Caching in Web memory hierarchies

Dimitrios Katsaros
Department of Informatics, Aristotle University
Thessaloniki, 54124, Greece
dkatsaro@csd.auth.gr

Yannis Manolopoulos
Department of Informatics, Aristotle University
Thessaloniki, 54124, Greece
manolopo@skyblue.csd.auth.gr

ABSTRACT

Web cache replacement algorithms have received a lot of attention during the past years. Though none of the proposed algorithms deals efficiently with all the particularities of the Web environment, namely, relatively weak temporal locality (due to filtering effects of caching hierarchies), heterogeneity in size and origin of request streams. In this paper, we present the *CRF* replacement policy, whose development is mainly motivated by two factors. The first is the filtering effects of Web caching hierarchies and the second is the intention of achieving a balance between hit and byte hit rates. *CRF*'s decisions for replacement are based on a combination of the recency and frequency criteria in a way that requires no tunable parameters.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*World Wide Web (WWW) Performance Evaluation*; C.4 [**Performance of Systems**]: Measurement Techniques, Modelling Techniques, Performance Attributes.

Keywords

Caching, Latency, Proxy, Replacement policy, Temporal locality, World Wide Web.

1. INTRODUCTION

The growth of the Web resulted in a performance penalty for both the Web servers and its infrastructure, the Internet. Several solutions are investigated in order to cure this situation. The most extensively investigated solution is *content caching* (e.g., [9, 3, 1]). Other solutions include the technique of prefetching [11] and the cooperating caches.

A cache replacement policy assigns a value to every cached object, usually called *utility value* $\mathcal{UV}$, and evicts from cache the object with the least utility value. The aim of the replacement policy is to improve the cache's effectiveness by optimizing two performance measures, the *hit ratio* and *cost savings ratio*. The former is defined as: $HR = \frac{\sum h_i}{\sum r_i}$ and the later as: $CSR = \frac{\sum c_i * h_i}{\sum c_i * r_i}$, where h_i is the number of references to object i satisfied by the cache out of the r_i total references to i and c_i is the cost of fetching object i in cache. The cost can be defined either as the object's size s_i or as the downloading latency c_i. In the former case, the CSR coincides with the byte hit ratio (BHR) and in the latter case with the delay savings ratio (DSR).

SAC '04, March 14-17, 2004, Nicosia, Cyprus.

1.1 The challenge of designing a replacement policy for the Web

There are several factors that distinguish Web caching from caching in traditional computer architectures. These include [9, 3]: (a) the heterogeneity in objects' sizes, (b) the heterogeneity in objects' fetching costs, (c) the depth of the Web caching hierarchy and (d) the origin of the request streams, which are not generated by a few programmed processes, but mainly originate from large human populations with diverse and varying interests.

The majority of the replacement policies proposed so far [3, 14, 5, 2, 1] focus on the first two factors. The main drawback in the design of these policies is that they fail to achieve a balance between HR and CSR. Some of them, the *recency-based policies*, favor the HR, e.g., the family of GreedyDualSize algorithms [3, 7], whereas some others, the *frequency-based policies*, favor the CSR (BHR or DSR), e.g., LFU-DA [5]. Notable exceptions are the LUV [2] and GD* [7], which try to combine recency and frequency. The drawback of LUV though, is the existence of a manually tunable parameter λ, which is used to "select" the recency-based or frequency-based behavior of the algorithm. A similar drawback has GD* as well, since it requires manual tuning of the parameter β.

Regarding the depth of the caching hierarchy, only recently, Williamson [16] studied the effects of caching hierarchies. Its study revealed an alteration in the access pattern, which is characterized by weaker temporal locality. To deal with this Williamson proposed the use of different replacement policies (LRU, LFU, GD-Size) in different levels of the caching hierarchies. This solution though is not feasible, because in many cases the caches are administratively independent. Moreover, the adoption of a replacement policy (e.g., LFU) in a level of the hierarchy favors one performance metric (CSR) over the other (HR) as we explained earlier, and we are not willing to accept this bias.

The last factor has received little attention. This factor (in combination with the caching hierarchy depth) is responsible for the large number of one timers – objects requested

only once – at the request streams. Many studies ([16]) describe the large number of one-timers seen in Web request streams. Only SLRU [1] deals explicitly with this factor and proposed the use of a small auxiliary cache to maintain metadata for past evicted objects. This approach though needs to heuristically determine the size of the auxiliary cache. The most important drawback though, is that the preclusion of some objects from entering into the cache can result in slow adaptation of the cache in a changing request pattern.

