FRES-CAR: An Adaptive Cache Replacement Policy

George Pallis, Athena Vakali, Eythimis Sidiropoulos
Department of Informatics
Aristotle University of Thessaloniki,
54124, Thessaloniki, Greece
gpallis@ccf.auth.gr, {avakali, eythimis}@csd.auth.gr

Abstract

Caching Web objects has become a common practice towards improving content delivery and users' servicing. A Web caching framework is characterized by its cache replacement policy, which identifies the objects (i.e. the elements on a Web page, which include text, graphics, and scripts) to be replaced in a cache upon a request arrival. In this paper, we present a cache replacement algorithm (so-called FRES-CAR), which identifies the objects that should be evicted by considering together three important criteria: object's frequency, recency and size. Experimentation under synthetic workloads has shown that FRES-CAR achieves higher hit rates when compared with the most popular and existing algorithms.

1. Introduction

The explosive growth of the Web has imposed a heavy demand on networking resources over which Web servers and users often experience long and unpredictable delays (particularly) when retrieving Web pages. In a certain extend, the Web has become a "victim" of its own success. Increasing bandwidth would solve problems temporarily since it would ease the users to create more and more resource-hungry applications, bunching again the network.

To overcome the above limitations, Web caching has proven to be a valuable tool [7]. Typically, Web caching aims to improve the performance of Web-based systems by keeping and reusing Web objects that are likely to be used often in the near future. In this context, one of the most critical research issues in Web caching involve cache replacement as well as cache consistency and validation. Whenever the cache is full and the proxy needs to cache a new object, it has to decide which object to evict from the cache to accommodate the new object. The policy used for the eviction decision is referred to as the replacement policy.

In order to evaluate a cache replacement policy, a common practice is to measure two performance rates, the hit rate (HR - is the percentage of the number of requests that are served by the cache over the total number of requests) and the byte hit rate (BHR - is the percentage of the number of bytes that correspond to the requests served by the cache over the total number of bytes requested). A high HR indicates an effective cache replacement policy and defines an increased user servicing, reducing the average latency. On the other hand, a high BHR improves the network performance (i.e., bandwidth savings, low congestion etc.).

Usually, the cache past status is used to predict the future cache replacement actions. Each object is defined by a "value", the so-called cache utility value (CUV), for each object, where the objects with the smallest utility outcome will be the first candidates to evict from the cache. Earlier cache replacement policies have been categorized in:

- **Recency-based**: The object's CUV is determined by the time of the last reference to that object. LRU (Least Recently Used) is the most indicative algorithm of this category and has been applied in several proxy caching servers, such as Squid.
- **Size-based**: The object's CUV is determined by its size. The most indicative algorithm of this category is the SIZE [10] which considers object's size as the basic parameter (large objects are evicted first), assuming that users are less likely to re-access large objects because of the high access delay associated with such documents.
- **Frequency-based**: The object's CUV is determined by the number of requests to that object. Frequency-based approaches are variations of the LFU (Least Frequently Used) algorithm and they use the popularity of objects for the replacement decision.
- **Function-based**: The object's CUV is determined by a cost function, which involves multiple parameters or weights related to a performance metric (such as

Proceedings of the 2005 International Workshop on Challenges in Web Information Retrieval and Integration (WIRI'05)
0-7695-2414-1/05 $20.00 © 2005 IEEE

HR, BHR). The most indicative algorithm of this category is the Greedy-Dual Size [3].

Most systems use a recency-based policy (i.e. LRU), due to its simplicity, but the disadvantage of such policies is that they do not take into account the size of the objects and the latency time in the server transmission, resulting to high BHR. The size-based policies usually result in higher HR since they maintain smaller objects in cache (i.e., more objects reside in cache). The frequency-based policies keep in cache the most popular objects independently on their size and on their recency status i.e. they avoid keeping in cache recently accessed objects but with a low frequency (such as the so-called "one-timer" objects1). Finally, the function-based policies use complex formulas and usually suffer from heavy parameterization and high overhead.

