Adaptive Web Proxy Caching Algorithms

Geetika Tewari
and
Kim Hazelwood

TR-13-04

Computer Science Group
Harvard University
Cambridge, Massachusetts

Adaptive Web Proxy Caching Algorithms

Geetika Tewari and Kim Hazelwood
Harvard University
{gtewari,hazelwood}@eecs.harvard.edu

Abstract

Web proxy caching is a well-known technique for reducing access latencies and bandwidth consumption. As in other caching systems, a replacement policy is necessary for determining when and what to evict from the cache, and many proxy caching algorithms have been proposed and evaluated over the years. Yet many of these algorithms were motivated by the Internet traffic trends in place during the design of the algorithms (1997–1999). The recent proliferation then subsequent decline in the use of peer-to-peer systems suggests that the performance of proxy caching algorithms may have changed dramatically since their inception. This paper analyzes the distribution of current web content and re-evaluates various proxy cache replacement algorithms including LFU, LRU and several GreedyDual variants. Based on the change in Internet traffic trends identified, two new web caching algorithms are proposed: a local policy that maintains a list of popular URLs and a global policy that partitions the cache into distinct regions, each of which is sized according to the popularity of the type of object being cached. Our results indicate that for small cache sizes LFU and LRU have higher byte-hit rates than several GreedyDual variants on modern Internet traffic. We also found that maintaining a hot list improves byte-hit rate, while a cache-by-regions approach improves hit rate.

1 Introduction

Web caching is the temporary storage of remote web objects on a local server [5]. Advantages of this technique range from reduced access latencies to reduced server load and bandwidth consumption. While a replacement policy is necessary for determining when and what to evict from the cache, proxy caches have salient features that complicate the management policy, such as the potential for fragmentation, invalidations of cached data, and dramatically varying costs for cache misses.

Many proxy caching algorithms [7, 17] have been proposed and evaluated over the years—from traditional policies such as LRU, LFU, and FIFO to newer policies such as GreedyDual [19] and its variants [12, 13]. Yet, many of these algorithms were motivated by the Internet traffic trends in place during the design of the algorithms.

A 2002 study [16] revealed changes in web traffic since 1999 at the University of Washington. Results indicated that HTTP traffic had changed, with peer-to-peer traffic overtaking WWW traffic as the largest contributor to bandwidth consumption. Since that time, several news sources [1, 2] reported a measured decline in the use of peer-to-peer file sharing software such as Gnutella [15] and KaZaA [14] spurred by copyright infringement lawsuits [3] filed by the Recording Industry Association of America. These changes in the use of P2P systems warranted both a re-evaluation of web traffic trends, and a re-evaluation of the web caching algorithms that were generally driven by the trends of widely-available traces collected in 1997.

In this paper, we analyzed traces of modern web traffic from October–December 2003 and noted a decline in multimedia traffic. Through a detailed simulation of various cache-replacement policies executed using these modern Internet traces, we re-evaluated the predominant proxy cache replacement algorithms. We used the insight gathered during our analysis of the existing algorithms on modern workloads to design and implement two proxy caching algorithms.

The specific contributions of this paper are:

- A detailed evaluation of the web traffic trends of late-2003.
- A re-evaluation of existing web caching algorithms using modern web traces.
- The design and evaluation of two novel web caching algorithms designed to adapt to changes in web traffic.

The remainder of this paper is organized as follows. Section 2 discusses the problem of web caching and describes the specific challenges that distinguish the problem domain. Section 3 analyzes the modern trends in Internet traffic, and discusses implications for caching algorithms. Using a trace-driven simulator, we evaluate of many of the popular web caching algorithms that are in use today in Section 4. Next, Sections 5 and 6 introduce and evaluate HotList and GlobalRegions, two new web caching algorithms. Finally, Section 7 makes recommendations for future web caching design algorithms and concludes the paper.

