A Parameterized Hybrid Size-LFU Replacement Algorithm for Web Caches

Hwangcheng Wang, Junzhi Peng, and Yuting Wu
Department of Electronic and Information Engineering
National Ilan Institute of Technology
1, Section 1, Shen-Nong Road, Ilan 260 Taiwan
{hcwang|u8990053|u8990032}@ilantech.edu.tw

Abstract

Web caches serve two important purposes: to improve user-perceived response and to reduce network traffic. How well these goals are achieved depends on the caching algorithm employed and cache replacement scheme used. Substantial efforts have been made to devise suitable methods. Most existing replacement algorithms consider recency of object preferences, frequency of object references, or the fetching cost of objects. We propose a parameterized approach that combines reference pattern and object size in the design. Properties of the algorithm are analyzed. The algorithm is applied to a variety of workloads and its performance is compared against several well known methods. Preliminary results indicate that the algorithm is a viable choice.

Keywords: Web cache, object replacement algorithm, LRU, LFU, Size, SzLFU, hit ratio

1. Introduction

Network has become an integral part of everyday life. The rich information accumulated in the long past has been converted into a form suitable for access and display via the internet. Modern search engines boast of being able to sieve through billions of pages for information of interest to the user [6] [12]. The tremendous growth of information has also clogged the network, creating serious congestion on the network. Most users have experienced long waits for information to arrive. Besides the effort to continually install faster links and equipments on the network, an alternative approach is to deploy web cache proxies on the server side or client side.

Cache server holds frequently requested items in its storage system consisting of main memory and hard disks. Borrowing the idea from traditional computer caches, web caches try to exploit the regularity in object access patterns. By keeping objects that are likely to be requested over and over again, it's hoped that network congestion will be ameliorated, if not eliminated.

Evidence suggests that the goal has largely been achieved. If an item requested is already in cache (i.e. a cache hit occurs), the item can be delivered to the user from the cache directly instead of retrieving it from the originating web servers.

Admittedly, storage technology has advanced at a fast pace. Today a regular desktop personal computer is invariably equipped with hard disks of capacity on the order of multi-giga bytes, and web cache servers may have several disks attached to it. Yet, given the amount of data available on the network and the proliferation of multimedia data, eventually the storage space on a web cache will be filled up. At that point, a replacement algorithm will have to be invoked in order to allow newly requested items to be brought into the cache. Consequently, the effectiveness of the replacement algorithm may significantly affect performance of the cache server.

Much research has gone into the design and analysis of cache replacement algorithms. Some of the methods are extensions from those used in traditional cache systems. For instance, LRU and LFU are two widely used methods that predate the web age, and have been adapted for use in web cache environments [9] [11]. The popular Squid web cache server adopts a proactive LRU approach. It employs dynamically adjusted low and high water marks as thresholds to decide on the aggressiveness of invoking the replacement algorithm [15]. Nonuniformity in object size adds another dimension to the web cache and complicates the design issues of replacement algorithm.

The size factor has been considered alone [16] or been added to the cost model of quite a few replacement algorithms in different ways [1] [3] [8] [10] [13] [14] [18]. Following the common convention, we will refer to the scheme based solely on size as Size algorithm. A number of comprehensive surveys on these methods are available. In particular, a unifying framework is put forth in [2] within which various methods can be reformulated and compared. The article also discusses implementation issues, such as space and time complexities of the algorithms. A rigorous treatment of the regularity often found in the overall access pattern and the access to individual web pages/objects is given in [4]. Based on the observation, near optimal algorithms are proposed and examined. This regularity property also allows the use of partial traces in the study of web cache