1.1. Paper's Contribution

Here, we propose a cache replacement policy that will integrate the ideas proposed in frequency, recency and size based approaches. More specifically, the proposed algorithm aims at addressing the following issues:

- *tune recency with frequency* by replacing (recently accessed) one-timers with more frequently requested objects. This practice overcomes the disadvantage of a "pure" recency-based policy which might favor maintaining one-timer objects in cache.
- *tackle the cache fragmentation problem* by partitioning the cache into groups of exponentially increasing sizes. This practice will keep a "balanced" cache segmentation and will avoid overflowing the cache with large-scale objects.
- *provide a "cautious" cache replacement* that will deal with all the main deficiencies of the replacements that independently handle either recency (one-timers are not tracked), or frequency (size is not considered), or size (access delays due to large object capacities are not considered).

According to the authors knowledge, although several cache replacements approaches (e.g. [3,4,5]) have been proposed in the past, few efforts (such as in [5]) have been devoted in integrating the above characteristics together. The proposed algorithm is compared with the most popular and widely-used algorithms and based on extensive (synthetic) trace-driven simulations (generated by the ProWGen tool [2]) is shown to result in beneficial HR and BHR.

The rest of the paper is organized as follows. Section 2 formulates the problem, whereas the proposed replacement algorithm, the so-called FRES-CAR, is described in Section 3. Section 4 has the results of the experimentation and the algorithm's performance evaluation which are commented and discussed. Finally, we conclude the paper and give some future work plans in Section 5.

[1] *One-timers* are the objects which are requested only once, regardless of the duration of the access log studied.

2. Problem Formulation

A cache replacement problem involves some parameters that are used to monitor the cached objects replacement process. In practice, each cache replacement policy defines a CUV assigned to each cached object such that the object with the appropriate CUV will be evicted from cache. In this paper, the Web cache content is modeled by a linked list in which each node is associated with a particular cached object. Therefore, the number of nodes is bounded by the number of cached objects. Here, we consider each cached object to be identified by its corresponding stored object filename, along with a number of related attributes (e.g. object's size, time of object's request etc.). The basic goal of the proposed cache replacement problem is to maintain in cache the objects with the largest CUV in order to achieve high HR.

The total cache size is computed in bytes and is of fixed size (S). The total set of objects that are stored in the cache is not fixed, but it depends on which objects are cached each time (denoted by N_i). Let o_i be the Web object which is requested at the i-th request. If o_i is in the cache, then we have a cache hit. Otherwise, we have a cache miss and the object o_i should be inserted into the cache. In case that there is not enough space in cache, there is a need to evict one or more objects from the cache in order to free sufficient space.

For each cached object x we keep track of: the object id (o_x), the object size (s_x) in order to handle the *size* balance, the number of accesses since the last time object x was accessed ($hist_x$) in order to support *recency*, and the node id (r_x^k) for the object o_x which is updated according to its *frequency*.

It is important that the cache replacement process should guarantee enough space for the incoming objects. In this context, there are two actions related with the replacement process (i.e. either the object will remain stored in cache or it will be evicted from cache). We define a function to identify the action that should be taken for each cached object:

Definition 1: The function that will determine whether a cached object's will remain in cache or not is defined by

$$f(x) = \begin{cases} 1 & \text{if object x will be evicted from cache} \\ 0 & \text{otherwise} \end{cases}$$

Problem Statement: Suppose that N_i is the number of objects in cache at the i-th request and S is the total

capacity of the cache area, the cache replacement problem is to

$$\text{maximize} \sum_{x=1}^{N_i}(1-f(x))\cdot CUV_x \text{ subject to}$$

$$\sum_{x=1}^{N_i} s_x(1-f(x)) \le S$$

3. The FRES-CAR Replacement Policy

The proposed cache replacement algorithm, is called FRES-CAR (*F*requency *RE*cency and *S*ize *CA*che *R*eplacement), and it involves two phases:

1) Phase 1- Cache Segmentation*:* partition the cache into a number of segments which are in accordance to the objects' sizes. Therefore the objects are stored in cache into certain variable-sized segments.

2)Phase 2- Cache replacement*:* each of the defined cache-segments apply a cache replacement policy.

3.1. Cache Segmentation

The motivation for a cache-segmentation is the flexibility that it offers to manage the cached objects, since the cache segments may grow and shrink deliberately (according to request streams) and also at each segment a separate replacement policy may be applied. In this paper we segment the cache in a way inspired from the work presented in [1]. Next, we propose a hierarchical pyramid-like cache segmentation (such as PSS (Pyramidal Selection Scheme) [1]) depending on objects' sizes such that objects are placed in the appropriate segment according to theirs sizes. More specifically, we assume that the k-th cache segment will host all objects that have sizes ranging between 2^{k-1} and 2^k-1 bytes. Thus, the total number of bytes that can be stored in a cache is given by S= $2^M - 1$ (M is the total number of cache segments, i.e. the number of cache segments is $M = \lceil \log_2(S+1) \rceil$).

