Delayed Popularity-Aware Web Proxy Caching Algorithms

Yeim-kuan Chang
Department of Information Management
Chung Hua University
30 Tung Shiang,
Hsin Chu, Taiwan, 30067, R.O.C.
ykchang@mi.chu.edu.tw

Abstract:

World Wide Web has been a very successful distributed system that distributes and shares information on the Internet. Caching objects close to the user community provides an opportunity to allow other users to get the same objects quickly. Although caching can reduce the delays of retrieving popular objects, the utilization of the disk space in the caching servers is usually very low. Low disk utilization is the cause of low cache hit ratio. One of the reasons is that most objects cached in the disk of a caching server are unpopular. These unpopular objects are generally referenced once. Ideally, if the unpopular objects can be predicted and only popular ones are cached, the disk utilization will be improved. Thus, the hit ratio will also increase.

Currently, all existing caching replacement algorithms store the objects in cache when they are referenced the first time, without considering how popular these cached objects are. As long as a newly requested object is cacheable, replacement algorithm focuses on how to select old objects in cache in order to allocate sufficient cache space to hold the object. In this paper, we take different approach: the objects are not necessarily stored in cache when they are first referenced. We focus on how to determine if the newly requested objects can be stored in cache when they are first referenced. Based on this approach, we propose an algorithm that improves the cache hit ratio and effectiveness. We show that the proposed algorithm is superior to the traditional caching policies, using extensive trace-driven simulations.

Keywords: Caching Proxy Server, Cache Hit Ratio, Cache Replacement policy, Popularity,
Trace-Driven Simulation.

1 Introduction

The World Wide Web (WWW) [Best95,Wesl95,Liuc97] has been very successful for distributing and sharing information on the Internet. The exponential growth of the Internet usage has led many new problems to be solved. These new problems include web server load unbalance and the hot spots on the Internet links caused by a large number of accesses to the same object in a short period.

Caching mechanism [Gwer94, Ding96, Abra95] used in current proxy server provides an opportunity to allow users to benefit from data accessed by other users. Theoretically, in an infinite cache, any object that is accessed by a user will stay in the cache forever. Any request for a cached object will be serviced locally. Thus, the users experience a shorter access delay. The benefits of caching proxy are reductions of network traffic, user-perceived retrieval delay, and server load.

Since the cache size in a proxy server is limited, many objects stored in cache will be replaced by the newly incoming objects before they are referenced again. Therefore, issues on how to prevent the cached objects from being replaced before they are re-accessed are worth studying. Replacement algorithms focus on the efficiency of managing the cache. The information about object size, cost, reference frequency, and temporal locality is used to optimize the replacement algorithms. As the word "replacement" is implied, all replacement algorithms store any objects that are defined to be cacheable, no matter will they be reused in the near future or not. This blind *save-it-first* policy is the primary cause of low disk utilization, and thus, indirectly decreases the hit ratio. Since all the cacheable objects are stored in cache, the cache soon become full and some kind of replacement actions must be taken.

Better replacement algorithms improve the hit ratio of caching proxy servers. The most widely used cache replacement algorithm is least-recently-used (LRU). LRU utilizes the temporal locality in request stream to replace the least recently used objects. Other replacement algorithms include largest size first, popularity-aware algorithms, etc. Frequency-based [Jin99,Arli99] and cost-aware [Cao97] algorithms are the most recent work proposed in the literature.

Users tend to visit many new pages before they come back to the old ones. Based on the characteristics of the requests through a typical proxy server, more than 60% of the requests are for non-shared objects. If a blind save-it-first approach is employed, there will be more than 60% of the non-shared objects stored in cache all the time, no matter what replacement algorithm is used. We conjecture that if the storage of the non-shared objects can be decreased, the hit ratio will be increased.

In this paper, we take an opposite approach to the traditional save-it-first methods: the requested objects from original server are not necessarily stored in cache when they are first accessed. The objects are stored in cache when they are re-accessed. At the first glance, we immediately notice that there is a disadvantage: all possible hits when objects are accessed the second time seem to be lost. However, on the other end, the cache is more efficiently utilized than the save-it-first approaches. In other words, the cache can store more objects that can be reused than traditional algorithms. To avoid losing the cache hits when the requested objects are accessed the second time, we utilize main memory space in the proposed algorithm. The reason is that memory is used to buffer the objects before they are forwarded to the requestor. We will show that the proposed algorithm actually performs better.

The rest of the paper begins with a discussion of the characteristics of access streams through caching proxy servers in section 2. Section 3 illustrates the basic cache architecture we used in this paper. Section 4 describes the details of the proposed algorithm. The results of performance comparisons using an extensive trace-driven simulation are presented in section 5. Finally, a concluding remark is given in the last section.

2 Characteristics of the Caching Servers

In this section, we will first study the issue of increasing the cache utilization of a caching server. We will show that the disk utilization is low for the traditional caching algorithms. Note that the hit ratios of caching servers located in most LANs are between 20% to 60% , no matter what optimizations are used. The reasons are that the requested objects don't exist in cache or the objects existing in cache are expired.

We analyze the NLANR UC trace and our campus's CHU trace. The results are given in Table 1. The number of objects accessed only once is 2,903,145 (34 G Bytes) that accounts for 73.57% of the total unique objects (82.44% in bytes). We call the objects accessed once *non-shared* objects. The objects accessed only twice account for 12.24% (9.42% in bytes), and are called *shared-twice* objects. By carefully studying the trace, we find out that users tend to navigate many new pages before accessing old ones. Almost 80% of the objects stored in cache are never re-used before they are evicted. Thus, the times taken by the disk for storing and removing those non-shared objects are wasted. This useless disk activity indirectly increases the time for servicing a request hitting on the cache.