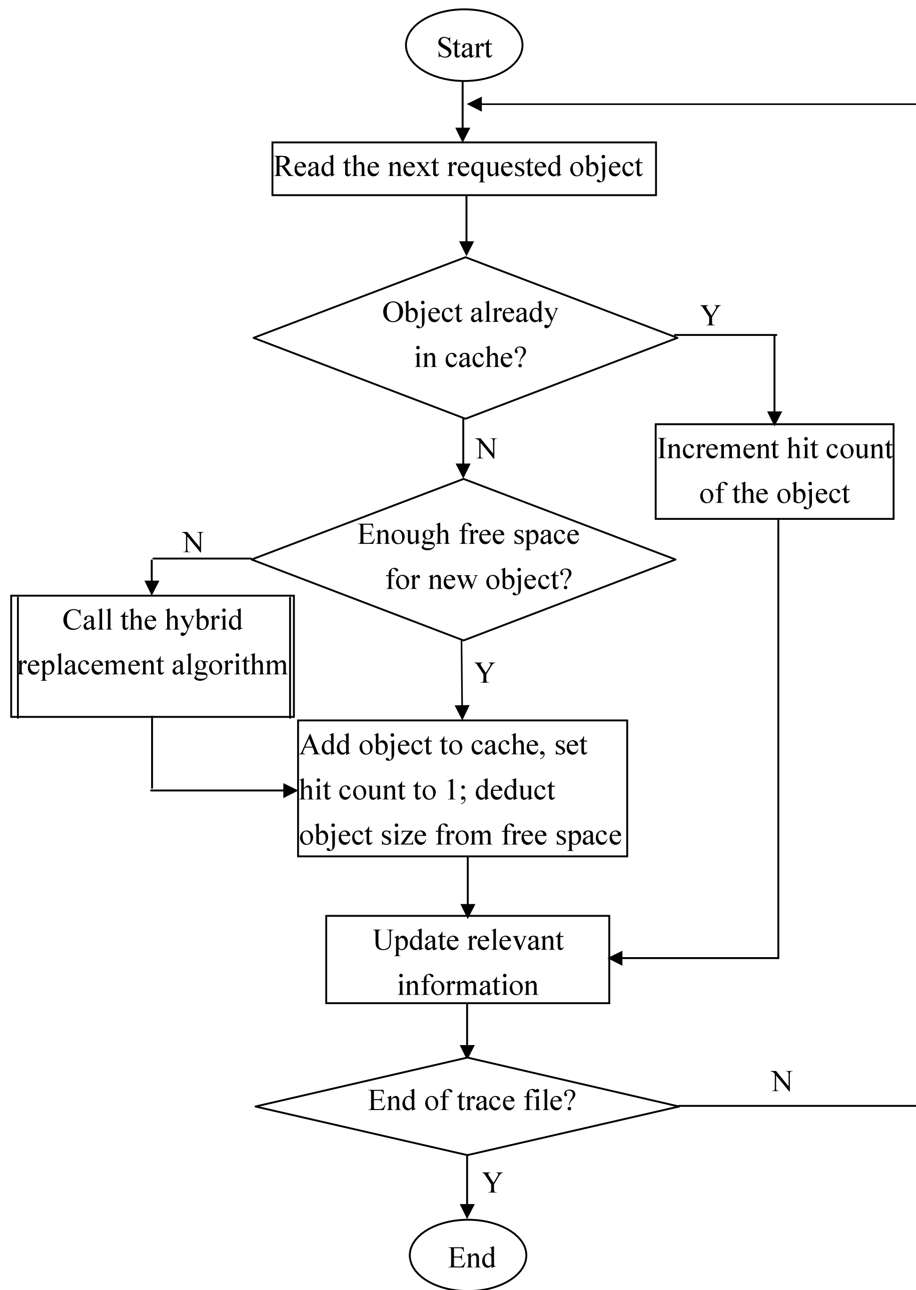
algorithms. A more theoretical study is also presented in [7], where it is shown that finding an optimal web cache replacement algorithm is essentially an NP-complete problem.

In this paper, we propose an algorithm that is based on a combination of object size and usage statistics. A distinctive feature of the method is that it incorporates a parameter which can be adjusted to reflect the relative weightings of the two factors. We have applied the method to a variety of workloads and find the method to show good performance in most cases under consideration. We analyze the properties of the algorithm and make observations on its behavior From the preliminary results, we believe that the method is a promising scheme that's worth further exploration.