2 Background and Related Work

Three distinct approaches to web caching currently exist, including client-side caching, server-side caching, and proxy caching. *Client-side caching* refers to caches that are built into most web browsers, which cache Internet objects for a single user, but from a variety of servers. *Server-side caching* (also known as reverse caching) refers to caches that are placed in front of a particular server to reduce the number of server requests. This paper focuses on *proxy caching*, a common form of web caching which aims to reduce the overall bandwidth consumption of a network. Proxy caches are often located near network gateways and serve many users with cached objects from many servers.

Web caching is different from conventional microprocessor caching or memory paging in several ways. First, cached web documents vary significantly in size depending on their type (text, image, video, etc.) and therefore caching algorithms may require a defragmentation step. Furthermore, same-sized pages can take different amounts of time to access, and thus caching algorithms must take into account the anticipated download latency when identifying an eviction candidate. Finally, the access stream seen by a single proxy cache is the union of several users' access streams, resulting in a Zipf-like distribution [6].

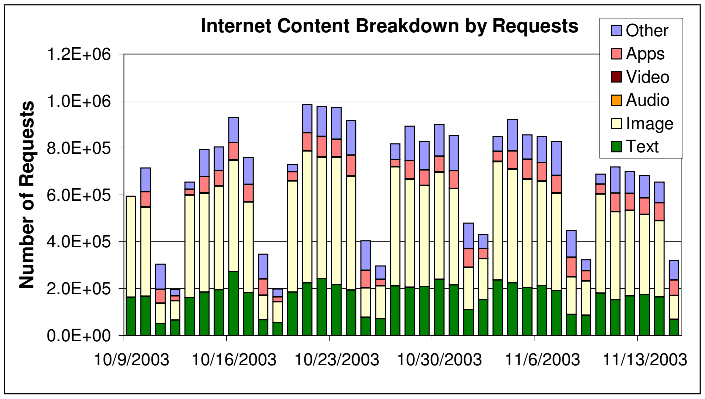
Web Proxy Caching Algorithms There has been extensive theoretical and empirical work done on exploring web caching policies that perform best under different performance metrics. Many algorithms have been proposed and found effective for web proxy caching. These algorithms range from simple traditional schemes such as Least-Recently Used (`LRU`), Least-Frequently Used (`LFU`), First-In First-Out (`FIFO`), and various size-based algorithms, to complex hybrid algorithms such as `LRU-Threshold`, which resembles `LRU` with a size limit on single cache elements, Lowest-Relative Value (`LRV`), which uses cost, size and last reference time to calculate its *utility*, and `GreedyDual` [19], which combines locality, size and cost considerations into a single online algorithm. Several studies have proposed and empirically evaluated variants on GreedyDual [7, 12, 13] that incorporate a combination of long-term popularity and frequency of access. In addition to these factors, some other studies have focused on the role of aging in web cache replacement policies [8]. Finally, theoretical work has employed randomization to derive new optimal replacement policies [4].

3 Current Internet Trends

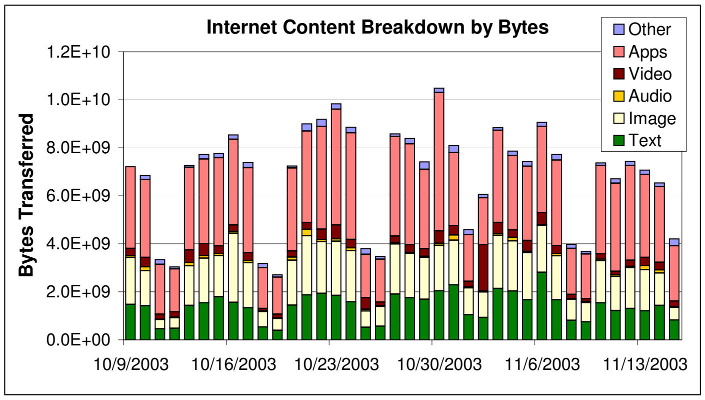
This section focuses on the current distribution of web content. Figure 1 shows the breakdown of Internet traffic by (a) number of requests and (b) number of bytes downloaded in Pittsburgh. This data was collected for six metropolitan areas, and the main trends were consistent across all cities. As Figure 1(a) shows, the majority of downloaded items are images or text. The majority of bandwidth, however, is consumed by downloading applications (Figure 1(b)). Closer inspection of our raw data indicated that the applications typically consist of patches from `windowsupdate.microsoft.com`. New caching algorithms could potentially leverage this fact.

