Role of Aging, Frequency, and Size in Web Cache Replacement Policies

Ludmila Cherkasova[1] and Gianfranco Ciardo[2]

[1] Hewlett-Packard Labs, 1501 Page Mill Road, Palo Alto, CA 94303, USA
cherkasova@hpl.hp.com
[2] CS Dept., College of William and Mary, Williamsburg, VA 23187-8795, USA
ciardo@cs.wm.edu

Abstract. Document caching on is used to improve Web performance. An efficient caching policy keeps popular documents in the cache and replaces rarely used ones. The latest web cache replacement policies incorporate the document size, frequency, and age in the decision process. The recently-proposed and very popular Greedy-Dual-Size (GDS) policy is based on document size and has an elegant aging mechanism. Similarly, the Greedy-Dual-Frequency (GDF) policy takes into account file frequency and exploits the aging mechanism to deal with cache pollution. The efficiency of a cache replacement policy can be evaluated along two popular metrics: file hit ratio and byte hit ratio. Using four different web server logs, we show that GDS-like replacement policies emphasizing size yield the best file hit ratio but typically show poor byte hit ratio, while GDF-like replacement policies emphasizing frequency have better byte hit ratio but result in worse file hit ratio. In this paper, we propose a generalization of Greedy-Dual-Frequency-Size policy which allows to balance the emphasis on size vs. frequency. We perform a sensitivity study to derive the impact of size and frequency on file and byte hit ratio, identifying parameters that aim at optimizing both metrics.

1 Introduction

Many replacement policies for web caches have been proposed [1, 3–9]. Some of them are quite simple and easy to implement, while others are heavily parametrized or have aspects that do not allow for an efficient implementation (thus they can exhibit good results that serve as theoretical bounds on the best practically achievable performance). Two essential features distinguish web caching from conventional caching in the computer systems: (i) the HTTP protocol supports whole file transfers, thus a web cache can satisfy a request only if the entire file is cached, and (ii) documents stored in a web cache are of different sizes, while CPU and disk caches deal with uniform-size pages.

One key to good web cache performance is an efficient cache replacement policy to determine which files should be removed from cache to store newly requested documents. Further improvements can be achieved when such a policy is combined with a decision about whether a document is worth caching at all.

A very good survey of currently-known replacement policies for web documents can be found in [4], which surveys ten different policies, comments on their

efficiency and implementation details, and proposes a new algorithm, Greedy-Dual-Size (GDS), as a solution for the web proxy replacement strategy. The GDS policy incorporates document size, cost, and an elegant aging mechanism in the decision process, and shows superior performance compared to previous caching policies. In [1], the GDS policy was extended, taking into consideration the document frequency, resulting in the Greedy-Dual-Frequency (GDF) policy, which considers a document's frequency plus the aging mechanism, and the Greedy-Dual-Frequency-Size (GDFS), which also considers a document's size.

The typical measure of web cache efficiency is the *(file) hit ratio*: the fraction of times (over all accesses) the file was found in the cache. Since files are of different size, a complementary metric is also important, the *byte hit ratio*: the fraction of "bytes" returned from the cache among all the bytes accessed. The file hit ratio strongly affects the response time of a "typical" file, since request corresponding to a file miss require substantially more time to be satisfied. The byte miss also affects the response time as well, in particular that of "large" files, since the time to satisfy such a request has a component that is essentially linear in the size of the file; furthermore, a large byte miss ratio also indicates the need for a larger bandwidth between cache and permanent file repository.

The interesting outcome of paper [1] was that, while GDFS achieves the best file hit ratio, it yields a modest byte hit ratio. Conversely, GDF results in the best byte hit ratio at the price of a worse file hit ratio. The natural question to ask is then how the emphasis on document size or frequency (and the related aging mechanism) impact the performance of the replacement policy: do "intermediate" replacement policies exist that take into account size, frequency, and aging, and optimize both metrics? In this paper, we partially answer this question in a positive way by proposing a generalization of GDFS, which allows to emphasize (or de-emphasize) the size, frequency, or both parameters. Through a sensitivity study, we derive the impact of size and frequency on the file and byte hit ratio.

