Popularity-Aware GreedyDual-Size Web Proxy Caching Algorithms*

Shudong Jin and Azer Bestavros
Computer Science Department
Boston University
111 Cummington St, Boston, MA 02215
{jins,bestavros}@cs.bu.edu

Abstract

Web caching aims at reducing network traffic, server load, and user-perceived retrieval delays by replicating popular content on proxy caches that are strategically placed within the network. While key to effective cache utilization, popularity information (e.g. relative access frequencies of objects requested through a proxy) is seldom incorporated directly in cache replacement algorithms. Rather, other properties of the request stream (e.g. temporal locality and content size), which are easier to capture in an on-line fashion, are used to indirectly infer popularity information, and hence drive cache replacement policies. Recent studies suggest that the correlation between these secondary properties and popularity is weakening due in part to the prevalence of efficient client and proxy caches. This trend points to the need for proxy cache replacement algorithms that directly capture popularity information.

In this paper, we (1) present an on-line algorithm that effectively captures and maintains an accurate popularity profile of Web objects requested through a caching proxy, (2) propose a novel cache replacement policy that uses such information to generalize the well-known GreedyDual-Size algorithm, and (3) show the superiority of our proposed algorithm by comparing it to a host of recently-proposed and widely-used algorithms using extensive trace-driven simulations and a variety of performance metrics.

1. Introduction

Web caching aims to reduce network traffic, server load, and user-perceived retrieval delay by replicating "popular" content on proxy caches [1, 14] that are strategically placed within the network—at organizational boundaries or major AS exchanges, for example.

It may be argued that the ever decreasing prices of RAM and disks render the optimization or fine tuning of cache replacement policies a "moot point". Such a conclusion is ill-guided for several reasons. First, recent studies have shown that Web cache hit ratio (HR) and byte hit ratio (BHR) grow in a *log-like* fashion as a function of cache size [2, 7, 8, 9]. Thus, a better algorithm that increases hit ratios by only several percentage points would be equivalent to a several-fold increase in cache size. Second, the growth rate of Web capacity is much higher than the rate with which memory sizes for Web caches are likely to grow. The only way to bridge this widening gap is through efficient cache management. Finally, the benefit of even a slight improvement in cache performance may have an appreciable effect on network traffic, especially when such gains are compounded through a cache hierarchy.

There are many factors that affect the performance of cache replacement policies. Among others, these factors include object size, miss penalty, temporary locality, and long-term access frequency.

- Unlike traditional caching in memory systems, Web caches are required to manage objects of *variable* sizes. Caching smaller and thus more objects usually results in higher hit ratios, especially given the preference for small objects in Web access [8]—though this preference seems to be weakening [5].
- The miss penalty (i.e. retrieval cost of missed objects from server to proxy) varies significantly. Thus, giving a preference to objects with a high retrieval latency can achieve high saving [22].
- Web traffic patterns were found to exhibit temporal locality [2, 9, 17], i.e., recently accessed objects are more likely to be accessed again in the near future. This has led to the use of LRU cache replacement policy and generalizations thereof [9]. Recent studies have documented a weakening in temporal locality [5].
- The popularity of Web objects was found to be highly

*This work was partially supported by NSF research grant CCR-9706685.

variable (i.e. bursty) over short times scales, but much smoother over long time scales [4, 13], suggesting the significance of *long-term* measurements of access frequency in cache replacement algorithms.

Motivation and key contributions: While key to effective cache utilization, *popularity* information (e.g., the relative access frequencies of objects requested through a proxy) is seldom maintained and rarely utilized *directly* in the design of cache replacement algorithms. Rather, other properties of the request stream (e.g., temporal locality and object size), which are easier to capture in an on-line fashion, are used to *indirectly* infer popularity information, and hence drive cache replacement policies.

To elaborate on this point, consider two widely used cache replacement policies: Least-Recently-Used (LRU) and Largest-File-First (LFF). LRU capitalizes on the temporal locality in a request stream, namely the recency of access, whereas LFF capitalizes on the negative correlation between popularity and object size. Both of these properties—namely, recency of a repeat access and size of requested object—are assumed to be indicative of the future popularity of the object, and hence reflective of the merit of keeping such an object in the cache. Recent studies [5] suggest that such relationships are weakening and hence may not be effective in indirectly capturing the popularity of Web objects.

In this paper, we (1) present an on-line algorithm that effectively captures and maintains an accurate popularity profile of Web objects requested through a caching proxy, (2) propose a novel cache replacement policy (termed GDSP) that directly uses such information to generalize the well-known GreedyDual-Size algorithm, and (3) show the superiority of our proposed algorithm by comparing it to a host of recently-proposed and widely-used algorithms with extensive trace-driven simulations using large DEC and NLANR proxy traces.