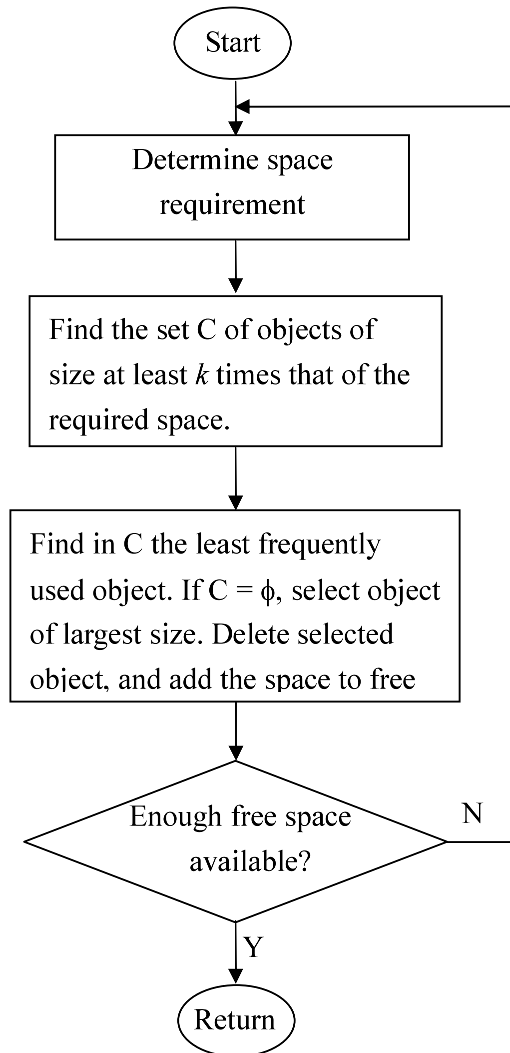
The rest of the paper is organized as follows. In Section 2, the algorithm is described with an example to illustrate the idea. Section 3 compares the method with several others and discusses their relative merits and weaknesses. The algorithm is put to test under a number of different conditions. Section 4 describes the workloads used in the study and experimental setup. Section 5 presents experimental results and analyzes the performance outcome. Concluding remarks and future research are given in Section 6.

2. **The SzLFU Algorithm**

The working of our algorithm, called hybrid Size-LFU and abbreviated as SzLFU, is depicted in Figure 1. Part (a) of the figure shows the overall structure of the program, whereas the replacement program per se is shown in part (b). The amount of available free space, denoted f, is tracked and updated as objects are added to or removed from the cache. For each object in the cache, the reference count and size data are maintained. Additional information associated with each object may include time of last reference and other useful information.



(a) Overall algorithm



(b) Replacement algorithm

Figure 1. The SzFLU object replacement algorithm

When a new object of size *s* is requested, the algorithm first checks if the object is already in cache. If so, the reference count of the object is increased by 1, and the algorithm proceeds to process the next request. If not, the required space of the object is compared with the free space. If the free space is large enough to accommodate the new object, it is brought into the cache with reference count set to 1. Free space and other information are updated as needed.

If the current free space is inadequate for the new object, the replacement mechanism is activated. First the space deficiency is calculated as the difference between the space needed and available free space. The difference is then multiplied by a parameter *k*, with the resulting number

used to select objects resident in the cache. All objects of size at least $k\times(s\text{-}f)$ are put in a set C. If no such object exists (i.e., C is empty), the object of largest size is removed, in a way similar to Size algorithm. Otherwise, members of C are scanned, and the object with lowest reference count will be chosen for eviction. When there is a tie (i.e. two or more objects in C have the same reference count), the one of largest size is selected so that more free space can be released. If the tie still cannot be resolved in this manner, other criteria such as last reference time may be used as well. Alternatively, one of the objects can be selected at random, or all such objects are removed. The selected object or objects are removed with occupied space reclaimed.

After this step, if enough space is created, the new object is brought into the cache, with all the relevant information properly updated. On the other hand, if the free space is still not large enough, another iteration of the replacement algorithm will be invoked. This process is repeated until eventually enough space is on hand to take in the requested object. (Here we make the implicit assumption that the size of any object is smaller than the cache size.) After that, the new object is inserted in the cache with necessary information updates.

To illustrate the idea underlying the SzLFU algorithm, let us consider an example. The following shows the snapshot of a web cache at a particular moment; the size and request profiles of each resident object are indicated. For convenience, the objects are randomly assigned a letter for reference.

object	a	b	c	d	e	f	g
size	12	9	7	10	8	6	4
request frequency	3	2	1	4	3	2	1

Assume the total capacity of cache is 64, and therefore the free space at the time is 8. Suppose a new object h of size 24 is requested, creating a shortage of 16 units in free space and necessitating the execution of the replacement algorithm. We consider three different cases with distinct values of k.

