Caching on the World Wide Web

Charu Aggarwal, Joel Wolf and Philip Yu

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

Abstract

With the recent explosion in usage of the world wide web, the problem of caching web objects has gained considerable importance. Caching on the web differs from traditional caching in several ways. The non-homogeneity of the object sizes is probably the most important such difference. In this paper we give an overview of caching policies designed specifically for web objects and provide a new algorithm of our own. This new algorithm can be regarded as a generalization of the standard LRU algorithm. We examine the performance of this and other web caching algorithms via event and trace driven simulation.

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 objects 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 object 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 objects closer to the user.

Consequently, web caching has become an increasingly important topic [1, 8, 14, 18, 19, 22]. 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 behalf of the client in order to locate a cached copy of a object if possible. More information on caching proxies may be found in [17]. 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. Such 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 objects between invocations. Mosaic, for example, uses a non-persistent cache.

In this paper, we shall discuss general main memory cache replacement policies designed specifically for use by web caches. The results are applicable to web server, proxy and client caches.

One of the key complications in implementing cache replacement policies for web objects 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 object. This is because it is possible to store a larger number of objects of smaller size. In the standard *least recently used (LRU)* caching algorithm for equal sized objects 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 the one which was least recently used. Clearly the LRU policy needs to be extended to handle objects of varying sizes.

In addition to non-homogeneous object sizes, there are several other special features of the web which need to be considered. First, the hit ratio may not be the best possible measure for evaluating the quality of a web caching algorithm. For example, the transfer time cost for transfering a large object is more than that for a small object, though the relationship is typically not straightforward. It will depend, for instance, on the distance of the object from the web server. Furthermore, web objects will typically have expiration times. So, when considering which objects to replace when a new object enters a web cache, we must consider not only the relative frequency, but also factors such as object sizes, transfer time savings and expiration times.

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.

We list below some cache replacement policies evaluated in [1, 26]. Each of them can be combined with an admission policy.

(1) **LRU:** In the most straightforward extension of LRU for handling non-homogeneous sized objects, 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.

(2) **LRUMIN:** This policy is biased in favor of smaller sized objects so as to minimize the number of objects 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, we remove the least recently used such object from the cache. If there are no objects with size at least S, then we start removing 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.

(3) **SIZE policy:** In this policy, the objects are removed in order of size, with the largest object removed first. Ties based on size are somewhat rare, but when they occur they are broken by considering the time since last access. Specifically, objects 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, 26] that policies which take into account the size tend to perform better than those which do not. This is because removing larger objects makes room for multiple smaller ones.

In this paper we devise a web cache replacement policy which appears to achieve performance better than any of the above schemes. We describe a corresponding admission control policy as well. The scheme we propose is quite general purpose, is easy to implement, and works well on many different kinds of workloads.

We briefly survey and categorize some additional cache replacement schemes for the web. The list below is certainly not exhaustive, though many replacement algorithms can be classified into one or more of the following categories.

(1) **Direct extensions of traditional policies:** Besides LRU, traditional policies such as *Least Frequently Used (LFU)* and *First In First Out (FIFO)* are well-known cache replacement strategies for paging scenarios [24]. Just as with LRU, it is possible to extend these policies to handle objects of non-homogeneous size. The policy in [22] can be regarded as an LRU extension, though time since last access is rounded to the nearest day. The difficulty with such policies in general is that they fail to pay sufficient attention to object sizes.

(2) **Key-based policies:** The idea in key-based policies is to sort objects based upon a primary key, break ties based on a secondary key, break remaining ties based on a tertiary key, and

Name	Primary Key	Secondary Key	Tertiary Key
LRU	Time Since Last Access		
FIFO	Entry Time of Object in Cache		
LFU	Frequency of Access		
SIZE	Size	Time Since Last Access	
LOG2-SIZE	$\lfloor \log_2(Size) \rfloor$	Time Since Last Access	
HYPER-G	Frequency of Access	Time Since Last Access	Size

Table 1: Some examples of Key Based Policies

so on. This class of policies by proposed in [26]. For example, a policy called *LOG2-SIZE* discussed in [26] uses $\lfloor \log_2\{\text{Size}\} \rfloor$ as the primary key and the time since last access as the secondary key. The *HYPER-G* policy uses frequency of access as the primary key, breaks ties using the recency of last use, and then finally uses size as the tertiary key. Some additional examples of key-based policies are illustrated in Table 1. The idea in using the key based policies is to prioritize some replacement factors over others. However, such prioritization may not always be ideal.