An interesting observation from Figure 1 is that audio and video constitute a very small proportion of the total workload, indicating that users have decreased their exchange of multimedia files since 2002. A potential implication of this is that web caching algorithms should place less emphasis on caching multimedia files and greater emphasis on text and image files, but a more versatile solution is for future algorithms to be able to adapt to changing distributions.

In order to determine whether material downloaded from some specific domains should be prioritized, we sorted our access logs by domain. We found that the most commonly accessed video domains across many cities include *www.lolitampegs.com* and *www.melovempegs.com*, while the the most popular domains in general are *www.google.com* and *www.windowsupdate.com*. The consistency in these trends over multiple cities suggest that it may be useful to incorporate this popularity information into a web caching algorithm, and motivated the design of our HotList algorithm presented in Section 5.



(a) By Number of Requests



(b) By Total Bytes Requested

Figure 1: Breakdown of Internet traffic load. Data is taken from Pittsburgh logs for the period of October 8 – November 8, 2003.

4 Existing Proxy Cache Algorithm Performance

In this section, we evaluate the performance of eight existing web caching algorithms, outlined in previous studies [18], including five standard algorithms and three variants of GreedyDual-Size.

4.1 Baseline Algorithms

The existing proxy caching algorithms we implemented are briefly described in the following subsections.

Least-Recently Used (LRU) Evicts the item in cache that was not requested for the longest amount of time.

Least-Frequently Used (LFU) Evicts the item in cache that has been requested the fewest number of times.

First-In First-Out (FIFO) Evicts items in the order that they were inserted into the cache.

Full Cache Flush (Flush) Evicts all items when the cache fills.

Random Evicts randomly selected cache items.

4.2 GreedyDual-Size Variants

Our GreedyDual-Size implementation is based on the specification of Cao and Irani [7]. The algorithm associates a value–H–with each cached object, which is initially set to $cost/size$. When the cache is full, the algorithm evicts the object with the lowest H–min_H–and all other objects reduce their H values by min_H. If a cached page is accessed, its H value is reset to its initial value.

The GreedyDual-Size variants we implemented differ in their definition of the GreedyDual value, H:

GreedyDual-Size(1) (GDS) Sets the $cost$ of each object to 1 in order to minimize miss ratio.

GreedyDual-Size(Packets) (GDSP) Sets the $cost$ of each object to the estimated number of network packets sent and received in order to minimize the network traffic resulting from the misses. This is defined as $2 + (size/536)$, where 536 is the default maximum TCP segment size (RFC-783) [7].

GreedyDual-Size(Latency) (GDSL) Sets the $cost$ of each object to the latency that was required to download the object in order to minimize overall latency.

4.3 Experimental Framework

We collected 22 GB of compressed web access logs from the `IRCache` distributed web cache servers during October–December 2003. The web logs were collected for Boulder, Palo Alto, Pittsburgh, San Diego, San Jose, and Silicon Valley. As indicated in Figure 2, these raw web logs were converted into a format that could be input to our own trace-driven proxy cache simulator. This required eliminating unnecessary log fields and generating several new fields–such as object type, unique ID, and access latency. Our proxy cache size was varied to investigate how well each algorithm scaled under increasing cache pressure. The web cache was sized at a fixed percentage of the total workload: 0.001%, 0.005%, 0.01%, 0.05%, 0.1%, 0.5%, 1.0% and 5.0%, where the largest cache size is on the order of several gigabytes.