1.2 Motivation for a new replacement policy

From the discussion in subsection 1.1, we make two observations. The first regards the heterogeneity in size (cost). This heterogeneity forced the replacement policies proposed so far to favor either the hit or the cost savings ratio. Our position is that we should achieve a balance between the two, since both are equally important. Achieving high hit rate *reduces the average latency that the user sees*. On the other hand, the high byte hit rate has beneficial results on the Internet performance, i.e., low congestion, low TCP packet miss rate, *bandwidth savings*, etc.

The second observation concerns the weak temporal locality in Web request streams. The replacement policy should take into account both the short-term locality and the frequency of reference. Recently, a couple of efforts are reported in [6, 10] for the design of replacement policies in environments with low or moderate temporal locality in operating and database systems. The proposed solutions though are not applicable in the Web environment.

In addition, we state one more requirement for any Web cache replacement policy. We wish the lack of any "administratively" tunable parameters. The existence of parameters whose value is derived from statistical information extracted from Web traces (e.g., LNC-R-W3 [14] or LRV [12]) is not desirable due to the difficulty (and feasibility) of tuning these parameters.

Our motivation can be summarized into the following requirements/constraints. The new policy should: (a) Deal with the weak temporal locality in Web request streams; (b) Achieve a balance between hit rates and cost saving rates; (c) Lack any administratively tunable parameters. The major contributions of this work are summarized as follows:

- We identify the paradox of attempting to optimize only one of the two performance measures in a heterogeneous caching environment, like the Web. We claim that both metrics are equally important. The HR increases user satisfaction, whereas the BHR increases network performance.

- We isolate a number of qualitative characteristics of Web request streams that can help us in designing a new replacement policy. We propose a novel Web replacement policy, namely the *CRF* (standing for Combined Recency and Frequency) replacement policy.

This study is the first effort to deal with all the particularities of the Web environment. The rest of this paper is organized as follows. In section 2, we proceed step-by-step in the design of *CRF*, explaining its features. In section 3 we present the results of the performance evaluation of the examined policies. Finally, section 4 summarizes the major findings of this work.

2. THE *CRF* REPLACEMENT POLICY

2.1 Design principles

Choosing the cost savings metric. In section 1 we saw that the cost savings ratio can be expressed either as delay savings ratio or byte hit ratio. The question is which of the two metrics should we choose to optimize? The drawback of choosing the delay savings ratio is that we can not have a relatively accurate estimation for the downloading delay of an object. The transient network and Web server conditions affect it very much. Two more reasons discourage us from adopting the fetching delay as a measure of cost. The first reason is the *persistent HTTP connections*, which avoid reconnection costs and the second is *connection caching* [4], which reduces connection costs. These two reasons bring about significant variation in the connection time for identical connections. Hence, we favor the size instead of the latency of fetching an object as a measure of the cost.

Dealing with the one-timers. To "isolate" these objects we partition the cache space. Cache partitioning has been followed by some prior algorithms, e.g., FBR [13], but not for the purpose of the isolation of one-timers. The only policy that adopted the partitioning for the purpose of isolating the one-timers is the Segmented LRU [8]. This policy though was designed for disk systems and its replacement decisions are not appropriate for the Web. After testing it, we find out that it suffers from cache pollution. We decide our cache to have two segments, the *R-segment* and the *I-segment*. The cache segments are allowed to grow and shrink deliberately depending on the characteristics of the request stream. The one-timers are accommodated into the R-segment. The choice of further partitioning the I-segment (having many segments like [1, 12]) is not a wise decision, since it makes very difficult to decide the segment from which the victim will be selected and it incurs maintenance cost for moving the objects from one segment to the other. For the determination of one-timers we follow a simple strategy; any object requested only once while being in the cache, is considered to be an one-timer.

Ranking of objects within the segments. The two-segment cache requires a couple of decisions to be made, which regard the ranking of objects within each segment and the selection of replacement victims. These decisions must assure three constraints/targets: (a) the balance between hit and byte hit ratio, (b) the protection of the cache from one-timers, but without preventing the cache from adapting to a changing access pattern and finally (c) the exploitation of the weak temporal locality.

