GD-GhOST: A Goal-Oriented Self-Tuning Caching Algorithm*

Ganesh Santhanakrishnan   Ahmed Amer   Panos K. Chrysanthis   Dan Li

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260 U.S.A.

{ganesh, amer, panos, lidan}@cs.pitt.edu

ABSTRACT

A popular solution to internet performance problems is the widespread caching of data. Many caching algorithms have been proposed in the literature, most attempting to optimize for one criteria or another, and recent efforts have explored the automation and self-tuning of caching algorithms in response to observed workloads. We extend these efforts to consider the goal of optimizing for selectable performance criteria. With our proposed algorithm, we have shown performance matching and exceeding the best performance of the known greedy dual-size algorithms for either object or byte hit ratios across different web workloads. GD-GhOST consistently outperforms the other algorithms tested, at its worst observed performance GD-GhOST exhibited equivalent miss rates to those of the best applicable Greedy-Dual variant, while achieving miss rates that were 25% lower than the worst performing variant. For byte miss rates, GD-GhOST consistently demonstrated rates lower than the best applicable Greedy-Dual variant. At its best, GD-GhOST offered byte miss rates 10% lower than the best variant.

Categories and Subject Descriptors

H.3.m [**Information Storage and Retrieval**]: Miscellaneous—*Caching, Web Caching, Adaptive Caching*; D.4.3 [**Operating Systems**]: File Systems Management—*Distributed File Systems*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Adaptive Caching, Web Caching, Greedy-Dual Algorithms

---

1. INTRODUCTION

One of the larger problems with the web is the efficient delivery of content from a site to a user. This problem continues to increase and leads to longer delays with the increase of new web sites and users. Data caching has been shown as one of the most effective ways to reduce user perceived latency, as well as to reduce network congestion. Web caching can be done at various levels [1, 16]; in browsers, in intermediate servers and in the web servers. Cache location and variations in the request workload all result in different caching policies being suitable for different purposes. Several factors influence and have been used to decide the data that is to reside in cache memory including cache coherence, admission and replacement policies [7]. In this paper we focus on replacement policies. While admission polices are generally based on heuristics or predictive models in the case of prefetching algorithms, replacement algorithms are based on available statistics. Considerable research has gone into this area leading to the development of numerous policies. Among the most successful policies for web caching, the Greedy Dual-Size (GD-Size) algorithms have proven to be very effective [4]. The variants of Greedy Dual-Size algorithms are well known for their abilities to maximize different performance metrics. Example metrics include hit ratios and byte hit ratios.

Recently, research efforts have produced caching policies that, in addition to optimizing a specific performance metric, attempt to automate policy parameter tuning. These efforts hope to effectively eliminate the need for an administrator or programmer to select a particular parameter, and thus allow the algorithm to adapt to the observed workload. In this paper, we present one such policy, GD-GhOST (a Goal-Oriented Self-Tuning caching algorithm based on GD-Size variants), that is aimed at web caching applications, but can be used at other levels such as device caching, active networks [15], routers, Content Distribution Networks (CDN's) and smart routers, and that attempts to facilitate the specification of desirable performance metrics in addition to eliminating the need to preset any algorithm parameters. GD-GhOST differs from adaptive algorithms in the sense that, at any given time, it does not select a single policy out of several ones, but combines all of them based on their weights.

In the next section, we briefly describe the three primary GD-Size schemes which are considered in our evaluation. Section 3 describes our GD-GhOST replacement policy, and Section 4 presents our experimental results based on real-world traces from Boston University and the 1998 World Cup. Finally we describe related work in Section 5 and conclude in Section 6.

2. BACKGROUND

The original Greedy-Dual algorithm based its page replacement on the retrieval cost of each page, which we will refer to as the *H* value. The page with the lowest retrieval cost, *H* value, was the page removed. To account for varying object sizes, the Greedy Dual-Size (GD-Size) algorithm [4] calculates the *H* value for a page *p* as

$$H = \frac{cost(p)}{size(p)}$$

There are many different variants of Greedy Dual Size, each attempting to maximize a slightly different metric, based on the definition of the cost function [4]. GD-Size(1) is considered to be well-suited for maximizing hit ratio or reducing average latency, while GD-Size(packets) is more appropriate when we wish to maximize byte hit ratio. When we compare these algorithms based on file access frequency, GD-Size(frequency) [5] is often the best performer. In short, the choice of metric you wish to optimize can decide what particular caching algorithm is considered the best for the workload at hand. The specific GD-Size algorithms used in our initial tests are:

1. GD-Size(1)
2. GD-Size(packets)
3. GD-Size(frequency)

Using hit ratios and byte hit ratios as two example metrics we will now go on to describe how an arbitrary selection among these three algorithms can be used by GD-GhOST to determine its replacement policy.

