IEICE TRANS. INF. & SYST., VOL.E92–D, NO.4 APRIL 2009

LETTER

An Effective Self-Adaptive Admission Control Algorithm for Large Web Caches

Chul-Woong YANG[†], ***Member***, **Ki Yong LEE**[†a)], ***Nonmember***, **Yon Dohn CHUNG**[††], ***Member***, **Myoung Ho KIM**[†], ***and*** **Yoon-Joon LEE**[†], ***Nonmembers***

SUMMARY In this paper, we propose an effective Web cache admission control algorithm. By selectively admitting objects into the cache, the proposed algorithm can significantly reduce the amount of disk I/O on a Web cache while maintaining a high hit ratio. The proposed algorithm adaptively adjusts its own admission control parameter, requiring no user-supplied parameters. Through extensive experiments, we show the effectiveness of the proposed algorithm.

key words: *Web cache, cache admission control, WWW*

1. Introduction

Over the past few years, most studies for Web cache management have focused on Web cache replacement algorithms, such as LRU, LFU, LRU-K, and GD-SIZE that determine which objects in the Web cache should be replaced when a new object is brought in [1]. However, most of them have not considered the issue of *admission control*, which determines whether a new object should be cached or not. It may not always be profitable to insert an object into the cache, because objects with low access frequency degrade the cache performance.

With the recent advent of user-created contents (UCC), Web object size is rapidly increasing [2]. If a Web cache admits *every* object merely because it is accessed once, the cache may spend a lot of time in writing large objects that would be rarely referenced in the future. This can impose heavy disk I/O overhead on the cache server while reducing cache hit ratio.

We propose an effective admission control algorithm for large Web caches, called *Adaptive Frequency-based Admission Control (AFAC)*. When the Web object not in the cache is requested, admission control determines whether the object should get into the cache or not. In our algorithm, we use two criteria for admission control, i.e., the re-reference likelihood and the size of the object. We show through various experiments that, by selectively admitting objects based on these two criteria, the amount of disk I/O can be significantly reduced while a high hit ratio is maintained.

2Q [3] is the most representative caching algorithm that utilizes the concept of admission control. When an object is first referenced, 2Q inserts the identifier of the object into an auxiliary FIFO queue called `A1`, instead of inserting the object into the cache. If the object is re-referenced while its identifier is in `A1`, it is considered as a *hot*, i.e., frequently requested, object and inserted into the cache. If an object is not re-referenced while its identifier is in `A1`, it is not inserted into the cache. Although this admission control is simple, it has been shown to provide a higher hit ratio than performing no admission control at all.

However, the admission control in 2Q has two limitations. First, the size of `A1` is difficult to tune. If the size of `A1` is too large, many *cold*, i.e., infrequently requested, objects may have high chances to be admitted into the cache. If the size of `A1` is too small, even hot objects may not be admitted into the cache, which results in a low utilization of the cache. Second, 2Q does not consider the non-uniformity of object sizes. Admitting one very large object into the cache may cause dozens of objects to be evicted from the cache. It also incurs a large amount of disk I/O. Thus, large objects should be admitted more carefully.

Note that cache admission control and cache replacement are orthogonal concepts. In other words, our proposed cache admission control algorithm will be used in combination with cache replacement algorithms such as LRU, LFU, LRU-K, and GD-SIZE, rather than substituting for those replacement algorithms.

2. The Proposed Method

Figure 1 shows the proposed admission control architecture. In the figure the FIFO queue F that resides in main memory plays a central role in our proposed admission control method. Let N be the size of F, and $F(n)$ denote a set of last n elements in F ($1 \leq n \leq N$). $F(n)$ is called the *selection window* of F. Let d_i and s_i be the identifier and the size

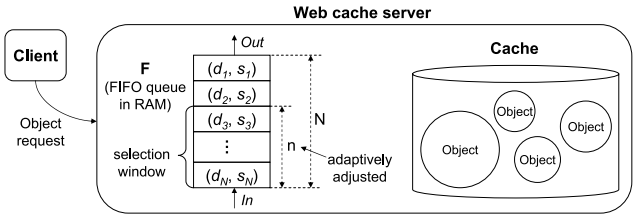

Fig. 1 The proposed cache admission control architecture.