Our implementation of GDSP algorithm addresses a number of important problems, namely (a) How to capture the temporal locality exhibited in Web access streams, although it is weak? (b) How to avoid *cache pollution*—i.e., the tendency of previously popular objects to linger in the cache? (c) How to efficiently maintain the popularity profile of a large working set of Web objects? and (d) How to use such a profile to accurately estimate the long-term access frequency of individual objects?

The remainder of this paper is organized as follows. We first review earlier work on Web proxy cache replacement algorithms. Next, we describe our generalization of the GreedyDual-Size algorithm. Then we evaluate the performance of our proposed algorithm by comparing it to alternative techniques. We conclude with a summary.

2. Related work

There is a large body of work on caching in general and on Web caching research in particular. In this section, we restrict our presentation to cache replacement policies for Web proxies and servers.

Basic policies: Simple Web cache replacement policies leverage *a single basic* property of the reference stream. Least-Recently-Used (LRU) leverages temporal locality of reference—namely, that recently accessed objects are likely to be accessed again. Least-Frequently-Used (LFU) leverages the skewed popularity of objects in a reference stream—namely, that objects frequently accessed in the past are likely to be accessed again in the future. Previous studies [7] indicate that the independent reference model [10] explains well the distribution properties of Web access, supporting the use of of frequency-based policies. Largest-File-First (LFF) leverages the negative correlation that exists between object sizes and likelihood of access—small objects have a higher probability of being referenced again in the future.

Early characterizations of Web access patterns suggested the presence of strong temporal locality of reference [2, 4, 13]. However, more recent studies have concluded that this temporal locality is weakening [5]. One reason for this trend is effective client caching. To understand this, it suffices to note that the request stream generated by a client using an efficient caching policy is precisely the set of requests that missed in the client cache. Such a request stream is likely to exhibit weak temporal locality of reference—in particular, a recently accessed object is *unlikely* to be accessed again in the future! This trend suggests that LRU is not an adequate policy for cache replacement at proxies.

Early characterizations of Web access patterns suggested a strong preference for small objects [8]. However more recent studies have concluded that this preference is significantly weakening [5, 7]. Again, this weakening could be related to the presence of more efficient client caching, which tend to mask the correlation between size and frequency of access. It suggests that LFF is not an adequate policy for cache replacement at proxies.

Unlike LRU and LFF, LFU infers object popularity directly from the reference history. While caching the most popular objects would yield optimal performance, recent studies of Web access patterns suggest that the popularity of Web objects is highly bursty [4, 13]. Objects that are popular over short time scales are not necessarily popular over longer time scales (and vice-versa). This property limits the performance of LFU due to the cache pollution phenomenon to which we alluded earlier.

To summarize, the unique characteristics of Web accesses patterns observed at caching proxies (e.g., variable-size objects, variable-cost requests, burstiness of access stream, weakening temporal locality, etc.) limit the effectiveness of basic cache replacement algorithms.

Hybrid policies: Several studies have generalized LRU to make it more sensitive to the variability in object size and retrieval delays. The GreedyDual algorithm [23] was proposed to deal with variable-cost uniform-size page caching problem. Cao and Irani [9, 15] generalized the GreedyDual algorithm to deal with variable-size Web objects. The resulting algorithm, GreedyDual-Size (GDS), enables a cache replacement strategy to be sensitive to both the variability in Web object sizes and retrieval costs (miss penalty). The GDS implementation described in [9] uses $cost/size$ as the *utility value* of an object. This value is deflated over time to dynamically "age" objects in the cache. GDS was proven to possess an *optimal competitive ratio*—meaning that its cost of cache misses is within K times that of an off-line optimal algorithm, where K is the ratio of the cache size to the size of the smallest object in the trace.

Other generalizations of LRU have attempted to incorporate access frequency information into LRU. LRU-K [19] computes the average reference rate of the last K accesses. It was shown to outperform LRU for database disk buffering applications. LNC-W3 [20, 21] is another generalization that incorporate object size, retrieval costs, and the average reference rate into LRU. LNC-W3 uses these aspects to compute the *profit* of caching an object.

The Hybrid algorithm presented in [22] is aimed at reducing total latency by estimating the utility of retaining an object in the cache based on the object size, load delay, and frequency. The LRV algorithm presented in [17] uses the cost, size, and last access time of an object to calculate a utility value. The calculation is based on extensive empirical analysis of trace data. These algorithms have several drawbacks. Most importantly, they are heavily parameterized, requiring extensive tuning and parameter estimation. This makes them susceptible to changes in access patterns and the location of the cache.

