Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy

Ludmila Cherkasova
Computer Systems Laboratory
HPL-98-69 (R.1)
November, 1998

E-mail: cherkasova@hpl.hp.com

Web, HTTP, WWW proxies, caching policies, replacement algorithms, performance

Web proxy caches are used to improve performance of the WWW. Since the majority of Web documents are static documents, caching them at WWW proxies reduces both network traffic and response time. One of the keys to better proxy cache performance is an efficient caching policy which keeps in the cache popular documents and replaces rarely used ones. This paper introduces the Greedy-Dual-Size-Frequency caching policy to maximize hit and byte hit rates for WWW proxies. Proposed caching strategy incorporates in a simple way the most important characteristics of the file and its accesses such as file size, file access frequency and recentness of the last access. Greedy-Dual-Size-Frequency is an improvement of Greedy-Dual-Size algorithm – the current champion among the replacement strategies proposed for Web proxy caches.

Internal Accession Date Only

Contents

1 Introduction

With the growth of the World Wide Web, an increasingly large fraction of available bandwidth on the Internet is used to transfer Web documents. Recent studies show a significant increase of network traffic due to HTTP requests: 40% in 1996 against 19% in 1994.

Since the majority of Web documents are static documents, caching them at HTTP proxies reduces both network traffic and request response time. HTTP proxies serve as intermediateries between the browsers on a client side and web servers on Internet. Earlier studies have shown that the hit ratio for Web proxy caches can be as high as 50% and can potentially reduce traffic up to 20%. Thus proxy caching can significantly reduce network traffic and improve end-to-end request latency seen by a client.

One of the keys to better proxy cache performance is an efficient caching algorithm which intends to keep in the cache popular documents and replace rarely used ones. Additional improvement can be achieved when the replacement strategy is combined with the decision whether some documents are worth caching or not.

There are three essential features distinguishing Web proxy caching from conventional caching in the computer systems:

- Since HTTP protocol supports whole file transfers only – Web proxy cache can satisfy request only if the entire file is cached.
- Web documents stored in a proxy cache are vastly of different sizes while conventional caches (CPU caches and virtual memory) deal with uniform size pages.
- The access stream seen by a conventional cache could strongly exhibit some algorithm specific patterns because it is generated by one or few applications currently running on a system. The access stream seen by the proxy cache is a union of access streams coming from the users sitting behind the proxy (often the number of clients can be as high as several hundreds or thousands).

Due to all of these special Web features, there are a number of replacement policies proposed for Web proxy caches by the scientific community. Some of them are quite simple (i.e. – easy to implement), and some of them are heavily parametrized or have parts that do not have an efficient implementation, and therefore in case of showing some good results, they still could rather serve as a theoretical bound of best achievable performance than a practical choice.

The other problem which complicates a fair comparison of the proposed caching (or replacement) algorithms concludes in the following. The results of trace-driven simulation studies often have a strong dependency on the traces used to run simulation. In some cases, variation in the traces could lead to contradictory simulation results.

A very good survey of currently known Web replacement policies is done in [Cao-Irani97]. We use their survey to give an overview of the current state of the art in this field in Section 2.

In [Cao-Irani97], a new algorithm, called the Greedy-Dual-Size, is proposed as a solution for the Web proxy replacement strategy. The authors make an additional effort to obtain traces used in simulation studies by different authors. This additional effort makes their results more reliable and less dependent on particular trace characteristics.

They convincingly show that the Greedy-Dual-Size algorithm, with appropriate cost definitions, outperforms currently known caching algorithms on a number of performance metrics (see more discussion on performance metrics in Section 4).

Greedy-Dual-Size is an extension of a very elegant and efficient algorithm, called Greedy-Dual [Young91], which was designed to handle uniform-size variable-cost cache replacement. Greedy-Dual-Size is extended to deal with caching and replacing of variable size documents – typical for the Web.