Manuscript received September 30, 2008.
Manuscript revised November 20, 2008.

[†]The authors are with the Department of Computer Science, KAIST, Daejeon, Korea.
[††]The author is with the Department of Computer Science and Engineering, Korea University, Seoul, Korea.
a) E-mail: kylee@dbserver.kaist.ac.kr (Corresponding author)

DOI: 10.1587/transinf.E92.D.732

of object i, respectively. When an object i is first referenced, we insert the pair (d_i, s_i) into F only, rather than immediately inserting the object to the cache. If an object is re-referenced while in the queue, we will give it a high chance to be admitted into the cache. Otherwise, it will go out of the queue without having a chance to get into the cache. Note that the amount of memory for F is very small in practice because F stores only the identifier and size of objects, rather than the objects themselves.

The FIFO queue F has something in common with `A1` queue in [3], but has two fundamental differences. First, only $F(n)$, i.e. the last n elements in F, rather than all the elements in F, are used for admission control. The value of n will be dynamically adjusted. Second, when an object in $F(n)$ is re-referenced, we will go through another admission test, instead of simply allowing the object to get into the cache as in [3]. This test is based on the relative sizes of objects in $F(n)$, so that a smaller object will have a higher admission probability than a larger object.

2.1 Adaptive Adjustment of the Selection Window

Consider the value of n in $F(n)$. If n is too large, many cold objects could be inserted into the cache, which will degrade the cache performance. If n is too small, the cache may not properly accommodate even hot objects. Since no fixed value of n can be good for all cases, the value of n needs to be dynamically determined, depending on the situation. The value of n can be decided based on how frequent the objects in $F(n)$ have been admitted into the cache. Though there can be various ways to implement this idea, we will use a simple strategy as follows:

For n number of object requests (regardless of those objects being in $F(n)$ or not), we count the number of objects that are admitted into the cache. If more than one object were admitted into the cache, we decrease n by rate β to make the admission control more restrictive. Here, β, called the window adjusting factor, is a predetermined real number between 0 and 1. If no object was admitted, we increase n by rate β to make the admission control less restrictive. This process is repeated. Experimental results in the next section show that this simple strategy can successfully maintain the effectiveness of the admission control.

2.2 Size-Based Additional Admission Test

Now we describe how the size of objects is considered in the proposed algorithm. If an object i in $F(n)$ is re-referenced, we insert i into the cache with a probability p_i, which depends on the relative sizes of objects in $F(n)$. We will use the following formula for p_i, though other formulations are also possible.

$$p_i = 1 - \frac{s_i - s_{min}}{2 \cdot (s_{max} - s_{min})}.$$

Here, s_{min} and s_{max} are the size of the smallest and largest objects in $F(n)$, respectively. p_i has the maximum value 1 at $s_i = s_{min}$ and the minimum value 1/2 at $s_i = s_{max}$. It decreases with the increase of s_i. More specifically, if an object i in $F(n)$ is re-referenced, we draw a random value r_i from the uniform distribution on $[0, 1]$. If $r_i \leq p_i$, we admit i into the cache. Otherwise, object i has to wait until the next reference. Thus, the smaller the size of object is, the higher the chance of admission is. Since admitting a large object into the cache causes many objects to be evicted from the cache and a large amount of disk I/O, we admit large objects more conservatively than smaller ones. For example, if the smallest object i in $F(n)$ is re-referenced, it will be absolutely admitted into the cache since $p_i = 1$. On the other hand, if the largest object i in $O(n)$ is re-referenced, it is admitted into the cache with $p_i = 1/2$. Note that, since $p_i \geq 1/2$, the largest object takes at most two tests on average for admission.

```
/* Procedure to be invoked upon a reference to object i */
 1: if (i is in the cache)     /* cache hit */
 2:     serve i from the cache;
 3: else if (the pair (d_i, s_i) is in F(n))
 4:     /* perform the admission test of section 2.2 */
 5:     draw r_i from the uniform distribution on [0, 1];
 6:     if (r_i ≤ p_i)          /* admission test passed */
 7:         evict objects from the cache if needed;
 8:         insert i into the cache and serve i from the cache;
 9:         numAdmitted++;
10:     else                    /* admission test failed */
11:         insert (d_i, s_i) into F;
12:         serve i from the origin server;
13: else
14:     insert (d_i, s_i) into F;
15:     serve i from the origin server;
16: m++;
17: if (m ≥ n) adjustSelectionWindow();
18: Procedure adjustSelectionWindow()
19: if (numAdmitted > 1)
20:     n ← n × (1 − β);   /* β: window adjusting factor */
21: else if (numAdmitted = 0)
22:     n ← n × (1 + β);
23: m = numAdmitted = 0;
```

Fig. 2 Algorithm AFAC.