Case I: $k = 0.8$

First, $16 \times 0.8 = 12.8$ is larger than any object in cache, therefore the largest object a is selected. After its deletion, the free space is increased to 20, still not enough for the new object. Again, the space deficiency of 4 is multiplied by 0.8 and used as the criterion to select objects. All the objects currently in cache meet the size criterion. Objects c and g are candidates for removal because of low demand. Among the two, c is chosen since it has a larger size. Now there is sufficient space and we end up with the following cache content and a free space of size 3.

object	b	d	e	f	g	h
size	9	10	8	6	4	24
request frequency	2	4	3	2	1	1

Case II: $k = 0.5$

In this case, objects of size at least $16 \times 0.5 = 8$ are considered at first, out of which object b is selected because it has lowest usage. When its size is added to free space, the deficiency reduces to 7. In the second round, objects larger than 3.5 in size are plucked. All the objects satisfy the condition. Thus, least demanded items, objects c and g are potential candidates. Eventually c is removed for reasons discussed in Case I and the resulting cache content is shown below with no free space left.

object	a	d	e	f	g	h
size	12	10	8	6	4	24
request frequency	3	4	3	2	1	1

Case III: $k = 0.2$

We proceed as before. In the first iteration, all the objects are eligible for removal since they are all of size greater than $16 \times 0.2 = 3.2$. Therefore, object c and then object g are removed, making the total free space equal to 5. In the final iteration, object b is selected, whose removal will

leave a free space of 4 after the new object is brought into cache. The following cache configuration is obtained:

object	a	d	e	f	h
size	12	10	8	6	24
request frequency	3	4	3	2	1

As a comparison, Size algorithm will remove objects a and d; LFU may remove c, g, b or c, g, f, depending on the tie-breaking rule employed, relative order of references, and other factors.

3. **Comparison with Other Methods**

LRU removes in turn the object which has not been used for the longest time until enough space is created. As a result, the objects remaining in cache tend to be those in high demand. Therefore, the hit ratio is generally high. LFU evicts objects whose reference frequency is lowest. This process is repeated until a newly requested object can be accommodated. After rarely used items are removed, the rest tend to be items in heavy demand, leading to high hit ratio.

Size algorithm selects objects of largest size, then second largest size, and so on, for removal, until enough space is obtained. The policy increases the amount of space and in general takes fewer deletions to produce the required space. Often the extra space thus obtained can hold additional items, which can boost hit ratio.

The hybrid method searches only a fraction of the cached objects of size at least k times as large as the required space and then applies a method similar to LFU to choose objects in this set for removal. In doing so, the number of operations is effectively reduced while preserving the high hit ratio property of LFU method.

However, each of the methods may have its shortcomings. LRU is simple and fast, but may get rid of objects that are still in high demand, simply because the inter-reference interval is long. Overall, the hit ratio can be lower than LFU in many cases. LFU generally exhibits high hit ratio

because of the way it retains objects in cache. But the need to sort objects after each access makes the bookkeeping overhead high. Indeed it has been shown that LFU has a higher implementation complexity than LRU [9].

Size algorithm favors the retention of small objects. In the long run, most objects are relatively small in size. Objects that are still being requested can be victimized just because of their sizes. Eventually, the cache may be filled with small yet seldom requested objects. If this happens, the performance of cache will be hampered.

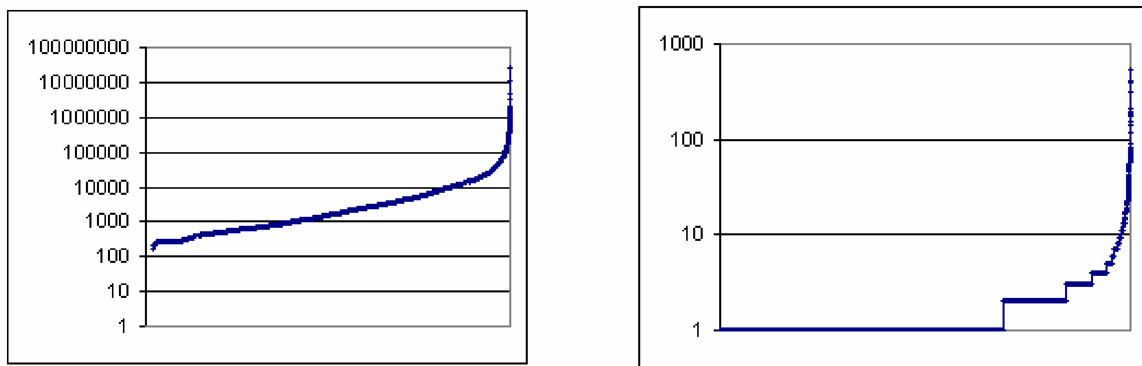
SzLFU skips items below a threshold determined by the incoming object and currently available space. This accelerates the selection process relative to LFU. Also, objects removed tend to be larger than those purged by LFU. In other words, number of items resident in cache will be larger than LFU on the average. This is likely to improve hit ratio compared to LFU. Furthermore, objects retained in cache may be just slightly smaller compared to the case when LFU is used. Hence with a higher bit ratio, we expect the byte hit ratio performance to be comparable with LFU in most cases.