Our aim for the R-segment is to accommodate as many objects as we can and at the same time to exploit any short-term temporal locality existing in the request stream. Thus, we rank the objects into the R-segment according to the ratio of their entry time to their size.

The I-segment comprises the heart of the cache and the replacement policy adopted for it must provide a balance between HR and BHR and also deal with the weak temporal locality. Thus, the replacement policy of the I-segment must: a) avoid weighting by the object's size, and b) encompass frequency-based replacement criteria.

Now, arises the question of how to quantify the frequency of reference to an object. Frequency-based policies like LFU, FBR, GDSF and LFU-DA, suffer from the large reference counts accumulated by the objects. The (dynamic) aging

mechanisms introduce an undesirable parameter, the *cache age factor*. Thus, we select to describe the frequency of reference to an object by the time interval between the ultimate and penultimate reference to it. Let us call this time interval the *last inter-reference time*.

To achieve the amalgam of a recency and frequency-based policy, the ranking function for the I-segment is the product of the last inter-reference time of an object times the recency of the object. The recency describes a transient preference to an object, whereas the inter-reference time describes the steady-state popularity of an object.

Selection of the replacement victim. We are faced with the problem of selecting the replacement victim. If we always choose to purge from the cache the victim of the R-segment, then the cache cannot discriminate between the objects that are hot and become colder and the objects that are cold and become hotter. If we always choose to purge from the cache the victim of the I-segment, then the policy becomes vulnerable to the one-timers.

For our convenience, let us call the candidate victim originating from the R-segment, the *R-victim* and the candidate victim originating from the I-segment, the *I-victim*. Let t_c be the current time and R_1 be the reference time of the R-victim. Let I_1 be the time of the penultimate reference to the I-victim and I_2 be the time of the last reference to it. Let also δ_1 $(= t_c - I_2)$ be the reference recency of the I-victim, δ_2 $(= t_c - R_1)$ be the reference recency of the R-victim, and δ_3 $(= I_2 - I_1)$ be the last inter-reference time of the I-victim.

We must estimate whether or not the I-victim gradually loses its popularity and at the same time estimate the potential of the R-victim to get a second reference. The estimation of these possibilities will be based on the comparison between the intervals δ_1 and δ_3 and between δ_1 and δ_2. We implicitly assume that at time t_c we have two concurrent references for the I-victim and R-victim. In order to select the final victim among the R-victim and the I-victim, we have to take into account both the recency and the frequency of reference (inter-arrival interval). We favor the I-victim in all the cases where $\delta_1 < \delta_3$. When $\delta_1 > \delta_3$, the only case in which we favor the I-victim is when $\delta_1 < \delta_2$. In this case we care to protect the cache from one-timers.

2.2 The implementation of *CRF*

For each cached object $\mathfrak{o}_i$ we need to keep its size $\mathfrak{s}_i$, the time $\mathfrak{t}_l$ of the last reference to it and the time $\mathfrak{t}_p$ of the penultimate reference to it. The inter-access time is measured using *virtual time* – the virtual time clock advances at one unit after every request. This information comprises all the metadata we need for each cached object ($O(1)$ *space complexity* per object).

The records for the objects are kept in two separate heaps. The first heap, called the *R-heap*, stores the records (entries) for the objects of the R-segment. The second heap, called the *I-heap* stores the entries for the objects of the I-segment. Both heaps are *max-heaps*. The sorting key of the R-heap is the ratio $\mathcal{UV}_{\mathrm{R-heap}} = \frac{-\mathfrak{t}_l}{\mathfrak{s}_i}$, whereas the sorting key of the I-heap is the ratio $\mathcal{UV}_{\mathrm{I-heap}} = \frac{-1}{(t_c - \mathfrak{t}_l) * (\mathfrak{t}_l - \mathfrak{t}_p)}$. The dependence of the $\mathcal{UV}_{\mathrm{I-heap}}$ on the current time introduces a complication. In order to select from the I-segment the object with the least $\mathcal{UV}_{\mathrm{I-heap}}$ we must sort the metadata. Obviously, the metadata need not be completely sorted, because we are interested only in the object with the least $\mathcal{UV}_{\mathrm{I-heap}}$. This is the reason behind the selection of the heap as the data structure to keep the metadata. The pseudocode for the implementaion of *CRF* is depicted in Figure 1.

```
Algorithm CRF
// Cache space = cs. Free cache space = a.
// Consider a request to an object o_i of size s_i.
begin
if( s_i > cs )    return;
if( I-heap-inHeap(o_i) )  update-statistics;
else if( R-heap-inHeap(o_i) )
      R-heap-remove(o_i);
      I-heap-insert(o_i);
else
      if( s_i > a )
            I-heap-build-heap();
            while( a < s_i )  evictOne();
      R-heap-insert( o_i );
      a - = s_i;
end
procedure evictOne
begin
If(empty(I-heap))   finalVictim= R-heap-extract-max();
else if(empty(R-heap))  finalVictim= I-heap-extract-max();
else
      d_R = R-heap-return-max();
      d_I = I-heap-return-max();
      if( (d_I.t_l < d_R.t_l) AND
                  ((t_c - d_I.t_l) > (d_I.t_l - d_I.t_p)) )
            finalVictim= I-heap-extract-max();
      else
            finalVictim= R-heap-extract-max();
a + = s_finalVictim
end
```