Theoretically, the space wasted for storing non-shared objects is roughly $(1-h)S$ bytes, where S and h denote the cache size and the average hit ratio, respectively. We can convert $h(S)$ to $h_{eff}(S_{eff})$ where h_{eff} is the hit ratio for an effective cache size of $S_{eff}=hS$ bytes. Figure 1 also shows the plots of $h_{eff}(S_{eff})$ in dashed lines for UC and CHU traces, using LRU replacement algorithm. Thus, with an optimal algorithm that predicts which objects are accessed only once, the hit ratio can be improved from $h_{eff}(hS)$ to $h_{eff}(S)$. For example, for a cache of 1 GB in size, an optimal caching algorithm can improve the hit ratio from 34% to 41% and the byte hit ratio from 17% to 22% for UC trace.

Obviously, finding an optimal caching algorithm is not an easy task. The popularity of an object may give us a hint about how frequently it will be referenced again. Many relevant factors need to be considered. First, all the objects accessed on the Internet form a hierarchical structure. The top-level objects of the tree are accessed ahead of the lower level objects. Therefore, the top-level should be accessed more often than the lower level objects. Secondly, the objects in a popular site tend to be more popular than that in an unpopular site. Thus, the objects in a popular site should be cached.

To predict the object popularity, some kind of statistics may be used. In other words, a large number of historic accesses are needed to perform the statistical estimations. As the characteristics of UC trace shows, more than 90% of the requested objects are accessed once or twice. For these non-shared and shared-twice objects, we don't have enough historic information to estimate how popular an object is. Do we store the objects in cache when they are first accessed? Traditional caching algorithms take the positive approach: the objects are stored in cache when they are first accessed. If we use the LRU caching algorithm, the probability that an object is accessed once or twice before it is evicted is more than 90%, no matter what is the size of the cache.

Generally, it is difficult to determine whether or not we save the objects in cache when they are first accessed. If we save the non-shared objects, we can use the statistics of all pages to determine if they should be removed some time later. But this approach is in fact a page replacement method. We know that there is no perfect replacement algorithm reported in the literature.

We now study the possibility of taking an opposite approach: the objects are not stored in the cache when they are first accessed until some kind of information appears. We use the information about popularity as the indicator to determine whether the newly requested objects are stored in cache. We call this approach as *delayed popularity-aware* approach.

Is there any advantage that we can gain from this approach? Intuitively, the cache utilization will increase. Thus, the hit ratio will also increase. The proposed approach uses an extra memory space called URL memory cache to record the URLs of the objects when they are first accessed. The objects are stored in cache when their URLs have been recorded earlier. Obviously, the non-shared objects will never be stored in disk cache. The useless disk activity for storing non-shared objects is eliminated completely. However, the possible hits on the objects in cache when they are accessed the second time may be lost. Take the UC trace as an example again. As shown in Table 1, we will at best have 45% of the hit ratio with a cache of size 7.2 GB in the extreme case where the requested objects are not stored in disk cache in the first time. This approach does not seem to be a good solution because the traditional LRU caching algorithm gets a hit ratio of about 50% for a cache of size 7.2 GB shown in Figure 1. However, based on our early study, even in this extreme case, the hit ratio of our approach using LRU as a base-line algorithm performs better than traditional LRU algorithm when cache size is smaller than 2 GB. The extreme case gets a hit ratio of 45% in a cache of infinite size. But it is definitely worse than the better replacement algorithms such as GDSP and LFU-DA proposed in [Jin99,Arli99].

The 11% hit ratio difference between the traditional caching algorithm using the cache of infinite size and the extreme case comes from the misses when objects are accessed the second time. We further analyze the UC trace for the distribution of arrival time difference between the first time and the second time that objects are accessed. About 20-30% of all references to an object occurred within less than 10000 other references to the same object. Based on this property, we will utilize the memory space that is originally used as buffers for disk cache to reduce the miss possibility when objects are referenced the second time. This memory space does store the objects when they are first referenced.

3 The Basic Cache Architecture

Basic cache architecture used in caching proxy is illustrated in Figure 2. The cache architecture described in this section is also implemented in Squid proxy, the most popular caching proxy server. A directory is maintained in main memory of the system in order to access to the cached objects. The MD5 codes of URLs are employed as the keys in the directory. Thus, the directory is usually implemented as a hash table to support efficient search for requested objects.

A special data field is needed to implement specific cache replacement policy. For example, the *lru* field shown in Figure 2 implements a doubly linked list of stored entries to facilitate the LRU policy. There are two levels of caches, memory and disk caches. This 2-level cache model does not maintain a cache inclusion property. The cacheable objects stored in memory cache are called *hot* objects. The hot objects must have a copy in disk cache. However, the memory cache is also used to store the *in-transit* objects. The in-transit objects are the non-cacheable objects that caching server fetches from original server on behalf of clients. The memory cache can be seen as buffering space for disk cache while data needs to be sent to clients. When a request hits in the caching proxy, the requested object stored in disk cache must first copied to memory cache before it is ready to be sent to the client.

Only a single directory exists for both memory and disk caches. If there is no sufficient space in memory cache to hold the requested object, the same replacement policy as the disk cache is executed. To be more specific, the detailed cache operations for both memory cache and disk cache are described as follows.

Cache hits: Upon receiving a request, the caching proxy first looks up the directory for the requested object using the MD5 code of the URL as the key. Assume it is a hit. The caching proxy now checks to see if the requested object exists in memory cache. If a copy in memory cache does not exist, a sufficient memory space must be allocated first by replacing unused non-cacheable objects or unpopular hot objects in memory cache. Assuming the LRU replacement policy is used, the caching proxy goes through the lru doubly linked list in directory to find out which memory objects are old enough to be replaced. The replacing process continues until the free space in memory cache is large enough to hold the requested object. At last, the data of the requested object is in memory cache and ready to be sent to client. Notice that the replacing process for memory cache does not affect the objects stored in disk cache.