Note that k plays an important role in deciding the behavior of the SzLFU algorithm. A small k makes our algorithm behave more like LFU algorithm. For a larger k, the algorithm inclines toward Size algorithm, as size effect becomes more prominent. In the case $k = 0$, the SzLFU algorithm behaves much like the pure LFU algorithm in principle. But on close examination, they may be different in practice when implementation details are considered. For example, if LFU uses time of reference as the tie breaker and SzLFU uses object size, as is often the case, the two will show difference in actual behavior.

4. **Experimental Study and Workload Characteristics**

To gain a deeper understanding of the SzLFU algorithm, we apply it to a number of trace files and collect performance data. For comparison, we also implement LRU, LFU, and Size algorithm and test them on the same set of traces. In this section, we describe the traces used and present experimental results in the next section.

For completeness, five different workloads are used, referred to as W1 through W5 hereafter. W1 is extracted from the access records of the web proxy server operated by the Computer Center of Ilan Institute of Technology. It exhibits Pareto or power-law property in object size and usage distributions typically found in traces obtained in a similar fashion [5] [17]. Most accesses are concentrated on a small number of web objects, and objects of low reference counts (many as low as 1) account for a great portion of the access records. Figure 2 shows the size and access frequency distributions of the workload.



(a) Size distribution

(b) Access frequency distribution

Figure 2. Object size and access frequency distributions of workload W1

W2 has uniform distributions in access frequency and object size. Objects are accessed more or less evenly and they vary from very small to very large in size. Such a size distribution is somewhat similar to that seen on the web. But access distribution lacks strong clustering; this may be similar to the situation where students access their individual grades in different classes.

In W3, the distribution of both object size and access frequency is nonuniform. In W4, object size is distributed uniformly over a wide range, but access frequency distribution shows a bias similar to that in W1. W5 is the inverse of W4, with uniform distribution in access frequency, but nonuniform distribution in object size.

Table 1 summarizes the main size and frequency statistical information of the workloads. As can be seen, they have quite different characteristics. The label “unique object size” indicates the

total size of objects if each object is accessed exactly once. This parameter is important since a cache of capacity larger than it can achieve a hit ratio of 100%. In contrast, “cumulative object size” represents the total size of objects ever requested, taking into account individual reference counts. On the last row of the table is shown the size of caches simulated.

Table 1. Characteristics of the workloads used in the experiments

Workload	W1	W2	W3	W4	W5
Number of objects	24819	10000	10000	10000	10000
Unique object size	287771238	150709567	4282743	150371531	4204735
Average object size	11558	15071	428.27	15037	420.47
Standard deviation of object sizes	195145	4994	1182.70	6183	1571.0
Cumulative object size	443873829	1506245018	41045478	1530323406	43055560
Mean accesses per object	2.175	10	10	10	10
Standard deviation of accesses	7.396	1.833	107.11	107.066	1.810
Simulated cache size	100MB	100MB	1MB	100MB	1MB

5. **Performance Results and Analysis**

The performance results are summarized in Table 2. For each of the methods considered, hit ratio and byte hit ratio are shown. In the case of the SzLFU algorithm, performance data are given for different values of k.

Table 2. Performance results of various algorithms for different workloads

workload	metrics	LFU	LRU	Size	SzLFU				
					k=0.1	*k*=0.2	*k*=0.5	*k*=0.7	*k*=1
W1	HR	53	52	53	NA	53	53	NA	NA
	BHR	32.4	34.6	32.2	NA	32.3	32.5	NA	NA
W2	HR	8.4	6.5	24.1	NA	8.5	10.7	17.5	21.7
	BHR	6.7	6.5	6.6	NA	6.7	6.7	6.6	6.6
W3	HR	71	64	78.4	80	80.1	79.8	79	NA
	BHR	65.9	62.3	39.4	55.3	52.2	45.4	42.6	NA
W4	HR	34	21	10.7	NA	34	34.7	NA	24.3
	BHR	34.8	22.3	2.5	NA	34.8	34.8	NA	12.8
W5	HR	34	22.7	67.7	64	66	67	NA	68
	BHR	22.6	23.6	22.5	23.0	22.3	22.6	NA	22.2

Note: HR = hit rate; BHR = byte hit rate; NA = result has not been collected for the value of *k*.