In this paper, we propose a new algorithm for Web proxy caching, called Greedy-Dual-Size-Frequency, which takes into account how many times the document was accessed in the past. It does improve the Greedy-Dual-Size algorithm further to reflect file access pattern. We describe it in rigorous form in Section 5.

By exploiting the same idea, we also propose an extension to Greedy-Dual algorithm in Section 7. A new algorithm, called Greedy-Dual-Frequency, incorporates the frequency count for uniform-size references and is proposed as a solution for conventional paging problem.

2 The Overview of Existing Replacement Strategies for Web Proxies

In this section, ten different known algorithms are outlined and commented on. Greedy-Dual-Size algorithm is of special interest and is described separately in Section 4.

- **Least-Recently-Used** (LRU) replaces the document which was requested least recently.
  This traditional policy is the most often used in practice and has worked well for CPU caches and virtual memory systems. However it does not work as well for proxy caches because the locality of time accesses for Web traffic often exhibits very different pattern.
- **Least-Frequently-Used** (LFU) replaces the document which has been accessed the least number of times.
  This strategy tries to keep more popular objects and replace rarely used ones. However, some documents can build a high frequence count and be never accessed again. Tradi-

tional LFU strategy does not provide any mechanism to remove such documents and this leads to cache pollution.

- **SIZE**[WASAF96] replaces the largest document.
  This strategy tries to minimize a miss ratio by replacing one large document rather than a bunch of small ones. However, some of the small documents brought to a cache may never be accessed again. The SIZE strategy does not provide any mechanism to evict such documents, which leads to pollution of the cache.

- **LRU-Threshold**[ASAWF95] is the same as LRU, except it does not cache documents larger than a certain threshold size.

- **Log(Size)+LRU**[ASAWF95] replaces the document which has the largest log(size) and is the least recently used among the same log(size) documents.

- **Hyper-G**[WASAF96] is a refinement of LFU with last access time and size considerations.

- **Pitkow/Recker**[WASAF96] replaces the least recently used document, except if all the documents are accessed today. In this case, the largest document is replaced.
  This strategy tries to catch the daily time access pattern specific for the Web documents. This replacement policy is also proposed to run at the end of the day for freeing a space occupied by "old", least recently accessed documents.

- **Lowest-Latency-First**[WA97] removes the document with the lowest download latency. It is explicitly aimed to minimize the average latency.

- **Hybrid**[WA97] targets at reducing the average latency too. For each document a utility of retaining the document in the cache is computed. The document with the smallest utility value is replaced. The utility function for a document f located at server s is defined as using the following parameters: c_s - the connection time to server s, b_s – the bandwidth to server s, fr_f – frequency count for f (i.e. number of accesses since f was brought into the cache), $size_f$ – the document f size in bytes. The utility function for a document f is defined as:

$$\frac{(c_s + K_1/b_s) \times (fr_f)^{K_2}}{size_f}$$

  where K_1 and K_2 are constants. Estimates for c_s and b_s are based on the times to get the documents from the server s in the recent past.

- **Lowest Relative Value (LRV)**[LRV97] is also based on computing a utility of retaining a document in the cache. The document with the smallest utility value is then replaced. The calculation of utility value is based on extensive empirical analysis of the trace data. For a given i, let P_i denote the probability that a document was requested $i+1$ times given that it is requested i times. P_i is estimated in online manner by taking

the ratio D_{i+1}/D_i, where D_i is the total number of documents seen so far which have been requested at least i times in the trace. $P_i(size)$ is defined similar to P_i taking into account the only documents of size $size$. Furthermore, let $1 - D(t)$ be the probability a document requested again as a function of time (in seconds) since its last access t. $D(t)$ is estimated as

$$D(t) = 0.035log(t+1) + 0.45(1 - e^{\frac{-t}{2e6}}).$$

Then for a particular document f of size $size$ and cost $cost$ requested i times in the past, with the last access t seconds ago, the utility function V is computed in the following way:

$$V(i,t,size) = \begin{cases} P_1(size) \times (1 - D(t)) \times cost/size & \text{if } i = 1 \\ P_i \times (1 - D(t)) \times cost/size & \text{otherwise} \end{cases}$$

Performance studies conducted in [ASAWF95, WASAF96] show that SIZE outperforms LFU, LRU-Threshold, Log(size)+LRU, Hyper-G and Pitkow/Recker in terms of hit ratio.

