RC 20619 11/08/96 (revised 03/05/97)
Computer Science 22 pages

# Research Report

## On Caching Policies for Web Objects

Charu C. Aggarwal, Joel L. Wolf, Philip S. Yu

IBM Research Division
Thomas J Watson Research Center
Yorktown Heights, New York 10598

Marina Epelman

Operations Research Center
Massachusetts Institute of Technology
Cambridge, Mass 02139



Research Division
Almaden · T.J. Watson · Tokyo · Zurich

# On Caching Policies for Web Objects

Charu Aggarwal*, Joel Wolf*, Philip Yu* and Marina Epelman**

* IBM T.J. Watson Research Center, Yorktown Heights, New York

** Massachusetts Institute of Technology, Cambridge, Massachusetts

**Abstract**

With the increasing popularity of the world wide web, it is desirable to store copies of popular documents in client, proxy or server caches and thus reduce the delay times for URL requests. In this paper we study the caching problem for web documents. Specifically, we propose a pyramidal selection scheme which maintains multiple stacks based on document sizes in order to perform efficient cache replacement. In addition we discuss the concept of "admission control", which decides whether an object should or should not be cached in the first place. We analyze the performance of our schemes and others by means of event and trace driven simulation. We shall see that the result is an efficient general purpose scheme, whose performance is relatively insensitive to the nature of the work loads.

# 1 Introduction

The recent increase in popularity of the world wide web has led to a considerable increase in the amount of traffic over the internet. As a result, the web has now become one of the primary bottlenecks to network performance. When documents are requested by a user who is connected to a server on a slow network link, there is generally considerable latency noticeable at the client end. Further, transferring the document over the network leads to an increase in the level of traffic. This has the effect of reducing the bandwidth available for competing requests, and thus increasing latencies for other users. In order to reduce access latencies, it is desirable to store copies of popular documents closer to the user.

Consequently, web caching has become an increasingly important topic [1, 6, 11, 15, 16, 19]. Caching can be implemented at various points in the network. On one end of the spectrum, there is typically a cache in the web *server* itself. Further, it is increasingly common for a university or corporation to implement specialized servers in the network called caching *proxies*. Such proxies act as agents on the behalf of the client in order to locate a cached copy of a document if possible. More information on caching proxies may be found in [14]. Usually caching proxies and web servers behave as secondary or higher level caches, because they are concerned only with misses left over from *client* caches. Client caches are built into the web browsers themselves. They may store only those accesses from the current invocation (so-called *non-persistent* cache), or they may retain documents between invocations. Mosaic, for example, uses a non-persistent cache.

In this paper, we shall discuss a general cache replacement policy for non-uniform size objects for main memory caching. The results are applicable to web server, proxy and client caches.

One of the complications in implementing cache replacement policies for web documents is that the objects to be cached are not necessarily of homogeneous size. For example, if two objects are accessed with equal frequency, the hit ratio is maximized when the replacement policy is biased towards the smaller document. This is because it is possible to store a larger number of documents of smaller size. At the same time, the transfer time saving obtained by caching a larger document is more, though the relationship of such saving to size is rarely linear. Thus, while deciding which documents to replace when a new document enters, we must take into account not only the relative frequency, but also factors such as the sizes or transfer time savings. In the standard *least recently used (LRU)* caching algorithm we maintain a list of the objects in the cache which is ordered based on the time of last access. In particular, the most recently accessed object is at the top of the list, while the least recently accessed object is at the bottom. When a new object comes in and the cache is full, one object in the cache must be pruned in order to make room for the newly accessed object. The object chosen is that one which was least recently used. In the most straightforward extension of LRU, one would prune off as many of the least recently used objects as is necessary to have sufficient space for the newly accessed object. This may involve zero, one or many replacements. Thus, this extension of LRU takes size into account only peripherally while performing the cache replacement decisions. As we shall see, such a replacement policy turns out to be naive in practice.

A related issue to that of replacement is *admission control.* In other words, when should we allow an object to enter the cache at all? It may not always be favorable to insert an object into the cache, because it may lower the probability of a hit to the cache.

Abrams et. al., Williams et. al [1, 20] have considered some cache removal policies which do take sizes at least partly into account. Below are some of the policies they have evaluated:

(1) **LRU policy:** In the LRU policy, the idea is to remove objects based on their recency of use. For example, if we need to make space for an object of size 10 MB, entering into the cache, then we remove the object which was used least recently. If this object has size less than 10 MB, then we again remove the least recently used object among the remaining objects and so on until as much as 10 MB of space has been created.

(2) **LRU-MIN policy:** This policy tries to bias in favor of smaller sized objects so as to minimize the number of documents replaced. Let the size of the incoming object be $S$. Suppose that this object will not fit in the cache. If there are any objects in the cache which have size at least $S$, remove the least recently used such object from the cache. If, on the other hand, there are no objects with size at least $S$, then we start replacing objects in LRU order of size at least $S/2$, then objects of size at least $S/4$, and so on until enough free cache space has been created. This policy can also be combined with an admission control policy.

(3) **SIZE policy:** In this policy, the objects are removed in order of size, with the largest object removed first. Needless to say, ties on size are somewhat rare, though such ties are broken by taking into account the time since last access. (Documents with higher time since last access are removed first.)

Note that all of these policies take into account either the size or the time since last access or both. It was concluded in [1, 20] that policies which take into account the size tend to perform better than those which do not. This is because removing larger documents makes room for many small documents.

In this paper we devise a replacement and admission control policy which appears to achieve performance better than any of the above schemes. Further, we shall see that the scheme we propose is quite general purpose, and works well on different kinds of workloads. We show simulations on both server as well as proxy traces to this effect.

The paper is organized as follows. In Section 2 we concentrate on formulating a theoretical model which discusses the impact of sizes on an LRU policy. We build a model which incorporates the effect of this impact. We show how this optimization model can be approximately solved by a simple heuristic, and how access costs may be handled by the model. In Section 3 we show how to make this heuristic more easily implementable in practice. We call the resulting policy the *pyramidal selection scheme (PSS).* In Section 4 we discuss an admission control policy. Empirical results and simulations are presented in Section 5. Finally in Section 6, we present a summary and conclusion.

## 2 What is LRU replacement?

The base policies considered in Abrams et. al., Williams et. al. [1, 20] and other work as "LRU removal policies", are not the only ways of achieving a generalization of LRU in order to handle nonuniform object sizes. When a document is to be inserted into the cache, more than one document may need to be removed in order to make way for the incoming object. In the LRU removal policies discussed in [1, 20], documents are greedily removed from the cache in the order of recency of last access, until enough space is created for the incoming document.