We compare the performance of the algorithms based on **hit ratio**, **byte-hit ratio**, and **latency reduction**. The hit ratio is the number of requests that hit in the proxy cache as a percentage of the total requests. The byte-hit ratio is the number of bytes that hit in the proxy cache as a percentage of total bytes requested. Finally, latency reduction measures the percentage of the download times of objects that hit in the cache.

From a user's perspective, latency reduction is the most important metric. From a network perspective, however, byte-hit ratio is most important as it results in reduced bandwidth consumption. Finally, hit ratio is a good indicator of the effectiveness and fairness of a cache replacement policy.

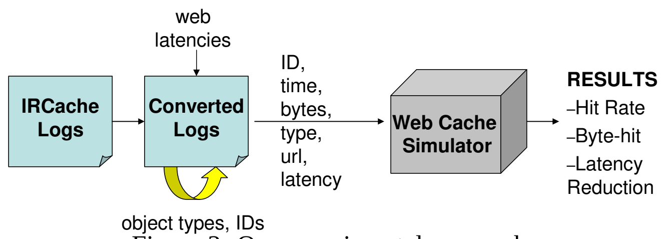


Figure 2: Our experimental approach.

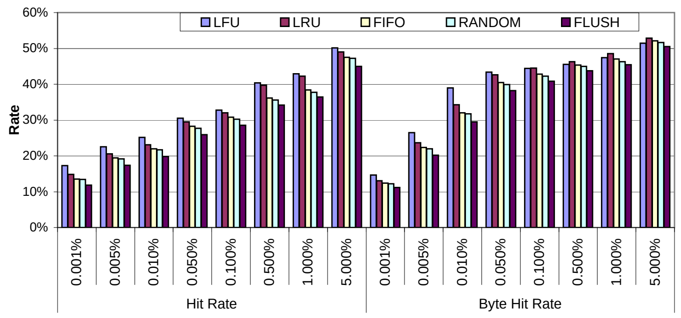


Figure 3: Baseline algorithm performance.

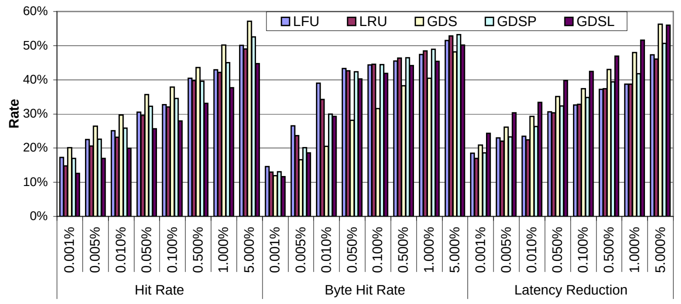


Figure 4: Performance of LFU, LRU and GreedyDual-Size variants.

Latency information was not available in the original input logs, therefore we calculated the latency with respect to Harvard University. Latency was estimated as $TcpHandshakeTime * size$, where $size$ is measured in bytes. We collected the TCP handshake time for a request of zero bytes from every unique domain in our web logs. For domains that were unavailable, we assumed the average latency across all domains (calculated to be 0.708058 seconds). Since we were unable to run our latency programs from the actual source city and we do not take outgoing link bandwidth into account, our latency numbers should be considered a relative approximation of the actual latencies.

4.4 Experimental Results

Figures 3–4 show the results of the eight cache replacement policies described in Sections 4.1 and 4.2 executed on web traces from the six metropolitan areas listed in Section 4.3, averaged across all cities. Of the five baseline algorithms shown in Figure 3, it is clear that `LFU` had the best hit rate across all cache sizes. For smaller cache sizes, `LFU` had the highest byte-hit rate until we reached 0.5%, when `LRU` began having the highest byte-hit rate. The `Flush` policy performed worst across almost all cache sizes. This is to be expected, as `Flush` was designed for a simple implementation rather than to maximize hit rate.