3. THE GD-GHOST POLICY

GD-GhOST is a replacement policy based on a combination of several Greedy Dual-Size variants, that attempts to satisfy a given goal using a fully adaptive combination of these individual component algorithms. To describe it in more detail we will consider how it combines the component algorithms, and how it evaluates and adjusts its behavior to a specific goal. GD-GhOST combines individual component algorithms using a master-algorithm approach similar to that employed in the ACME algorithm [2].

GD-GhOST combines the *H* values calculated by the three GD-Size variants, and based on an on-line evaluation of each variant's performance it produces its own *H* value. We employ the variant that is most appropriate in the following manner: At any given point of time, we can calculate three different *H* values. Since the *H* values themselves cannot be compared among the different algorithms, we normalize them. We then proportionally weight the three variants' *H* values based on the performance of each algorithm.

The evaluation of each algorithm's performance is derived from a user-specified weighting of the importance of each metric. For our preliminary results we tested complete bias toward byte or object hit ratios. In other words, this weighting was based on the user-specified weightings for hit ratio and byte hit ratio. The weighting of the policies uses a fixed number of credits that are shared among the policies. This component is similar to the weighting mechanism employed by ACME [2]. For cache eviction decisions, the items with the lowest combined H values (weighted by credit values) are the highest priority for eviction. Initially we assign equal credits to each of the policies. Based on the performance of the algorithms, we distribute the credits for each algorithm in the following manner.

$$Credit_c = (Perf_o - Perf_i) \cdot Credit_p$$

where:

$Credit_c$ is the current credits for the algorithm.
$Credit_p$ is the previous credits for the algorithm.
$Perf_o$ is the overall combination/selection of hit ratio and byte hit ratio.
$Perf_i$ is algorithm *i*'s performance based on the combination/selection of hit ratio and byte hit ratio.

The credits are distributed among the algorithms every time we evaluate their relative performance. This is not done with every access, but at a variable interval. There is no need to manually set this interval, as it's automatically adjusted based on variations in relative algorithm performance. If the workload is such that there is a consistent combination of algorithms to maximize the desired combination of metrics, then the period is automatically lengthened. As updates are needed more rapidly, the period is automatically reduced. This on-line update of credits ensures that at any instant in time, we are most likely to follow the leader among the three algorithms, for the metrics that are considered most important. When the best performing algorithm degrades in performance, the redistribution of credits ensures that it does not degrade the overall performance. As a matter of fact, as we can observe in our results, we follow the best performance of the three component algorithms, and frequently exceed it.

4. EXPERIMENTAL RESULTS

We conducted simulation-based experiments on real-world traces to evaluate the GD-GhOST policy and its ability to exhibit performance similar to the best policy for the selected performance metric. Specifically we tested its ability to maximize hit ratios and byte hit ratios, which we will present in this section through minimizing miss rates and byte miss rates. At its worst observed performance GD-GhOST was within approximately 1% of the best policy's miss ratio, and at its best, GD-GhOST reduced byte miss rates by well over 50%. Specifically, for miss rates, GD-GhOST exhibited equivalent miss rates (on average within 0.3%) to those of the best applicable Greedy-Dual variant, while achieving miss rates that were 25% lower than the worst performing variant. For byte miss rates, GD-GhOST consistently demonstrated rates lower than the best applicable Greedy-Dual variant. At its best, GD-GhOST offered byte miss rates 10% lower than the best variant. This 10% is measured on an absolute scale, and is equivalent to more than halving the byte miss rates of the competing Greedy-Dual variants.

4.1 Workload Description

Experiments were conducted using traces run against a cache simulator implemented in Java. The three Greedy Dual-Size algorithms were implemented, along with the GD-GhOST policy. For GD-GhOST, the credits for the individual replacement policies were updated on-line using the proportional weighted averaging described in Section 3. As described above, the trial period for updating the credits was dynamically adjusted and required no *a priori* settings.