3.2. The Cache Replacement Algorithm

A typical replacement approach involves updating the cache content under a certain criterion or over a considered time period. Here, we consider that a cache replacement is occurred when there is a need for cache disk space. Specifically, whenever an object is requested and it is a cache hit, we update the place where the requested object is stored in the list (namely, we re-order the list by moving the requested object towards to the tail of the list). On the other hand, in case of a cache miss, the requested object is located to the appropriate cache segment with respect to its size. Thus, regarding to the amount of free cache space, we have two options. If there is enough free space (available space $C \ge s_x$) to accommodate the object o_x in the cache, then we locate the requested object to the appropriate cache segment. If not, we should make space by evicting from the cache one or more objects. Then, we locate the requested object to the appropriate cache segment.

3.2.1. Locating Objects within Segments. If there is enough space in order to cache the requested object, we assign it to the appropriate cache segment with respect to its size. Considering that we have M segments the total cache size S is computed by the sum of the sizes of all cache segments. In particular $S = \sum_{k=1}^{M} S_k$, where S_k is the size of cache segment k

Each cache segment is modeled by a linked list where each node of the list is associated with a particular cached object. The list structure used for each cache segment is organized with respect to the three major characteristics: frequency, recency and size. This is realized by holding in the tail of the list the most frequently, recently and (moderate)-sized objects. Given that the cache segment k stores N_i^k objects ($N_i = \sum_{k=1}^{M} N_i^k$) at the i-th request, we adopt a similar to [6] approach by considering a variable γ_k ($\frac{1}{N_i^k} \le \gamma_k \le 1$) for each cache segment k. In our case, the location of the objects is performed as follows:

a. If the object is not in cache, then insert it at node $\lceil \gamma_k \cdot N_i^k \rceil$
b. If it is currently at node r_i^k, move it to node $r_i^k + \lceil \gamma_k (N_i^k - r_i^k) \rceil$

The above process updates the location of objects within cache segments even when the object is already in cache such that we support a dynamic and adaptive cache. Therefore, frequently accessed objects will remain in (close to) the tail nodes.

The locating objects process uses a linked list to maintain the order of the objects in the cache. Therefore, it has $O(N_i^k \log N_i^k)$ complexity, since it needs to perform insertions in the list at arbitrary nodes.

3.2.2. Free Space in Cache for Replacement. When the cache is full, the cache replacement policy should determine which objects should be purged from the cache in order to make room for the incoming objects. Typically, a function is needed in order to determine

which object should be evicted from the cache and in our case we use the following CUV:

Definition 2: Each object x in cache is characterized by its CUV which is evaluated by $CUV_x = \frac{1}{s_x hist_x}$, where s_x is the size of object x, and $hist_x$ keeps the track of the number of accesses since the last time object x was accessed.

According to the above definition, it follows that the utility of each object in cache is lower for large-sized objects as well as for objects that have not been requested for a long time. Whenever we need to decide which object to eject from the cache, we compare the CUV of all objects residing at the head of each cache segment list. The object with the smallest CUV will be evicted from the span of the cache, and not necessarily from the cache segment where the incoming object is meant to be cached. This process is continued until to free the required space from the span of cache area in order to make room for the new object. Therefore, each time we have a cache action (add an object to a cache segment or delete an object from a cache segment), the cache segments are re-sized so that the condition $S = \sum_{k=1}^{M} S_k$ is always satisfied. The proposed FRES-CAR algorithm involves two phases:

PHASE 1: Cache Segmentation

Step 1.1: The number of cache segments is computed.

Step 1.2: An object o_i in a cache segment k is identified by its corresponding stored object filename (f_i), its size (s_i), the number of accesses since the last access of object o_i ($hist_i$), and the place (node-id) object o_i is assigned in the list k (r_i^k) .

Step 1.3: We only cache the object's o_i of size (s_i) if there is cache free space.

Step 1.4: Depending on the object's o_i size (s_i) we select an appropriate cache segment will be chosen.