Cache misses for cacheable objects: The caching proxy first creates an entry in the directory for holding the requested objects. After the data of requested object is fetched from original server, it is stored in both memory and disk caches. A similar memory cache replacing process as the cache hit case is performed. To hold the requested object in disk cache, a replacement policy is also needed. The replacing process for the disk cache completely removes all the data associated to the old cached objects in the directory. The removed data include the entry in the hash table, the memory copy in memory cache if exists, and disk copy in disk cache, etc.

Cache misses for non-cacheable objects: The caching proxy also creates an entry in directory for holding the requested objects. After the data of requested object is fetched from original server, it is stored only in memory cache. A similar memory cache replacing process as the cache hit case is performed. A replacement policy for disk cache is not needed since non-cacheable objects are not stored in disk cache. Unused non-cacheable objects in memory cache are the first ones to be selected for replacement.

Notice that in the case of cache miss for cacheable objects, a requested object must be stored in disk cache as long as it is cacheable. In other words, no matter will the requested object be re-used in the future or not, it is stored in disk cache. It is reason that the typical caching proxy contains over 60% of the cached objects that are only referenced once before they are evicted from the system. We will propose a new algorithm that can predict whether or not the newly requested objects be re-used in the future. Thus, the possibility that the objects are cached but not referenced again before they are evicted is reduced. This is the point on which we will elaborate in the next section.

4 The Proposed Caching Algorithms

The main task of the proposed *delayed popularity-aware* algorithm is to determine whether or not we will store the newly requested object in disk cache right after the object is fetched from original server. Many characteristics of web objects can be considered as pointers for make this decision. Typical examples include object size, object type, popularity (access frequency), temporal property, and even the depth of the pages and the length of URLs. We use the following heuristic to make this decision: *the most frequently accessed objects recently will most likely be accessed again.* The words "frequently" and "recently" imply that access frequency of objects and a decay function applied on frequency [Jin99] are needed. Therefore, in the proposed algorithm, an extra space called *URL memory cache* is introduced to store URLs and the associated access frequency of the requested objects. If the requested object is cacheable, the process of storing the object in disk cache is delayed until the same object is accessed again. Or we can say that cacheable objects are not stored in disk cache unless they have been accessed before. The above heuristic determines which objects can stay in URL cache longer than other ones. Thus, it indirectly affects which object can be stored in disk cache right after it is fetched.

Based on the idea described, it is possible that a cacheable object has a copy of data in memory cache, but not in disk cache. Since the access stream is infinite, the size of URL cache must be limited. A replacement policy is also needed in URL cache. Detailed cache operations are described as follows.

Cache hits: The operations are similar to the original algorithm. In addition to unused non-cacheable objects and hot objects in memory cache, the cacheable objects without disk copies are also the candidates for replacement in memory cache. Consider the case that a copy of the requested object exists in memory cache but not in disk cache. The reference count associated with the requested object in memory cache is incremented by one and the data is then stored in disk cache. If the evicted objects from memory cache are cacheable, its URL along with its reference count is then stored in URL cache.

Cache misses for cacheable objects: For a cache miss, if the requested object is cacheable, the caching algorithm checks if its URL is recorded in URL cache. If not, replacement operations are performed for allocating enough space for holding the requested object. The URL of the replaced object is now stored in URL cache along with its reference count. The replacement operations in URL cache must be performed. The evicted URLs from URL cache are released. The requested object itself is not stored in disk cache at this moment. Thus, no replacement in disk cache is needed. If the URL of the requested object is in URL cache, its associated record in URL cache is removed, the requested object is stored in disk cache, and the reference count is set to one. Similarly, the replacement operations in disk cache must be performed. The URLs of the evicted objects from disk cache are stored in URL cache and again the replacement operations in URL cache are performed.

Cache misses for no-cacheable objects: For a cache miss, if the object is non-cacheable, the operations are similar to original algorithm. If the evicted object from memory cache is cacheable and it does not exist in disk cache, its URL along with the reference count is stored in URL cache.

Notice that the proposed approach may lose some possible hits on the disk cache when objects are accessed the second time. However, it removes all the disk activity that disk cache stores the objects that will not be accessed again before evicted.

Efficient Management of URL Cache

A separate hash table similar to that in memory/disk cache is used in URL cache to support efficient search for the URL of requested object. The MD5 of URL is employed as the search key. We employ a replacement policy that is based on the URL access frequency. The least frequently accessed entry in URL cache is first selected for replacement. A priority queue with access frequency as the key is a suitable implementation for such replacement policy. As suggested in [Jin99], the O(logn) time cost can be reduced to O(1) by aggregating entries in one group with the same access frequency. The priority queue is constructed by groups instead of single entries.

Each entry of the URL cache records the MD5 of URL, access frequency, and a few pointers for facilitating priority queue and hash table data structures. The required memory space for each entry in URL cache is constant. The size of the hash table and priority queue itself is small and does not depend on the number of entries hashed, thus can be ignored. Based the size of the UC trace we studied in this paper, keeping all the URLs of the requests from one-day period in URL cache is reasonable. This accounts for 400k URLs. Therefore, assuming 80 bytes is needed for each entry in URL cache, 32 MB of the memory space is needed for the URL cache.

5 Performance Evaluation

We adopt the caching policy of the Squid proxy server: if the URL of the requested object contains '?' or 'cgi', the object is not cached. If the size of the requested object is larger than 1 MB, we don't cache the object either. Since there is no cache-control headers recorded in *access.log* of Squid proxy server, all other URLs are considered cacheable. Note that the size field in access.log is the number of bytes transferred to the requesting client. This size may not be equal to real object size. Therefore, to make our simulation more accurate, we replace the sizes in access.log with the real sizes logged in *strore.log* file by a 2-pass preprocess.