We intentionally leave unspecified the possible location of the cache: at the client, at the server, or at the network. The workload traces for our simulations come from four different popular web sites. We try to exploit the specific web workload features to derive general observations about the role of document size, frequency and related aging mechanism in web cache replacement policies. We use trace-driven simulation to evaluate these effects.

The results from our simulation study show that de-emphasizing the impact of size in GDF leads to a family of replacement policies with excellent performance in terms of both file and and byte hit ratio, while emphasizing document frequency has a similar (but weaker, and more workload-sensitive) impact.

2 Related work and Background

The original Greedy-Dual algorithm introduced by Young [10] deals with the case when pages in a cache (memory) have the same size but have different costs to fetch them from secondary storage. The algorithm associates a "value" H_p with each cached page p. When p is first brought into the cache, H_p is defined as the non-negative cost to fetch it. When a replacement needs to be made, the

page p^* with the lowest value $H^* = \min_p\{H_p\}$ is removed from the cache, and any other page p in the cache reduces its value H_p by H^*. If a page p is accessed again, its current value H_p is restored to the original cost of fetching it. Thus, the value of a recently-accessed page retains a larger fraction of its original cost compared to pages that have not been accessed for a long time; also, the pages with the lowest values, hence most likely to be replaced, are either those "least expensive" ones to bring into the cache or those that have not been accessed for a long time. This algorithm can be efficiently implemented using a priority queue and keeping the offset value for future settings of H via a *Clock* parameter (aging mechanism), as Section 3 describes in detail.

Since web caching is concerned with storing documents of different size, Cao and Irani [4] extended the Greedy-Dual algorithm to deal with variable size documents by setting H to *cost/size* where *cost* is, as before, the cost of fetching the document while *size* is the size of the document in bytes, resulting in the Greedy-Dual-Size (GDS) algorithm. If the *cost* function for each document is set uniformly to one, larger documents have a smaller initial H value than smaller ones, and are likely to be replaced if they are not referenced again in the near future. This maximizes the file hit ratio, as, for this measure, it is always preferable to free a given amount of space by replacing one large document (and miss this one document if it is referenced again) than many small documents (and miss many of those documents when they are requested again). From now on, we use a constant cost function of one and concentrate on the role of document *size* and *frequency* in optimizing the replacement policy.

GDS does have one shortcoming: it does not take into account how many times a document has been accessed in the past. For example, let us consider how GDS handles hit and miss for two different documents p and q of the same size s. When these documents are initially brought into the cache they receive the same value $H_p = H_q = 1/s$, even if p might have been accessed n times in the past, while q might have been accessed for a first time; in a worst-case scenario p could then be replaced next, instead of q. In [1], the GDS algorithm was refined to reflect file access patterns and incorporate file *frequency* count in the computation of the initial value: $H = \mathit{frequency}/\mathit{size}$. This policy is called the Greedy-Dual-Frequency-Size (GDFS) algorithm. Another important derivation related to introducing the frequency count in combination with GDS policy is the direct extension of the original Greedy-Dual algorithm with a frequency count: $H = \mathit{frequency}$. This policy is called Greedy-Dual-Frequency (GDF) algorithm.

Often, a high file hit ratio is preferable because it allows a greater number of clients requests to be satisfied out of cache and minimizes the average request latency. However, it is also desirable to minimize the disk accesses or outside network traffic, thus it is important that the caching policy results in a high byte hit ratio as well. In fact, we will show that these two metrics are somewhat in contrast and that it is difficult for one strategy to maximize both.

3 GDFS Cache Replacement Policy: Formal Definition

We now formally describe the GDFS algorithm (and its special cases, GDS and GDF). We assume that the cache has size *Total* bytes, and that *Used* bytes

(initially 0) are already in use to store files. With each file f in the cache we associate a "frequency" $Fr(f)$ counting how many times f was accessed since the last time it entered the cache. We also maintain a priority queue for the files in the cache. When a file f is inserted into this queue, it is given priority $Pr(f)$ computed in the following way:

$$Pr(f) = Clock + Fr(f)/Size(f) \tag{1}$$

where $Clock$ is a running queue "clock" that starts at 0 and is updated, for each replaced (evicted) file $f_{evicted}$, to its priority in the queue, $Pr(f_{evicted})$; $Fr(f)$ is the frequency count of file f, initialized to 1 if a request for f is a miss (i.e., f is not in the cache), and incremented by one if a request for f results in a hit (i.e., f is present in the cache); and $Size(f)$ is the file size, in bytes. Now, let us describe the caching policy as a whole, when file f is requested.