In another attempt to leverage access frequency, Arlitt *et al* proposed and evaluated two replacement policies—GDSF and LFU-DA [3]. GDSF simply incorporates access count into GDS. It uses $\frac{access\_count \times cost}{size}$ as its base value. LFU-DA is a special case of GDSF in which *cost* is proportional to $size$. Simulations show that GDSF(1), in which *cost* is the same for all objects, obtains the highest hit ratio while LFU-DA obtains the highest byte hit ratio. A recent paper [16] found the combination of a simple GD-LFU policy(same as GDSF(1)) and a Hybrid coherency policy obtains the lowest average cost.

3. Popularity-aware GDS cache management

One of the weaknesses of GDS is its inability to capture and leverage the knowledge of the *long-term access frequencies* of Web objects. Recent studies [5, 7, 8] have shown the prevalence of Zipf-like distributions (Power-law) in Web access characteristics. One such distribution is identified when characterizing object popularity P as a function of object rank ρ. In particular, $P \sim \rho^{-\alpha}$, $0 < \alpha < 1$. This leads to the following property: *The number of objects accessed at least k times is proportional to $k^{-1/\alpha}$.* This implies that the probability of future references is dependent on past access frequencies—suggesting the relevance of taking into consideration long-term access frequencies in cache replacement strategies. In this section, we present GDS-Popularity (GDSP), a generalization of GDS that enables it to leverage the knowledge of the skewed popularity profile of Web objects.

3.1. Overview of GDSP

We incorporate access frequency into the GDS algorithm through the use of a new *utility value* for a given object. The utility value $u(p)$ for an object p is defined as the expected normalized cost saving as a result of having p in the cache:

$$u(p) = \frac{f(p) \times c(p)}{s(p)},$$

where $s(p)$ is the size of p, $c(p)$ is its retrieval cost (i.e. miss penalty), and $f(p)$ is its access frequency. Thus, $u(p)$ represents the cost saved per byte of p as a result of all accesses to it in a given period of time.

To capture access recency and to avoid pollution by previously popular objects, we use a dynamic aging mechanism similar to that used by GDS. In particular, we represent the *cumulative value* of an object p by $H(p)$. The cumulative value of the object last evicted from the cache is denoted by L. Thus, an invariant of our algorithm is that $H(p) \geq L$ for any object p the cache.

The resulted algorithm is called GDS-Popularity or GDSP. Its general steps are described in Figure 1. It has nearly the same time and space cost as GDS. The object meta data can be maintained with a priority queue with key $H(p)$. The processing overhead on each hit or replacement is $O(\log n)$. Another element of overhead comes from maintaining the popularity profile and estimating the access frequency. The next subsection shows that the time and space requirements for doing so is very low.

3.2. Capturing object popularity

GDSP maintains "meta" information for a subset of the objects in the request stream. Such information includes the object size, the retrieval cost, the last access time, and the estimated access frequency. All but the last of these quantities are readily available from the request stream (e.g. HTTP headers, etc.).

```
Algorithm GDS-Popularity:
L ← 0.0
for each request for object p do
  if p is in cache
    then H(p) ← L + f(p) × c(p)/s(p)
    else while there is not enough free cache for p
              do L ← min{H(q)|q is in cache}
                 Evict q which satisfies H(q) = L
         fetch p
         H(p) ← L + f(p) × c(p)/s(p)
```

Figure 1. Pseudo code of GDSP Algorithm

Popularity information: One simplistic way of computing the relative access frequency of Web objects is to keep track of the full reference history of every Web object requested through the proxy. This is obviously unrealistic due to the huge scale of the Web. Instead, our solution keeps the popularity information, i.e. access frequency and last access time of only a small fraction of all Web objects requested through the proxy. This method allows us to bound the space used to maintain such information. In particular, in our implementation, we bound the space by satisfying two conditions: (1) less than (say) 1% of the cache is used to keep the popularity information, and (2) the total number of objects for which this information is kept is no more than (say) 20% of the total number of objects expected in a given access stream. Under such conditions, and for the NLANR and DEC traces (discussed later) we considered, the total space requirement including the auxiliary space, is only a few mega bytes.