In this paper, we discuss a policy which attempts to more closely emulate LRU replacement. The idea in this replacement policy is to try to devise an objective function, and globally choose all the outgoing objects from the cache at one time (rather than choosing one by one in order of sort keys) while optimizing this objective function. In order to emulate LRU replacement effectively, this objective function should be such that it should try to achieve the same criteria as the LRU policy for uniform sized objects.

We assume that there are $N$ objects, indexed by $1, 2, \ldots, N$. We assume that the size of the $i$th object in bytes is $S_i$.

An iteration counter is maintained which is incremented each time there is a request for an object. Let the set of objects in the cache at the $k$th iteration be denoted by $C(k)$. Let $i_k$ denote the index of the object accessed at the $k$th iteration. If $i_k$ is already present in the cache before the $k$th iteration, then we have a hit, and it does not need to be brought into the cache. On the other hand, if $i_k$ is not present, then we have a miss. Assuming $i_k$ satisfies the admission control requirements, we have to decide which objects to purge from the cache.

Consider the decision variable $y_i$ for object $i$ which is 1 if we wish to purge it, and 0 if we want to retain it. The decision variable $y_i$ is defined only for objects which are present in the cache. We assume that $\Delta T_{ik}$ is the number of accesses since the last time document $i$ was accessed. with the most recently used object ranked first in the queue. This number is well-defined for all objects which have, in fact, been accessed before.

For ease in exposition, we shall refer to $1/\Delta T_{ik}$ as the *dynamic frequency* of object $i$, at iteration $k$.

Note that the LRU policy for uniform size objects removes the object with the least dynamic frequency from the cache, and thus tends to retain the objects with high frequency of access. While the dynamic frequency is a poor estimator of the true relative frequency of an object, it tends out to be a very robust algorithm in practice at least for the case of uniform size objects. When multiple objects need to be removed from the cache, we choose the outgoing objects such that the sum of their dynamic frequencies is as small as possible.

This interpretation of dynamic frequencies leads to a model which is a direct generalization of LRU, though in a different way from the schemes previously described. Specifically, we get the following model:

$$\text{Minimize} \sum_{i \in C(k)} y_i / \Delta T_{ik}$$

$$\text{subject to:} \quad \sum_{i \in C(k)} S_i \cdot y_i \geq R$$

$$y_i \in \{0, 1\}.$$

This is a knapsack problem. By the nature of the formulation we can make the following observation: *We seek to eject those objects for which the sum of their* $1/\Delta T_{ik}$ *values is as small as possible.*

Recall that the LRU scheme for uniform size objects purges the object with the least value of $1/\Delta T_{ik}$. In fact, when this model is applied to a problem in which all the sizes $S_i$ are the same, we get the trivial solution in which we eject the object with the least value of $1/\Delta T_{ik}$.

## 2.1 A Simple Heuristic for the Replacement Policy

As we have noted, the mathematical programming problem considered in the previous section is a version of the knapsack problem [3]. This problem is known to be NP-hard. However, fast heuristics exist and they do well in practice.

One well-known knapsack problem heuristic is the following greedy policy: Order the objects by the ratio of cost to size. Then choose the objects with the best cost-to-size ratio, one by one, until no more can fit into the knapsack.

The cost-to-size ratio for the object $i$ is $1/(\Delta T_{ik} \cdot S_i)$. So, reindex the objects $1, 2, \ldots, |C(k)|$ in order of non-decreasing values of $\Delta T_{ik} \cdot S_i$. After sorting we have:

$$\Delta T_{1k} \cdot S_1 \leq \Delta T_{2k} \cdot S_2 \leq \ldots \leq \Delta T_{|C(k)|k} \cdot S_{|C(k)|} \tag{1}$$

Then we greedily pick the highest index objects one by one and purge them from the cache until we have created sufficient space for the incoming object. This policy may however be somewhat hard to implement in practice because of the difficulty in comparing the product of the size and the time since last access for each and every object in the cache. A practical solution is illustrated in a later section.

## 2.2 Handling General Access Costs

The schemes of the previous section are intended to maximize the hit probability of the cache. Although the hit probability may be a good measure to maximize while designing cache replacement strategies, it could be argued that not all objects are the same in terms of access costs. Thus, although we have been essentially assuming that a fixed cost predominates, the transfer time savings for larger objects are higher. The relationship of transfer time savings to size is however somewhat noisy, and far from linear. For example, the benefits obtained by caching a document in a proxy cache which is requested from a more distant server is likely to be more than that which is requested from a nearby server. The purpose of this subsection is to notice that the above schemes may easily be extended to handle non-uniform access costs.

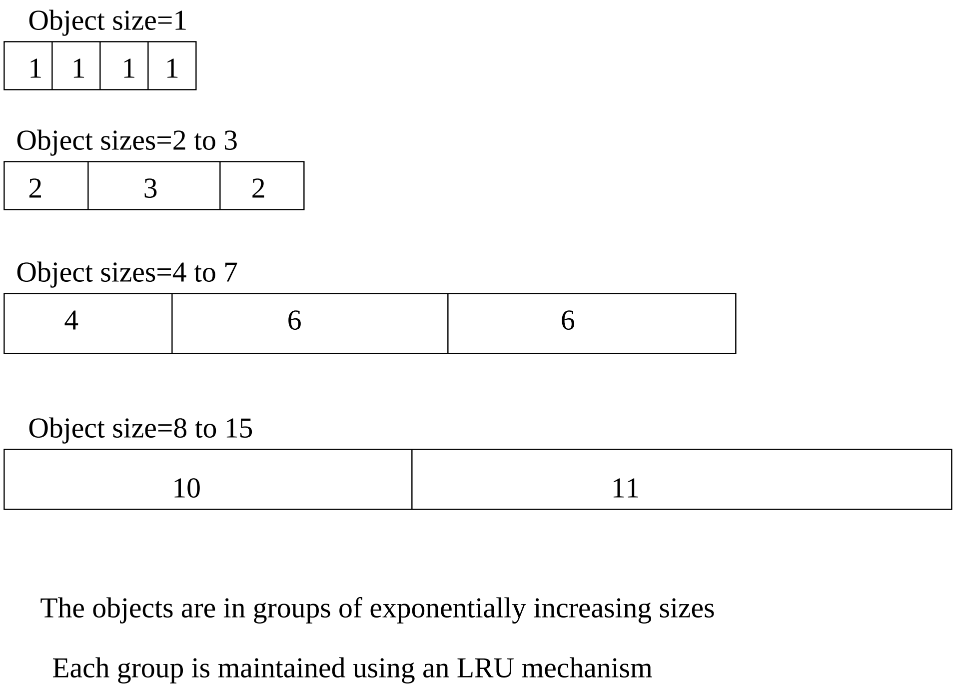


Figure 1: The Pyramidal Selection Scheme

Access costs may often be an important issue while handling documents from proxy servers. We assume that the cost of accessing a document is equal to the transfer-time avoided. This value can be actually measured at the cache itself.