SIZE also outperforms LRU in most situations as shown in [WASAF96]. However, results in [LRV97] show that LRU outperforms SIZE in terms of byte hit ratio.

LRU outperforms LFU in most situations [Cao-Irani97]. LFU can only slightly outperform LRU for small caches.

In terms of minimizing latency, [WA97] shows that Hybrid performs better than Lowest-Latency-First.

Performance study in [LRV97] shows that LRV outperforms both LRU and SIZE in terms of hit ratio and byte hit ratio.

This has narrowed the choice of replacement policies for Web proxy caches to LRU, SIZE, Hybrid and LRV. Note that Hybrid and LRV are heavily parametrized strategies that make them to serve rather as a theoretical bound of best achievable performance than a practical choice.

The performance study conducted in [Cao-Irani97] showed that Greedy-Dual-Size algorithm is a current champion outperforming LRU, SIZE, Hybrid and LRV for different metrics and for a variety of different proxy traces used in other performance studies.

3 The Original Greedy-Dual Algorithm

The original Greedy-Dual algorithm was introduced by Young [Young91]. It deals with the case when pages in a cache (memory) have the same size but have different costs to fetch them from the secondary storage.

The algorithm associates a value, H, with each cached page p. Initially, when a page is brought to a cache, H is defined to be a cost to bring the page into the cache (note that the cost is non-negative).

When a replacement needs to be made, the page with the lowest H value, min_H, is replaced, and then all the pages reduce their cost values H by min_H. If a page p is accessed again its current cost value H is restored to the original cost of bringing this page to the cache.

In such a way, the H values of recently accessed pages maintain a larger amount of the original cost as compared to the pages that have not been accessed for a long time.

The algorithm uses the pages with the lowest H values to be replaced first. These are either "least expensive" pages to bring into the cache or the pages that have not been accessed for a long time.

There is different, an efficient implementation of the algorithm using a priority queue and keeping offset value for future settings of H. We describe it in detail in Section 4.

4 Greedy-Dual-Size Algorithm for Web Proxies

Web Proxy caching is concerned with storing the documents of different sizes. Cao and Irani in [Cao-Irani97] extend Greedy-Dual algorithm to deal with variable size documents by setting H to *cost/size* where the *cost* is the cost of bringing the document and *size* is the size of the document in bytes. They call this algorithm as Greedy-Dual-Size algorithm (abbreviation: GD-Size).

The simulation results provided in [Cao-Irani97] show that proposed algorithm outperforms other currently known algorithms for different performance metrics with accordingly chosen cost functions.

The two most common metrics used to evaluate the proxy cache performance are the following:

- Hit ratio – the number of requests satisfied from the proxy cache as a percentage of total requests.
- Byte hit ratio – the number of bytes that transferred from the proxy cache as a percentage of total number of bytes for all the requests.

To get the best hit ratio, the *cost* function for each document has to be set to 1. In such a way, larger documents have a smaller priority key than the smaller ones, and are likely to be replaced if they are not referenced again in the near future. To maximize the hit ratio it is always more "profitable" to replace one large document (and miss this one document if it is referenced again) than to replace many short documents to regain the same amount of space

(and miss many of those documents when they are requested again). The cost of 1 favors small documents and replaces large documents, especially those rarely referenced. We denote this strategy GD-Size(1).

Thus, GD-Size(1) achieves the best hit ratio. However, this high hit ratio for GD-Size(1) is achieved at a price of a lower byte hit ratio.