It is important to note the necessity of keeping reference popularity information for cached and evicted objects alike. This is necessary not only to improve the accuracy of access frequency estimation, but also to avoid the pollution phenomenon, to which we alluded earlier. This phenomenon is analogous to thrashing whereby a popular, newly cached object is evicted before building up enough "inertia" (in terms of access frequency) to resist eviction due to a burst of references to an object that is popular only over a short time scale [4, 13]. The situation is exacerbated further as cached objects age—the longer the request stream, the larger the "inertia" needed to resist eviction. Since the access frequency of a new popular object is always computed from scratch, it has no fair chance to stay in the cache *unless* its reference popularity information is maintained even when the object itself is temporarily evicted.

In the remainder of this paper, we use the term "Popular Objects" to refer to the set of objects for which popularity information is maintained at the proxy at any particular point in time. Also, we use the term "Cached Objects" to refer to the set of objects cached at the proxy at any particular point in time. Note that "Cached Objects" is a subset of "Popular Objects", which are in turn a subset of all objects in the request stream.

Efficient management of popularity information: To support an efficient search for popularity information associated a given URL, a hash table (on the URL) can be used. As explained earlier, we maintain entries in this hash table for only popular objects. To that end, we employ a replacement policy that evicts the least frequently accessed entry. A faithful implementation of such a policy would require the maintenance of a priority queue with access frequency as the key. This results in an expensive $O(\log n)$ time cost for each replacement. Fortunately, since there is a significant number of entries with identical frequency, an efficient approximation is possible. Namely, by aggregating the entries with the lowest frequency in a linked list, our implementation selects the oldest such entry as a candidate for eviction. This implementation needs only $O(1)$ time for each replacement.

Frequency computation: As described earlier, we need to keep track of the access frequency for popular objects. Keeping a reference count, while simple, may result in some inaccuracies. Below, we discuss two such inaccuracies and present the refinements adopted in our implementation.

We denote by $f_i(p)$ the access frequency estimate for object p after being accessed i times since its inclusion as a popular object.

First, access frequency estimates are time varying. To account for this, a mechanism must be adopted to give preference to more recent references in prediction. In our implementation, we use a decay function to de-emphasize the significance of past accesses. In particular, on the $(i+1)$-th reference to an object p, its frequency is iterated as:

$$f_{i+1}(p) = f_i(p) \times 2^{-t/T} + 1,$$

where t is the elapsed time since the last reference and T is a constant that controls the rate of decay. In our experiments, we set $T = 2$ days.[1]

Second, the Zipf-like nature of popularity distribution implies that there can be arbitrarily many accessed-once objects in a request stream. The probability of future accesses to such objects is very low. To account for this, a mechanism must be adopted that de-emphasizes accesses made to unpopular objects. In our implementation, we discount the significance of the first access to an object. In particular, the weight of a first reference is set to $f_1 = 1/3$. This value was chosen because, in the traces we considered in our experiments, the fraction of objects that were referenced twice or more was around $1/3$.

[1] The frequency value reflects the merit of keeping an object over a long period. The significance of past accesses should not decay too fast. As for the traces considered in our experiments, it is usually suitable to half the effect of a reference every at least one day.

4. Performance evaluation

In this section, we present the results of extensive trace-driven simulations that we have conducted to evaluate the performance of GDSP.

4.1. Traces used

In our trace-driven simulations we used traces from DEC[12] and NLANR[18]. We only present the results obtained from the first week (08/29 - 09/04, 1996) of the DEC trace (results from the other weeks were similar). We have also run our simulations with traces from a multitude of NLANR proxy sites since April, 1999. Since the results of our simulations were similar across all sites, we present here only the results obtained from the site UC traces(April 7-10, 1999). Some of the characteristics of these traces are shown in Table 1. Note, HR_∞ and BHR_∞ are the hit ratios when the cache size is infinite, i.e., the upper bound of the ratios.

Table 1. Traces used in our simulations

Traces	DEC	NLANR
All requests	3,543,968(44.9GB)	4,278,480(62.4GB)
Unique files	1,354,996(21.9GB)	1,464,799(30.7GB)
HR_∞	48.7%	55.8%
BHR_∞	35.8%	50.1%

Our preprocessing of the DEC traces followed the same procedures described in [9]. In particular, we excluded non-cache-able requests, including cgi-bin requests and queries. In addition, in our experiments, we count a request as a hit if the last modification times of the cached object and the actual reply to users are the same when both are known, or if the object size has not changed when both last modification times are unknown.

Our preprocessing of the NLANR traces was more elaborate. The NLANR traces include many IMS (If-Modified-Since) and REFRESH requests with a reply code of “304” (Not Modified). In order to include such requests in the workload, we had to find the sizes of the objects of such requests. We do so through a 2-pass scanning of the entire trace. This process was 96%-successful in identifying cache-able requests (The remaining 4% were IMS and Refresh requests for which we were unable to identify the object sizes). In addition to this preprocessing, we have also excluded non-cache-able requests, including cgi-bin requests and queries.