1. If the request for f is a hit, f is served out of cache and:
   - *Used* and *Clock* do not change.
   - $Fr(f)$ is increased by one.
   - $Pr(f)$ is updated using Eq. 1 and f is moved accordingly in the queue.
2. If the request for f is a miss, we need to decide whether to cache f or not:
   - $Fr(f)$ is set to one.
   - $Pr(f)$ is computed using Eq. 1 and f is enqueued accordingly.
   - *Used* is increased by $Size(f)$.

   Then, one of the following two situations takes place:
   - If $Used \leq Total$, file f is cached, and this completes the updates.
   - If $Used > Total$, not all files fit in the cache. First, we identify the smallest set $\{f_1, f_2, \ldots, f_k\}$ of files to evict, which have the lowest priority and satisfy $Used - \sum_{i=1}^{k} Size(f_i) \leq Total$. Then:
     (a) If f is not among $f_1, f_2, \ldots, f_k$:
        i. *Clock* is set to $\max_{i=1}^{k} Pr(f_i)$.
        ii. *Used* is decreased by $\sum_{i=1}^{k} Size(f_i)$.
        iii. $f_1, f_2, \ldots, f_k$ are evicted.
        iv. f is cached.

     (b) If f is instead among $f_1, f_2, \ldots, f_k$, it is simply not cached and removed from the priority queue, while none of the files already in the cache is evicted. This happens when the value of $Pr(f)$ is so low that it would put f (if cached) among the first candidates for replacement, e.g., when the file size is very large – thus the proposed procedure will automatically limit the cases when such files are cached.

We note that the above description applies also to the GDS and GDF policies, except that, in GDS, there is no need to keep track of the frequency $Fr(f)$ while, in the GDF policy, we use the constant 1 instead of $Size(f)$ in Eq. 1.

Let us now consider some properties of GDFS. Among documents with similar size and age (in the cache), the more frequent ones have a larger key, thus a better chance to remain in a cache, compared with those rarely accessed. Among documents with similar frequency and age, the smaller ones have a larger key compared to the large ones, thus GDFS tends to replace large documents first,

to minimize the number of evicted documents, and thus maximize the file hit ratio. The value of *Clock* increases monotonically (any time a document is replaced). Since the priority of files that have not been accessed for a long time was computed with an old (hence smaller) value of *Clock*, at some point, the *Clock* value gets high enough that any new document is inserted behind these "long-time-not-accessed" files, even if they have a high frequency count, thus it can cause their eviction. This "aging" mechanism avoids "web cache pollution".

4 Data Collection Sites

In our simulation study, we used four access logs from very different servers:
HP WebHosting site (WH), which provides service to internal customers. Our logs cover a four-month period, from April to July, 1999. For our analysis, we chose the month of May, which represents well the specifics of the site.
OpenView site (www.openview.hp.com, OV), which provides complete coverage on OpenView solutions from HP: product descriptions, white papers, demos illustrating products usage, software packages, business related events, etc. The log covers a duration of 2.5 months, from the end of November, 1999 to the middle of February, 2000.
External HPLabs site (www.hpl.hp.com, HPL), which provides information about the HP Laboratories, current projects, research directions, and job openings. It also provides access to an archive of published HPLabs research reports and hosts a collection of personal web pages. The access log was collected during February, 2000.
HP site (www.hp.com, HPC), which provides diverse information about HP: HP business news, major HP events, detailed coverage of the most software and hardware products, and the press related news. The access log covers a few hours[1] during February, 2000, and is a composition of multiple access logs collected on several web servers supporting the HP.com site (sorted by time).

