2Q and 2Q/2 - Two Efficient Web Cache Management Techniques

Arun Dev, Prem Kumar Kandalu and Arunabha Sen[*]
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287

Abstract

In recent times, *proxy caching* has become an integral part of any efficient web *content distribution* scheme. Effective use of proxy caching can reduce user latency, load on the web server and save network bandwidth. Performance of proxy caching depends to a large extent on the *admission control policy* that decides on the web objects to be cached and the *replacement policy* that determines the object to be removed from the cache in case of contention for cache space. Many replacement policies, from very simple such as *LRU*, to fairly complex, such as the *greedy-dual-size-frequency* have been proposed in the last few years. Although it is well known that LRU is not particularly effective as a replacement policy for the web traffic, it is still extensively used due to its inherent simplicity. The computational complexity of LRU is $O(1)$, whereas the complexity of many other *sophisticated* policies are $O(log\ n)$, where n is the number of objects in the cache. In this paper we consider a policy, known as the *2Q replacement algorithm*, that was originally proposed for the database environment. We also develop a variation of the 2Q algorithm and call it 2Q/2. We evaluate the performance of these two schemes through simulation using synthetic and trace data. The simulation results (with 95% confidence interval) demonstrate significant performance gain over both LRU and LRU/2 scheme.

Keywords: proxy caching, replacement algorithm, least recently used, web content distribution

1 Introduction

With the explosive growth of the World Wide Web, proxy caching has become an integral part of any efficient web *content distribution* scheme. With the drastic reduction in storage costs, now it is viable to increase both the size and the number of proxy caches to alleviate the latency experienced by the user and at the same time reduce network traffic and server load. With the advent of cable modems and DSL technologies, the Internet user at home has access to higher bandwidth .

[*]Corresponding author: Arunabha Sen, Telephone: 480-965-6153, Fax: 480-965-2751, e-mail: asen@asu.edu

To reduce user latency and the load on web servers, addition of proxy caches becomes even more important in this environment. The addition of these end user technologies is slowing down an already overloaded Internet backbone. One way to deal with this problem is the introduction of web object caches. The term *flash crowd* is used to refer to the phenomenon of a web server being inundated with a very large number requests for a popular web object(s). For web objects, the problem of flash crowds can also be effectively handled by proxy caching.

In a network with proxy servers, a client's request for a web object is initially directed to a web proxy. The proxy server evaluates the request to determine if the request can be met from the objects present in its local cache. If the object is present in the cache, a *cache hit* is said to have occurred. The requested object is then transmitted (served) to the client from the proxy cache. As the proxy servers are typically located at the *edge* of the network (close to the client site), this leads to a saving in bandwidth and reduction of user latency. If the requested object is not found in the cache, a *cache miss* is said to have occurred. In this case, the proxy contacts either a *peer cache* to find out if it has a copy of the object or the main server. The requested object is then served to the client, either from the peer cache or the main server. At this point, the proxy has to make a decision on whether to cache this object for future requests. The proxy cache has a fixed amount of storage and if the proxy server decides to cache this object, it may have to remove some of the current cache resident objects to make room for this new object. A *replacement policy* is then invoked to determine the objects to be removed from the cache. In recent times, some computer scientists have argued that with the significant reduction of the cost of disk storage, search for an *ideal* replacement policy has become irrelevant, as one can easily install very large size caches. However, this argument is somewhat flawed as many content distributors (including the industry leader Akamai Technologies) serve their frequently requested objects, not from disks, but from RAM. Because the RAM is considerably more expensive than the disks, admission control and replacement policy is still important in this environment.

The two common metrics used to evaluate the performance of the replacement algorithms are *cache hit rate* and *byte hit rate*. The cache hit rate is expressed as a percentage of the number of requests serviced from the proxy cache to the total number of requests. The byte hit rate is expressed as a percentage of the number of bytes that was delivered from the proxy caches to the total number of bytes for all requests. A high cache hit rate would indicate that a large number of user requests were serviced from the proxy cache. As the proxy caches are nearer to the client than the main servers, a high cache hit ratio may indicate a reduction in client observed latency. However, the latency experienced by the user is difficult to measure as it depends on a number of factors like network congestion and web server load. A high value for the byte hit rate indicates a reduction in the core network traffic. This may also be an indirect indicator of improvement in user observed latency. A replacement policy that optimizes both the cache hit rate and the byte hit rate would be a very good choice for a cache replacement policy. However, sometimes one may have to choose between maximizing the cache hit ratio or maximizing the byte hit ratio. In order to maximize the hit rate, the policy would select a number of smaller files instead of one large file. This discrimination against the size of the object increases the hit rate but in turn reduces the byte hit rate. A single large file can impact the byte hit rate by its presence or absence in the cache.