Figure 4 evaluates three variants of the GreedyDual-Size algorithm, and compares their performance to that of the best performers of the baseline algorithms – `LFU` and `LRU`. Confirming observations in previous studies [7], `GDS` consistently had the highest hit rate. In terms of byte-hit rate, however, performance varied by cache size. For smaller cache sizes, `LFU` and `LRU` out-

HotList in LRU

```
doReplacement(){
  While free space < new object size
    Find next LRU object, x
    If x is on hot list, find next LRU object
       If no other LRU object found
          Evict item from bottom of hot list
    Else evict x
  EndWhile
}
```

Figure 5: Pseudocode for HotList in LRU.

performed GDS, GDSP and GDSL. When cache sizes increased, GDSP began to slightly outperform LFU and LRU. Therefore for high cache-pressure environments, it seems more appropriate to implement LFU or LRU rather than a GreedyDual-Size variant in order to maximize byte-hit rate. Furthermore, since the byte-hit rate of GDSP does not differ significantly from the byte-hit rates of other baseline algorithms across larger cache sizes, it may be best to choose a more consistent performer, such as LRU.

Since GDSL is optimized for latency effects, we evaluated each GDS-variant in terms of the latency reduction, and also report the results in Figure 4. As expected, GDSL achieves the highest latency reduction, at the cost of lower hit rates than GDS and lower byte-hit rates than GDSP.

These results suggest that popular [7, 12, 13] GreedyDual based algorithms may not be optimal for caching modern Internet traffic, and motivate the investigation of new web caching strategies. Therefore, in the following sections, we present the design of two caching algorithms. These algorithms can be adjusted to account for Internet trend changes in the future.

5 HotList

Our first algorithm, HotList, is motivated by Section 3, where we note that many of the same websites are consistently popular across various U.S. cities. We hypothesize that items that have been popular in the past are likely to be valuable in the future. Many researchers have studied algorithms for determining popular items in an online manner [9, 10, 11]. We extend their work by exploring the trade-offs of using these techniques in proxy caching algorithms, and empirically demonstrating their effectiveness on current web proxy workloads.

5.1 The HotList Algorithm

HotList is designed to complement some of the most effective baseline algorithms. The main idea is to allow a hot list of the *n*-most popular items in the data stream to be given priority in the eviction policy. This approach can be implemented in conjunction with an existing replacement policy as illustrated in Figure 5 where LRU is the underlying replacement policy. While Figure 5 features LRU, any replacement policy can be used as the basis for this algorithm.

5.2 HotList Implementation

The challenge in implementing an accurate hot list is maintaining the *n*-most popular items in the data stream without having to update a count for every unique item in the data stream. A Synopsis data structure such as a *concise sample* or a *counting sample* is an effective and accurate solution for maintaining a

Counting Sample Algorithm

```
X = 1
for all t in the dataset
  if t already in the list
     increment its count
  else
     add t to the list with probability 1/X
  if list overflows
     X' = X * factor
     for each unique s in the list
         flip a biased coin, decrementing the count on each
         tails, until a head is flipped.
     X  = X'
```

Figure 6: Counting Samples: A sequential approximate sampling algorithm. In our experiments, we set factor to 3.

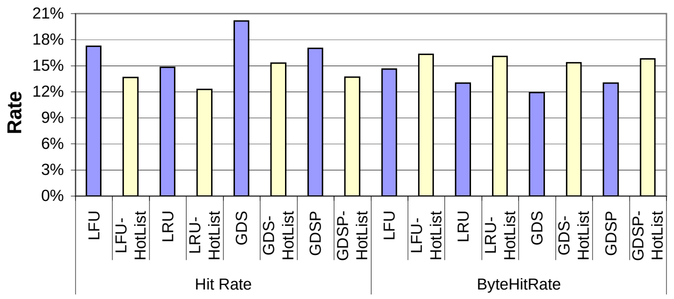


Figure 7: Performance comparison between baseline algorithms and their HotList counterparts. The cache size is 0.001% of the total bytes requested and the hot list size is 200 items.

sample of heavy hitters in a data stream [11]. Synopsis data structures work in an online manner, which is appropriate for our application as well as any other streaming application. They compute approximate (bounded error) frequencies, while automatically adjusting to changing trends in the web-access stream. Counting samples, shown in Figure 6, provide stronger guarantees on the counts than concise samples [11].