We tested with different cache sizes and using both client and proxy web traces. Specifically, we used traces from Boston University [6] and the 1998 World Cup [3]. The Boston University traces contain records of the HTTP requests and user behavior of a set of Mosaic clients running in the Boston University Computer Science Department, spanning from 21 November 1994 through
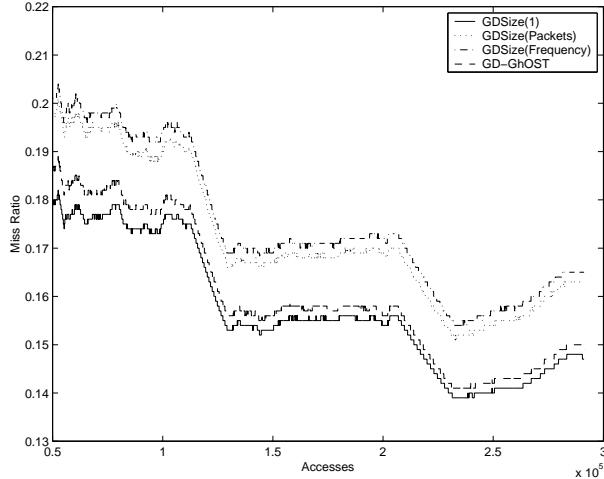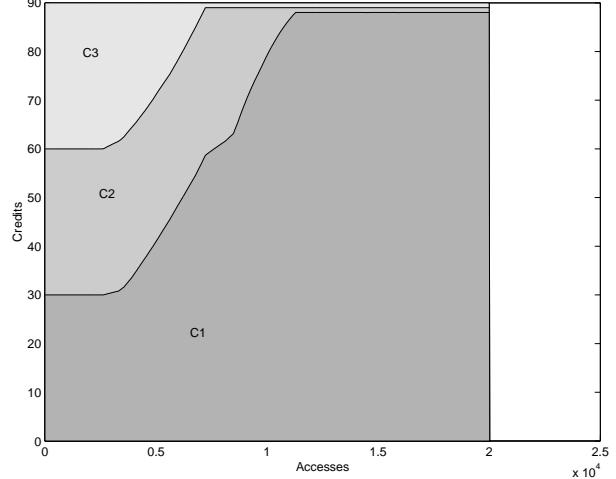
(a) *Miss ratios*　　　　　　　　　　　　　　(b) *Credit Changes*

Figure 1: Minimizing miss ratios.

8 May 1995. During the data collection period a total of 9,633 Mosaic sessions were traced, representing a population of 762 different users, and resulting in 1,143,839 requests for data transfer. There were 5-32 workstations. The World Cup 98 data set consists of all the requests made to the 1998 World Cup Web site between April 30, 1998 and July 26, 1998. During this period of time the site received 1,352,804,107 requests. We performed the simulation using varying cache sizes from 100K to 100MB.

4.2 Optimizing Hit Ratios

The first set of results were conducted with performance biased completely in favor of maximizing hit ratios (minimizing miss ratios). This is equivalent to setting our goal to be 100% weighted for hit ratios (and ignoring byte hit ratios). At its worst observed performance for miss ratios, GD-GhOST remains within approximately 1% of the best algorithm. For example, Figure 1 tracks the miss ratios for four algorithms with the Boston University traces during the period of April, 1995 with a cache size of 10MB.

In Figure 1(b), we show the variation of the credits with the accesses. In the figure, C1 stands for the region representing the variation of the credits of GD-Size(1), C2 stands for the region representing the variation of the credits of GD-Size(packets) and C3 stands for the region representing the variation of the credits of GD-Size(frequency). We show the results for a region of 20,000 accesses. The reason for this is that the experiments represented by Figure 1 clearly showed the shifting of credits towards GD-Size(1) after which they remained fairly constant for the rest of the accesses. In Figure 1(a), we show the variation of the Miss Ratios of the three algorithms and GD-GhOST. The fact that the Miss Ratio curve for GD-GhOST closely follows the curve for the best performing policy reiterates our claim that our policy always performs within some Δ of the best performing policy.

4.3 Optimizing Byte Hit Ratios

Figure 2 is for user specified weight of 100% for Byte Hit Ratio. In Figure 2(b), we show the variation of the credits with the accesses. Again, C1, C2, and C3 represent the relative credits of GD-Size(1), GD-Size(packets) and GD-Size(Frequency) respectively.