Performance Metrics:

Four performance metrics are considered in this paper: namely, Hit Ratio (HR), Byte Hit Ratio (BHR), cache effectiveness (CE), and byte cache effectiveness (BCE). All the cacheable and non-cacheable requests are counted in the metrics. HR is ratio of the number of the requests hit in the cache to the total number of the requests. BHR is ratio of the number of the bytes hit in the cache to the total number of the bytes requested. CE and BCE are defined as follows:

$$CE(BCE) = \frac{\sum evicted(i) \times (i-1)}{\sum evicted(i)}$$

, where *evicted(i)* is the number of objects (or bytes) that are referenced *i* times before they are evicted from the cache. CE and BCE reflect how effective the cache is utilized to store and re-use the requested objects. Ideally, objects are stored in cache once and reused many times before they are evicted.

Simulated caching algorithms:

In the simulations, we compared the proposed algorithm to LRU and GDSP(1) [Jin99,Arli99]. GDSP(1) treats all misses as having constant cost, i.e., using *frequency/size* as the basic value. GDSP(1) algorithm has been shown to outperform other replacement algorithms. The server name based algorithms [Kur98] is not compared since its hit ratio is worse than LRU algorithm. For the replacement policy of the proposed algorithm, we adapt GDSP(1) for memory and disk caches and LRU for URL cache. The size of URL cache is set to 32 MB which is capable of storing 400K URLs. For all the simulated algorithms, we assume that the memory cache size is 128 MB.

Figures 3 and 4 show the results for HR, BHR, CE, and BCE for UC trace. For HR and BHR, we plot the relative ratios to that in a cache of infinite size. As shown in Table 1, the HR and BHR are 0.556 and 0.23 in a cache of infinite size, respectively. The HRs of the proposed algorithm and GDSP(1) are must higher than that of LRU. The HR difference between the proposed algorithm and GDSP(1) is negligible. For BHR, the proposed algorithm outperforms the other two and GDSP(1) is slightly worse than LRU. The CE and BCE decreases as the size cache increases because average probability of cache space re-use reduces when the cache size becomes bigger. As we can see in the figures, the proposed algorithm has the highest CE and BCE. The cached objects in a cache with higher CE value can be re-accessed more often than that with lower CE value. In other words, the cache utilization using the proposed algorithm is better than other algorithms.

As mentioned earlier, the proposed algorithm may lose many possible hits in disk cache for objects that are accessed the second time. However, the proposed algorithm gains in that the disk cache tends to store the objects that will be re-accessed many times. In Figure 5, we plot the access frequency of the objects in disk cache and that of evicted objects with 1 GB in size. Other results for 2 and 4 GB are similar and not shown for the space limit. The proposed algorithm brings 4.4% (4.3% in bytes) and 31.8% (26.6% in bytes) of the objects stored the cache that are accessed once and twice, respectively. However, for GDSP(1), these numbers are 48% (57.2% in bytes) and 15.9% (11.5%), respectively. The access frequencies of the evicted objects that are accessed only once are 26.8% (25.8% in bytes), 90% (92.6% in bytes), 85.346% (89.4% in bytes), for the proposed algorithm, GDSP(1), and LRU, respectively.

Figures 6 and 7 show the results for HR, BHR, CE, and BCE for CHU trace. The proposed algorithm performs the best in BHR, CE and BCE. For the results of HR, the proposed algorithm is better than GDSP(1) when disk cache size is 1 GB or less. The HR difference between the proposed algorithm and GDSP(1) is less than 1% when disk cache size is 2 GB or larger. The CE of LRU is generally better than GDSP(1). However, The BCE of LRU is worse than GDSP(1).

6 Conclusions

In this paper, we took a different caching policy that determines whether or not the newly requested objects are stored in cache when they are first referenced. This is totally different from the replacement caching algorithms that always store the objects in cache when they are first referenced. Using an extensive trace-driven simulation, we showed that the proposed algorithms improve the hit ratio when cache size is small and the cache effectiveness over the traditional replacement algorithms.

References

1. [Jin99] S. Jin and Azer Bestavros. Popularity-aware greedy dual-size algorithms for Web access. In Proceedings of the 20th Int'l Conf. on Distributed Computing Systems (ICDCS), April 2000.
2. [Cao97] Pei Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms", Usenix Symposium on Internet Technlogy and Systems (USITS-97), 1997.
3. [Gwer94] Gwertzman, J. and Seltzer, M., "World Wide Web Cache Consistency", USENIX 1996, http://www.eecs.harvard.edu/~vino/web/usenix.196/.
4. [Ding96] Dingle, A. and Partl, T., "Web Cache Coherence", In Proceedings of the Fifth International World Wide Web Conference, Paris, France, May 6-10, 1996, http://www5conf.inria.fr/fich_html/papers/P2/Overview.html.
5. [Abra95] Abrams, M., Standridge C. R., etc., "Caching Proxies: Limitations and Potentials", In Proceedings of the Fourth International World Wide Web Conference, December 1995, http://ei.cs.vt.edu/~succeed/WWW4/WWW4.html.
6. [Arli99] M. Arlitt, et. al., "Evaluating Content Management Techniques for Web Proxy Servers", In *Proceedings of the 2nd Workshop on Internet Server Performance*, May, 1999.
7. [Liuc97] Liu, C. and Cao, P., "Maintaining Strong Cache Consistency in the World-Wide Web", In Proceedings of the 17th IEEE Int'l Conference on Distributed Computing Systems, May 1997.
8. [Bern95] Berners-Lee, T., "Propagation, Replication, and Caching on the Web", 1995, http://www.w3.org/pub/WWW/Propagation/Activity.html.
9. [Wesl95], D. "Intelligent Caching for World-Wide Web Objects", Proceedings of INET95, 1995.
10. [Best95] Bestavros, A., "Demand-based document dissemination for the World Wide Web", Technical Report BU-CS-95-003, Boston Univ. CS Department, Boston, MA, February 1995.
11. [NLANR] NLNAR Traces ftp://ftp.ircache.net/Traces/.
12. [CHU] Chung Hua University (CHU) Traces. http://proxy.chu.edu.tw/.