PHASE 2: Cache Update and Replacement

Step 2.1: We search the resulted in Step 1.4 cache segment, which was previously selected, for the existence of object o_i.

- In case of object o_i being in the cache segment k, we use the cache location scheme (subsection 3.2.1) to update object's o_i place (r_i^k) in the cache segment k.

Step 2.2: In case the object o_i has not been found in cache segment k and the free size in the whole cache area is greater than object's o_i size (s_i) we insert the object in the previously selected cache-segment k based on cache location scheme.

Step 2.3: Otherwise, until the free size of the cache is enough for the object to be cached, we evict objects from the whole cache area (i.e., meaning from different cache segments and not just from the one where the object is meant to be cached) according to the following rules:

- We compute the CUV for each of the objects in the heads of the linked-lists of the cache segments and we evict that with the smallest CUV.
- Finally after making enough space for the new object to be cached, we insert it into the specific cache-segment computed earlier by means of cache location scheme.

In conclusion, it is being understood that the cache location scheme (subsection 3.2.1) is used to find the place (node id), in a specific cache segment (list), in which we will cache our newly requested object and in the appropriate change of its position in its parent cache segment in case of a cache hit. On the other hand, the logic of the cache segmentation is based on the choice of which object to evict in order to make space for a new request that will need to be cached.

4. Experimentation – Results

In our experiments, we use extensive (synthetic) trace-driven simulations, which are generated by the ProWGen[2] tool [2]. The motivation to use the ProWGen is that it incorporates five selected workload characteristics, which are deemed relevant to caching performance. Due to its flexibility, the ProWGen workload generator has been used in several earlier research efforts [4, 11].

Table 1. Input Parameters to ProWGen Tool

Parameters	Default Values	Range
Total requests	1,500,000	1,500,000
Unique objects	30%	20%-40%
One timers	70%	50%-80%
Zipf slope	85%	50%-95%
Pareto tail index	1.0	1.0
Correspondence size-popularity	No correlation	No correlation
Temporal locality	Dynamic	Dynamic

ProWGen traces capture the most important characteristics of Web proxy workloads that are most relevant to Web proxy cache performance [11]. The input parameters of ProWGen tool are summarized in Table 1. The choice of these input parameters' values, as default values, was made in order to have a more realistic Web trace. These parameters are discussed in the next subsections.

We examined the performance of FRES-CAR against that of LRU, LFU, γ-LRU for variable-sized objects [6], HLRU [8], and PSS [1] under several workloads. Due to the lack of space we will present the most indicative of the obtained results. Before we experiment on the

[2] The ProWGen is a synthetic workload generation tool for simulation evaluation of Web proxy caches and it is very flexible.

performance of these algorithms, we examine the behavior of γ-LRU for variable-sized objects as γ varies from $1/N_i$ to 1, and FRES-CAR, as γ_k varies from $1/N_i^k$ to 1 (for each cache segment k). Furthermore, we should investigate the behavior of HLRU with respect of h value (h indicates the number of references for a certain Web object in a specific time period). The motivation for this evaluation is to use the values of γ, γ_k and h that will result in better performance. After several experiments, we conclude that the highest HR was observed when γ=0.6 in case of γ-LRU for variable-sized objects, γ_k=0.8 in case of FRES-CAR for each cache segment k, and h=6 in case of HLRU. The experimentation is carried out by considering the percentage of one-timers, the skewness of the access patterns (zipf slope), and the percentage of unique objects. Then we evaluate the impact of the above parameters on each replacement policy for a specific cache size. The size of the cache is expressed in terms of the percentage of the total number of bytes of all objects in a Web log file. In our experiments, we consider that the default value of cache size is defined as the 1% of bytes of all objects in a Web log.

4.1. One-Time Referencing

Initially, we examine the sensitivity of the proposed caching policy to one-timer objects. As mentioned earlier, one-timers are defined as the objects which are requested only once, regardless of the duration of the access log studied. Indeed, there is not use to cache one-timers, since they are never accessed again. Thus, the cache replacement algorithms should distinguish such one-timer objects in order not to reduce the cache effectiveness. Authors in [11] studied various percentages for one-timers and observed that as their percentages increase, the performance of cache replacement algorithms get a small improvement. As depicted from the Figure 1, we confirm this observation since the HR and BHR increase as the percentage of one-timer objects increase.