Let $c_i$ be the cost of accessing object $i$. This factor can be taken into account by modifying the objective function value appropriately. Thus, the generalized objective function is $\sum_{i \in C(k)} c_i \cdot y_i / \Delta T_{ik}$. Similarly, the generalized size adjusted LRU rule would place objects in non-increasing order of $S_i \cdot \Delta T_{ik} / c_i$.

# 3 The Pyramidal Selection Scheme

So far we have talked about how a good selection factor may be used in order to maximize the hit ratio of an object from the cache. An important question which arises is the implementation of such a rule. In several cases, it may be impractical to scan each and every object in the cache and compare their $S_i \cdot \Delta T_{ik}$ values in order to select which objects to eject from the cache. We shall use the intuition gained from the previous section in order to propose the following *pyramidal selection scheme*, or *PSS*.

The primary idea behind the PSS scheme is that we make a pyramidal classification of objects depending upon their size. Thus, all objects of group $i$ have sizes ranging between $2^{i-1}$ and $2^i - 1$. Thus, there will be $N$ different groups of objects: $\{1, 2, \ldots, N\}$, where the value of $N$ is chosen in such a way that $2^N$ is an upper bound on the capacity of the cache itself. In other words, $N = \lceil \log(\mathrm{M}+1) \rceil$. (One could imagine combining some of the resulting groups if objects of corresponding sizes occur infrequently. This would have the effect of reducing the number of partitions $N$ accordingly.)

For the main cache, as illustrated in Figure 2, the objects in each group $i$ are maintained as a separate LRU list. (In the auxiliary cache, we maintain all the URL and time stamps of the objects on the same LRU list.) Whenever we need to decide which object to eject from the cache, we compare the $S_i \cdot \Delta T_{ik}$ values of only the least recently used objects in each group. The result of using this mechanism is that we will choose the object with the largest overall value of $S_i \cdot \Delta T_{ik}$ to within a factor of 1/2 even in the worst case.

**Theorem 3.1** *Let $Z^*$ be the least value of $S_i \cdot \Delta T_{ik}$ among all the objects $i$ in the cache, and let $Z^0$ be the corresponding value for the PSS scheme. Then $Z^0 \geq (1/2)Z^*$.*

**Proof:** Let us define the PSS *group leaders* as the set of least recently used objects in each of the $N$ groups. Let $j$ be the object among these group leaders chosen by the PSS scheme, and let $m$ be the object with the optimal value of $S_i \cdot \Delta T_{ik}$ when request $k$ is received. Let $l$ be the least recently used object (group leader) in the group to which object $m$ belongs. Then, we must have $S_l \geq S_m/2$ and $\Delta T_{lk} \geq \Delta T_{mk}$. Consequently, we must have $S_l \cdot \Delta T_{lk} \geq S_m \cdot \Delta T_{mk}/2$. But, since $j$ is the optimal object within the set of group leaders and $l$ is also a group leader, it must be the case that $S_j \cdot \Delta T_{jk} \geq S_l \cdot \Delta T_{lk}$. Combining the above two inequalities, we get that $S_j \cdot \Delta T_{jk} \geq S_m \cdot \Delta T_{mk}/2$. In other words, $Z^0 \geq (1/2)Z^*$. ■

In reality, the value of $Z^0$ is typically so close to $Z^*$ that the performance difference between the PSS scheme, and a policy which uses a direct size-time product is almost imperceptible. We shall illustrate a comparison of the hit ratios of these two policies in the next section.

Note that a scheme called log(SIZE), discussed in the Williams et. al.[20] paper has an interesting connection with the PSS scheme, which was derived completely independently. The log(SIZE) scheme always chooses the least recently used item in the non empty stack with the largest range of sizes. However, the PSS scheme looks at the least recently used object of each stack, and among these picks the object which has the least product of the $S_i \cdot \Delta T_{ik}$.

Again, the PSS scheme can be adapted to the case when there are access costs. Instead of grouping together documents based upon the value of their sizes, we group together documents based upon geometrically increasing ranges for the value of $S_i/c_i$. Here $c_i$ is the transfer time latency measured at the cache; in other words it is the effective transfer time saving.

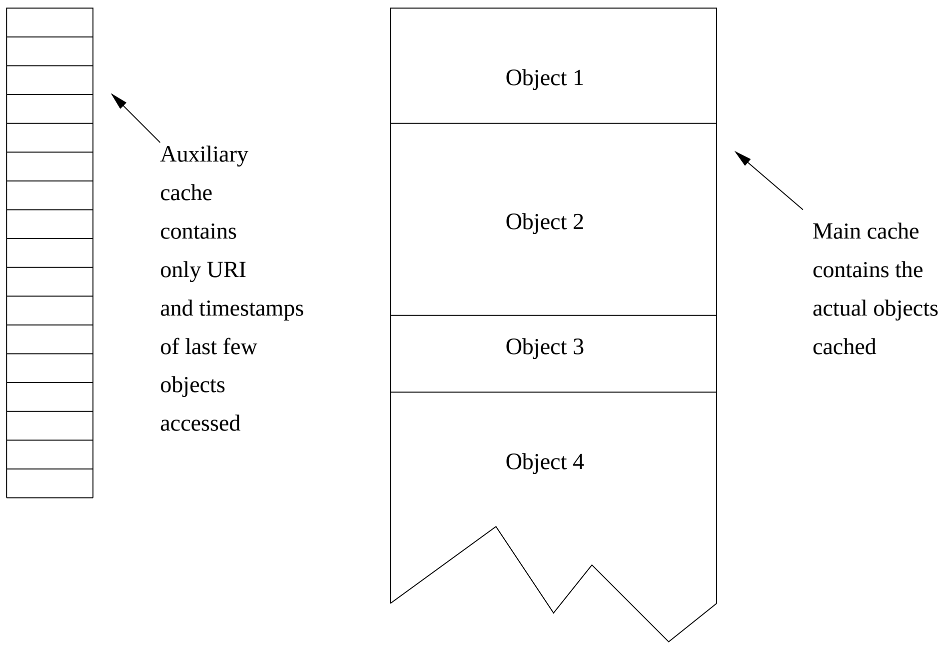


Figure 2: Performing Admission control

# 4 An Admission Control Policy

An *admission control policy* decides whether or not it is worthwhile caching an object in the first place. Having a good admission control policy is especially important when caching non-uniform size objects, because a considerable amount of disruption can be caused when an object is added and others are purged from the cache. Having too frequent replacements may lead to wasted space and to storing objects which are never hit at all. Admission control makes the scheme less sensitive to the transients in the workload.