We also show the results for a region of approximately 20,000 accesses. Again we see an eventual dominance of one of the algorithms, GD-Size(packets), but it's interesting to note that there was an initial favouring of GD-Size(1), which for a short while was the best algorithm, to be replaced eventually by GD-Size(packets).

In Figure 2(a), we see the variation of the byte miss ratios of the three policies with that of GD-GhOST. The curves for the best performing policy, GD-Size(packets), and that of GD-GhOST are almost inseparable. At the same time we can also see that the overall Byte Hit Ratio is frequently better than that of the best performing policy.

4.4 Summary Results

Figures 1 and 2 demonstrated the ability of GD-GhOST to self-tune towards a specific selected goal and match the best component algorithm. We now look at the summary results for varying cache sizes, and see how in at least one instance, byte miss rates are reduced by as much as 50%.

In Figure 3(a) we give the average miss ratios for different cache sizes of 100KB, 10MB and 100MB measured for both Boston University traces and the World Cup 98 traces. These represent cache sizes that are restricted, small and reasonable, respectively. In Figure 3(b) we give the average byte miss ratios for the same cache sizes and traces. For both figures we see GD-GhOST as comparable in performance to the best component algorithm. But the most notable result is for the byte miss ratios for the World Cup traces and a 10MB cache. In this particular case, GD-GhOST performs almost twice as well as its best component algorithm. This strongly indicates that the combined algorithm is not only capable of matching the best performing policy, but can on occasion perform significantly better than the best of its parts.

5. RELATED WORK

While ghost (or shadow) caches are a known technique of tracking more entries than the cache capacity, GD-GhOST is actually an adaptive caching algorithm based on a dynamic combination of GD-Size variants [4] as component algorithms. The Greedy-Dual * (GD-*) Web caching algorithm [9] is known to be superior to many
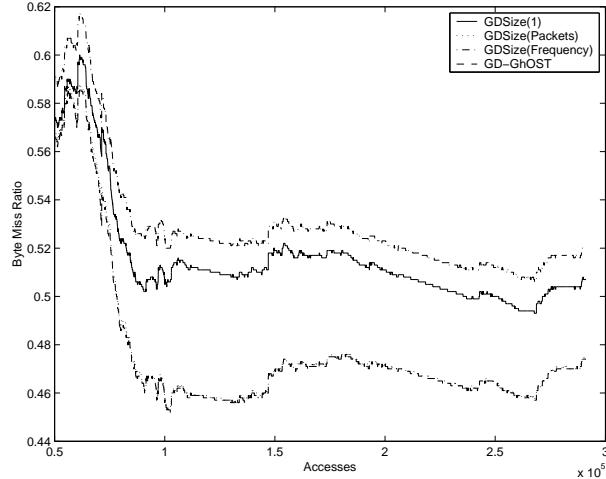
(a) *Byte miss ratios*
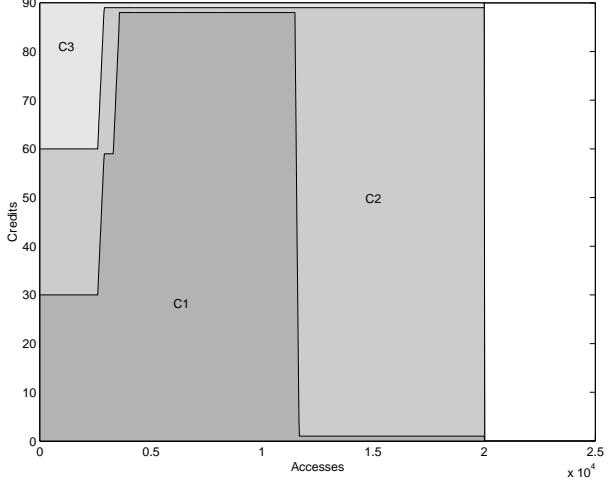
(b) *Credit Changes*

Figure 2: Minimizing byte miss ratios.

other Web cache replacement policies, but our approach differs in that we attempt to optimize a user-specified combination, or selection, of performance goals.

Greedy-Dual algorithms differ from simpler caching schemes such as LRU and LFU in that they use a cost function to form a priority list of items in the cache. Other algorithms have also used such techniques to rank in-cache data. The Least-Normalized-Cost replacement (LNC) [13] policy inserts the documents into a priority queue with a priority key. The problem with this policy is that it has tunable parameters which are workload dependent. Another policy known as Low Interference Recency Set (LIRS) [8] maintains a variable size LRU stack whose LRU page is the L(lirs)-th page that has been seen at least twice recently, where L(lirs) is a parameter. As suggested in the paper, the setting of L(lirs) to 1% of the cache size will be good for Independent Reference Model (IRM) workloads. It does not perform as well for LRU Stack Depth Distribution (SDD) workloads.