Figure 1: The *CRF* replacement policy.

3. PERFORMANCE EVALUATION

We examined the performance of the *CRF* against that of LRU, LFU, Size, LFUDA [5], GDS [3] (as representative of the familily which includes GDS, GDSF, GDSF), SLRU [1], LUV [2], HLRU [15], LNCRW3 [14]. Based on [15], we selected the HRLU(6) to be the representative of the HLRU family. The LNCRW3 is implemented so as to optimize the BHR instead of DSR. Regarding the tuning of LUV, we tried several values for the λ parameter, and finally we selected the value 0.01, based on the fact that this value gave the best performance for small caches and also the best performance im most cases.

To generate synthetic Web request streams we used the *ProWGen* tool [16]. Table 1 summarizes the values of various input parameters to the *ProWGen* tool for the generation of the request traces.

We performed a performance evaluation of the selected caching schemes for three parameters, the percentage of one-timers, the skewness of the access pattern (Zipf slope) and the "amount" of temporal locality. We partitioned the examined algorithms into two families. The family of recency-based algorithms includes the GDS, SLRU, Size, LUV and the family of frequency-based includes the LRU, LFU, LFUDA, LNCRW3, HLRU. The impact of each factor on each family of algorithms was examined for three characteristic cache sizes. The first cache is approximately equal

Parameter	Range
Total requests	2000000
Unique docs (% requests)	20%
One-timers (% unique)	65% – 80% [def:70%]
Zipf slope	0.65 – 0.95 [def:0.85]
Pareto tail index	1.0
Corr. size–popularity	no correlation
Temporal locality	dynamic

Table 1: Input parameters to the *ProWGen* tool.

to 0.15% of the infinite cache size. The second is approximately 0.75% and the last is equal to 1.5%. Due to lack of space we will present only a very small subset of the obtained results. Specifically, we will present graphs for small-sized caches when we vary the percentage of one-timers and the value of the zipfian skew. For the medium and large caches we will provide some descriptions of the results and will present some tables that contain aggregated information.

Sensitivity to one-timers. The previous studies [16] examined only a couple of values for one-timers (60% and 70%). They concluded that as their percentage increases, the cache's performance gets only a small improvement (1%-4%). We confirmed this observation, but observed a different situation beyond the value of 70%, (Figures 2 and 3). The general trend is that the HR of the recency-based algorithms drops. This drop is more steep and happens immediately after the value of 70% for small caches, but less steep and appears for higher percentages for other cache sizes. The explanation is that the recency-based policies cannot distinguish and isolate one-timers. A larger cache can absorb part of this effect, but the degradation of the performance is still apparent when the percentage of one-timers gets large enough, e.g., 80%. We make similar observations w.r.t. BHR.

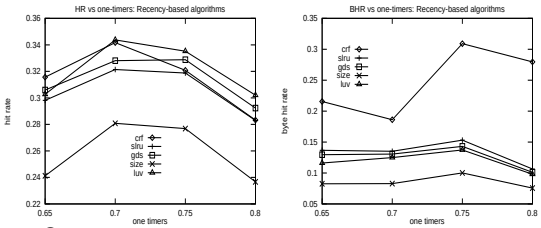

Figure 2: Recency-based policies for small caches. *Left*: HR vs. one-timers, *Right*: BHR vs. one-timers.