In order to do a good job with admission control, we propose the construction of a small auxiliary cache which maintains the *identities* of some number $X$ of objects. [1] In this auxiliary cache we also maintain a "timestamp" of the last access of an object, measured in terms of number of object accesses.[2] Thus, a counter is incremented each time an object is requested from the cache, whether or not that request can be filled. Note that the auxiliarly cache contains identities of objects rather than the objects themselves, so that its size is negligible compared to that of the main cache. The auxiliary cache is maintained in strict LRU order. Thus, this cache contains a list of the identities of the more popularly accessed objects, though

[1] As a rule of thumb, we set $X$ to be about twice the average number of objects in the main cache fro the purpose of this paper. Needless to say, other choices may exist which result in even better performance. However, we shall leave the problem of finding the optimum auxiliary cache size to future work.

[2] If access cost savings rather than hit ratios are considered as the caching objective function, then the transfer times for the last access of the corresponding documents are stored in the auxiliary cache.

the number of objects in it is approximately twice the average number of objects in the main cache. Figure 1 illustrates the auxiliary and main caches.

It is important to note that the replacement policy in the auxiliary cache does not depend upon the replacement policy in the main cache in any way. Whenever there is a request for an object, a replacement occurs in the auxiliary cache using pure LRU replacement policy, whether or not that object is available in the main cache.

We would like to have an admission control policy which ensures that at the $k$th iteration the incoming object $i_k$ is popular enough to offset the loss of the objects it displaces. So we proceed as follows: In the event that there is enough free space available for the object $i_k$, we simply bring $i_k$ into the cache. Otherwise, we check if $i_k$ occurs in the auxiliary cache. If it does not, then the object $i_k$ does not enter the main cache. (It is however, added to the auxiliary cache in accordance with LRU rules.) On the other hand, if $i_k$ does occur in the auxiliary cache, then we decide if the decision which the replacement policy heuristic makes would be profitable. That is, we compare the dynamic frequency $1/\Delta T_{i_k k}$ with the sum of the dynamic frequency values (or $\sum_i 1/\Delta T_{ik}$) of the set of candidate outgoing objects determined using the scheme described in the previous section. We admit an object only if it is profitable to do so. Observe that the information needed for the dynamic frequency calculation of object $i_k$ can be obtained from the timestamp in the auxiliary cache. After this iteration, the time stamp of the object $i_k$ is changed to $k$. Thus, the candidate set of possible objects to be purged from the cache is determined before the decision is reached about whether or not an object ought to enter the cache. If the admission control policy says that the new object is not to be brought in, then the candidate set is never actually purged from the main cache.

The above method of admission control is for the case when the hit ratio is used as primary objective function. In order to consider the access costs of the documents, we modify the comparison conditions in the admission control policy. In other words we compare the value $c_{i_k}/\Delta T_{i_k k}$ of the incoming document with the value $\sum_i c_i/\Delta T_{ik}$ of the candidate outgoing objects.

The idea of admission control bears some resemblance in spirit to the 2Q approximation of LRU-K proposed by Johnson and Shasha [12, 17] in making the cache more robust to transients in the workload; however, the method of doing so is different in this case.

# 5 Empirical Results

In this section we compare the performance the performance of three different caching schemes: the naive LRU policy extension, the LRU-MIN policy of [1], the SIZE policy discussed in [20] and the PSS policy which we have proposed in this paper. We employ both event and trace driven simulations.

Note that the PSS scheme was obtained as a direct extension of a size-adjusted replacement policy, which we call SLRU (Size-adjusted LRU). We compare SLRU to PSS in order to show that the performance of the two schemes is virtually identical. Both of these schemes were implemented in conjunction with admission control.

We are interested in examining the performance of the algorithms under the assumption that objects have varying sizes, relative frequencies, and combinations of these two factors. The following were the key performance metrics.

(1) A primary measure is, of course, the hit ratio.

(2) Often it may be desirable to find the transfer time savings (or cost savings) instead of the hit ratio. We discuss the performance of the different schemes in terms of cost savings.

(3) A secondary measure is the *robustness* of the scheme. It is well known that the performance of caching policies for web objects often depend upon whether smaller objects have higher frequency or vice versa. In order to check the performance of the schemes for varying size-frequency correlations, that the LRU scheme is very robust for uniform size objects and varying distributions of relative frequencies. All the schemes that are compared are in fact generalizations of LRU, and consequently it is useful to see how the correlation of size and frequency factors into the robustness of the proposed schemes.

## 5.1 Issues in the Simulation

Our primary motivation for the event driven simulation was to understand the effect of the varying parameters on the performance of the schemes. The effects on real workload are discussed in the trace driven simulation. An issue that we faced in the trace driven simulation was the fact that access costs were not present in the traces available to us. In order to test the effects of costs we used the event driven simulations. Our motivation in this simulation was to test how the performance of the schemes varied depending upon correlation of cost with size. This is quite important considering the fact that the transfer of an object from its original location to a proxy cache is not necessarily directly proportional to its size. For example, for a proxy cache in the US, the transfer time for a document located at a site in Japan may be quite high, even though the document may be small. On the other hand, for the case of a server cache, the transfer time saved for the documents because of caching may be very strongly correlated to their size. We shall see that the PSS scheme performs well under all varieties of correlations.

## 5.2 Event Driven Simulation

We test the caching schemes on objects of sizes uniformly distributed between 1 and 500. That is, we assume that there are 500 different objects, with one object of each size $i$. The objects were chosen to have Zipf-like frequency distributions [13]. The reason that we choose a Zipf-like distribution was to study the effect of frequency skew on the caching policies. Thus the frequency of object $i$ is proportional to $1/\pi(i)^\theta$, where $\theta$ is the Zipf parameter, and $\pi$ is a permutation vector. By varying $\pi$ we can affect the relationship between the size and frequency.