LRU and LFU schemes are clearly affected by the actual object reference pattern. In cases where the references are uniformly distributed and hence lack locality property, the two methods exhibit relatively poor performance. If the references to individual objects vary widely (some objects are referenced much more frequently than others), the performance of the two methods are more favorable.

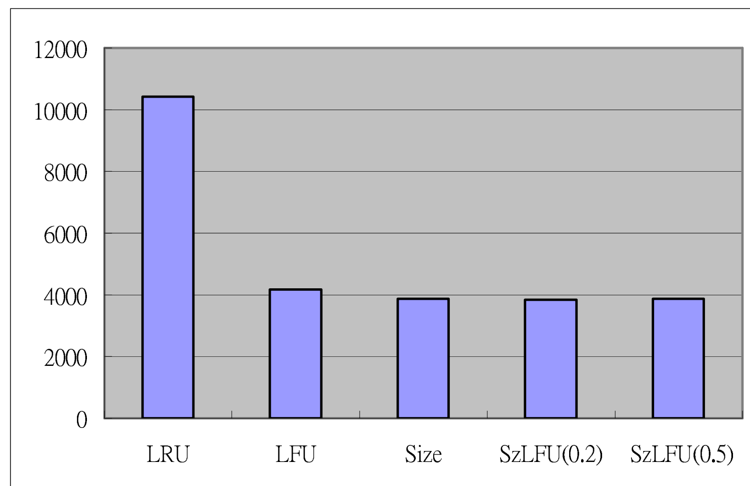
Size algorithm has difficulty in handling W5, where the reference frequencies show great fluctuations but object size is uniformly distributed. The low performance can be attributed to the fact that small-sized objects do not necessarily have strong demand. For the other cases, because cache retains only smaller objects, the byte hit rate suffers, even though hit ratio is not low.

SzLFU takes into account both usage pattern and object sizes. As such, it can adapt to the inherent properties of the traces by adjusting the parameter *k*. From performance data shown in the

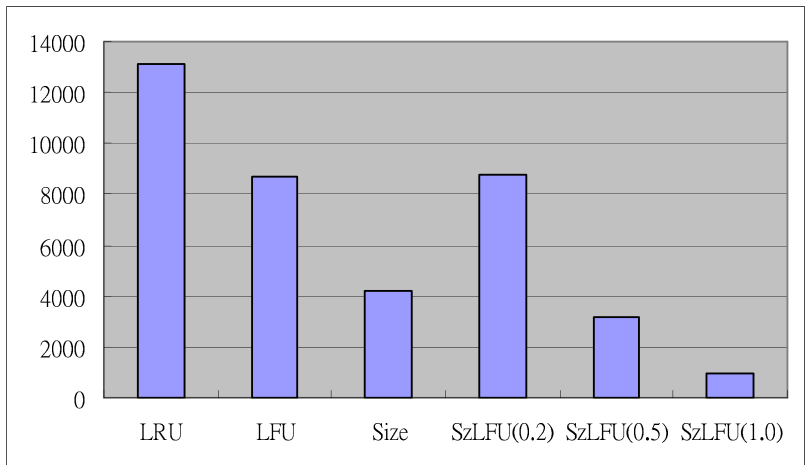
table, we see that it manages to maintain a satisfactory performance level judged by either hit ratio or byte hit ratio. In fact, it's the best performer in many cases with a proper choice of *k*. While other methods might fluctuate for different workloads, SzFLU is more consistent in performance.

The value of *k* also affects the hit ratio and byte hit ratio, but in different ways for some workloads. For instance, in the case of W2, hit ratio favors a larger *k*, but byte hit ratio favors a smaller *k*. In the case of W3, hit ratio does not show a clear relation with the value of *k*, but byte hit ratio is apparently higher for smaller values of *k*. This is consistent with our previous remark that larger values of *k* makes the algorithm more like Size algorithm and the latter shows poor byte hit ratio performance for W3.