Traces	UC from NLANR site July 7 – 31, 2000	CHU from Chung Hua U. July 4 – August 9, 2000
All requests	9,851,129	13,439,638
All cacheable requests	9,423,761 (70.4 GB)	11,266,147 (95.3 GB)
All unique cacheable objects	3,946,291 (41.3 GB)	2099505 (25.3 GB)
All unique objects shared more than once	1,043,146 (7.2 GB)	790530 (8.4 GB)
All unique objects shared more than twice	559,972 (3.9 GB)	4640030(4.8 GB)
Hit ratio	55.6%	68.21%
Byte hit ratio	26%	48.57%
Hit ratio (first-not-save)	45%	62.32%

Table 1: The summarized characteristics of UC and CHU traces.

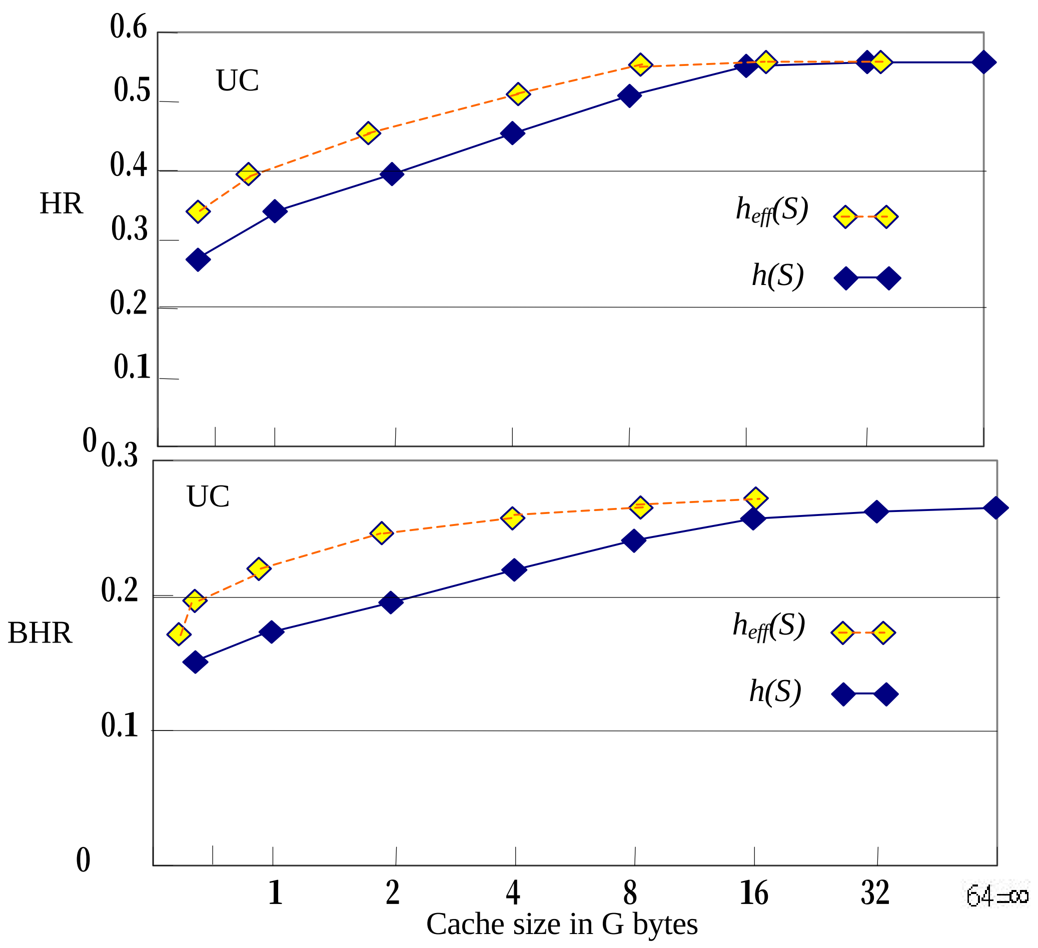


Figure 1: (a) plot of hit ratio *h(S)* and byte hit ratio $h_b(S)$ for UC.