Since we used the PSS scheme as a more implementable surrogate for SLRU, it is useful to compare the performance of the PSS and the SLRU schemes. In Figure 3 we consider the

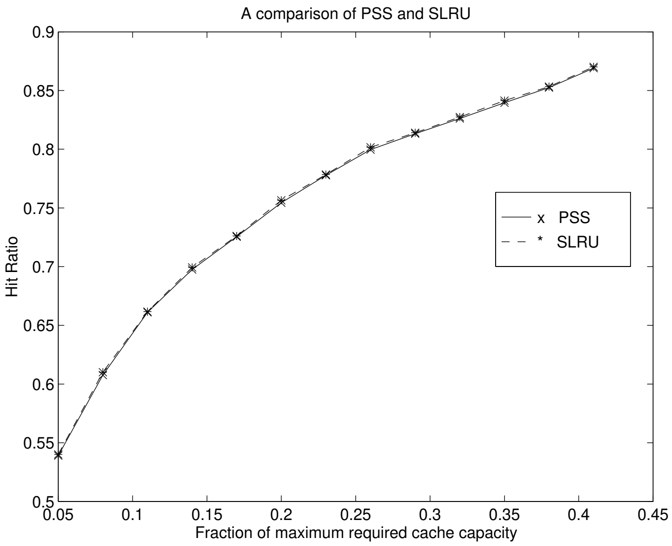


Figure 3: Performance of the PSS and SLRU Schemes

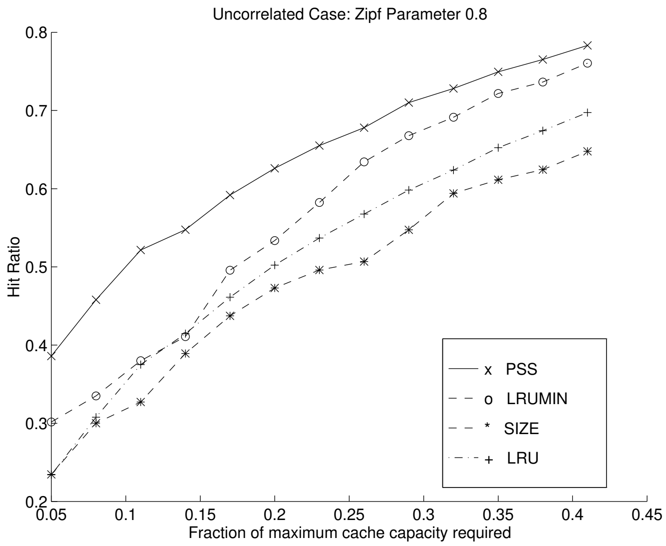


Figure 4: The Uncorrelated Zipf Frequency Case: Hit Ratio

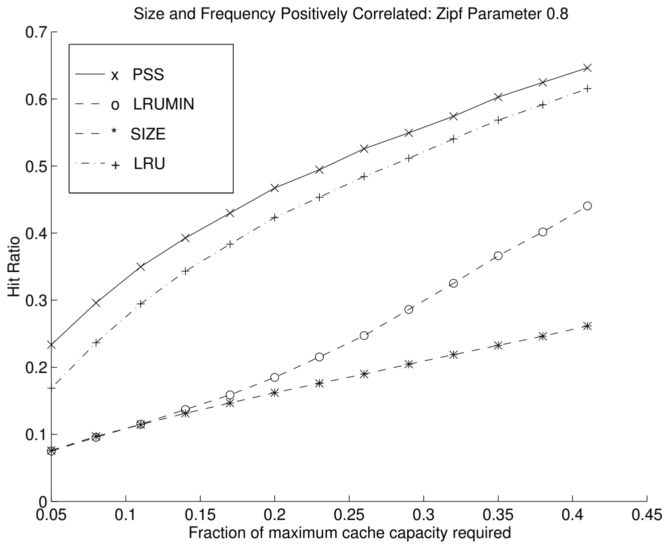


Figure 5: The Positively Correlated Zipf Frequency Case: Hit Ratio

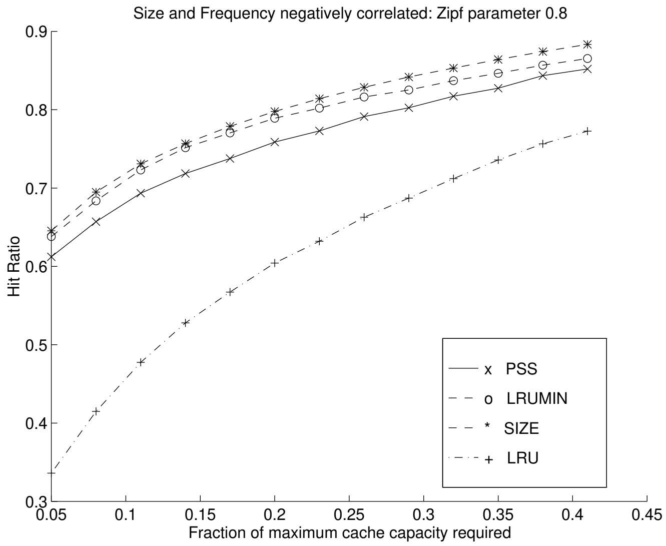


Figure 6: The Positively Correlated Zipf Frequency Case: Hit Ratio

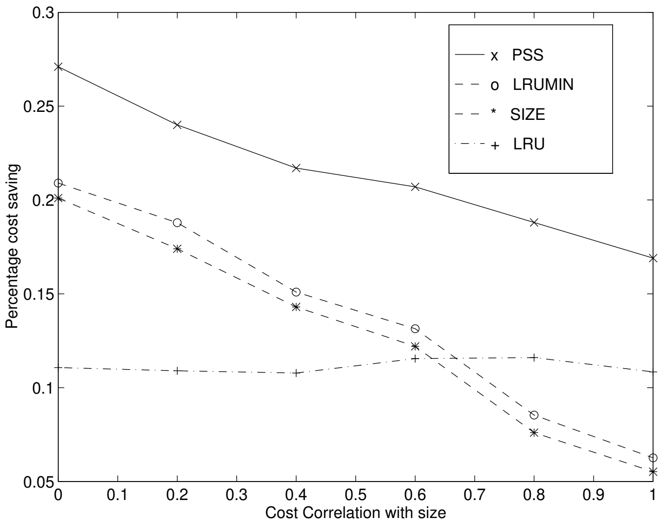


Figure 7: The Uncorrelated Case: Cost Savings

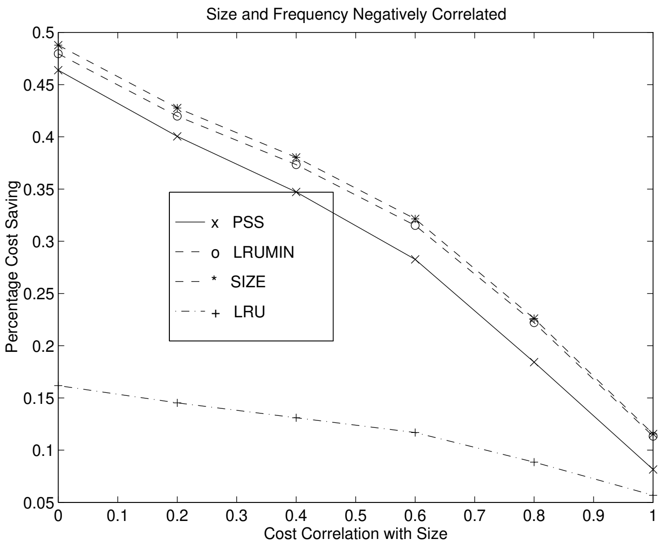


Figure 8: The Negatively Correlated Case: Cost Savings

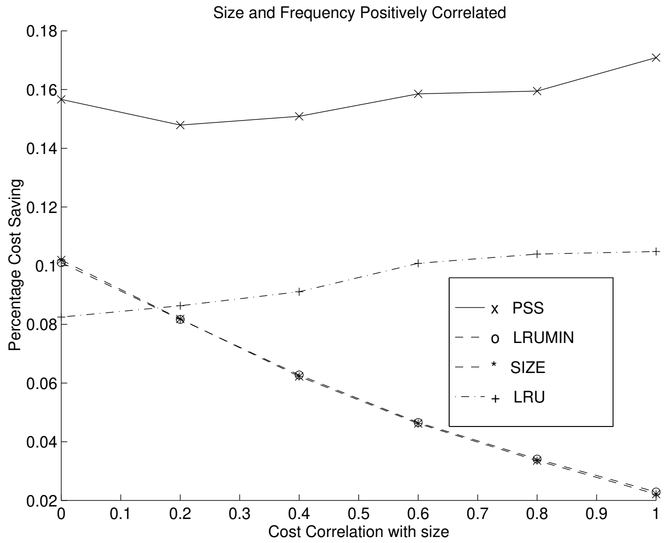


Figure 9: The Positively Correlated Case: Cost Savings

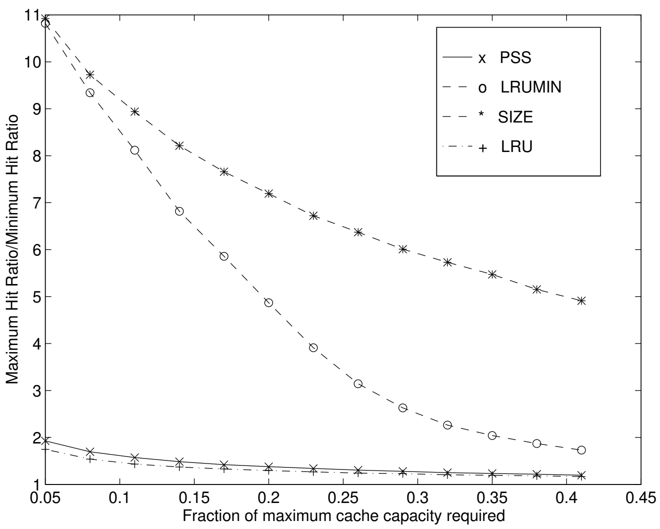


Figure 10: Robustness

case where object size and frequency are uncorrelated. That is, we choose $\pi$ randomly. In this case, the frequency of object $i$ is chosen to be $1/\pi(i)$. As we see from the figure, the hit ratios of these two are so similar that it is almost impossible to distinguish between them. In other words, the PSS scheme is in practice no worse than its theoretical ancestor.

We test three different relationships between the frequency of an object and its size. For each of these cases, we consider a frequency Zipf parameters equal to 0.8.

(1) There is no correlation between the object size and the frequency. In this case, we see that the LRU-MIN scheme outperforms LRU most of the time, though for relatively small cache sizes and very skewed frequency distributions the situation is reversed. The SIZE policy is outperformed by both the LRU as well as the LRUMIN policies. The reason for this is that the SIZE policy is unable to take advantage of the frequency skew. A lack of correlation between size and relative frequency prevents this. The PSS policy consistently outperforms all of these schemes.

(2) There is complete positive correlation between object size and frequency. This the value of $\pi(i)$ is chosen to be $501 - i$, so that the $i$th object has size $i$ and relative frequency $1/(501 - i)^{\theta}$. Thus the largest object has the highest frequency, and the smallest object has the lowest frequency. The corresponding hit ratio curve is shown in the Figure 5. As we can see, in this case the LRU-MIN policy and the SIZE policy perform very poorly since they are biased too strongly towards objects of smaller size. In the positively correlated case, smaller size objects have a lower frequency. Thus, the policies corresponding to