The Frequency based replacement (FBR) [14] policy combines both the frequency of access and recency of access. It divides the LRU list into three sections, and maintains a counter for every document in the cache. The algorithm has several tunable parameters. In order to prevent cache pollution from stale pages with high reference counters, all the reference counters must be periodically rescaled. A class of policies known as Least Recently/Frequently Used (LRFU) [10] were shown to subsume the Least Recently Used (LRU) and Least Frequently Used (LFU) policies. These policies have an update rule which is a form of exponential smoothing that is widely used in statistics. It effectively balances both the LRU policy and LFU policies. There is an adaptive version of the policy known as Adaptive Least Recently/Frequently Used (ALRFU) [11] policy. This policy basically has a mechanism to dynamically adjust the parameter used in the basic LRFU policy. Again, both these LRFU schemes require tunable parameters.

A recent policy known as the Adaptive Replacement Cache (ARC) policy [12] was proposed. It maintains two variable sized LRU lists. It captures both recency and frequency of access well, and provides an elegant, efficient and effective mechanism for combining them. It is intended as a page replacement policy, and so it does

not consider the different delays in fetching a document, and variable object sizes, which are important factors in the web caching context.

Our approach is most similar to Adaptive Caching using Multiple Experts (ACME) [2]. ACME uses a mixture of arbitrary policies which are treated as experts. Machine learning algorithms are applied to combine the recommendations of the different policies based on their ordering. We differ from ACME in two ways: by providing a more direct evaluation of each element's relative importance, and allowing the evaluation of an arbitrary selection of performance criteria. By using GD-Size variants as component algorithms we are able to proportionally scale the credits based on the normalized cost functions. ACME restricts policy information to orderings in exchange for a greater generality in terms of applicable policies. This means that GD-GhOST produces a new evaluation function instead of a switching among policies or a rigid mixed-weighting policy. As with GD-* we also differ from ACME in our ability to incorporate arbitrary combinations or selections of performance criteria. Finally, a motivating factor for these dynamic schemes, as for multi-queue algorithms [18], is to adapt policies automatically for cases where multiple caches can have adverse interactions. A recent work that addresses harmful cache interactions is that of Wong and Wilkes [17], where demotions were used as a mechanism to ensure cache exclusivity.

6. CONCLUSIONS AND FUTURE WORK

In summary, we can see that when considering the byte or object hit rates, the GD-GhOST policy adapts on-line and performs on par with the best-performing policy for whichever goal is selected. In some instances, such as with byte miss ratios and the World Cup traces, it is possible for the combined policy to exceed the performance of the best component algorithm. The only true preset parameter for GD-GhOST is the selection of the relative weighting of different performance criteria. The evaluation and self-tuning of the algorithms is fully automated, including the selection of update intervals.

Our initial results, reported in this paper, indicate the potential gains that GD-GhOST can offer. In order to fully evaluate its poten-
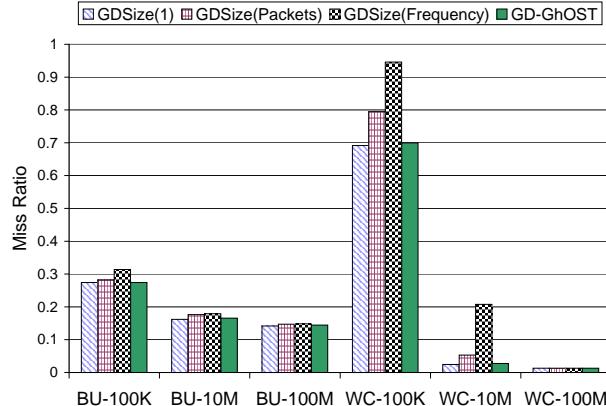
(a) *Miss ratios*
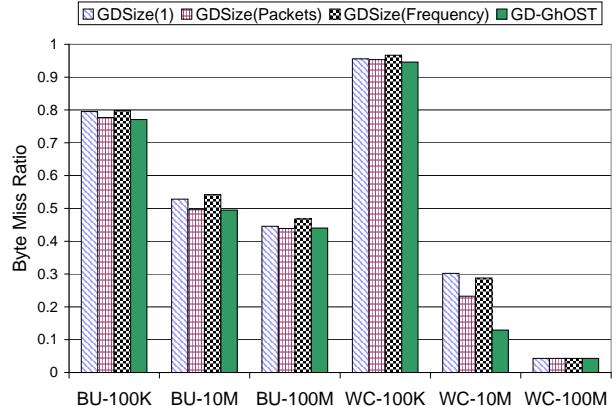
(b) *Byte miss ratios*