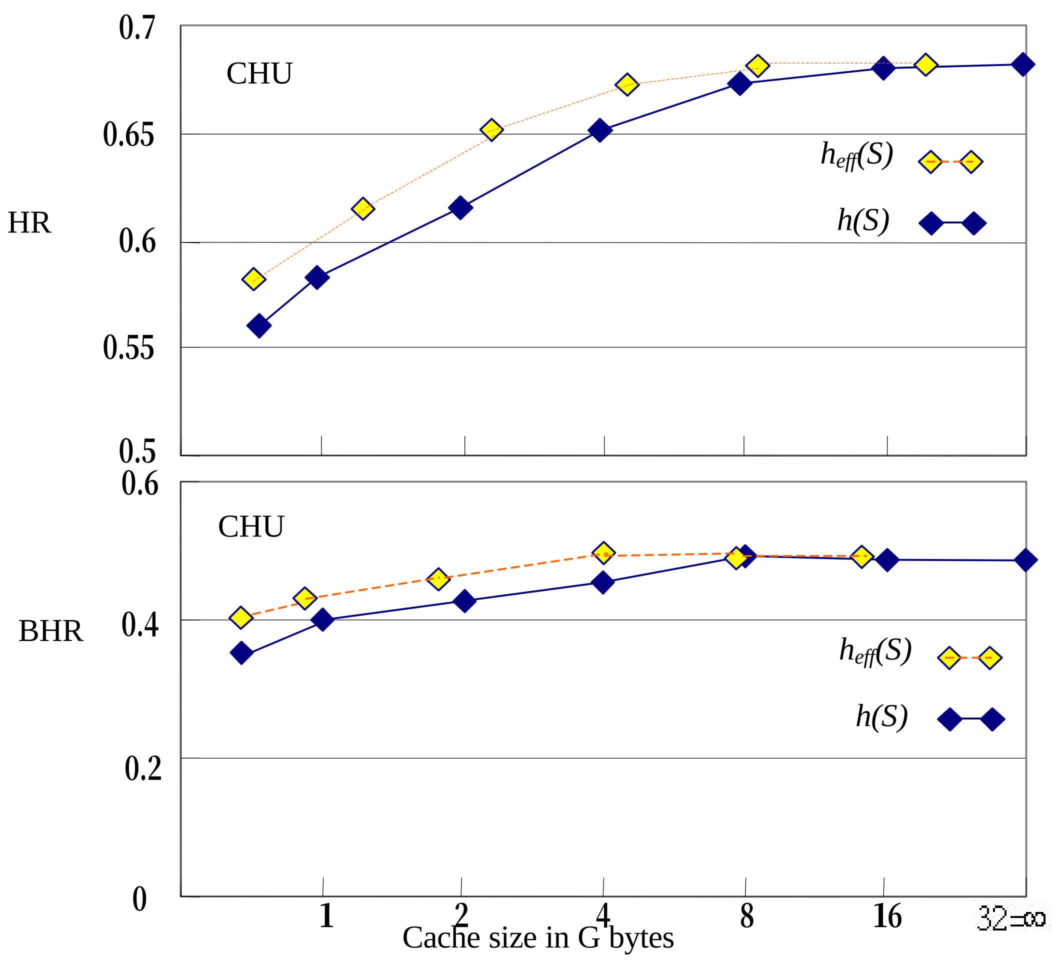


Figure 1: (b) plot of hit ratio *h(S)* and byte hit ratio $h_b(S)$ for CHU.

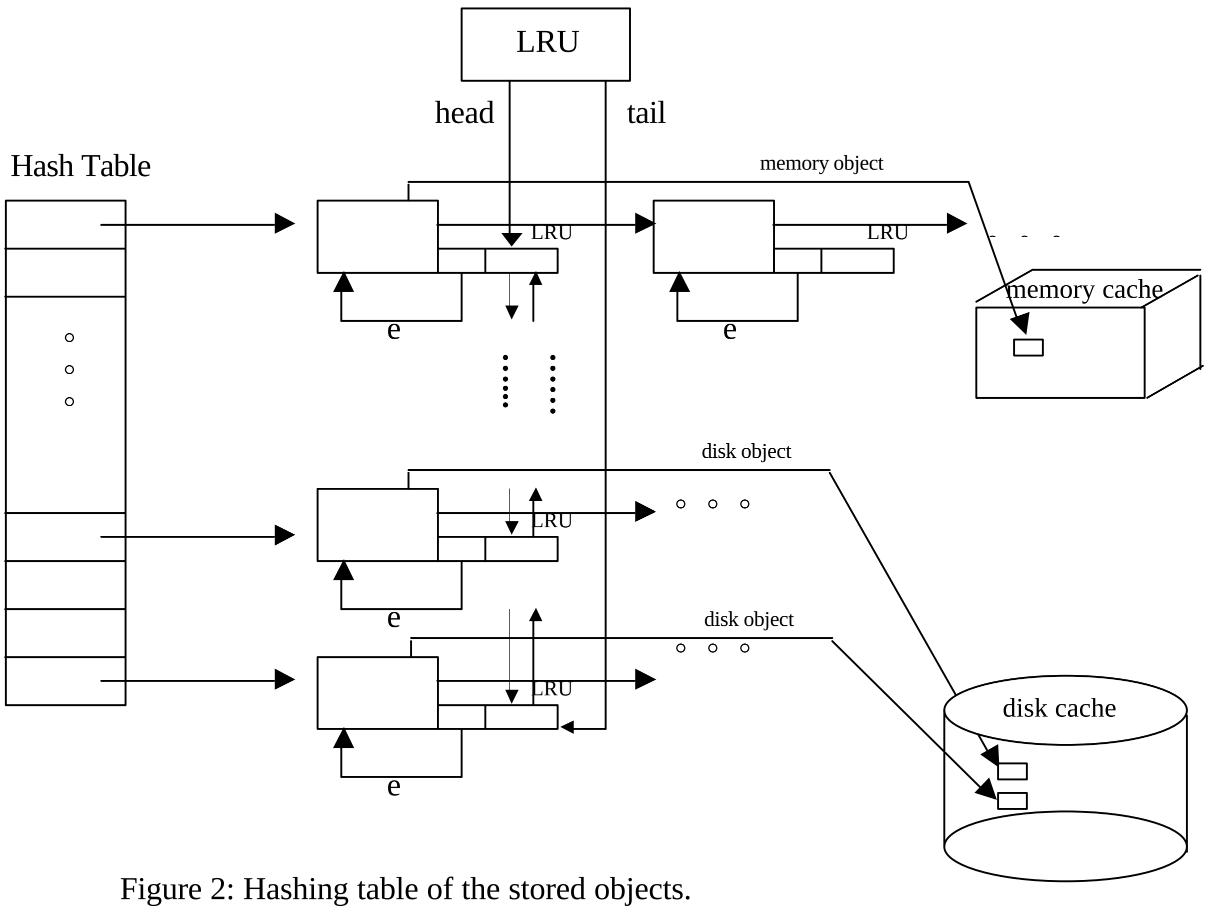


Figure 2: Hashing table of the stored objects.