SIZE and LRUMIN are both bad strategies for the positively correlated case, since it often results in a more frequently accessed object being displaced for the sake of a less frequently accessed object. The normal advantage of keeping small objects in the cache is offset by the fact that the cache gets clogged with many infrequently accessed objects. In this case, even the LRU policy outperforms LRU-MIN by a substantial margin. The performance of the PSS policy, however, is clearly best.

(3) There is complete negative correlation between object size and frequency. In other words, $\pi(i)$ is chosen to be equal to $i$, so that the $i$th object has a size of $i$ and frequency of $1/i^{\theta}$. Thus, the largest object has the lowest frequency and vice-versa. Figures 8 and 9 show the corresponding hit ratio curves. This is the most favorable case for the LRU-MIN policy: The small objects have high frequency, and the LRU-MIN scheme generally retains these objects in its cache. However, even in this case, we see that it performs only marginally better than the PSS policy. The LRU policy, on the other hand, does not do well at all. This is because large objects will occasionally enter the cache and displace many frequently accessed small objects.

We also tested how the various policies performed when access costs instead of hit ratios were considered. Note that the access cost of an object may not necessarily be directly related to size. This is especially the case for proxy caches in which web documents may be at varying distances from the cache. In order to test the effect of different kinds of relation between the size of an object and its access cost, we used the following method to assign access costs.

$$\text{Cost(document)} = K_1 \cdot \text{Size Component} + K_2 \cdot \text{Noise Component}$$

By varying $K_1$ and $K_2$, it is possible to test the effect of different kinds of access costs on the performance. More specifically, the values of $K_1$ and $K_2$ are decided as follows. We choose a cost correlation parameter $q$ which lies between 0 and 1. Let $A$ be the average size of a document. In other words, $A = \sum_{i=1}^{500} S_i \cdot f_i$, where $f_i$ is the relative frequency of document $i$. Then, the cost $C_i$ of accessing a document $i$ is as follows:

$$C_i = q \cdot S_i + 2 \cdot (1 - q) \cdot A \cdot \text{random}(0, 1)$$

Here $\text{random}(0, 1)$ is a random number between 0 and 1. Note that when $q = 0.5$, the noise and size components contribute equally to the access costs on the average because of the way in which the noise component is scaled by the value $A$. Increasing $q$ from 0 to 1 increases the size component in the access cost.