The factors that should be taken into account while attempting to develop a good cache replacement policy include,

- *Recency of reference*: The recency of reference indicates how recently the web object was requested. The *least recently used (LRU)* replacement algorithm uses the recency of reference as the sole criteria for deciding on the web object (page) to be evicted from the cache. LRU works best in environments where the access pattern shows a strong temporal locality. Although some studies on the web proxy workloads have found locality of reference in the user request pattern, some other studies have indicated very little locality of reference in the access pattern.

- *Frequency of Reference*: The frequency of reference indicates how frequently the web object was requested. This measure separates the "popular" web objects from the "unpopular" ones. The *least frequently used (LFU)* replacement algorithm uses the frequency of reference as the sole criteria for deciding on the web object (page) to be evicted from the cache. Many studies on web access request statistics show that a large number objects are requested only once. Arlitt *et al* in [1, 2, 3, 4, 8] report that over 60% of the requested objects were distinct. They refer to such objects as the *one-timers*. A good replacement policy should avoid putting the one-timers in the cache.

- *Object size* The size of the objects varies over the World Wide Web. Recent studies have shown that the size of a large number of web objects is fairly small. The size of the median object was found to be around 4KB in one of the web proxy workloads studied in [3]. This observation is likely to change drastically with an increased presence of a large number of audio and video objects on the web [12].

In this paper we consider a policy, known as the *2Q replacement algorithm*, that was originally proposed for the database environment [9]. We evaluate the performance of this simple and low overhead scheme ($O(1)$ complexity), through simulation with both synthetic and trace data. The simulation results (with 95% confidence interval) demonstrate significant performance gain over both LRU and LRU/2 scheme. The trace data used to evaluate the performance of the replacement algorithms was obtained from the NLANR web cache logs [10].

2 Prior Work

As discussed in the previous section, there are primarily three factors, *recency of reference, frequency of reference* and *size of the object* that should be taken into account while designing a replacement policy. In the last few years quite a few policies have been proposed where some or all of these three factors have been taken into account. A few examples of these policies is given below:

- *Least Recently Used* (LRU) replaces the least recently requested web object.
- *Least Recently Used/k* (LRU/k) [11] replaces the web object whose k-th most recent request is least recent.
- *2Q cache management technique* (2Q) [9] low overhead technique that simulates the LRU/k
- *Least Frequently Used* (LFU) replaces the least frequently requested web object.
- *Size* [14] replaces the largest size document in the cache.
- *Lowest Relative Value* (LRV) [13] computes the *utility* value of a web object in cache and then replaces the one with the smallest utility value.
- *Lowest Latency First* [15] is designed to minimize the average latency and as such removes the document with the lowest download latency.
- Greedy Dual (GD) [16] is designed to handle uniform size, variable cost cache objects.
- Greedy Dual Size (GDS) [6] An extension of the GD to handle variable size web objects.
- Greedy Dual Size Frequency (GDSF) [7] An extension of the (GDS) that takes into account the *popularity* of the web objects, measured in terms of frequency of request in recent past.