4.2. Performance metrics and algorithms

Our performance evaluation metrics reflect the various objectives of the proxy caching algorithms. We considered three metrics: Hit Rate (HR), Byte Hit Rate (BHR), and Latency. Optimizing HR aims to maximize the fraction of all requests found in the cache. Optimizing BHR aims to minimize the total traffic between the proxy and servers. Optimizing latency aims to minimize the average response time perceived by end-users. To achieve these objectives, one must tune what a proxy perceives as the miss penalty. In particular, to optimize HR, one should treat all misses as having identical cost. We refer to this by the *constant cost* assumption. To optimize BHR, one should relate the miss penalty to the size of the missed object (in number of packets, defined as $2 + \frac{size}{536}$). We refer to this by the *packet cost* assumption. To minimize latency, one should relate the miss penalty to the latency of retrieving the missed object from the server. We refer to this by the *latency cost* assumption.

We compared GDSP to LRU, LFU, GDS, and GDSF algorithms. LRU and LFU were selected because they represent widely used policies that exploit fundamental characteristics of the request stream—LRU capitalizes on recency of access and LFU capitalizes on long-term access frequency. We excluded algorithms that were known to be inferior to GDS as established in [9]. These include the SIZE-based, Hybrid [22], and LRV [17] algorithms. Given that GDSF [3] (like our proposed GDSP algorithm) is an extension of GDS, which enables it to account for access frequency, we have also included comparisons of GDSF and GDSP.

An important aspect of LFU is the policy used for eviction when two objects have the same access count. To that end, a *tie breaker* is necessary. In our simulations, the last access time is used as the tie breaker. This also means that to some extent the LFU algorithm considers access recency. It is not clear whether different tie breakers lead to the different performance between the LFU algorithms in [7] and this paper. Even if no tie breaker is used, the hit ratios of LFU in our experiments were not as low as those in [7].

GDS is a family of algorithms, each with a different definition of *cost*. Three versions of the GDS algorithm, GDS(1), GDS(packets), and GDS(latency) are simulated, reflecting the constant cost, packet cost, and latency cost assumptions described above. Clearly, if the cost of transferring each byte is the same (i.e. retrieval cost is proportional to object size), then the GDS algorithm degenerates into LRU. This implies that the performance of GDS(packets) will be close to that of LRU since the number of packets is roughly proportional to the object size. Similar to GDS, we also consider three versions of our GDSP algorithm—namely, GDSP(1), GDSP(packets), and GDSP(latency).

In simulations, we varied the cache size from less than 0.5% to 30% of the total unique file size. This corresponds to hundreds of MB to 10GB. Too large cache size (e.g., close to 100% of total unique file size) makes any algorithms travial. It is also unrealistic since the aggregated size

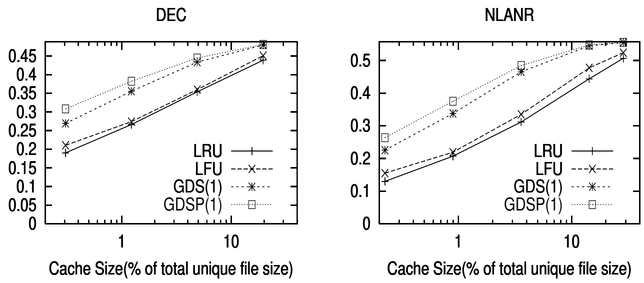

Figure 2. HR under the constant cost assumption.

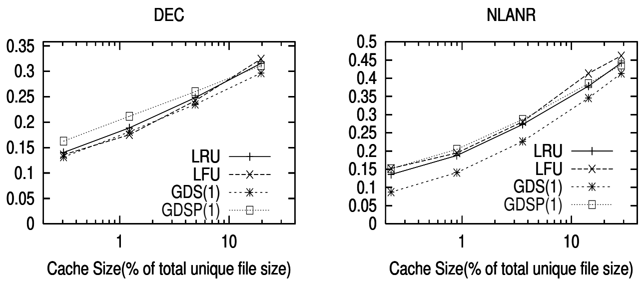

Figure 3. BHR under the constant cost assumption

of the unique files requested through a proxy is not bounded (e.g., in one month this aggregated size for the NLANR site UC was up to 200GB), while the proxy server resources are restricted in terms of both memory and CPU.