In addition to synopsis data structures, we also considered data structures such as Count-Min sketch [9], and multistage filters with conservative update [10], as potential candidates for maintaining a hot list. While a Count-Min sketch can serve as a plausible data structure for maintaining a hot list, it is more difficult to implement than synopsis structures. And while multistage filters are straightforward to implement, they are more suitable for reporting whether the count of items in the data stream exceed a certain numeric threshold, rather than identifying the *top-n* items in the stream.

5.3 HotList Performance

We incorporated a hot list into the best performing baseline algorithms: LFU, LRU, GDS and GDSP. Figure 7 shows a performance comparison of the baseline algorithms and their HotList counterparts. The results indicate that while incorporating a hot list in the baseline algorithms lowered the hit rate on average by 3.57%, the byte-hit rate improved by 2.74% on average. This could be because the hot list contained very large objects causing fewer objects to fit into the cache, and resulting in higher byte-hit rates than hit rates.

Several factors have important implications on the perfor-

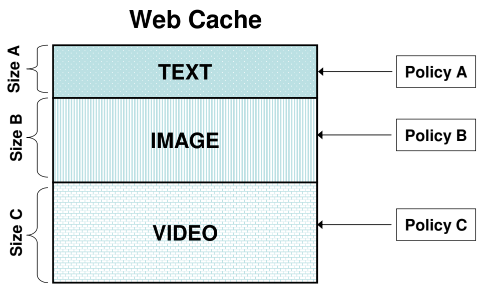


Figure 8: Conceptual view of GlobalRegions in a web cache.

mance of HotList. First, the size of the hot list should be sufficiently large in relation to the total cache size in order to have a visible impact on performance. We empirically determined that the trend displayed in Figure 7 remains only if we increase the size of the hot list as we increase the size of the cache. The performance improvements from this, however, are offset by the cost of referencing the list. Second, the algorithm used to implement the hot list dictates the accuracy of the hot list items and the amount of time spent maintaining the list. For instance, while counting samples produce a somewhat more accurate sample than concise samples, they are slower to update than concise samples. Such factors should be taken into account when designing and optimizing a hot list.

6 GlobalRegions

Our next algorithm, shown in Figure 8, works to combine several existing cache replacement policies into a single global policy.

Currently, most web caches consist of a single unified cache. A single cache eviction policy is applied regardless of the different web object types interspersed throughout the unified cache. An alternate approach is to partition the proxy cache into multiple, distinct regions, each of which stores a single object type.

We call this approach *GlobalRegions*. Aside from storing a single internet content type, each region can apply its own replacement policy independently of the other regions. Furthermore, each region can make up a different percentage of the overall proxy cache. The intended advantage is that caching large objects, such as video files or applications, will not interfere with smaller objects, such as text or images. Furthermore, the replacement policy in each region can be customized to the access patterns of a single type of internet content.

6.1 GlobalRegions Design Decisions

There are several design decisions that must be made before implementing a proxy cache with global regions. In this section, we discuss our approach for making each of these decisions.

Table 1: Global cache region proportions for each configuration.

Object	Balanced	Requests	Bytes
Text	16.7%	30%	15%
Image	16.7%	40%	25%
Video	16.7%	2.5%	7%
Audio	16.7%	2.5%	3%
App	16.7%	10%	45%
Other	16.7%	15%	5%

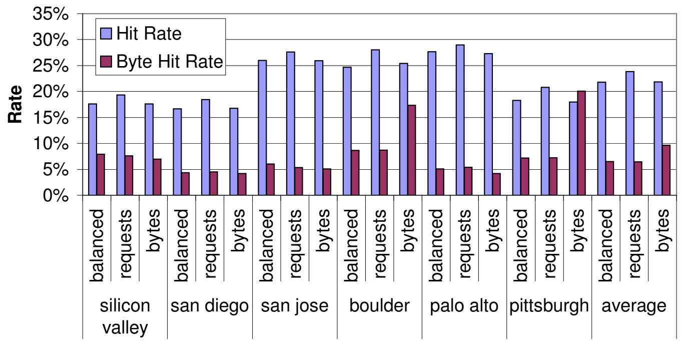


Figure 9: Evaluation of the three proportion configurations for global cache regions.