We tested the three schemes for differing values of the parameter $q$. We choose the cache capacity to be $0.05 * \sum_{i=1}^{500} S_i$ and the zipf parameter to 0.06. Again, we tested for the different cases, depending upon whether size and frequency were uncorrelated, positively correlated, or negatively correlated. These cases were plotted in the Figures 7, 8, and 9 respectively. On the X-axis we have the value of the parameter $q$, while on the Y-axis, we have the percentage cost saving by caching. The following were some of the observations we made:

(1) The effect of size-frequency correlation on the relative performance of the schemes was pretty much the same, as it was for the case when hit-ratio was used as the performance measure. This is substantiated by the fact that in the negatively correlated case (Figure 8), the LRU-MIN and SIZE policies perform very well. However, in other cases these policies do not fare very well.

(2) The effect of cost correlation with size had a greater effect on the performance of the LRU-MIN and SIZE policy and the least effect on the LRU scheme. For example, in the case of the Figure 7, as the cost correlation with size increases, the performance of the LRU scheme remains virtually unchanged. This is because the LRU scheme does not discriminate against smaller or larger objects, and a differing structure of access cost does not affect the performance of the policy. All the other schemes (PSS, LRU-MIN and SIZE) discriminate against larger objects; consequently when cost correlation with size increases, the cost savings are reduced as well, even though the hit ratios are the same. Since, the LRU-MIN and the SIZE schemes are most aggressive in discriminating against larger objects, this effect is felt most strongly in these schemes. In the uncorrelated case for example, the LRU policy is actually better than the LRU-MIN and SIZE policies when cost correlation to size is high.

(3) On the whole, in terms of access costs, the PSS policy usually performs competitively or better than the best of the other three schemes (LRU-MIN, SIZE, and LRU). Even in the negatively correlated case, the PSS scheme is only marginally worse off than the LRU-MIN and SIZE schemes.

We examine the robustness of the schemes to the size-frequency correlation. In general, for any given scheme and choice of Zipf Parameter and cache capacity, we expect the hit ratio

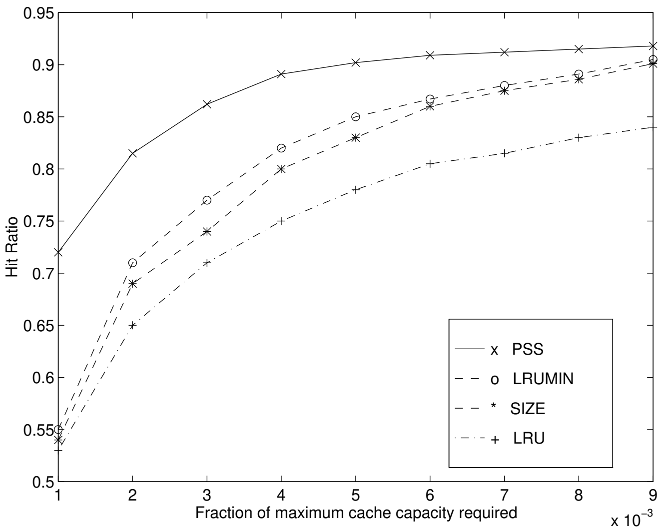


Figure 11: Server Trace Driven Simulation

of the scheme corresponding to the negatively correlated case to be much higher than the hit ratio for the postively correlated case. This is because in the negatively correlated case smaller objects have higher frequency, and this is beneficial from the point of view of efficiency in the occupancy of the cache.

$$\text{Robustness(Policy A)} = \frac{\text{Hit Ratio(Policy A, Negatively Correlated Case)}}{\text{Hit Ratio(Policy A, Positively Correlated Case)}} \tag{2}$$

For a given scheme, we expect this ratio to be larger than 1. In general, we desire a policy to be predictable, and not vary too much depending upon the characteristics of the workload. Thus good schemes will have lower values of robustness which are closer to 1. As we see from Figure 10, the LRU-MIN and SIZE policies are the least robust. This is because these policies always tends to keep the smallest objects in the cache, even if they are less frequently accessed. Consequently, (as evidenced by Figures 5 and 9) in the positively correlated case, these schemes perform very poorly. Both the PSS and the LRU schemes are the most robust. However, the LRU policy is dominated by the PSS scheme on the primary performance measure (hit ratio) in almost every case, no matter how size and frequency are correlated. Furthermore, the PSS policy is only marginally worse than the LRU scheme in terms of robustness.

## 5.3 Trace Driven Simulation

Aside from the parametric simulations which have been described above, we also performed some trace driven simulations using data from both server as well as proxy traces. We chose both kinds of traces in order to show that the PSS algorithm is quite general purpose in that the performance is good for both kinds of workloads. For each of the logs, we had traces whose lengths were between 100000 to 150000 user accesses each. Most of the frequently accessed