The HR of the frequency-based algorithms drops only for small caches, whereas for all other cache sizes it increases slightly or remains stable. When the cache is small, they do not have many opportunities to filter the one-timers, since they flood the whole cache space. The great advantage of the frequency-based policies is the constantly increasing BHR with increasing percentage of one-timers. This is a natural consequence of their decision to avoid weighing by the object's size. Their BHR drops, only when the cache is small and the number of one-timers too large.

Let us look now at the performance of the *CRF*. From a qualitative point of view, *CRF*'s behaviour combines the best features of the two families. *CRF*'s HR is very close to that of the recency-based policies and its BHR is only moderately lower than that of the frequency-based policies. Moreover, its HR remains stable or increases slightly,

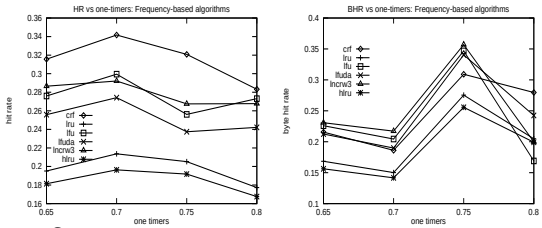

Figure 3: Frequency-based policies for small caches. *Left*: HR vs. one-timers, *Right*: BHR vs. one-timers.

	Recency-based		*Frequency-based*	
Cache	HR	BHR	HR	BHR
small	-2	+11	+3	-2
medium	-1	+15	+9	-5
large	-2	+11	+10	-6
Average	-2	+12	+7	-4

Table 2: *CRF*'s gain-loss w.r.t. one-timers.

similarly to the HR of the frequency-based policies (except from the case of small cases, where the HR of all algorithms drops). Its BHR increases significantly with increasing number of one-timers, similarly to the behaviour of the frequency-based policies. From a quantitative point of view, Table 2, summarizes *CRF*'s performance with respect to that of the best performing algorithm in each case. Each table's cell shows the average *CRF*'s performance improvement taken over all measurements for a particular cache size (a plus sign indicates better performance, and a minus indicates worse performance). We can see that *CRF* sacrifices 2% of HR to achieve 12% greater BHR when compared with the recency-based policies. Similarly when compared with the frequency-based policies, it sacrifices a certain amount of BHR, i.e., 4%, in order to offer (almost) double savings in HR, i.e., 7%.

Sensitivity to Zipfian slope. The second experiment studies the effect of the Zipf slope on the performance of the replacement policies (refer to Figures 4 and 5). The general pattern is that the HR and BHR of all policies increases with increasing skewness. Increased skewness means stronger temporal locality. Thus, the recency-based policies can exploit this skewness quite effectively. Similarly, the frequency-based polices recognize easily the objects that are more popular, since they get more references. The second generic observation concerns the BHR of all policies. Beyond a specific value of skewness, the BHR shows an abrupt increase. This increase appears at the value of 0.80 and it is more steep for the frequency-based policies. Beyond that value for the slope, the larger part of the "popularity mass" is concentrated into a very small number of objects. These objects are recognized by the policies and stay longer in the cache. The frequency-based policies more easily identify these objects and since they do not weigh by the object's size, they achieve higher BHR.

We have observed that the general pattern is a constant increase of the BHR with increasing skewness. A closer examination reveals a more complex situation. There are cases, where higher skewness (e.g., 0.95) leads to slightly smaller BHR for the frequency-based policies than lower skewness (e.g., 0.85). This situation is mainly encountered in medium and large caches. Such cases do not appear for the recency-

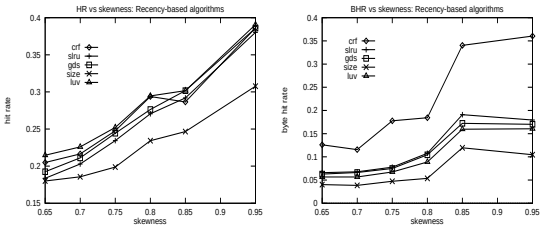

Figure 4: Recency-based policies for small caches. *Left*: **HR vs. Zipf slope,** *Right*: **BHR vs. Zipf slope.**