4.3. Performance under constant cost assumption

First, assume that the objects have the same cost. We compare LRU, LFU, GDS(1), and GDSP(1). Figures 2 and 3 give the ratios (y-axis) for these algorithms as a function of the cache size (x-axis, logarithmic scale). We show HR_∞ and BHR_∞ as upper bounds on performance when the cache size is large enough (the total unique file size).

For the DEC trace, LFU and LRU have the lowest HR. They are far worse than GDS(1) and GDSP(1)—e.g. when cache size is 1GB, LRU's HR is 35.4% and LFU's HR is 36.0% while both GDS(1) and GDSP(1) obtain about 44%. This is because LRU and LFU are not sensitive to object sizes. The BHR of LFU is low when the cache size is small; it increases the fastest when cache size increases. This may be due to two reasons: (1) when cache size is larger, popular objects have a better chance of being hit again, thus increasing their likelihood of staying in the cache, and (2) since LFU uses the last access time as the tiebreaker, a larger cache allows LFU to benefit from temporal locality.

GDSP(1) consistently outperforms GDS(1), especially with small cache. GDS(1) has the lowest BHR. This is not surprising since GDS(1) favors small files independent of their popularity—thus, a large popular object stands less chance of being cached under GDS(1). On the other hand, GDSP(1) achieves superior HR without significant degradation in BHR. This is due to GDSP's sensitivity to access frequency, which enables it to cache large popular files.

For the NLANR trace, the results are similar.[2] The HR of GDSP(1) is consistently the highest. Its BHR is lower than LFU only when the cache is very large, but much higher than GDS(1). The disadvantage of GDS(1) in BHR is more obvious for this trace. Another difference is the consistently better performance of LFU when compared to LRU. This is probably due to the weaker temporal locality in the NLANR trace (compared to the DEC trace). This weakening in temporal locality (from 1996 to 1999) is in line with the findings in [5]. Also, this weakening may be due to the diversity of users of upper-level NLANR proxies.

To summarize, when HR is the main objective, GDSP(1) is the best choice. It outperforms GDS(1) without significantly compromising BHR.

4.4. Performance under packet cost assumption

Figures 4 and 5 show the hit ratios when the cost is the number of packets transferred. Figure 6 shows the number of packet transfers due to the various algorithms. We compare LRU, LFU, GDS(packets), and GDSP(packets).

For the DEC trace, both hit ratios for GDS(packets) and LRU are close. This is because when the cost is roughly proportional to object size, GDS(packets) is nearly equivalent to LRU. GDSP(packets) consistently outperforms the others—in terms of both hit ratios and packet transferred. The relative BHR improvement of GDSP(packets) over GDS(packets) and LRU is as much as 15%. The relative HR improvement is as much as 30% when the cache is small.

For the NLANR trace, the results are similar. The BHRs of GDS(packets) and LRU are nearly equal. GDS(packets) outperforms LRU slightly in HR. This difference is due to the fact that $\frac{cost}{size}$ is not an exact constant and GDS(packets) slightly favors small objects, resulting in increased hits. LFU is the closest to GDSP(packets) when the cache is large. LFU does well due to the weak temporal locality.

When cache size is larger than 4% of the total unique file size, GDSP(packets) is superior to GDS(packets) along several performance metrics. (1) The relative improvement with respect to HR and BHR is 20% and 17%, respectively. For example, for the NLANR trace, when cache size is 1GB, the HR and BHR of GDS(packets) are 33.7% and 27.5%, respectively, whereas those of GDSP(packets) are 40.3% and 32.1%, respectively; When cache size

[2] For the NLANR trace, we do not count a REFRESH request as a hit since the proxy must contact the server. However, we count the bytes hit since a server's 304 reply does not lead to an object transfer. This assumption does not change the relative performance of the algorithms in simulations.