The strategy GD-Size(packets) sets *cost* function for each document to $2 + size/536$, which is the estimated number of network packets sent and received to satisfy a cache miss for requested document. Te GD-Size(packets) strategy achieves both: a high hit and a high byte hit ratio. This cost function provides greater key for large documents than for small ones. It allows the documents of small sizes to be replaced more likely than the large ones (especially if these large documents are often referenced). If the large document is never referenced again, then it gets replaced due to aging mechanism.

Thus, GD-Size(1) is aimed to minimize miss ratio, while GD-Size(packets) tries to minimize the network traffic resulting from the misses.

In fact, as we can see, those two metrics are somewhat contradictory, and it is very difficult for one strategy to achieve the best performance results in both metrics.

Typically, a high hit ratio is a preferred choice because it allows a greater number of clients requests to be satisfied out of proxy cache, and minimizes average request latency.

However, if minimizing the outside network traffic is more desirable then a strategy providing a higher byte hit ratio has to be used.

Greedy-Dual-Size algorithm with appropriate cost definitions outperforms currently known caching algorithms on a number of performance metrics including hit ratio and byte hit ratio.

However, Greedy-Dual-Size algorithm does have one shortcoming. It does not take into account how many times the document was accessed in the past.

Example: Let us consider how GD-Size(1) handles hit and miss for two different documents of the same size. When initially the documents are brought to a cache they both get an H value as

$$H = 1/size.$$

The document *doc1* which was accessed n times in a past will get the same H value as the document *doc2* accessed for a first time, and in a worst case scenario the document *doc1* will be replaced instead of document *doc2*.

The Greedy-Dual-Size algorithm can be improved to reflect file access pattern by incorporating file frequency count *frequency* in the computation of *H:*

$$H = frequency \times \frac{cost}{size}.$$

We will call it the Greedy-Dual-Size-Frequency algorithm (abbreviation: GD-Size-Frequency).

5 Greedy-Dual-Size-Frequency Caching Policy

This section describes the proposed algorithm in a rigorous way.

Let us consider a cache of size *Total*.

Let *Used* be amount of cache which is used to store cached files.

At the beginning, $Used = 0$.

With each file f present in a cache we associate a frequency count $Fr(f)$: how many times it was accessed. File f which is not in a cache but is going to be cached is assigned a frequency count of 1: $Fr(f) = 1$.

To define which files are going to be replaced when the cache capacity is exceeded, we maintain a priority queue on files.

The file f is inserted into the priority queue with a priority key $Pr(f)$ computed in the following way:

$$Pr(f) = Clock + Fr(f) \times \frac{Cost(f)}{Size(f)} \qquad (1)$$

where

- The parameter *Clock* is a running queue "clock" that starts at 0 and is updated for each replaced (evicted) file $f_{evicted}$ to the priority key of this file in the priority queue: $Pr(f_{evicted})$.
- The parameter $Fr(f)$ is a file f frequency count. If file f is a hit (i.e. it is present in a cache) then $Fr(f)$ is increased by one: $Fr(f) = Fr(f) + 1$. If file f is a miss (i.e. it is not in a cache) it is assigned a frequency count of 1: $Fr(f) = 1$.
- The parameter $Size(f)$ is the file size.
- The parameter $Cost(f)$ is the cost associated with file f to bring it to a cache. We have discussed the most popular cost functions and correspondent performance metrics in Section 4.

Now, let us describe the caching policy as a whole.

1. If the requested file f is a hit in the cache then this file request is served out of cache, and

   - The Amount of cache *Used* does not change.
   - File frequency count $Fr(f)$ is increased by one.
   - The priority key $Pr(f)$ is recomputed using formula (1).
   - File f with recomputed key $Pr(f)$ is reinserted into the priority queue to reflect its new state, and

   it completes this case.

2. If the requested file f is a cache miss then this file request is served from the original server and may be cached by a Proxy cache on its way back from the original server to the client that requested this file. To do this

   - File frequency count $Fr(f)$ is set to one.
   - The priority key $Pr(f)$ is computed using formula (1).
   - File f is inserted into the priority queue with computed key $Pr(f)$.
   - Amount of cache *Used* is reevaluated in the following way:

   $$Used = Used + Size(f) \tag{2}$$

   After that one of the following two situations takes place:

   - If $Used \leq Total$ then this means that there is enough space to store the file f, and no files should be replaced. The file f is cached, and it completes this case.
   - If $Used > Total$ then this means that there is not enough space to store the file f, and some files should be replaced.