(3) **Function-based replacement policies:** The idea in function-based replacement policies is to employ a potentially general function of the different factors such as time since last access, entry time of the object in the cache, transfer time cost, object expiration time and so on. For example, the algorithm described in [7] employs a weighted rational function of the transfer time cost, the size and the time since last access. The algorithm described in [27] employs a weighted exponential function of the access frequency, the size, the latency to the server and the bandwidth to the server. The *Least Normalized Cost Replacement (LNC-R)* algorithm described in [23] employs a rational function of the access frequency, the transfer time cost and the size. This algorithm is certainly the most similar to our own scheme, which is also function-based.

The paper is organized as follows. In Section 2 we concentrate on formulating a theoretical optimization model for web caching which generalizes LRU. We devise an optimization model and show how this can be approximately solved by a simple heuristic. We call the policy derived here *Size-adjusted LRU*, or *SLRU*. In Section 3 we show how to make this heuristic more easily implementable in practice. We call the resulting policy the *Pyramidal Selection Scheme*, or *PSS*. Section 4 describes web-specific extensions to the above two algorithms in order to handle general access costs and expiration times. In Section 5 we discuss an admission control policy. Results of event and trace driven simulations are presented in Section 6. In particular, we compare PSS with the LRU, LRUMIN and SIZE schemes. Finally in Section 7, we present a summary and conclusion.

2 Generalized LRU Replacement

When an object is to be inserted into the cache, more than one object may need to be removed in order to create sufficient space. In the LRU extension discussed in [1, 26], objects are greedily removed from the cache in the order of recency of last access until enough space is created for

the incoming object. But such a policy is not the only possible LRU generalization for handling objects of non-uniform size. In this section we describe the theoretical foundations of another such policy. Specifically, we define and heuristically solve an optimization problem which mimics but generalizes the LRU criteria for uniform sized objects.

First we shall need some notation. We assume that there are N objects, and that object i has size S_i. A counter is maintained and incremented each time there is a request for an object. The set of objects in the cache at the kth iteration is denoted by $C(k)$. Let i_k denote the object accessed at the kth iteration. If i_k is present in the cache in the $(k-1)$st iteration 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. Let $R \geq 0$ denote the amount of additional space in the cache which must be created in order to accommodate i_k, an easy calculation. Consider the decision variable y_i for object i defined to be 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 ΔT_{ik} is the number of accesses since the last time object i was accessed. This number is well-defined for all objects which have been accessed before. 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 smallest dynamic frequency from the cache, and thus tends to retain the objects with high frequency of access. While the dynamic frequency is an imperfect estimator of the true frequency of an object, LRU turns out to be a very robust algorithm in practice, at least for the case of uniform size objects.

Roughly speaking, for non-uniform sized objects we would like the sum of dynamic frequencies for the outgoing objects to be as small as possible. Specifically, we have the following model:

$$\begin{aligned} \text{Minimize} \quad & \sum_{i \in C(k)} y_i / \Delta T_{ik} \\ \text{such that} \quad & \sum_{i \in C(k)} S_i \cdot y_i \geq R \\ & \text{and } y_i \in \{0, 1\}. \end{aligned}$$

The above mathematical programming problem is a version of the *knapsack problem* [4]. (Said precisely, the objects we place in the knapsack are actually those which will be purged from the cache.) The knapsack problem is known to be NP-hard. However, there exist fast heuristics which 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 $1/(S_i \cdot \Delta T_{ik})$. So, we reindex the objects $1, 2, \ldots, |C(k)|$ in order of non-decreasing values of $S_i \cdot \Delta T_{ik}$. After sorting we have:

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

Then we greedily pick the highest index objects one by one and purge them from the cache until we

Figure 1: The Pyramidal Selection Scheme

have created sufficient space for the incoming object. We call this replacement scheme *Size-adjusted LRU*, or *SLRU*.

3 The Pyramidal Selection Scheme

We should note that the SLRU policy described in the previous section may be somewhat unrealistic to implement in practice, because of the difficulty in comparing the product of the size and the time since last access for every object in the cache. A somewhat more practical variant, known as the *Pyramidal Selection Scheme*, or *PSS*, will be described in this section.