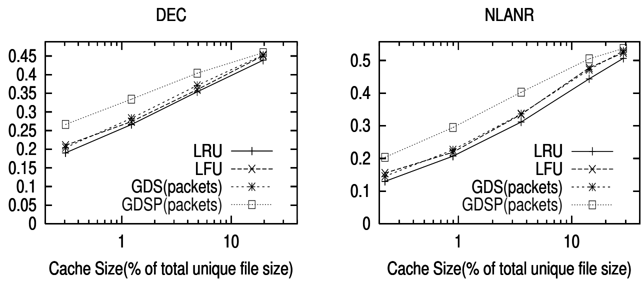

Figure 4. HR under the packet cost assumption

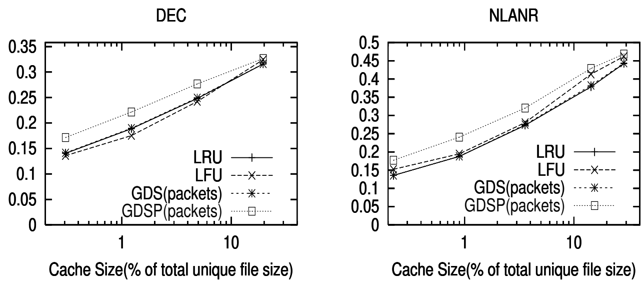

Figure 5. BHR under the packet cost assumption

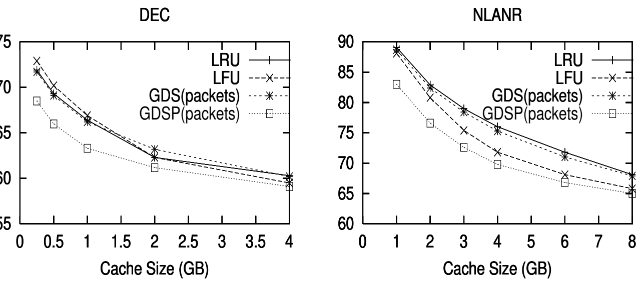

Figure 6. Packets(mega) under packet cost assumption

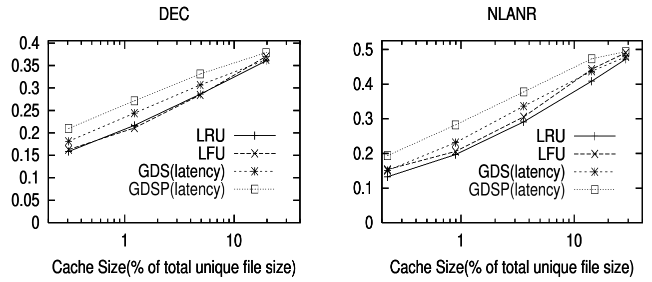

Figure 7. LSR under the latency cost assumption

is 4GB, the BHR of our algorithm is 42.9%, compared with 38.2% of GDS(packets). (2) GDS(packets) needs about twice the cache size to obtain the same HR and BHR of GDSP(packets). (3) GDS(packets) produces 8% more packets on the network for the NLANR trace and same cache size as shown in Figure 6 (right). When cache size is 1GB, GDS(packets) needs 88.6M packets while GDSP(packets) needs 83.0M packets; when cache size is 4GB, GDS(packets) needs 75.2M packets while GDSP(packets) needs only 69.8M packets.

4.5. Performance under latency cost assumption

The retrieval delay for fetching an object from a remote server can be modeled by: $c(p) = t_{conn} + t_{byte} \times s(p)$, where t_{conn} is the time to establish the connection and t_{byte} is the average time to transfer a byte. We simply estimate these two parameters for all servers in the trace, instead of determining these parameters for each server separately.[3] We do so by computing the average delay for objects of different sizes and estimating the parameters with a *least-square fit*. For the DEC trace, we computed $t_{conn} \approx 1.5 secs$, and $t_{byte} \approx 0.00021 sec/byte$. For the NLANR trace, we computed $t_{conn} \approx 1.7 secs$, and $t_{byte} \approx 0.00013 sec/byte$.

[3] We had originally attempted to compute these parameters on a per-server basis using the techniques proposed in [22] and used in [9]. However, our findings revealed wide inaccuracies. We suspect that these inaccuracies are due to the variability in network conditions as observed in [11] and server load conditions as observed in [6].

Figure 7 shows the latency saving ratio (LSR) for both DEC and NLANR traces under LRU, LFU, GDS(latency), and GDSP(latency). The results show that latency reduction is minimal for LRU and LFU. GDSP(latency) clearly outperforms GDS(latency). The relative improvement over LRU and LFU is up to 25%; the relative improvement over GDS(latency) is higher than 10%.

4.6. Comparison to GDSF

As suggested in [3] and [16], a simple generalization of the GDS algorithm uses the exact access count as $f(p)$ in our algorithm and does not maintain a popularity profile for accurate frequency computation. This algorithm is called GDSF in [3] and GD-LFU in [16]. Studies have shown that this algorithm outperforms GDS. In this section we compare GDSF and GDSP.

Figure 8 gives the hit ratios of GDSF(1), GDSF(packets), GDSP(1), and GDSP(packets) for the NLANR trace. The results are similar for the DEC trace and are not included here for space limitations. As evident from Figure 8, the GDSP algorithms consistently outperform the corresponding GDSF algorithms, which in turn are only slightly better than the corresponding GDS algorithms. These findings confirm the value of GDSP's popularity profile maintenance and frequency estimation techniques.