Figure 3: Average results for varied cache sizes.

tial we are currently experimenting with more performance metrics beyond byte and object hit/miss ratios. This includes the evaluation of fractional combinations of metrics, as opposed to an absolute bias toward specific goals. We are also currently evaluating the effects of multi-stage caching on the GD-GhOST policy. We are also considering approaches to reduce the cost of implementing a GD-GhOST policy.

7. ACKNOWLEDGMENTS

We would like to thank our colleagues at the University of Pittsburgh: Milos Hauskrecht for valuable discussions, and Huiming Qu for early contributions to this work. We would also like to thank members of the Storage Systems Research Center, and the Machine Learning Group at the University of California, Santa Cruz, for valuable discussions.

8. REFERENCES

[1] M. Abrams, C. Standridge, G. Abdulla, and S. Williams. Caching proxies: Limitations and Potentials. In *Proc. of 4th Int'l World Wide Web Conf.*, pp.119–133, Dec. 1995.

[2] I. Ari, A. Amer, R. Gramacy, E. L. Miller, S. Brandt, and D. D. E. Long. Adaptive Caching using Multiple Experts. In *Proc. of the 2002 Workshop on Distributed Data and Structures*, pp.143–157, Mar. 2002.

[3] M. Arlitt and T. Jin. 1998 World CupWebSite Access Logs - Available at http://www.acm.org/sigcomm/ita/. Aug. 1998.

[4] P. Cao and S. Irani. Cost aware WWW proxy caching algorithms. In *Proc. of USENIX symposium on Internet Technologies and Systems*, pp.193–206, Dec. 1997.

[5] L. Cherkasova. Improving WWW proxies performance with Greedy-Dual-Size-Frequency caching policy. In *HP Technical Report, Palo Alto, California*, Nov. 1998.

[6] C. A. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of WWW Client Traces. *Boston University Department of Computer Science, Technical Report TR-95-010*, Apr. 1995.

[7] A. Dingle and T. Partl. Web cache coherence. In *Computer Networks & ISDN Systems*, Vol.28, pp.907-920, May 1996.

[8] S. Jiang and X. Zhang. LIRS: an efficient Low Inter-reference Recency Set replacement policy to improve buffer cache performance. In *Proc. of the ACM SIGMETRICS Conf.*, pp.31–42, Jan. 2002.

[9] S. Jin and A. Bestavros. GreedyDual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams. In *Int'l Journal on Computer Communications,*, Vol.24(2), pp.174–183, Feb. 2001.

[10] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim. LRFU: A spectrum of policies that subsumes the Least Recently Used and Least Frequently Used policies. In *Proc. of IEEE TOC 1990*, pp.1352–1361, 2001.

[11] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) policies. In *Proc. of ACM SIGMETRICS Conf.*, pp.134–143, June 1999.

[12] N. Megiddo and D. S. Modha. ARC: a self-tuning, low overhead replacement cache. In *Proc. of the USENIX File and Storage Technologies Conf.*, pp.115-130, Mar. 2003.

[13] R. V. Peter Scheuermann, Junho Shim. A case for delay-conscious caching of web documents. In *Computer Networks & ISDN Systems*, Vol.29, pp.997-1005, Sept. 1997.

[14] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. of ACM SIGMETRICS Conf.*, pp.134–142, May 1990.

[15] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. In *IEEE Communications Magazine*, Vol. 35(1), pp.80–86, Jan. 1997.

[16] B. Williams. Transparent web caching solutions. In *Proc. of Third Int'l WWW Caching Workshop*, June 1998.

[17] T. M. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *Proc. of the USENIX Annual Technical Conf.*, Monterey, CA, pp.161–175, June 2002.

[18] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue replacement algorithm for second level buffer caches. In *Proc. of the USENIX Annual Technical Conf.*, pp.91–104, June 2001.