2.3 Algorithm AFAC

The algorithm in Fig. 2 shows the overall procedure of our admission control method. Initially, the value of n is set to half of the maximum number of objects the Web cache can store, and will be adaptively adjusted during the execution of our algorithm. The maximum size of FIFO queue F, i.e., N, is the maximum allowed memory size. m and *numAdmitted* is initially set to 0, respectively. Note that eviction of objects in line 7 will require a certain cache replacement policy. As indicated before, we focus only on a new admission control algorithm, not cache replacement policies. Most existing cache replacement algorithms can be used for eviction of objects in line 7 of Algorithm AFAC.

3. Performance Evaluation

To evaluate the performance of the proposed algorithm, we compared the following five algorithms: *LRU*, *LFU*, *GD-SIZE*, *2Q*, and *LRU+AFAC*. LRU+AFAC uses LRU and

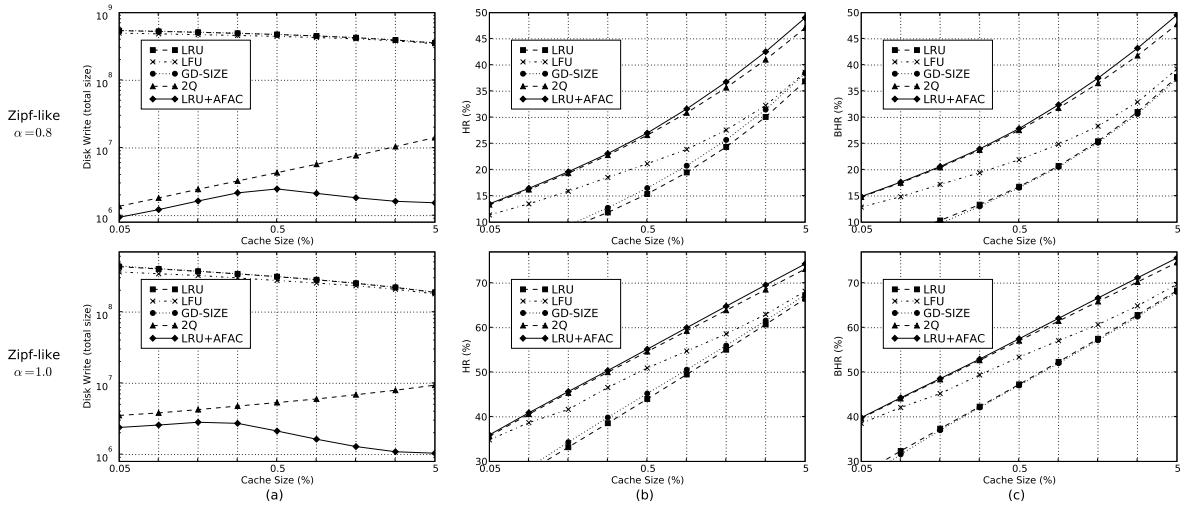

Fig. 3 Performance results on synthetic dataset.

Table 1 The specifications of traces used in the experiments.

Trace	UCC (Japan)	UCC (USA)	UCC (Europe)
Period	3/10/2008	3/10/2008	3/10/2008
Total requests	14,062,314	10,352,340	2,520,239
Distinct objects	185,720	193,352	130,102
Total transfer bytes	128 TB	101 TB	23 TB
Total unique bytes	1.5 TB	1.4 TB	0.9 TB

AFAC as the replacement algorithm and admission control algorithm, respectively. For LRU, LFU, and GD-SIZE, no admission control is applied. LRU, LFU, and GD-SIZE are representatives of frequency-based, recency-based, and size-based cache replacement algorithms, respectively. Note that 2Q uses LRU as the replacement algorithm. For 2Q, we set the size of `A1` to half the expected average number of objects in the cache as suggested in [3]. We conducted trace-driven simulations using the following datasets.