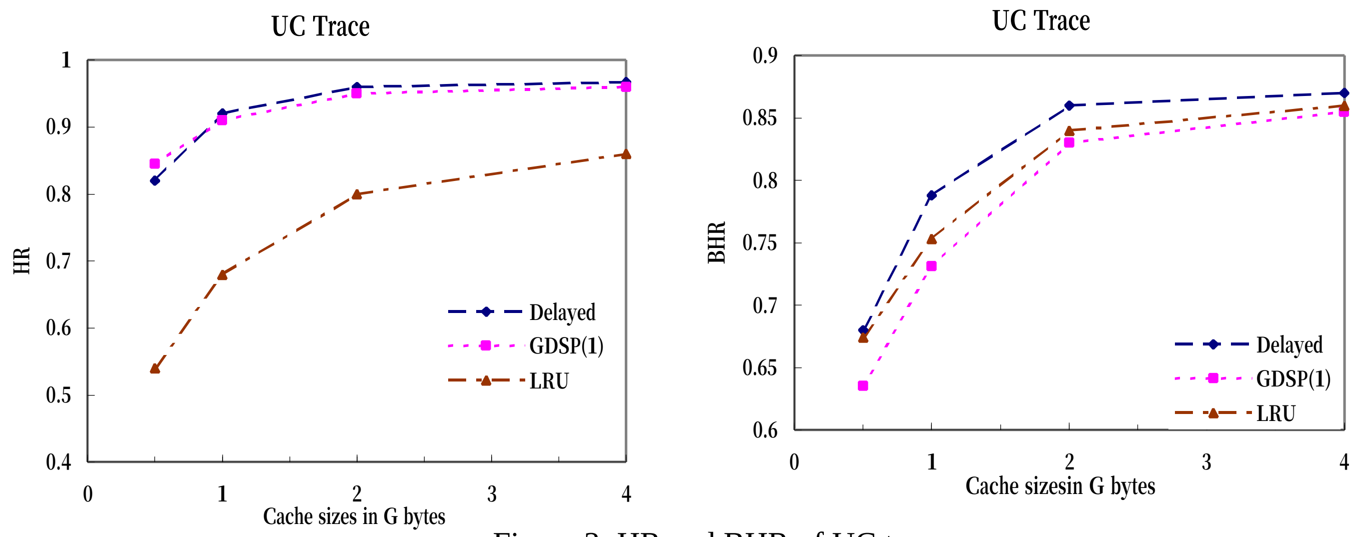


Figure 3: HR and BHR of UC trace.

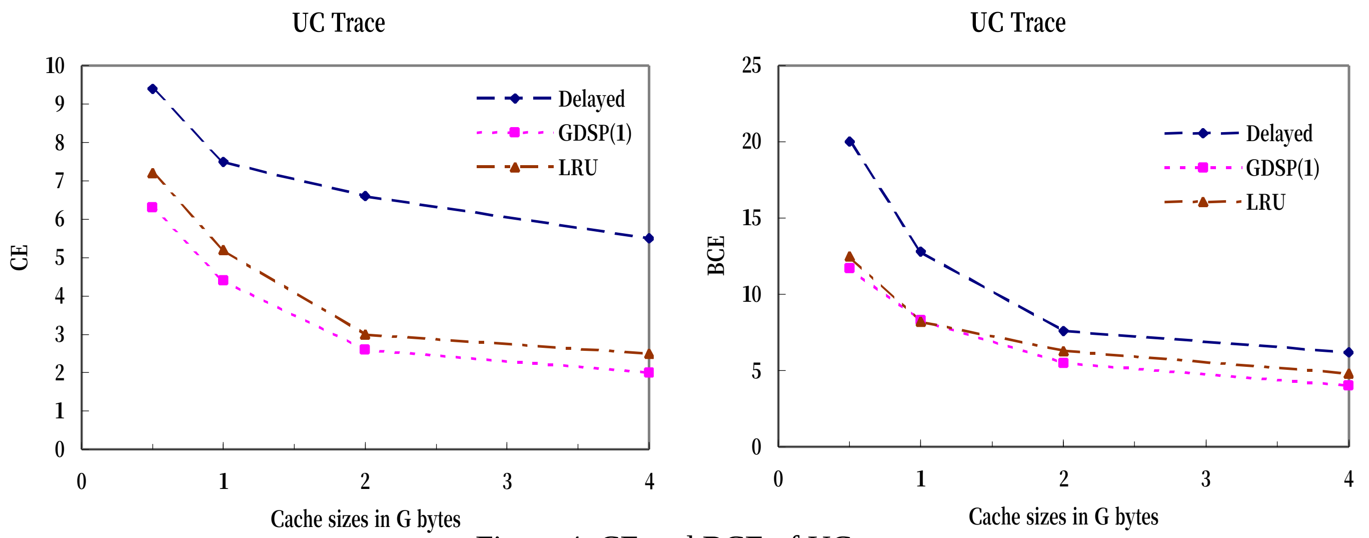


Figure 4: CE and BCE of UC trace.