     First, we identify a minimal set of files to evict using the following procedure: The first k files (k might be equal to 1) with lowest priority in the priority queue are chosen, f_1,f_2, ..., f_k, to satisfy the property $UsedEstimate \leq Total$ where

     $$UsedEstimate = Used - \sum_{i=1}^{k} Size(f_i) \tag{3}$$

     (a) If the original file f we are going to cache is not among the files f_1,f_2, ..., f_k then the following actions take place:

     i. The parameter $Clock$ (running "clock" of the priority queue) is recomputed in the following way:

     $$Clock = \max_{i=1}^{k} Pr(f_i) = Pr(f_k) \tag{4}$$

ii. Amount of cache *Used* is reevaluated:

$$Used = Used - \sum_{i=1}^{k} Size(f_i) \qquad (5)$$

iii. The files f_1,f_2, ..., f_k are evicted.

iv. The file f is cached.

(b) If the original file f we are going to cache is among the files f_1,f_2, ..., f_k which has to be removed in order to store the file f then:

i. The file f is not cached and its $Pr(f)$ is removed from the priority queue.

ii. None of the files in a cache are evicted.

This completes the caching algorithm.

The main innovation proposed in this section is the introduction of the frequency count $Fr(f)$ associated with each file f present in a cache and incorporating it in the formula (1). This reflects file access patterns and proposes non-trivial performance improvement of GD-Size algorithm across the set of different performance metrics with corresponding cost functions.

The part of the algorithm related to the file eviction has an interesting part related to the case when the file is not cached due to its very low key value which puts this file (if cached) among the first candidates for replacement. Typically, it can happen when the file size is very large – the proposed procedure automatically will restrict the cases when such files are cached.

6 GD-Size-Frequency Algorithm Properties

Let us consider the properties of GD-Size-Frequency(1).

In this case, the priority queue key for a file f is computed in the following way:

$$Pr(f) = Clock + Fr(f) \times \frac{1}{Size(f)} \qquad (1)$$

In such a way, the documents with a higher frequency count get the larger key, and have a better chance to stay in a cache, compared with documents rarely accessed.

GD-Size-Frequency(1) assigns greater keys to a small documents comparing with the large ones aiming to maximize hit ratio by minimizing the miss ratio when the documents get replaced.

Parameter *Clock* has a monotonically increasing value (it is increased any time when some document get replaced. The documents which were not accessed for a long time do not change

their key in the priority queue. At some point, the *Clock* value gets high enough that any new document is inserted behind these "long time not accessed" documents. In such a way, small documents and the documents with high frequency count get replaced if they are not accessed again. This "aging" mechanism prevents proxy cache from pollution.

Similar properties has GD-Size-Frequency(packets) aimed to achieve both: a high hit and a high byte hit ratio. It has the only difference comparing with GD-Size-Frequency(1) that it does not discriminate against the large documents.

7 Greedy-Dual-Frequency Algorithm

Another important derivation of Section 5 related to introducing the frequency count in combination with GD-Size policy is the direct extension of the original Greedy-Dual algorithm with a frequency count for uniform size cache (memory) pages. The original Greedy-Dual algorithm can be improved further to reflect the cache page access pattern by incorporating for each cache page its frequency count *frequency* in the computation of *H:*

$$H = frequency \times cost.$$

Following definitions introduced in Section 5: the page f is inserted into the priority queue with a priority key $Pr(f)$ computed in the following way:

$$Pr(f) = Clock + Fr(f) \times Cost(f)$$

We will call it Greedy-Dual-Frequency algorithm (abbreviation: GD-Frequency).