The access log records information about all the requests processed by the server. Each line from the access log describes a single request for a document (file), specifying the name of the host machine making the request, the time the request was made, the filename of the requested document, and size in bytes of the reply. The entry also provides the information about the server's response to this request. Since the *successful* responses with code 200 are responsible for all of the documents (files) transferred by the server, we concentrate our analysis only on those responses. The following table summarizes the characteristics of the reduced access logs:

Log Characteristics	WH	OV	HPL	HPC
Duration	1 month	2.5 months	1 month	few hours
Number of Requests	952,300	3,423,225	1,877,490	14,825,457
Combined Size, or Working Set (MB)	865.8	5,970.8	1,607.1	4,396.2
Total Bytes Transferred (GB)	21.3	1,079.0	43.3	72.3
Number of Unique Files	17,489	10,253	21,651	114,388

The four access logs correspond to very different workloads. WH, OV, and HPL had somewhat comparable number of requests (if normalized per month),

[1] As this is business-sensitive data, we cannot be more specific.

while HPC had three orders of magnitude heavier traffic. If we compare the characteristics of OV and HPC, there is a drastic difference in the number of accessed files and their cumulative sizes (working sets). OV's working set is the largest of the four sites considered, while its file set (number of accessed files) is the smallest one: it is more than 10 times smaller than the number of files accessed on HPC. In spite of comparable number of requests (normalized per month) for WH, OV, and HPL, the amount of bytes transferred by OV is almost 20 times greater than for WH and HPL, but still an order of magnitude less than the bytes transferred by HPC.

5 Basic Simulation Results

We now present a comparison of *Least-Recently-Used* (LRU), GDS, GDFS and GDF on a trace-driven simulation using our access logs. Fig. 1 compares GDS, GDFS, GDF and LRU according to both file and byte miss ratio (a lower line on the graph corresponds to a policy with better performance). On the X-axis, we use the cache size as a percentage of the trace's working set. This normalization helps to compare the caching policies performance over different traces.

The first interesting observation is how consistent the results are across all four traces. GDFS and GDS show the best file miss ratio, significantly outperforming GDF and LRU for this metric. However, when considering the byte miss ratio, GDS performs much worse than either GDF or LRU. The explanation is that large files are always "first victims" for eviction, and *Clock* is advanced very slowly, so that even if a large file is accessed on a regular basis, it is likely to be repeatedly evicted and reinserted in the priority queue. GDFS incorporates the frequency count in its decision making, so popular large files have a better chance of remaining in the queue without being evicted very frequently. Incorporating the frequency in the formula for the priority has also another interesting side effect: the *Clock* is now advanced faster, thus recently-accessed files are inserted further away from the beginning of the priority queue, speeding-up the eviction of "long time not accessed" files. GDFS demonstrates substantially improved byte miss ratio compared to GDS across all traces except HPL, where the improvement is minor.

LRU replaces the least recently requested file. 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 web caches because web workloads exhibit different traffic pattern: web workloads have a very small temporal locality, and a large portion of web traffic is due to "one-timers" — files accessed once. GDF incorporates frequency in the decision making, trying to keep more popular files and replacing the rarely used ones, while files with similar frequency are ordered accordingly to their age. The *Clock* is advanced much faster, helping with the eviction of "long time not accessed" files. However, GDF does not take into account the file size and results in a higher file miss penalty.