Concerning the performance of FRES-CAR, as depicted from the Figure 1a, its HR is significantly higher than the recency-based and frequency-based policies (10% higher HR compared to the other examined policies). On the other hand, its BHR is moderately lower than that of the recency-based policies (Figure 1b). Therefore, we conclude that FRES-CAR algorithm can effectively distinguish and isolate one-timers.

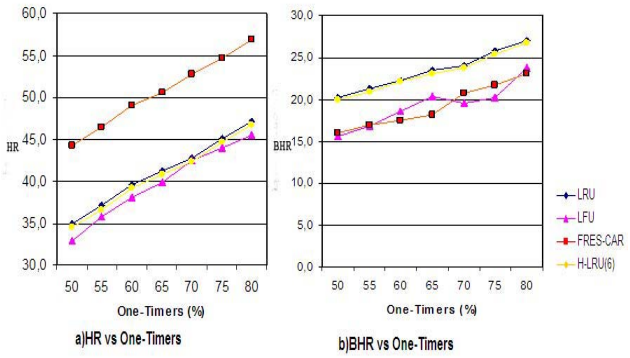

Figure 1. HR and BHR vs. One-Timers Ratio

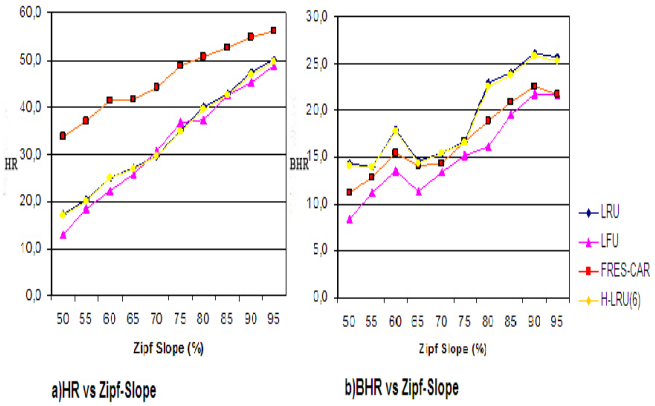

Figure 2. HR and BHR vs. Zipf Slope

4.2. Zipf-like Popularity Distribution.

A common characteristic in Web traces is the highly "uneven" distribution of references to files. As observed in the literature [2], Zipf's law and power-law are quite common on proxies. Particularly, Zipf-law has been applied to model file popularity since Zipf's law defines a power-law relationship between the popularity (*P*) of an object and its frequency (*l*). This relationship is formulated by $P=c/l^{\beta}$, where c is a constant and β is often close to 1. From the previous discussion follows that increased skewness means stronger temporal locality. Here, further experimentation was carried out in relation to the impact of the cache replacement policies on the zipf-like popularity distribution of objects. Thus, as depicted from the Figure 2, the recency-based policies are benefited from this skewness. Furthermore, we observe that the HR and BHR of all policies increase with increasing skewness. Regarding the performance of FRES-CAR, its HR achieves the highest performance comparing with all the other policies. On the other hand, we observed that its BHR is moderately lower than the recency-based policies at the slope value of 75% and higher.

4.3. Percentage of Unique Objects.

The third experiment studies the impact of the cache replacement policies on the percentage of unique objects that include in a Web log file. We observe that as the percentage of unique objects increase, the HR and BHR for all the algorithms decreases. Concerning the performance of FRES-CAR, Figure 3a depicts that it achieves a 10% to 14% higher HR comparing with the recency-based and frequency-based policies. Regarding its BHR, we observe that FRES-CAR has a quite similar performance if the percentage of unique documents is 30% or smaller.

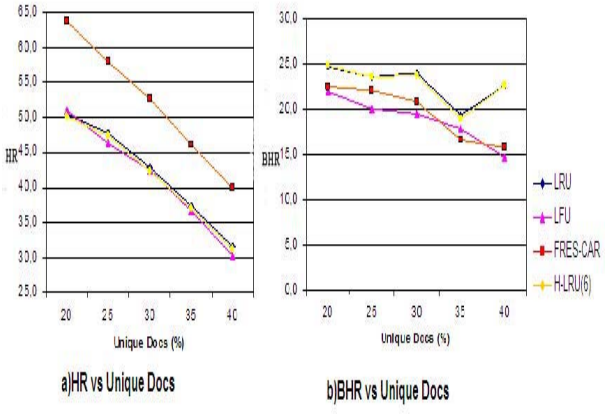

Figure 3. HR and BHR vs. Unique Docs Ratio

The most similar to FRES-CAR policies are the PSS and the γ-LRU for variable-sized objects policies. Table 2 summarizes the performance of FRES-CAR with respect to the PSS and γ-LRU for variable-sized objects.