- *Synthetic dataset:* We used 100,000 distinct objects whose sizes range from 100 KB to 10 MB and follows a Zipf-like distribution with $\alpha = 1.0$, centered around 5 MB. As for queries, we used 10,000,000 requests that follow a Zipf-like distribution with parameter $\alpha = 0.8$ and 1.0 [5].
- *Real dataset:* We used three Web cache traces from a YouTube-like world-wide UCC video service site[†]. The contents of the site are delivered by Content Delivery Networks (CDN) to several Web caches located in various countries. Each of three traces, namely *UCC (Japan)*, *UCC (USA)*, and *UCC (Europe)*, contains user access logs for flash video objects on the site during 24 hours from Japan, USA, and Europe, respectively. Table 1 shows the specification of each trace.

We measured the following three performance metrics: (1) *Disk writes*: The total bytes written on the Web cache. This metric indicates disk I/O overhead. (2) *Hit ratio (HR)*: the total number of hits in the cache divided by the total number of requests. (3) *Byte hit ratio (BHR)*: the total bytes served by the cache divided by the total requested bytes.

Figure 3 shows the performance of each algorithm for the synthetic datasets with increasing cache size. For disk writes in Fig. 3 (a), the algorithms with admission control, i.e., LRU+AFAC and 2Q, show a substantial improvement over the other algorithms. In Fig. 3 (a), LRU+AFAC outperforms 2Q significantly in larger cache sizes. i.e., cache sizes of more than 0.5% of the total working set size. In 2Q, the size of `A1` increases as the cache size increases. Thus, as the cache size increases, `A1` will contain more number of object identifiers, resulting in more cold objects being cached. In AFAC, however, by adaptively adjusting the selection window, we can avoid caching objects that are not re-referenced within a short period of time. Note that admitting more objects incurs more disk writes, but does not necessarily affect HR. Figures 3 (b) and (c) shows that LRU+AFAC performs better than the other algorithms for both HR and BHR.

Figure 4 shows the performance of each algorithm for the real datasets with increasing cache size. As in the cases of synthetic dataset, we can see that LRU+AFAC and 2Q outperform the others in all cases and LRU+AFAC outperforms 2Q for disk writes. Note that GD-SIZE considers the object sizes as in AFAC. However, it considers the object sizes in evicting objects, not in admitting objects. Therefore, many cold objects can still be inserted into the cache, resulting in more disk I/O and a lower hit ratio than LRU+AFAC.

Figure 5 shows the performance of *LFU+AFAC*, which uses LFU and AFAC as the replacement algorithm and admission control algorithm, respectively. Here, *Modified 2Q* represents the 2Q algorithm that is modified to use LFU as its replacement algorithm. Similarly to the LRU case, LFU+AFAC provides better performance than LFU and Modified 2Q in all metrics.

From the experimental results, we can confirm that AFAC can be used in combination with cache replacement algorithms to improve the Web cache performance still fur-

[†]We received the traces from CDNetworks Co., Ltd. [6].