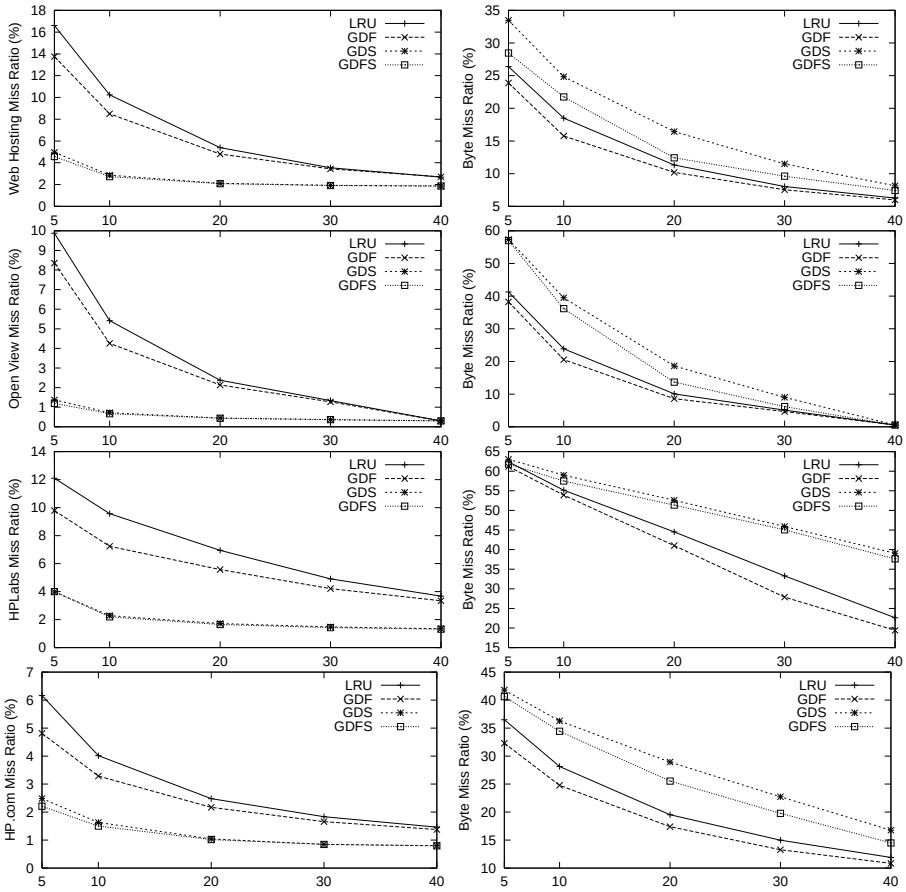

Fig. 1. File and byte miss ratio as a function of cache size (in % of the working set).

6 Generalized GDFS Policy and Its Simulation Results

The *Clock* in GDFS is updated for each evicted file $f_{evicted}$ to the priority of this file, $Pr(f_{evicted})$. In such a way, the clock increases monotonically, but at a very slow pace. Devising a faster increasing clock mechanism leads to a replacement strategy with features closer to LRU, i.e., the strategy where age has greater impact than size and frequency. An analogous reasoning applies to $Size(f)$ and $Fr(f)$: if one uses $Fr(f)^2$ instead of $Fr(f)$, the impact of frequency is stressed more than that of size; if one uses $log(Size(f))$ instead of $Size(f)$, the impact of size is stressed less than that of frequency.

With this idea in mind, we propose a generalization of GDFS (g-GDFS),

$$Pr(f) = Clock + Fr(f)^{\alpha}/Size(f)^{\beta} \tag{2}$$

where α and β are rational numbers. Setting α or β above one emphasizes the role of the correspondent parameter; setting it below one de-emphasizes it.

Impact of Emphasizing Frequency in g-GDFS. Introducing frequency in GDS had a strong positive impact on byte miss ratio and an additional slight improvement in the already excellent file miss ratio demonstrated by GDS. Led by this observation, we would like to understand whether g-GDFS can further improve performance by increasing the impact of the file frequency over file size in Eq. 2, for example setting $\alpha = 2, 5, 10$. Fig. 2 shows a comparison of GDF, GDFS, and g-GDFS with $\alpha = 2, 5, 10$ (and $\beta = 1$). The simulation shows that, indeed, the additional emphasis on the frequency parameter in g-GDFS improves the byte miss ratio (except for HPL, for which we already observed in Section 5 very small improvements due to the introduction of the frequency parameter). However, the improvements in the byte miss ratio come at the price of a worse file hit ratio. Clearly, the idea of having a different impact for frequency and size is sound, however, frequency is dynamic parameter that can change significantly over time. Special care should be taken to prevent $Fr(f)^{\alpha}$ from overflow in the priority computation. As we can see, the impact is workload dependent.

Impact of Deemphasizing Size in g-GDFS The question is then whether we can achieve better results by de-emphasizing the role of size against that of frequency instead. If this hypothesis leads to good results, an additional benefit is ease of implementation, since the file size is a constant parameter (per file), unlike the dynamic frequency count. We then consider g-GDFS where we decrease the impact of size over frequency in Eq. 2, by using $\beta = 0.1, 0.3, 0.5$ (and $\alpha = 1$), and compare it with GDF and GDFS, in Fig. 3. The simulation results fully support our expectations: indeed, the decreased impact of file size parameter in g-GDFS improves significantly the byte miss ratio (and, at last, also for HPL). For example, g-GDFS with $\beta = 0.3$ has a byte miss ratio almost equal to that of GDF, while its file miss ratio is improved two-to-three times compared to GDF. The g-GDFS policy with decreased impact of file size parameter shows close to perfect performance under both metrics: file miss ratio and byte miss ratio.