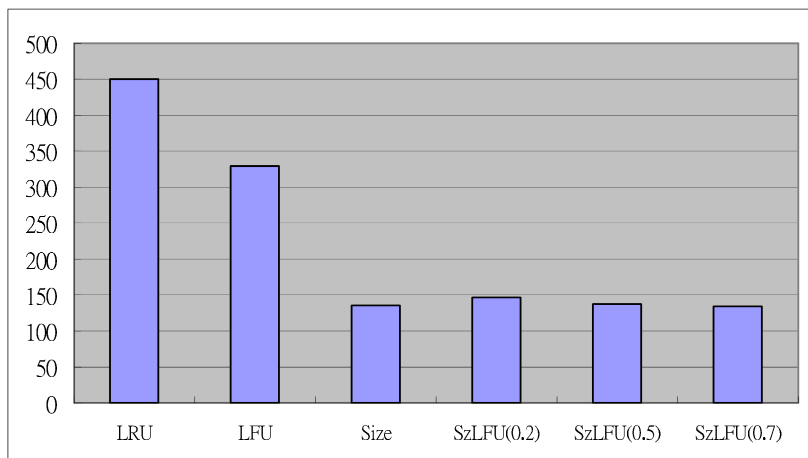
We can make further observations on other aspects of the algorithms. Figure 3 shows the average size of objects remaining in the cache at the end of simulation runs for the workloads examined. Consider the SzLFU algorithm. With large *k* values, the average size of objects remaining in cache tends to be small, due to the fact that larger objects are more likely to be selected for replacement, whereas smaller objects are exempted. As the value of *k* is lowered, smaller objects also become susceptible to being selected, and objects of larger size are able to survive, increasing the average size of objects remaining in cache. The diagrams in Figure 3 clearly illustrate the point.



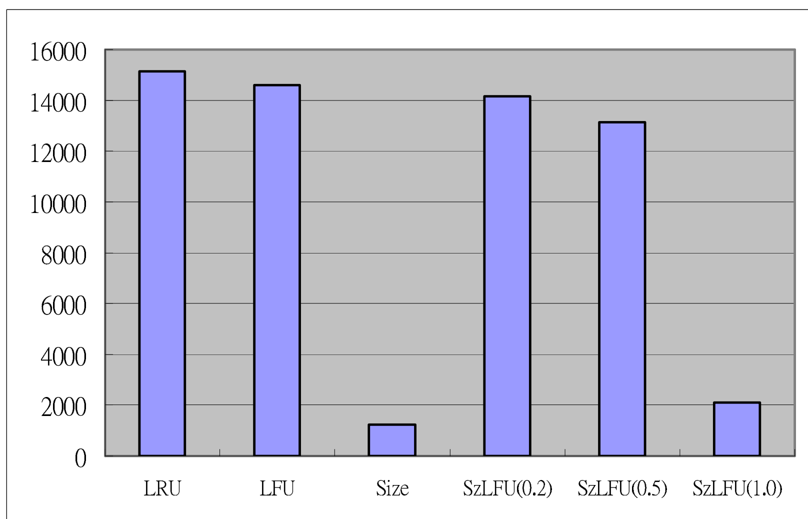
(a) Mean size of in-cache objects for W1



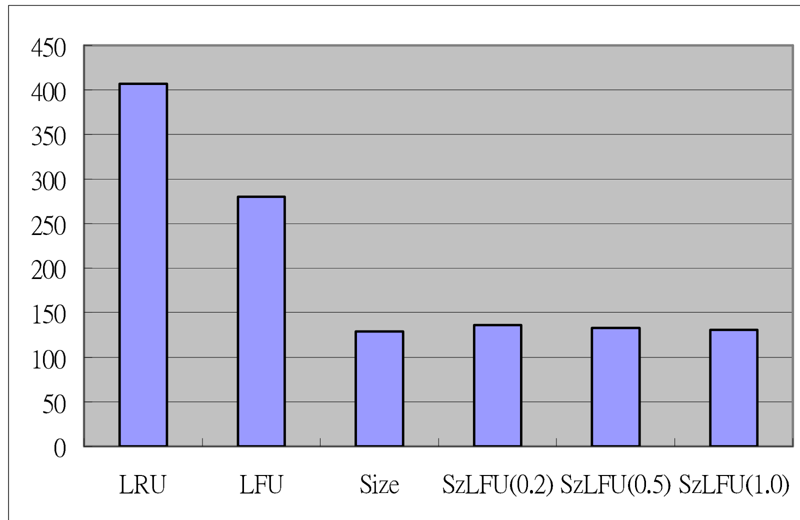
(b) Mean size of in-cache objects for W2



(c) Mean size of in-cache objects for W3



(d) Mean size of in-cache objects for W4



(e) Mean size of in-cache objects for W5

Figure 3. Average in-cache object size at the end of simulation for different workloads

Of the methods examined, LRU retains, on the average, largest objects in the cache, while Size algorithm keeps mostly small-sized objects in the cache. For the SzLFU algorithm, the average size of objects remaining in cache at the end of simulation lies between that of LRU/LFU and Size algorithms. This is not surprising since SzLFU possesses the properties of both LFU and size algorithms.