8 Directions for Future Research

The formula (1) for computing a file key in priority queue from Section 5:

$$Pr(f) = Clock + Fr(f) \times \frac{Cost(f)}{Size(f)}$$

has four basic parameters:

- The parameter $Clock$ which introduces the aging factor.
- The parameter $Fr(f)$ which is a file f frequency count.
- The parameter $Size(f)$ which is the file size.

- The parameter $Cost(f)$ which is the cost associated with file f to bring it to a cache. We have discussed the most popular cost functions and correspondent performance metrics in Section 4.

Clearly, all these parameters are important one. They could be defined in slightly different way to cover a wide range of replacement policies.

For example, the clock in GD-Size-Frequency policy starts at 0 and is updated for each replaced (evicted) file $f_{evicted}$ to the priority key of this file in the priority queue: $Pr(f_{evicted})$. In such a way, a clock is a monotonically increasing function, but it increases at a very slow pace. Designing a faster increasing clock function will lead to a replacement strategy with closer features to LRU, i.e. the strategy with greater impact of file resency over file size and frequency..

The same applies to parameters $Size(f)$ and frequency count $Fr(f)$. If one would use $log(Size(f))$ then the impact of file size will be decreased against the impact of file frequency.

It is an interesting problem to adapt this algorithm to achieve the best performance if one would know more details about workload specifics, and use it to emphasize the impact of right parameters in the formula (1). .

9 Conclusion

This paper introduces the Greedy-Dual-Size-Frequency caching policy to maximize hit and byte hit ratios for WWW proxies. The proposed caching strategy incorporates in a simple way the most important characterizations of the file and its accesses such as file size, file access frequency and recentness of the last access. Greedy-Dual-Size-Frequency is an improvement of Greedy-Dual-Size algorithm – current champion among the replacement strategies proposed for Web proxy caches. We describe it in rigorous form in Section 5.

The interesting performance question still remaines: to evaluate the potential gain of introducing a frequency count. We leave it for future study.

We speculate that this algorithm can also be successfully used for main memory caching at Web servers. This can be an interesting future study especially in the light of the results in [AW97]. They show that frequency based strategy with aging mechanism for reference count is a good choice for replacement strategy in Web server caching.

We also propose an extension to Greedy-Dual algorithm by exploiting the same frequency count idea. A new algorithm, called Greedy-Dual-Frequency, incorporating the frequency count for uniform-size references is proposed for conventional paging problem.

10 Acknowledgements

My thanks to Sekhar Sarukkai and John Wilkes who brought to my attention the paper [Cao-Irani97], when I missed Pei Cao's presentation given at HPLabs.

I would like to thank Rajiv Gupta and Josep Ferrandiz for their strong support during the writing of this paper.

John Dilley made a set of useful comments which author gratefully incorporated in the paper.

11 References

[AW97] M. Arlitt, C.Williamson: Trace-Driven Simulation of Document Caching Strategies for Internet Web Servers. The Society for Computer Simulation. Simulation Journal, vol. 68, No. 1, pp23-33, January 1997.

[ASAWF95] M.Abrams, C.Stanbridge, G.Abdulla, S.Williams, E.Fox: Caching Proxies: Limitation and Potentials. WWW-4, Boston Conference, December, 1995.

[Cao-Irani97] P.Cao, S.Irani: Cost Aware WWW Proxy Caching Algorithms. Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS), Monterey, CA, pp.193-206, December 1997.

[LRV97] P.Lorensetti, L.Rizzo, L.Vicisano. Replacement Policies for Proxy Cache. Manuscript, 1997.

[WASAF96] S.Williams, M.Abrams, C.Stanbridge, G.Abdulla, E.Fox: Removal Policies in Network Caches for World-Wide Web Documents. In Proceedings of the ACM Sigcomm96, August, 1996, Stanford University.

[WA97] R.Wooster, M.Abrams: Proxy Caching the estimates Page Load Delays. In proceedings of 6th International World Wide Web Conference, 1997.

[Young91] N.Young: On-line caching as cache size varies. In the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, 241-250,1991.