Number of Cache Partitions We chose to incorporate six distinct regions in GlobalRegions, including `TEXT`, `IMAGE`, `VIDEO`, `AUDIO`, `APPLICATIONS`, and `OTHER`. We felt that this partitioning was a good match for today's trends depicted in Figure 1 and our design is completely extensible to incorporate any future object types.

Top Algorithms per Region Next, we determined the top replacement algorithms for each of our six object types. We conducted this study by determining the best replacement policy for an object type's input stream and ignoring all of the cache accesses that were not of the selected object type. It is important to point out that implementing `LFU` for all six regions is actually very different than implementing a unified `LFU` policy. A regional `LFU` policy only considers evicting the `LFU` item that is of the same type as the object to be inserted.

We found that `LFU` was the best replacement policy for *text*, *images*, *applications*, and *other*, while `LRU` and `FIFO` outperformed other replacement strategies for *video* and *music*.

Cache Partition Proportions Our final design decision was to determine the best proportions for each region in the proxy cache. We evaluated three configurations. Our first configuration, which we called *balance*, sized each of the six partitions equally, at *cacheSize/6*. Our second configuration, *requests*, was motivated by Figure 1(a) which compares the number of requests for each object type. We used the proportions of requests for each object type to determine the size of their cache regions. Our final configuration, *bytes*, was motivated by Figure 1(b), where we found that nearly half of all of the internet bandwidth was used to download application files, and therefore sized each region according to these bandwidth trends. The final proportions of each cache region are shown in Table 1 for each configuration.

These proportions were evaluated and are compared in Figure 9. The figure shows several interesting results. First, we see that on average, the byte-hit rate was maximized by setting our cache size proportions using the `bytes` configuration, which not coincidentally, was optimized for the bytes traveling over the wire. Similarly, the configuration resulting in the best hit rate, on average, was the `requests` configuration. Again, this makes sense, because the `requests` configuration was based on the actual number of requests made for each object type. Therefore, when partitioning a cache to employ GlobalRegions, region

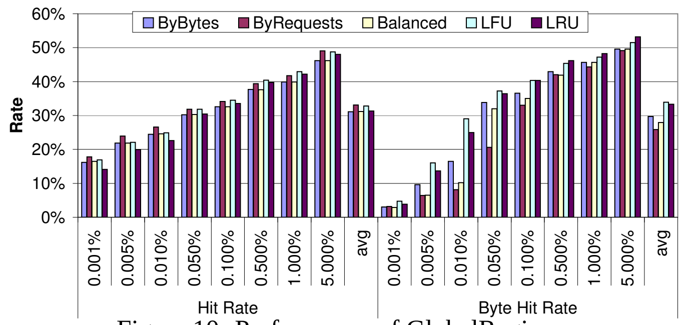


Figure 10: Performance of GlobalRegions.

sizes should be allocated according to the desired optimization criteria.

6.2 GlobalRegions Performance

We evaluated the performance of GlobalRegions by comparing the hit rate and byte-hit rate of a partitioned cache with a unified cache performing `LFU` or `LRU`. We chose these two because they were some of the top performers of the eight baseline replacement algorithms we tested. Our partitioned cache was sized such that the sum of all six regions was equal to the size of the single unified cache. Furthermore, the replacement algorithms used for each cache region were as follows: Text–`LFU`, Image–`LFU`, Video–`FIFO`, Audio–`LRU`, Application–`LFU`, and Other–`LFU`.