7 Conclusion and Future Work

We introduced the generalized Greedy-Dual-Size-Frequency caching policy aimed at maximizing both file and byte hit ratios in web caches. The g-GDFS policy incorporates in a simple way the most important characteristics of each file: size, file access frequency, and age (time of the last access). Using four different web server logs, we studied the effect of size and frequency (and the related aging mechanism) on the performance of web caching policies. The simulation results show that GDS-like replacement policies emphasizing the document size yield the best file hit ratio, but typically show poor byte hit ratio, while GDF-like replacement policies, exercising frequency, have better byte hit ratio, but result in worse file hit ratio. We analyzed the performance of g-GDFS policy, which allows to emphasize (or de-emphasize) size or frequency (or both) parameters, and performed a sensitivity study to derive the impact of size and frequency on file hit ratio and byte hit ratio, showing that decreased impact of file size over file frequency leads to a replacement policy with close to perfect performance in both metrics: file hit ratio and byte hit ratio.

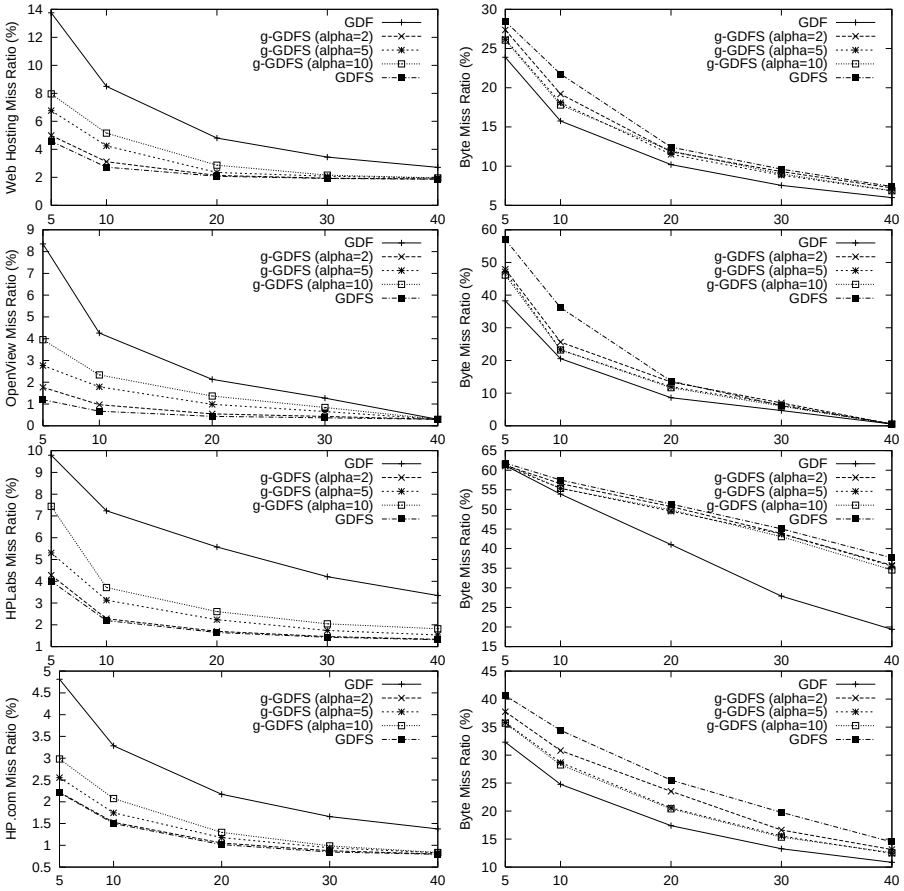

Fig. 2. File and byte miss ratio for new g-GDFS policy: $\alpha = 2, 5, 10$.