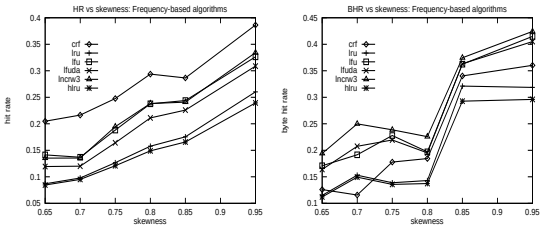

Figure 5: Frequency-based policies for small caches. *Left*: **HR vs. Zipf slope,** *Right*: **BHR vs. Zipf slope.**

based policies or the *CRF* and any such deterioration in BHR for these policies is attributed to the statistical variance. For instance, we observe that the BHR is lower at the slope value of 0.95 than it is for the value of 0.85. Lower Zipfian slopes means that it is quite possible that some large objects are popular enough to stay long in the cache and contribute in the BHR. Increasing only at a small amount the value of the slope, this population of objects decreases causing a degradation in BHR.

Regarding the performance of the *CRF* refer to Figures 4 and 5. We notice that its BHR does not drop (as happens with the frequency-based policies) but it constantly increases. From a quantitative point of view, *CRF* bridges the performance gap between the high HR of the frequency-based policies and the high BHR of the frequency-based policies. The former achieve a 5% to 10% higher HR than the latter and the latter achieve a 20% to 30% better BHR. Table 3 summarizes the quantitative measurements. When compared with the recency policies, *CRF* sacrifies 1% of HR to increase at 10% the BHR. When compared with the frequency-based policies, it sacrificies 6% of BHR to offer (almost) double savings in HR.

	Recency-based		*Frequency-based*	
Cache	HR	BHR	HR	BHR
small	-1	+10	+5	-7
medium	0	+15	+13	-4
large	-1	+9	+13	-6
Average	-1	+11	+10	-6

Table 3: *CRF*'**s gain-loss w.r.t. Zipfian slope.**

4. CONCLUSIONS

We have proposed a new replacement policy for Web caches, called *CRF*, which is the result of a careful selection of design alternatives. It was designed to adresses all the particularities of the Web environment. We evaluated *CRF*'s performance using synthetic request streams and examined its performance relative to that of the state-of-the-art algorithms. The results confirmed that *CRF* is truly a hybrid between recency and frequency-based policies. This finding was consistent across a range of cache sizes and request distributions.

5. REFERENCES

[1] C. Aggrawal, J. Wolf, and P.S. Yu. Caching on the World Wide Web. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):94–107, 1999.

[2] H. Bahn, K. Koh, S.H. Noh, and S.L. Min. Efficient replacement of nonuniform objets in Web caches. *IEEE Computer*, 35(6):65–73, June 2002.

[3] P. Cao and S. Irani. Cost-aware WWW proxy cahing algorithms. In *Proceedings of USITS*, pages 193–206, 1997.

[4] E. Cohen, H. Kaplan, and U. Zwick. Connection caching: Model and algorithms. *Journal of Computer and System Sciences*, 67(1):92–126, 2003.

[5] J. Dilley and M. Arlitt. Improving proxy cache performance: Analysis of three replacement policies. *IEEE Internet Computing*, 3(6):44–50, 1999.

[6] S. Jiang and X. Zhang. LIRS:An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM SIGMETRICS*, pages 31–42, 2002.

[7] S. Jin and A. Bestavros. GreedyDual* Web caching algorithm: Exploiting the two sources of temporal locality in Web request streams. *Computer Comnunications*, 24(2):174–183, 2001.

[8] R. Karedla, J.S. Love, and B.G. Wherry. Caching strategies to improve disk system performance. *IEEE Computer*, 27(3):38–46, 1994.

[9] D. Katsaros and Y. Manolopoulos. Cache management for Web-powered databases. In *Web-Powered Databases*, pages 201–242. IDEA Group Publishing, 2002.

[10] N. Megiddo and D. S. Modha. ARC: A self-tuning low overhead replacement cache. In *Proceedings of the USENIX FAST*, 2003.

[11] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized Web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1155–1169, 2003.

[12] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking*, 8(2):158–170, 2000.

[13] J.T. Robinson and M.V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the ACM SIGMETRICS*, pages 134–142, 1990.

[14] J. Shim, P. Scheuermann, and R. Vingralek. Proxy cache algorithms: Design, implementation and performance. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):549–562, 1999.

[15] A. Vakali. Proxy cache replacement algorithms: A history-based approach. *The World Wide Web Journal*, 4(4):277–297, 2001.

[16] C. Williamson. On filter effects in Web caching hierarchies. *ACM Transactions on Internet Technology*, 2(1):47–77, 2002.