6. **Summary and Conclusions**

In this paper, we propose a hybrid replacement scheme for use in web cache systems. The properties of the algorithm have been carefully examined and compared with several established algorithms. We have also tested the algorithm on a number of workloads with different characteristics and analyzed the performance data. Compared to the other methods, the proposed algorithm shows sound performance in most cases. The parameter *k* also gives it additional flexibility for tuning. Preliminary results suggest that it can be a viable choice in the selection of replacement algorithms.

Current research can be expanded in several ways in the future. More thorough study is needed to better appreciate the algorithm. This may include a more systematic way to determine the

effect different values of k have on the performance with respect to different workloads. In our experiments, we have found that large values of k can actually degrade rather than improve the performance. Although the use of partial traces is justified in some cases [4], test of the algorithm on larger trace files will be helpful in assessing its performance in a more realistic setting. Finally, as more schemes are being proposed and studied, it would be interesting to compare the performance of the SzLFU algorithm with other, more recent methods.

References

1. C. Aggarwal, J. Wolf, and P. Yu, "Caching on the world wide web," IEEE Trans. Knowledge and Data Eng., vol. 11, no. 1, pp. 94-107, 1999.
2. H. Bahn, K. Koh, S. H. Noh, and S. L. Min, "Efficient replacement of nonuniform objects in web caches," IEEE Computer, vol.35, no. 6, pp. 65-73, June 2002.
3. P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," Proc. Usenix Symposium on Internet Technology and Systems (USITS 97), Usenix, Berkeley, pp. 193-206, 1997.
4. E. Cohen and H. Kaplan, "Exploiting regularities in web traffic patterns for cache replacement," Proc. The 31st Annual ACM Symposium on Theory of Computing, pp. 109-118. 1999.
5. M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," Proc. SIGCOMM 99, pp. 251-262, 1999.
6. Google search engine, http://www.google.com.
7. S. Hosseini-Khayat, "On optimal replacement of nonuniform cache object," IEEE Trans. Computers, Vol. 49, no. 8, pp. 769 –778, Aug. 2000.
8. T.P. Kelly, Y.M. Chan, S. Jamin, and J. K. MacKie-Mason, "Biased replacement policies for web caches: differential quality-of-service and aggregate user value," Proc. 4th Int'l Web Caching Workshop (WCW 99), http://workshop99/ircache.net/Papers/kelly-final.ps, 1999.
9. D. Lee, J. Choi, J.H. Kim, S.H. Noh, S.L. Min, Y. Cho, and C.S. Kim, "LRFU: a spectrum

of policies that subsumes the least recently used and least frequently used policies," IEEE Trans. Computers, vol. 50, no. 12, pp. 1352-1361, 2001.

10. N. Niclausse, Z. Liu, and P. Nain, "A new and efficient caching policy for the world wide web," Proc. Workshop Internet Server Performance (WISP 98), http://www-sop.inria.fr/mistral/ personnel /Nicolas.Niclausse/articles/wisp98/,1998.
11. E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-k page replacement algorithm for database disk buffering," Proc. 1993 ACM Sigmod Int'l Conf. Management of Data, ACM Press, pp. 297-306, 1993.
12. Openfind search engine, http://www.openfind.com.tw.
13. L. Rizzo and L. Vicisano, "Replacement policies for a proxy cache," IEEE/ACM Trans. Networking, vol. 8, no. 2, pp. 158-170, 2000.
14. J. Shim, P. Scheuermann, and R. Vingralek, "Proxy cache design: algorithms, implementation, and performance," IEEE Trans. Knowledge and Data Eng., vol. 11, no. 4, pp. 549-562, 1999.
15. Squid Web Proxy Cache, http://www.squid-cache.org/Doc/FAQ/FAQ-12.html#ss12.25.
16. S. Willams, M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, "Removal policies in network caches for world-wide web documents," Proc. 1996 ACM Sigcomm Conf., ACM Press, New York, pp. 293-305, 1996.
17. W. Willinger, V. Paxson, M.S. Taqqu, "Self-similarity and heavy tails: structural modeling of network traffic, a practical guide to heavy tails: statistical techniques and applications," Birkhauser Boston Inc., 1998.
18. R.P. Wooster and M. Abrams, "Proxy caching that estimates page load delays," Computer Networks and ISDN Systems, vol. 29, nos. 8-13, pp. 977-986, 1997.