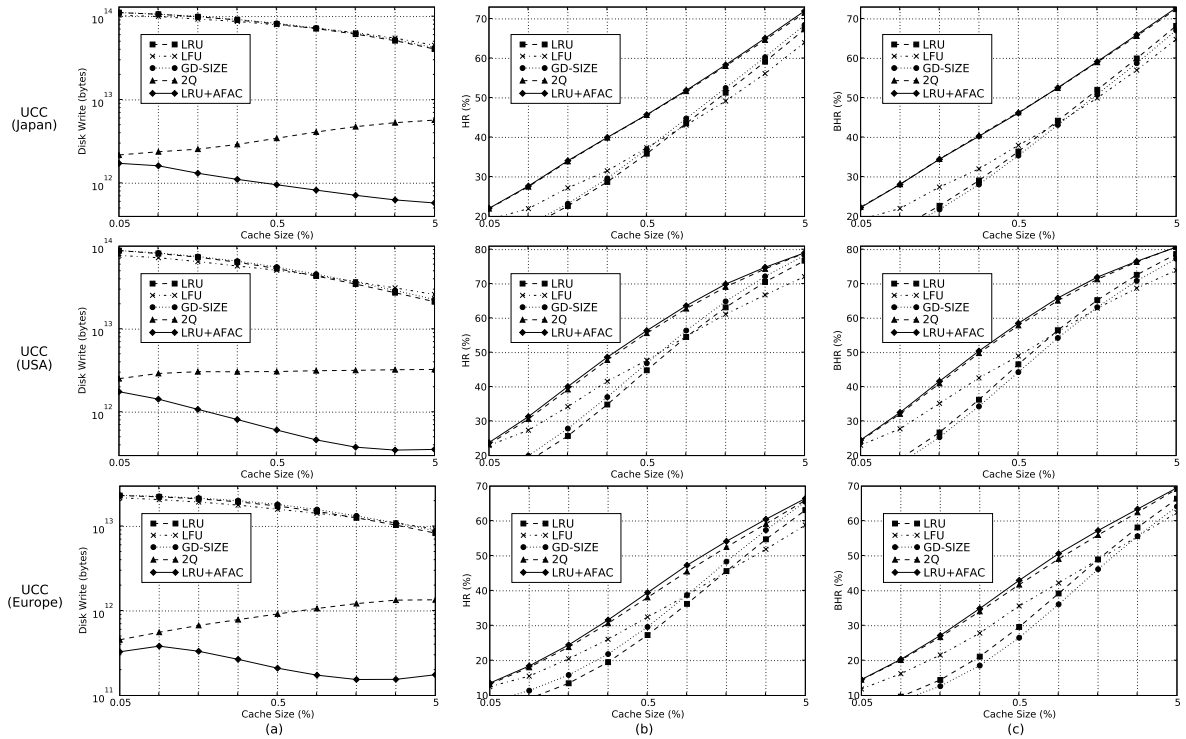

Fig. 4 Performance results on real dataset.

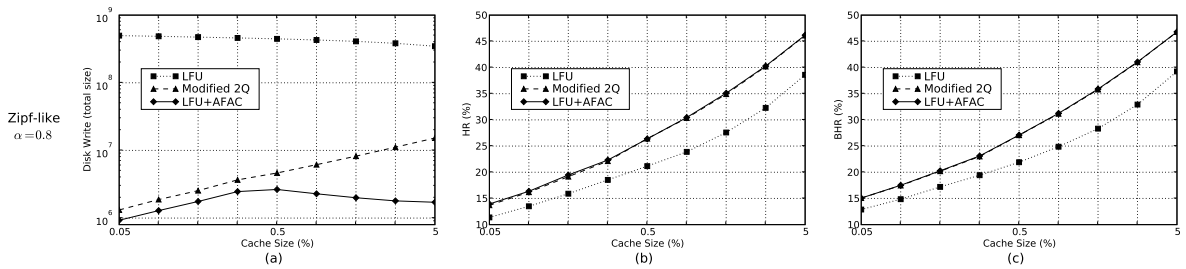

Fig. 5 Performance results when combined with LFU.

ther, by selectively admitting objects into the cache based on their re-reference likelihood and size.

4. Conclusion

We have proposed an effective admission control algorithm for large Web caches, called AFAC. By selectively admitting objects into the cache, AFAC can significantly reduce disk I/O while maintaining a high hit ratio. We have shown through extensive experiments that our proposed algorithm can substantially improve the Web cache performance in practice.

Acknowledgments

This research was supported by the Ministry of Knowledge Economy, Korea, under the Information Technology Research Center support program supervised by the Institute of Information Technology Advancement. (grant number IITA-2008-C1090-0801-0031).

Ki Yong Lee's work was supported by Brain Korea 21 Project, the School of Information Technology, KAIST in 2009.

References

[1] S. Podlipnig and L. Boszormenyi, "A survey of Web cache replacement strategies," ACM Comput. Surv., vol.35, no.4, pp.374–398, 2003.
[2] YouTube, LLC, http://www.youtube.com, 2008.
[3] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," Proc. 20th VLDB Conference, 1994.
[4] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," Proc. First USENIX Symposium on Internet Technologies and Systems, pp.193–206, 1997.
[5] L. Breslau, P. Cao, L. Fan, G. Philips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," IEEE INFOCOM '99, pp.126–134, 1999.
[6] CDNetworks Co., Ltd., http://www.cdnetworks.com, 2008.