The results, shown in Figure 10, reflect several interesting trends. First, employing GlobalRegions and partitioning the cache using our `requests` configuration performs better than the unified policies in terms of hit rate. For byte-hit rate, however, employing a unified policy performs better than all of the partitioned cache configurations. We believe that this is the result of the small size of certain partitions in GlobalRegions. For instance, video files have been allocated 2.5–16% of the overall cache size, which is already less than 1% of the actual byte traffic of the internet trace. It is likely that much of the byte-hit rate reduction is due to the large items that can no longer fit in this cache region. In support of this, Figure 10 shows a relative increase in the byte-hit rate as the cache size increases. We can conclude that partitioning the cache into distinct regions can improve the hit rate of proxy caches, and for large caches, can approach the byte-hit rate potential of a unified `LFU` cache.

7 Conclusions

Designing effective web caching algorithms is particularly challenging in light of the ever-changing trends in Internet content and usage. In this paper, we characterized today's Internet trends and the resulting impact on web caching algorithm design. We found that images and text files constitute a majority of downloaded items in our modern traffic traces, while downloads of applications such as operating system patches dominate bandwidth consumption. These changes have impacted the performance of several proxy cache replacement policies–such as GreedyDual variants, whose performance does not differ significantly from simple schemes such as `LFU` and `LRU`, and are outperformed by them in terms of byte-hit rate for small cache sizes.

The observations on Internet trends motivated the design of two adaptive proxy cache algorithms that can adjust to changing trends. HotList leverages the trend that many popular Internet sites are also popular across the country, and can operate effectively in conjunction with an underlying replacement policy. GlobalRegions leverages object-specific behavior within the cache, and supports distinct eviction policies for each object type, while reducing the cache contention from large multimedia object types. The improvements in performance—higher byte-hit rate with HotList and higher hit rate with GlobalRegions—support the importance of re-evaluating web caching algorithms in response to changing Internet traffic trends and designing adaptive web caching algorithms.

References

[1] Internet application usage continues to decline. Nielsen/NetRatings, September 2003.

[2] Music fans cut back on free file-swapping, 40% drop in Kazaa usage. Yahoo Technology News, October 2003.

[3] Record industry sues hundreds of internet music swappers. The New York Times, September 2003.

[4] O. Bahat and A. Makowski. Optimal replacement policies for non-uniform cache objects with optimal eviction. In *IEEE INFOCOM Conference on Computer Communications*, April 2003.

[5] G. Barish and K. Obraczka. World wide web caching: Trends and techniques. May 2000.

[6] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM*, pages 126–134, 1999.

[7] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.

[8] E. Cohen and H. Kaplan. The age penalty and its effect on cache performance. In *3rd Usenix Symposium on Internet Technologies and Systems*, pages 73–84, 2001.

[9] G. Cormode and S. Muthukrishnan. Improved data stream summaries: The count-min sketch and its applications. In *DIMACS Technical Report*, 2003.

[10] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. In *ACM Transactions on Computer Systems*, August 2003.

[11] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *ACM Conference on the Management of Data*, pages 331–342, 1998.

[12] S. Jin and A. Bestavros. Popularity-aware greedydual-size web proxy caching algorithms. In *20th Intl. Conf. on Distributed Computing Systems*, April 2000.

[13] S. Jin and A. Bestavros. Greedydual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams. *International Journal on Computer Communications*, 24(2):174–183, February 2001.

[14] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. Deconstructing the kazaa network. In *3rd IEEE Workshop on Internet Applications*, 2003.

[15] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[16] S. Saroui, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. Levy. An analysis of internet content delivery systems. In *Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[17] J. Wang. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review*, 25(9):36–46, 1999.

[18] R. Wooster and M. Abrams. Proxy caching that estimates page load delays. In *6th International World Wide Web Conference*, April 1997.

[19] N. Young. On-line caching as cache size varies. In *2nd ACM-SIAM symposium on Discrete algorithms*, pages 241–250, 1991.