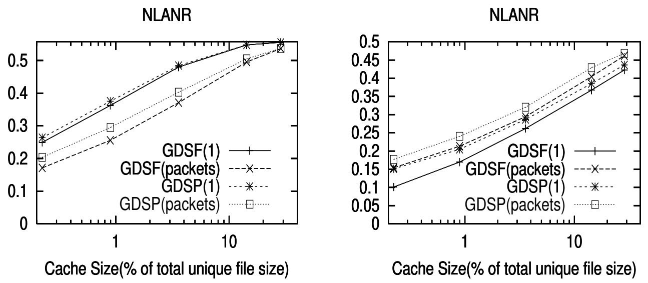

Figure 8. HR(left) and BHR(right) of GDSP vs GDSF

5. Summary

Popularity information is an important factor for effective Web cache replacement policies. In this paper, we presented an on-line policy that effectively captures and maintains an accurate popularity profile of Web objects requested through a caching proxy and designed a novel cache replacement algorithm that utilizes such information. To exploit temporal locality exhibited in the Web traffic as well as to avoid cache pollution by previously popular objects, the algorithm generalizes GreedyDual-Size by incorporating frequency information. A popularity profile of Web objects requested through the proxy is maintained efficiently, which makes it possible to accurately estimate the long-term access frequency of individual objects. Our evaluation using extensive trace-driven simulations and different performance metrics quantified the benefits and established the superiority of our proposed algorithms.

References

[1] Akamai Technologies. Freeflow content delivery system. Available at http://www.akamai.com.

[2] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing Reference Locality in the WWW. In *Proceedings of International Conference on Parallel and Distributed Information Systems*, December, 1996.

[3] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for Web proxy caches. In *Proceedings of the 2nd Workshop on Internet Server Performance*, May, 1999.

[4] Martin Arlitt and Carey Williamson. Internet Web servers: Workload characterization and implications. *IEEE/ACM Transactions on Networking*, 5(5): 631-644, 1997.

[5] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web client access patterns: characteristics and caching implications. *WWW Journal*, 2(1): 3-16, 1999.

[6] Paul Barford and Mark E. Crovella. Measuring Web performance in the wide area. *Performance Evaluation Review*, Special Issue on Network Traffic Measurement and Workload Characterization, August, 1999.

[7] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: evidence and implications. In *Proceedings of Infocom'99*, April, 1999.

[8] Carlos Cunha, Azer Bestavros, and Mark Crovella. Characteristics of WWW client-based traces. Technical Report BU-CS-95-010, April, 1995.

[9] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of USENIX Symposium on Internet Technology and Systems*, December, 1997.

[10] E. G. Coffman and P. J. Denning. Operating Systems Theory. Prentice-Hall, 1973.

[11] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[12] Digital Equipment Corporation. Digital's Web proxy traces, september, 1996. Available via ftp, ftp://ftp.digital.com/pub/DEC/traces/proxy/.

[13] Steven Gribble and Eric Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proceedings of USENIX Symposium on Internet Technology and Systems*, December, 1997.

[14] Infolibria Inc. Dynacache and mediamall caching solutions. Available at http://www.infolibria.com

[15] Sandy Irani. Page replacement with multi-size pages and applications to Web caching. In *Proceedings of the 12th annual ACM symposium on Theory of computing*, May, 1997.

[16] Balachander Krishnamurthy and Craig E. Wills. Proxy cache coherency and replacement–towards a more complete picture. In *Proceedings of the 19th IEEE ICDCS'99*, June, 1999.

[17] P. Lorenzetti, L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. Technical Report LR-960731, Univ. di Pisa, http://www.iet.unipi.it/∼luigi/research.html.

[18] National Laboratory for Applied Network Research. Weekly access logs at NLANR's proxy caches. Available via ftp, ftp://ircache.nlanr.net/Traces/.

[19] Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of ACM SIGMOD*, May, 1993.

[20] Peter Scheuermann, Junho Shim, and Radek Vingralek. A case for delay-conscious caching of Web documents. In *Proceedings of the 6th International WWW Conference*, April, 1997.

[21] Junho Shim, Peter Scheuermann, and Radek Vingralek. Proxy cache algorithms: Design, implementation, and performance. *IEEE Transactions on Knowledge and Data Engineering*, 11(4): 549-562, July/August, 1999.

[22] R. Wooster and M. Abrams. Proxy caching that estimates page load delays. In *Proceedings of the 6th International WWW Conference*, April, 1997.

[23] Neal E. Young. On-line caching as cache size varies. In *Proceedings of Symposium on Discrete Algorithms*, January, 1991.