In [7] the author claims that GDSF improves on the GDS, regarded as the *"current champion"*, of the various cache replacement algorithms. Inspite of various claims of performance improvement

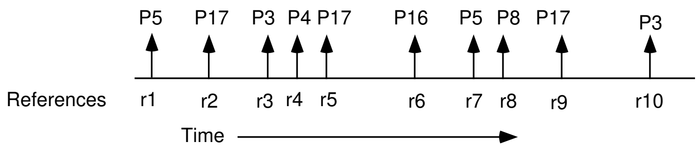

Figure 1: Access request reference string for the web pages

by these algorithms, arguably the simplest one, LRU, continues to be the most popular replacement algorithm in use. The computational complexity of LRU is $O(1)$. The more sophisticated algorithms, such as LRU/k, GDS and GDSF all maintain a *priority queue* of the objects in the cache to determine the next object to be removed from the cache. Because of the priority queue, the computational complexity of all such algorithms is $O(log\ N)$, where N is the number of web objects in the cache. In this paper we evaluate the performance of another $O(1)$ complexity algorithm, 2Q and compare its performance with LRU and LRU/k. In [11], the authors have reported that in the database environment LRU/k performed best when $k = 2$. The 2Q algorithm is designed to emulate the LRU/2 algorithm. In addition we develop a variation of the 2Q, called the 2Q/k algorithm and evaluate its performance.

3 LRU/k, 2Q and 2Q/2: Alogithms and Implementations

The LRU/k was proposed by O'Neil *et. al.* [11], to combine the *recency of reference* and *frequency of reference* into one single metric. The idea is to keep track of the access times of the last k references of the web objects. LRU/k gives priority to objects based on the k-th most recent access. When an object is to be evicted from the cache, in order to make room for another object, the object(s) deleted will be the one(s) with the maximum *backward k-distance.*

Backward k-distance: Given a reference string known upto time t, $r_1, \ldots, r_t$, the backward k-distance $b_t(p, k)$is the distance backward to the k-th most recent reference to the page p:

$$
b_t(p,k) = \begin{cases} x & \text{if } r_{t-x} \text{ has the value } p \text{ and there have been} \\ & \text{exactly k-1 other values } i \text{ with} \\ & t-x < i \leq t, \text{ where } r_i = p. \\ \\ \infty & \text{if } p \text{ does not appear at least} \\ & k \text{ times in } r_1, \ldots, r_t. \end{cases}
$$

In the example shown in figure 1, b_{10}(P17,3) = 8 and b_{10}(P5,2) = 9. O'Neil *et. al.* [11] studied the performance of the algorithm for different values of k. They reported that the best performance was obtained when k = 2. The LRU/2 algorithm replaces the page whose penultimate access is least recent among all penultimate accesses.

The 2Q algorithm was proposed by Johnson et. al. [9] for database buffer management. It is a low overhead, simple and easy to implement replacement policy. We have used a slight variation of the 2Q algorithm as a cache replacement policy for web objects. LRU works well because it tends to remove *cold* (not popular over time) objects from the cache to make room for the new object that is not in the cache. If the incoming object is cold, then LRU may displace a *warmer* object (more popular over time) to accommodate the new incoming object. The new cold object could reside for a long time in the cache. LRU/2 improves on LRU by using a stronger exit rule to remove cold objects from the cache. The operation of 2Q is as follows - the cache is partitioned into two parts, the main or hot cache and the intermediate cache. The size of the intermediate cache can be set to a tunable maximum threshold. Instead of cleaning cold objects from the buffer, 2Q only admits hot objects into the main cache. Similar to LRU/2, 2Q tests objects for the second reference time. On the first reference to an object, 2Q places the object in the intermediate cache (IC). The intermediate cache is maintained as a simple FIFO queue. If the object is re-referenced while it resides in the IC, the algorithm concludes that the object most likely is a popular object. It is then moved or upgraded to the hot or main cache (MC). The main cache is maintained as a LRU queue. If an object is not referenced while it is on the IC, 2Q removes it as it is most likely a cold object. An object that is removed from the MC is moved or down graded to the IC to give it a second chance. The reason we move it to the intermediate cache is that this object was popular earlier and the chances of it being accessed again are higher than that of a one-timer object.

```
begin
    if object (O) is on the MC queue then /* Cache Hit */
        put O on the back of the MC queue
        /* MC is managed as an LRU queue */
    else
        if O is on the IC queue then
            /* upgrade O to the Main Cache */
            remove O from the IC queue
            if there is space in the MC queue for O then
                put O on the back of the MC queue
            else delete object(s) from the front of MC
                put O on the back of the IC queue
                move the removed object(s) from the MC to the IC
                /* Give the removed object(s) a second chance */
            end if
        else /* first access we know concerning O */
            /* find free space for O */
            if there is free space available then
                put O on the back of the IC queue
            else if IC's size is above a (tunable) threshold
                delete object(s) from the front of IC queue
                put O on the back of the IC queue
                end if
            end if
        end if
    end if
end
```