Table 2. FRES-CAR vs. PSS and γ-LRU

FRES-CAR vs.	PSS		γ-LRU	
Cache Size	**HR**	**BHR**	**HR**	**BHR**
Large (1,5%)	+0.21%	-0.28%	+8.90%	-5.74%
Medium (1%)	+0.16%	+0.26%	+9.61%	-3.55%
Small (0,5%)	-0.10%	+0.25%	+9.45%	-0.90%
Average	+0.09%	+0.07%	+9.32%	-3.40%

Each table's cell shows the FRES-CAR's performance improvement taken over for a particular cache size. Note that a plus sign (+) indicates better performance and a minus sign (-) indicates worse performance. From this table it follows that our proposed algorithm achieves a 9% higher HR comparing with the γ-LRU for variable-sized objects. However, FRES-CAR sacrifices 3.4% of BHR in order to have such a great improvement in HR. When compared with the PSS algorithm, it improves a small amount of both HR and BHR. The low variation in hit rate values, compared with the PSS, can be explained by the common practice of segment-based cache and by the same γ_k values in all cache segments. A further experimentation with varying γ_k is underway.

5. Conclusion and Future Work

Caching is a typical approach for improving the performance of Web-based systems. In this paper, we introduced the FRES-CAR replacement algorithm, where the decision about which objects should be evicted is determined by unifying three object's factors: frequency, recency and size. Trace-driven simulation was employed to evaluate and comment on the performance of the proposed caching scheme. Results have indicated that FRES-CAR algorithm "sacrifices" a small amount (~3%) of BHR in order to improve significantly the HR.

Further research should extend the proposed work in order to study the performance of FRES-CAR under real data traces and under different values of γ_k per segment. Finally, research efforts are underway towards extending the proposed work under a Content Delivery Network (CDN) [9].

6. References

[1] C. Aggarwal, J. Wolf, P. Yu, "Caching on the World Wide Web", *IEEE Transactions on Knowledge and Data Engineering*, 11(1), 1999, pp. 94-107.

[2] M. Busari, C. Williamson, "ProWGen: A Synthetic Workload Generation Tool for the Simulation Evaluation of Web Proxy Caches", *Computer Networks*, 38(6), Jun. 2002, pp. 779-794.

[3] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms", Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems, Monterey, California, USA, Dec. 1997, pp. 193–206.

[4] D. Katsaros, Y. Manolopoulos, "Caching in Web Memory Hierarchies", Proceedings of the ACM Symposium on Applied Computing, Nicosia, Cyprus, Mar. 2004, pp. 1109-1113.

[5] S. Podlipnig, L. Boszormenyi, "A Survey of Web Cache Replacement Strategies", *ACM Computing Surveys*, 35(4), 2003, pp. 374-398.

[6] K. Psounis, A. Zhu, B. Prabhakar, R. Motwani, "Modeling Correlations in Web Traces and Implications for Designing Replacement Policies", *Computer Communications*, 45(4), Jul. 2004, pp. 379-398.

[7] M. Rabinovich, O. Spatsheck, *Web Caching and Replication*, Addison Wesley, 2002.

[8] A. Vakali, "Proxy Cache Replacement Algorithms: A history-based approach", *World Wide Web Journal*, 4(4), 2001, pp. 227-297.

[9] A. Vakali, G. Pallis, "Content Delivery Networks: Status and Trends", *IEEE Internet Computing*, Vol. 7(6), 2003, pp. 68-74.

[10] S. Williams, M. Abrams, C. Standridge, E. Fox, "Removal Policies in Network Caches for World Wide Web Documents", Proceedings of the ACM SIGCOMM Conference, Stanford University, Aug. 1996, pp. 293–305.

[11] C. Williamson, "On Filter Effects in Web Caching Hierarchies", *ACM Transactions on Internet Technology*, 2(1), 2002, pp. 47-77.