The interesting future research question is to derive heuristics that tie the g-GDFS parameters (in particular, β) to a workload characterization. Some promising work in this direction has been done in [6].

References

1. M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, T. Jin: Evaluating Content Management Techniques for Web Proxy Caches. In Proceedings of the 2nd Workshop on Internet Server Performance WISP'99, May, 1999, Atlanta, Georgia.
2. 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.
3. M.Abrams, C.Stanbridge, G.Abdulla, S.Williams, E.Fox: Caching Proxies: Limitation and Potentials. WWW-4, Boston Conference, December, 1995.
4. 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.

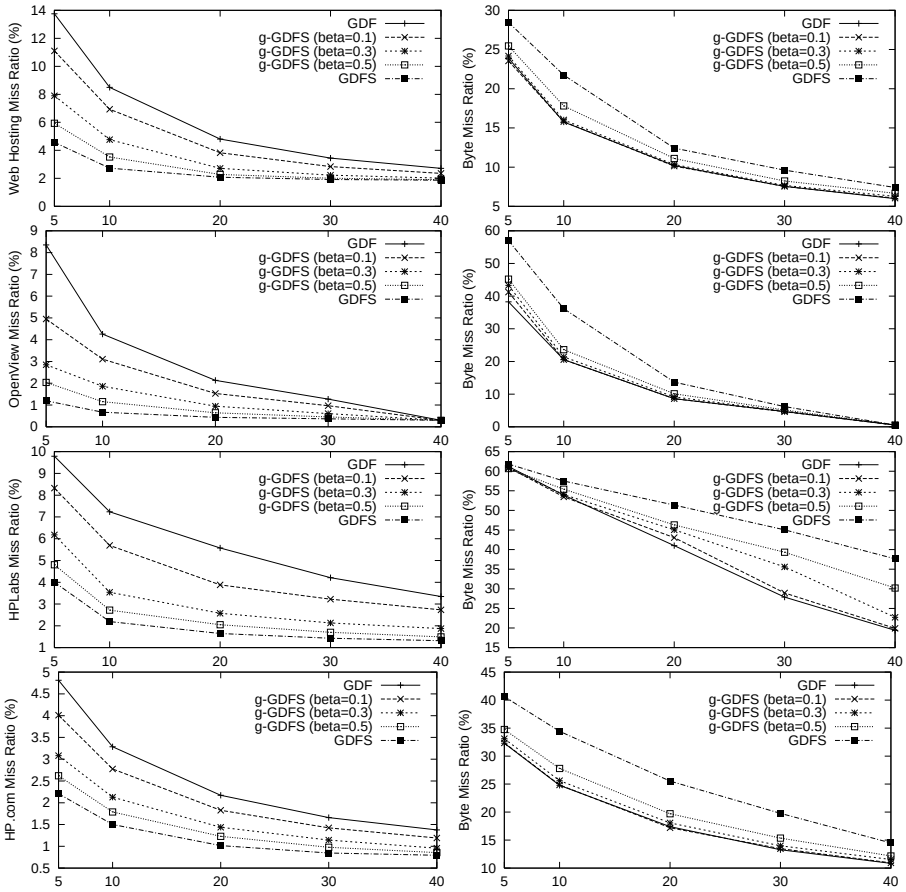

Fig. 3. File and byte miss ratio for new g-GDFS policy: $\beta = 0.1, 0.3, 0.5$.

5. S. Jin, A. Bestavros. Popularity-Aware GreedyDual-Size Web Proxy Caching Algorithms, Technical Report of Boston University, 2000-011, August 21, 1999.
6. S. Jin, A. Bestavros. GreedyDual* Web Caching Algorithm: Exploiting the Two Sources of Temporal Locality in Web Request Streams, Technical Report of Boston University, 1999-009, August 21, April 4, 2000.
7. P.Lorensetti, L.Rizzo, L.Vicisano. Replacement Policies for Proxy Cache. Manuscript, 1997.
8. 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.
9. R.Wooster, M.Abrams: Proxy Caching the estimates Page Load Delays. In proceedings of 6th International World Wide Web Conference, 1997.
10. N.Young: On-line caching as cache size varies. In the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, 241-250,1991.