It may be noted that in LRU/2, a priority queue has to be maintained to decide on the object to be removed from the cache. Maintaining such a priority queue requires $O(log\ N)$ amount of computation, where N is the number of objects in the cache. The complexity of the 2Q, on the other hand is $O(1)$. Thus from efficiency considerations, 2Q seems to be more attractive than LRU/2.

It may also be noted that LRU/2 specifies a policy to determine the *object to be removed* from the cache in case of contention. The 2Q algorithm on the other hand does not specify a policy to determine the *object to be removed*, but specifies a policy to determine the *object to be admitted* to the cache. In a sense, these to techniques are complementary to each other. Whereas the LRU/2 specifies a *removal policy*, the 2Q algorithm specifies an *admission* policy. We propose a new policy called 2Q/2 that uses the admission policy of 2Q and removal policy of LRU/2. Our experimental results show that 2Q/2 almost always outperforms LRU, LRU/2 and 2Q. The computational complexity of the 2Q/2 policy is same as the LRU/2.

4 Simulation Environment

The simulation package we developed is based on an earlier simulator developed by Arlitt and Williamson [1]. We modified the simulator so that we can plug in different replacement algorithm modules to record their performances.

Cold Misses: On running the simulator with initial requests, due to the absence of objects in the cache, a number of cache misses (called cold misses) are recorded. To overcome this phenomenon of cold misses, we run the simulator with the request logs and do not collect statistics until the cache is filled up. The period where no statistics are collected is the *warm-up* period. The simulator allows us the option to set a warm-up threshold, after which statistics will be collected. We used around 2% of the requests to warm up the cache before we started collecting the statistics.

Cache Sizes: The simulator provides the option to set different Cache sizes. We have considered six cache sizes for our study - 64 MB, 128 MB, 256 MB, 512 MB, 1024 MB and 2048 MB. The main statistics we collected for this study are (i) the cache hit rate and (ii) the byte hit rate.

Zipf-like Distribution

We considered the performance of the replacement algorithms with both synthetic and trace data. Initially, we studied the performance of the algorithms with synthetic data. This allowed us to study the effect of varying parameters on the performance of the scheme. Breslau et al in [5], report that the distribution of web objects follows a Zipf-like distribution with the exponent varying in different traces. The model we have considered is one where the web accesses are independent and the reference probability of the distribution follows a Zipf-like distribution. The relative probability of a request for the i-th most popular page is proportional to $1/i^{\alpha}$, with α taking on a value less than 1. The regular zipf distribution has $\alpha = 1$ but Zipf-like distribution has varying values of α. If there are n objects ordered such that object 1 is the most popular while object n being the least popular, then the probability of the j-th most popular object being requested is given by

$$p_j = K/j^{\alpha}, j = 1, \ldots, n$$

where

$$K = 1/(1 + 1/2^{\alpha} + \cdots + 1/n^{\alpha})$$

We have considered two values of α for our study: 0.86 and 0.7

The distribution of the file sizes considered are shown below in the tables.

File Size Distribution 1		File Size Distribution 2	
0-50 KB	60%	0-50KB	65%
50-100 KB	15%	50-100 KB	15%
100-500 KB	15%	100-500 KB	15%
500-5 MB	10%	500-5 MB	5%

5 Experimental Results

In this section we provide the results of the performance of the cache replacement policies with both synthetic and trace data.

The results of the replacement policies are shown in the tables below. We present four representative results:

- Cache Hit Rate with Synthetic Data
- Byte Hit Rate with Synthetic Data
- Cache Hit Rate with Trace Data
- Byte Hit Rate with Trace Data

In the following tables, we present the results of our simulation experiments with five sets of synthetic data and three sets of trace data.