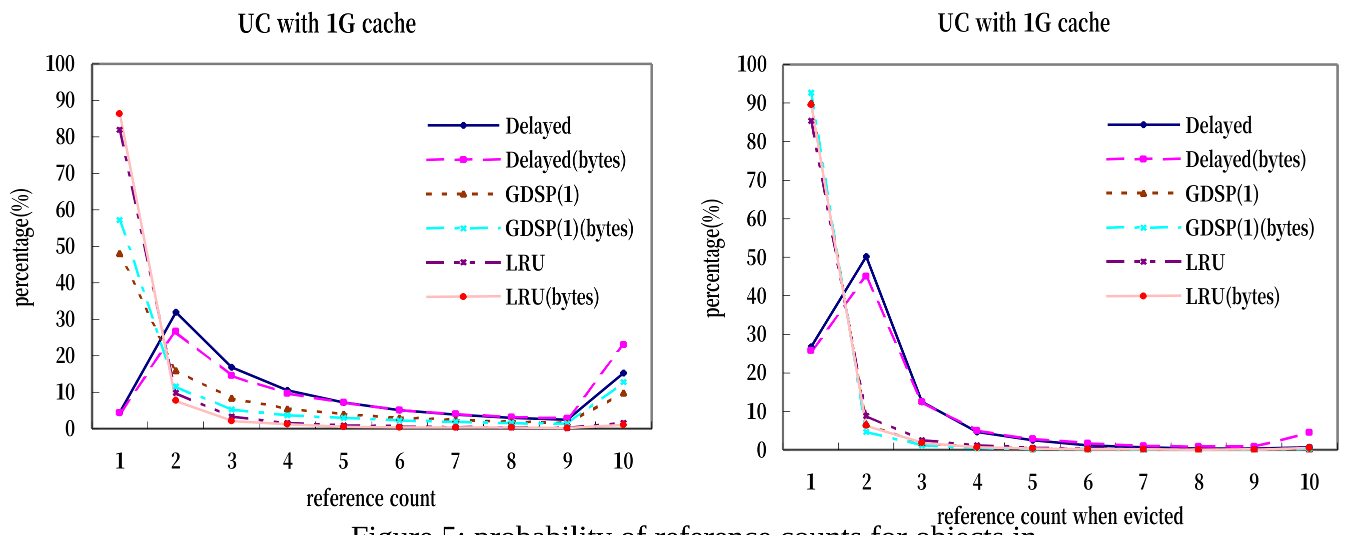
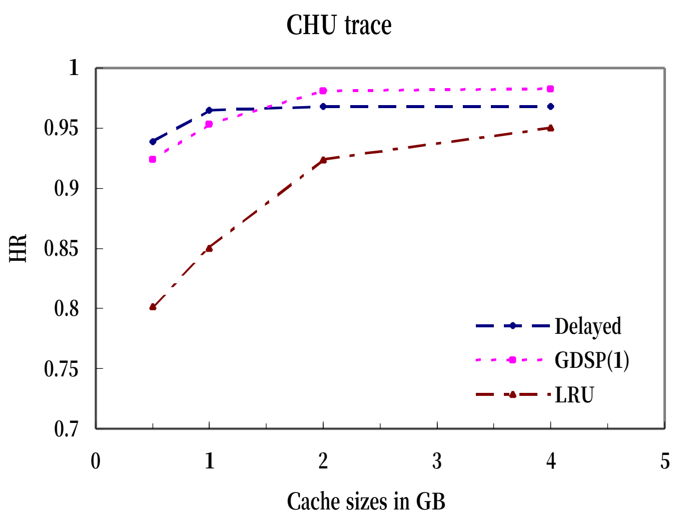


Figure 5: probability of reference counts for objects in cache and evicted objects.



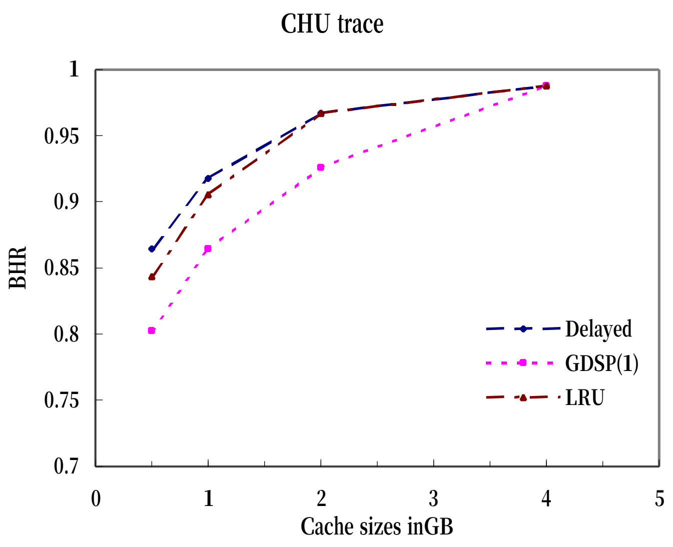
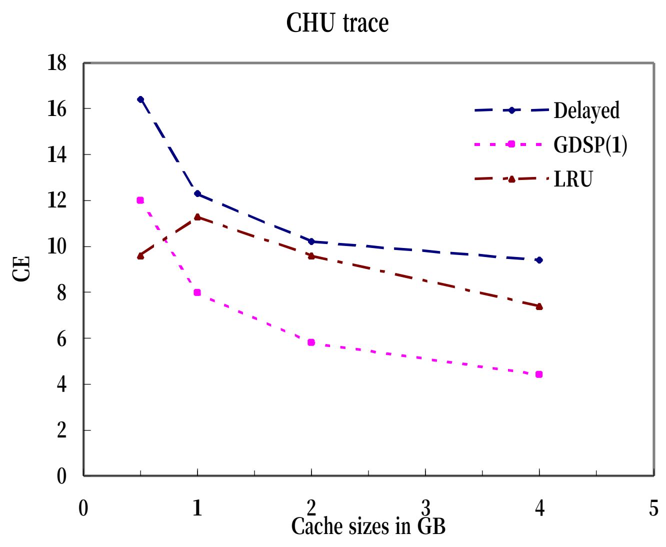


Figure 6: HR and BHR of CHU trace.



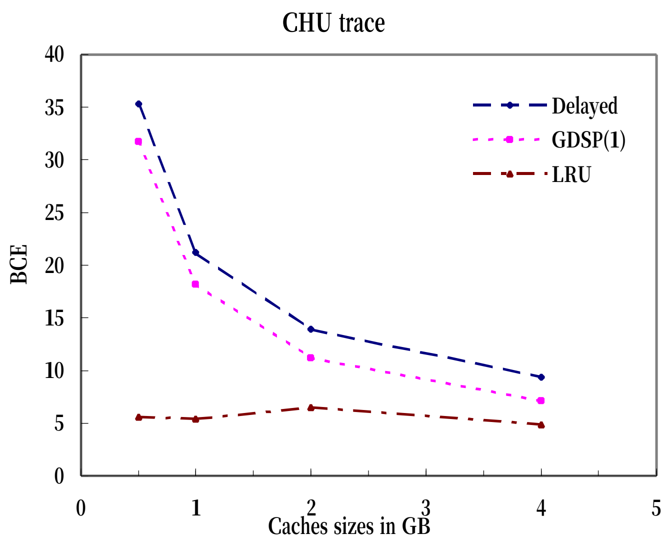


Figure 7: CE and BCE of CHU trace.