved. There are some limits (client-max-age and max-age) for all the pages in the cache. If the age of any page becomes greater than the limits, the page should be treated as old (line #1 and #7 of the algorithm). Similarly there is a minimum age parameter (client-min-age). All pages with age less than the limit are treated as new (line #2). expires is an optional field used to mark an object’s expiry date and the parameter is used to restrict some special pages to remain valid till expire date. now indicates the current date for the cache. If the page has already expired then it belongs to old class. Otherwise it should be treated as new (line #4 and #5). lm-factor is the ratio of age over how old was the object when it was retrieved. percent is the acceptable percentage of error. A simple observation leads the conclusion that, any page with lm-factor less than percent should be treated as fresh page.

VIII. SIMULATION RESULTS

From the simulation program, the graphs are as follows:

![Success Vs. Refresh Rate](Image)

Fig 2. $n = 5$

![Success Vs. Refresh Rate](Image)

Fig 3. $n = 10$
From Fig. 1-7, it is certain that, most of the time the success rate is above 90%. The average success rate found from the simulated data is about 94%, which is obviously good. The worst time success rate is found about 75%. The best cases has success ratio 100%. It is found that, if the refresh rate decreases the success ratio decreases with less fluctuation, but if refresh rate increases fluctuation occurs more frequently, on the other hand success rate also increases. As \( n \) increases, the success rate increases with fewer fluctuations which agree with the analyzed algorithm. Fig. 2 and Fig. 7 indicate that fluctuations decrease with increase in \( n \).

Fig. 8 below is found from simulation for obtaining hit versus total number of pages.

From the graph, it is obvious that, hit ratio is high and almost constant for increasing number of pages. It is almost constant for any variation of \( n \).

IX. CONCLUSION

In this paper, an approximate randomized algorithm is used to replace pages in web proxy cache. The worst case occurs when randomly selected bins does not contain any old page. The algorithm has a response time which is much lower than the well-known algorithms. Another major advantage of the algorithm is that, it
doesn’t depend on any complicated data structures and so it is very easy to implement.

As a rule, the algorithm can be implemented efficiently whenever there is a large collection of objects from which the old page is to choose randomly and in a very short span of time.

REFERENCES