For experiments with synthetic data we use the following parameters for all five data sets:

Number of different simulation runs = 50

For the first data set:
Number of requests = 1000000
Number of unique web objects = 500000
Zipf-like distribution parameter $\alpha = 0.86$

Cache Hit Rate with Synthetic Data Set 1				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64MB	8.67	17.49	18.48	21.1
128MB	12.61	21.53	22.65	24.61
256MB	17.23	25.84	27.33	29.56
512MB	22.6	30.78	33.07	35.9
1024MB	28.92	36.33	38.92	41.41
2048MB	36.64	42.69	45.29	46.69

Byte Hit Rate with Synthetic Data Set 1				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64 MB	1.18	2.43	3.66	4.49
128 MB	2.39	6.17	7.62	9.54
256 MB	4.81	11.58	14.04	16.72
512 MB	9.13	18.29	20.92	22.95
1024 MB	15.81	24.83	27.57	29.81
2048 MB	25.34	32.36	35.75	37.71

For the second data set:
Number of requests = 500000
Number of unique web objects = 10000
Zipf-like distribution parameter $\alpha = 0.8$

Cache Hit Rate with Synthetic Data Set 2				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64MB	9.27	16.44	17.02	20.01
128MB	13.38	22.12	22.78	25.87
256MB	19.69	28.25	29.37	32.45
512MB	27	35.31	37.16	40.37
1024MB	35.94	43.68	46.06	49.17
2048MB	47.12	53.58	55.83	58.51

Byte Hit Rate with Synthetic Data Set 2				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64 MB	3.26	6.1	7.66	9.38
128 MB	6.18	12.42	13.57	16.41
256 MB	11.09	19.51	21.04	24.23
512 MB	18.47	30.4	31.17	33.04
1024 MB	28.43	39.53	40.41	42.92
2048 MB	40.8	48.04	50.61	53.06

For the third data set:
Number of requests = 250000
Number of unique web objects = 10000
Zipf-like distribution parameter $\alpha = 0.76$

Cache Hit Rate with Synthetic Data Set 3				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64MB	10.66	18.85	19.14	21.78
128MB	15.79	23.09	25.48	28.6
256MB	22.65	30.84	32.87	36.62
512MB	31.07	38.96	41.35	44.55
1024MB	42.16	49	50.58	52.71
2048MB	56.47	59.095	59.59	60.98

Byte Hit Rate with Synthetic Data Set 3				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64 MB	7.24	12.23	14.33	16.33
128 MB	12.28	19.46	21.43	23.75
256 MB	18.99	26.96	29.12	31.6
512 MB	27.69	35.23	37.59	40.06
1024 MB	38.96	45.75	46.88	49.18
2048 MB	53.37	56.49	56.74	58.64

For the fourth data set
Number of requests = 500000
Number of unique web objects = 10000
Zipf-like distribution parameter $\alpha = 0.8$

Cache Hit Rate with Synthetic Data Set 4				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64MB	1.64	3.23	4.21	6.27
128MB	3.37	8.51	9.89	11.46
256MB	5.88	13.59	14.8	16.75
512MB	9.91	18.72	18.82	22.3
1024MB	14.65	24.27	24.87	27.53
2048MB	21.7	30.58	31.99	35.13

Byte Hit Rate with Synthetic Data Set 4				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64 MB	1.77	2.85	4.69	6.78
128 MB	3.74	9.67	9.93	13.1
256 MB	7.02	15.54	15.78	18.38
512 MB	11.66	20.14	20.33	23.63
1024 MB	17.27	25.79	26.34	29.66
2048 MB	23.53	31.93	33.27	36.51

For the fifth data set
Number of requests = 250000
Number of unique web objects = 5000
Zipf-like distribution parameter $\alpha = 0.86$

Cache Hit Rate with Synthetic Data Set 5				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64MB	15.99	23.2	23.59	27.68
128MB	22.87	31.26	31.52	35.7
256MB	31.13	39.3	40.32	45.23
512MB	40.27	48.35	49.56	53.05
1024MB	51.22	58.06	59.97	62.2
2048MB	64.7	69.26	70.2	71.56

Byte Hit Rate with Synthetic Data Set 5				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64 MB	11.89	16.96	18.13	19.07
128 MB	18.89	26.98	27.25	28.71
256 MB	27.36	36.92	37.32	38.56
512 MB	37.6	46.13	47.5	49.21
1024 MB	49.6	56.24	57.8	59.76
2048 MB	63.22	67.78	68.48	69.37

For experiments with web trace data we used the web log files maintained at the National Laboratory for Advanced Internet Research (NLANR) at the University of California at San Diego. From the total number of user requests in a day and the total number of unique pages requested, it is clear that a large number pages were requested only a few times.