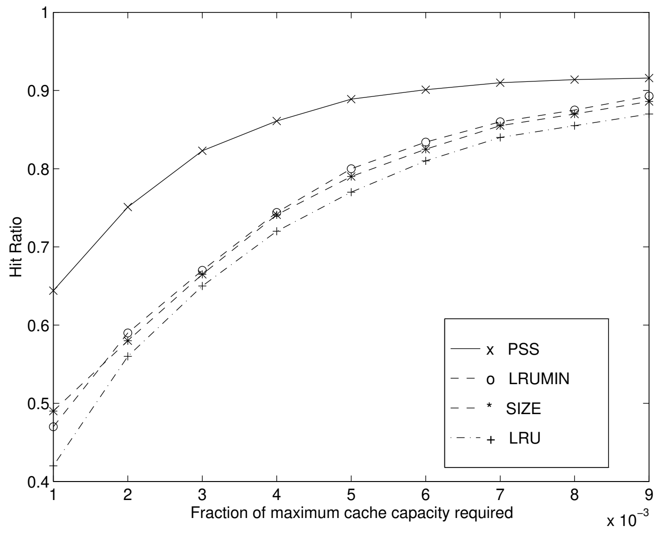


Figure 12: Server Trace Driven Simulation

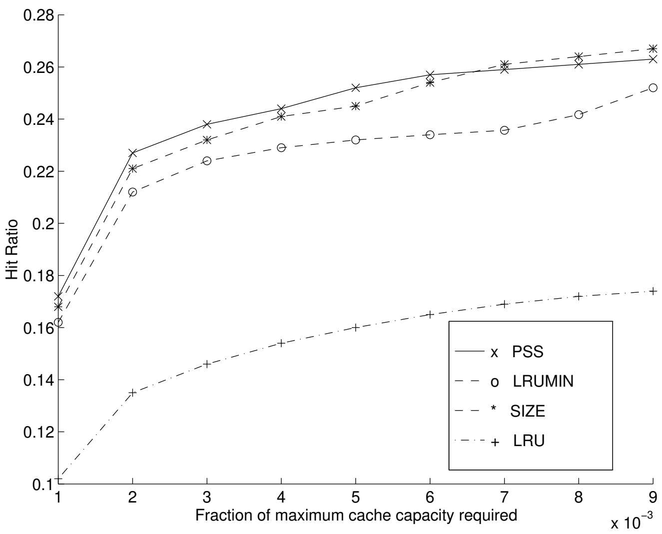


Figure 13: Proxy Trace Driven Simulation

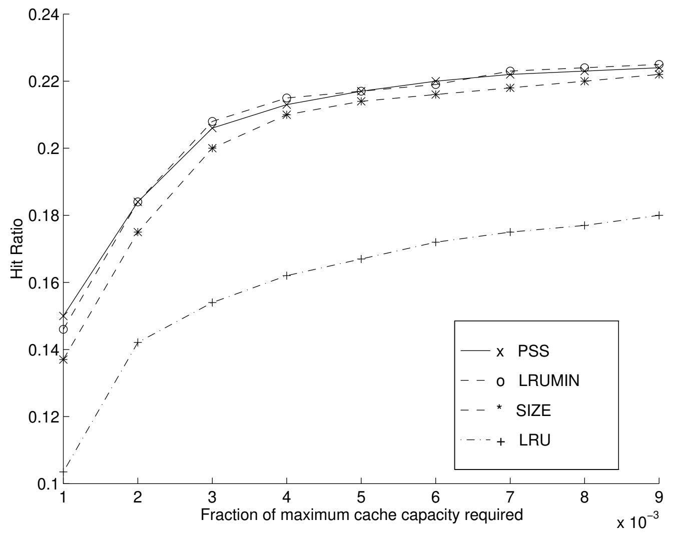


Figure 14: Proxy Trace Driven Simulation

pages seemed to have a relatively smaller size. We ran the simulation for varying values of the cache capacity.

The performance curves for the case of the server traces are illustrated in the Figures 11 and 12. Again, the PSS scheme performs significantly better than the LRU, LRU-MIN, and the SIZE schemes. In the case of the proxy traces, (Figures 13 and 14) the performance difference is less stark, though in both the cases, the PSS policy performs at least as well as the better of the two schemes (LRUMIN and SIZE). In the case of the proxy traces, the skew in the relative frequency is much less than that for the case of the server. As a result, size becomes the primary deciding factor in the hit ratio. This, in conjunction with the fact that objects of smaller size tended to have somewhat higher frequency accounted for the good performance of the PSS, LRU-MIN and the SIZE schemes.

# 6 Summary and Conclusion

In this paper we have discussed various schemes which handle the caching of web objects. Such schemes typically involve both a cache replacement and a cache admission policy. In addition to several existing policies, a new scheme called PSS has been introduced. This scheme has an additional advantage of being general purpose, and is not too sensitive to the characteristics of the workload.

We compared the hit ratios and robustness of the various algorithms through event and trace driven simulations. Based on these experimental results, we conclude that PSS is a practical and viable caching algorithm for web objects.

# References


[1] M. Abrams, C. Standridge, G. Abdulla, S. Williams and E. Fox. "Caching Proxies: Limitations and Potentials", *Fourth International World Wide Web Conference,* Boston, 1995.

[2] M. Abrams, S. Williams, G. Abdulla, S. Patel, R. Ribler, and E. A. Fox. "Multimedia Traffic Analysis Using CHITRA95." *Multimedia, 1995*, pages 267-276.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. "Network Flows: Theory, Algorithms, and Applications." *Prentice Hall, Englewood Cliffs, New Jersey.* 1993.

[4] M. F. Arlitt. "A performance study of internet web servers." Master's thesis, Computer Science Department, University of Saskatchewan, Saskatoon, Saskatchewan, May 1996.

[5] M. F. Arlitt and C. L. Williamson. "Web Server workload characterization: The search for invariants." In *Proceedings of SIGMETRICS*, Philadelphia, PA, April 1996.

[6] H. Braun and K. Claffy, "Web Traffic Characterization: An Assessment of the Impact of Caching Documents from NCSA's Web Server", *Second International World Wide Web Conference,* Chicago, 1994.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. "A hiearchial internet object cache." *Technical Report 95-611*, Computer Science Department, University of Southern California, Los Angeles, California.

[8] G. Copeland, W. Alexander, E. Boughter, and T. Keller. "Data Placement in Bubba." *Proceedings of the ACM SIGMOD, 1988,* pages 99-108.

[9] C. R. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of www client-based traces. Technical Report TR-95-010, Computer Science Department, Boston University, July 1995.

[10] P. B. Danzig, M. F. Schwarz, and R. S. Hall. "A case for caching file objects outside internetworks." *ACM SIGCOMM 1993 Conference*, pages 239-248, September 1993.

[11] S. Glassman, "A Caching Relay for the World Wide Web", *First International World Wide Web Conference,* Geneva, 1994.

[12] T. Johnson, and D. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm." *Proceedings of the 20th VLDB Conference.* Santiago, Chile, 1994.

[13] D. Knuth, "The Art of Computer Programming," Vol. 3, Addison Wesley, Reading MA, 1973.

[14] A. Luotonen and K. Altis, "World Wide Web Proxies", *First International World Wide Web Conference,* Geneva, 1994; also in *ISDN Systems 27, No. 2, 1994.*

[15] E. Markatos, "Main Memory Caching of Web Documents", *Computer Networks and ISDN Systems,* 28 (1996), pp. 893-905.

[16] R. Malpani, J. Lorch, and D. Berger, "Making World Wide Web caching servers cooperate." In *4th International World-wide Web Conference,* pages 107-117, Boston, December 1995.

[17] E. J. O'Neil, P. E. O'Neil, G. Weikum. "The LRU-K Page Replacement Algorithm For Database Disk Buffering". *Proceedings of the ACM SIGMOD, 1993.* Washington D. C., USA.

[18] V. N. Padmanabhan, and J. C. Mogul "Improving HTTP latency." *Computer Networks and ISDN Systems,* 28 (1 & 2): pages 25-35.

[19] J. Pitkow and M. Recker, "A Simple Yet Robust Caching Algorithm Based on Dynamic Access Patterns", *GVU Technical Report: VU-GIT-94-39*; also in *Second International World Wide Conference,* Chicago, 1994.

[20] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox "Removal Policies in Network Caches for World Wide Web Documents." *Proceedings of the ACM SIGCOMM, 1996.* pages 293-304.

[21] N. Smith "The UK national World-Wide Web proxy cache at HENSA Unix, 1995." http://www.hensa.ac.uk/wwwcache/intro.html