For the first set of trace data:
Total number of user requests in a day - 340,982
Total number of unique pages requested- 230,091

Cache Hit Rate with Trace Data Set 1				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
4 MB	13.45	14.46	14.8	15.03
8 MB	16.23	17.04	17.51	18.1
16 MB	18.61	20.13	21.78	22.54
32 MB	20.8	22.21	23.02	23.4

Byte Hit Rate with Trace Data Set 1				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
4 MB	19.18	21.08	23.78	24.56
8 MB	22.46	26.47	27.46	28.33
16 MB	26.26	29.83	30.04	31.45
32 MB	30.18	31.06	32.01	33.2

For the second set of trace data:
Total number of user requests in a day - 513,220
Total number of unique pages requested- 189909

Cache Hit Rate with Trace Data Set 2				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64 MB	27.97	29.2	29.8	29.87
128 MB	31.02	32.44	32.89	33.26
256 MB	34.2	35.31	35.43	36.34
512 MB	35.9	36.19	37.26	38.19
1024 MB	37.36	38.5	38.63	39.9
2048 MB	38.9	39.7	39.79	40.62

Byte Hit Rate with Trace Data Set 2				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
64 MB	27.29	29.6	32.94	35.33
128 MB	36.3	38.65	40.7	42.31
256 MB	42.51	44.93	47.92	49
512 MB	48.23	50.06	51.39	52.78
1024 MB	52.26	53.11	53.86	54.7
2048 MB	54.26	54.44	54.46	56.31

For the third set of trace data:
Total number of user requests in a day - 206,045
Total number of unique pages requested- 105164

Cache Hit Rate with Trace Data Set 3				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
16 MB	17.09	18.8	19.91	21.18
32 MB	19.83	20.5	21.61	22.62
64 MB	22.65	23.01	23.42	24.08
128 MB	24.01	24.69	25.08	26.65

Byte Hit Rate with Trace Data Set 3				
	Caching Policy			
Cache Size	LRU	LRU/2	2Q	2Q/2
16 MB	10.45	11.1	12.19	13.52
32 MB	12.05	12.9	13.86	14.91
64 MB	14.97	15.31	15.79	16.49
128 MB	17.1	17.9	18.16	18.96

It may be observed from the results that both 2Q and 2Q/2 significantly outperforms both LRU and LRU-K, both in cache hit rate and also in byte hit rate, when the experiments are conducted with synthetic data. The 2Q/2 policy performs slightly better than 2Q. The performance of 2Q and 2Q/2 did not appear to be significantly better when the experiments were conducted with the trace data. The reason for this disparity is the following: the main reason for improved performance by 2Q ands 2Q/2 is that it discriminates against *one-timers*, i.e., the web objects that are requested only one time. In contrast, LRU and LRU/2 do not discriminate against one-timers' and on the first reference to an object, they place it in the cache without worrying if this object will be requested a second time in the future. In case of LRU and LRU/2, if a web object is requested only two times, at the end of the first request, it will enter the cache and at the second request, it will register a cache hit. In case of 2Q and 2Q/2, unless an object is requested at least three times, the algorithm will not register a cache hit. Only at the end of second request, the object moves into the main cache and only at the third request, the first cache hit is registered. In the three NLANR trace data sets we used for our simulation, the number of user requests were 340982, 513220 and 206045 respectively. The number of unique web objects (pages) in these three cases were 230091, 189909 and 105164 respectively. From these statistics, it is clear that a large number of the pages were requested very few times, many only 1 or 2 times. The pages requested only one time, do not register a cache hit in all the algorithms, LRU, LRU/k, 2Q and 2Q/2. However, if the objects are requested two times, they register a cache hit for LRU and LRU/2 but do not register a cache hit for 2Q and 2Q/2. For this reason the results presented for the LRU and LRU/k and 2Q are comparable for the trace data. This will not be the case when a large number of objects are requested many times, not just one or two times. This is confirmed by the results obtained with the synthetic data, where a large number of objects are requested many times. The results presented in the tables are statistically significant with 95% confidence interval. It may also be remembered that the advantage of 2Q over LRU/2 lies not only in the higher hit rates but also in the computational complexity. Whereas, the computational complexity of LRU/2 is $O(log\ N)$, the complexity of 2Q is $O(1)$. If one is not very concerned about the $O(log\ N)$ complexity, 2Q/2 policy, which demonstrates significantly higher cache hit and byte hit rates over both the LRU and the LRU/2 policies, can be used.

6 Conclusion

In this paper we have introduced a variation of the 2Q database management policy for web caches. The algorithm proposed is easy to implement and has a very low overhead. We have compared the results of the 2Q and 2Q/2 algorithms with LRU and LRU/2 using both synthetic and trace data. In both cases, 2Q and 2Q/2 outperforms both LRU and LRU/2 in terms of hit rate as well as the computational effort needed.

Acknowledgements

We would like to acknowledge the support of the IBM Corporation, Martin Arlitt, Martin Reisslein, Felix Haranto and NLANR for this research.

References

[1] M. Arlitt and C. Williamson, "Trace-Driven Simulation of Document Caching Strategies for Internet Web Servers", *SIMULATION: The Journal of the Society for Computer Simulation*, vol. 68, no. 1, pp. 23-33, January 1997.

[2] M. Arlitt and C. Williamson, "Internet Web Servers: Workload Characterization and Performance Implications", *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 631-645, October 1997.

[3] M. Arlitt, R. Friedrich, T. Jin, "Performance Evaluation of Web Proxy Cache Replacement Policies", HP Labs Technical Report HPL-98-97 May, 1998.

[4] M. Arlitt, L. Cherkasova, J.Dilley, R.Friedrich, T. Jin, "Evaluating Content Management Techniques for Web Proxy Caches", HP Labs Technical Report HPL-98-173, April 1999.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," *Proceedings of IEEE INFOCOM'99*, New York, March 1999.

[6] P. Cao, S. Irani: "Cost Aware WWW Proxy Caching Algorithms", Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS), Monterey, CA, pp.193-206 December 1997.

[7] L. Cherkasova, "Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy", Caches", HP Labs Technical Report HPL-98-69, November 1998.

[8] J. Dilley, M. Arlitt, "Improving Proxy Cache Performance - Analyzing Three Cache Replacement Policies", HP Labs Technical Report HPL-1999-142. October 1999.

[9] T. Johnson and D. Shasha, "2Q: A low Overhead High Performance Buffer Management Replacement Algorithm", *Proc. of the 20th VLDB conference*, Santiago, Chile, pp. 439-450, 1994.

[10] National Laboratory of Applied Research. Sanitized access log. ftp://ircache.nlanr.net/Traces

[11] E. J O'Neil, P. E. O'Neil and G. Weikum, "The lru-k page replacement algorithm for database disk buffering",. *Proceedings of the 1993 ACM Sigmod International Conference on Management of Data*, pp.297-305, 1993.

[12] M. Reisslein , F. Hartanto , and K. W. Ross, "Interactive Video Streaming with Proxy Servers", *Proceedings of First International Workshop on Intelligent Multimedia Computing and Networking (IMMCN)*, pp. II-588-591, Atlantic City, NJ, February 2000.

[13] L. Rizzo, L. Vicisano, "Replacement Policies for a Proxy Cache", IEEE/ACM Transactions on Networking, Vol 8, No 2, April 2000.

[14] S. Williams, M. Abrams, C. Stanbridge, G. Abdulla, E. Fox: "Removal Policies in Network Caches for World-Wide Web Documents", In Proceedings of the ACM Sigcomm96, August, 1996, Stanford University.

[15] R. Wooster, M. Abrams: "Proxy Caching the estimated Page Load Delays", In Proceedings of the 6th International World Wide Web Conference, 1997.

[16] N. Young: "Online caching as the Cache size varies". In the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms", 241-250,1991