The primary idea behind the PSS scheme is that we make a pyramidal classification of objects depending upon their size. All objects of group i will have sizes ranging between 2^{i-1} and $2^i - 1$. Thus there will be $N = \lceil \log(\mathrm{M}+1) \rceil$ different groups of objects, where M is the cache size. For the cache, as illustrated in Figure 1, the objects in each group i are maintained as a separate 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 Section 6.

Notice that the LOG2-SIZE discussed in [26] bears at least some resemblance to the independently derived PSS scheme: The LOG2-SIZE scheme always chooses the least recently used items in the non-empty stacks corresponding to the largest size ranges. In contrast, the PSS scheme looks at the least recently used objects of each stack, and among these picks the objects which have the least product of the $S_i \cdot \Delta T_{ik}$.

4 Web-Specific Extensions

4.1 Handling General Access Costs

The scheme of the Section 2 attempts to maximize the probability of a cache hit. Although the hit probability is certainly a reasonable measure to maximize via a cache replacement strategy, it could also be argued that not all objects on the web have the same access costs. For example, the transfer time costs for larger objects are higher, though this relationship is somewhat noisy and far from linear. Similarly, the access cost of an object requested from a distant web server is likely to be more than that of one requested from a nearby server. In this subsection we notice that the above replacement scheme may easily be extended to handle non-uniform access costs, assuming of course that such costs are known.

Let c_i be the cost of accessing object i. Then the generalized objective function can be written as $\sum_{i \in C(k)} c_i \cdot y_i / \Delta T_{ik}$. Similarly, the generalized size adjusted LRU rule would place objects in non-decreasing order of $(S_i \cdot \Delta T_{ik})/c_i$, and greedily purge those objects with the highest indexes. Note that if all values of c_i are uniform (1, for example), then the replacement policy reverts back to our original one.

4.2 Handling Object Expiration Times

Objects on the web are often assigned expiration times, and our replacement algorithm should be able to factor these in effectively. It is tempting here to employ the *time to live* for an object, namely the difference between the expiration time of the object and the current time. For example, if an object has only a short time to live, or perhaps is already stale, it would seem to be a good candidate for replacement by the incoming object. Unfortunately, this approach can lead to a very unstable cache and also appears impractical to implement because of high maintenance costs. We shall use more static data instead, data which is computed once for each object at the time it enters the cache. In particular, suppose an object i enters the cache at time t. Let us define δt_{i1} to be the difference between t and the time when it was *last* accessed. Let δt_{i2} be the difference between

the object expiration time and t. (Note that we are using δ to refer to time differences; we had previously used Δ to refer to differences in terms the number of accesses.) We define the *refresh overhead factor* for an incoming object i to be $r_i = \min\{1, \delta t_{i1}/\delta t_{i2}\}$. This value is approximately the reciprocal of the number of expected accesses before the object needs to be *refreshed*. We incorporate the refresh overhead factor into the replacement policy by ordering objects in terms of non-decreasing values of $(S_i \cdot \Delta T_{ik})/(c_i \cdot (1 - r_i))$, and greedily purging those objects with the highest indexes.

If object i has no expiration date then δt_{i2} is infinite, so that the refresh overhead factor $r_i = 0$. Thus the replacement policy reverts back to our original one. On the other hand, if object i has not been accessed before, then we can assume that δt_{i1} is infinite, so that the refresh overhead factor $r_i = 1$. (In fact, the statement $r_i = 1$ holds whenever $\delta t_{i1} \geq \delta t_{i2}$.) This would result in a quick purge from the cache. However, the admission policy described in the next section will generally not allow such an object to enter the cache in the first place.) The refresh overhead factor will have only a marginal effect on the replacement policy if objects are accessed much more frequently than they expire. Conversely, when an object has an expiration rate nearly as big as the access rate, the refresh overhead factor will have a significant effect.

Similarly, the more implementable PSS scheme can be adapted to the case when there are access costs and/or expiration times. Instead of grouping together objects based upon the value of their sizes, we group together objects based upon geometrically increasing ranges for the value of $S_i/(c_i \cdot (1 - r_i))$.

5 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. For each object in this auxiliary cache we also maintain timestamps of the last access, measured both in terms of the number of object accesses and time, together with access cost and expiration time data. The access counter is incremented each time an object is requested from the cache, whether or not that request can be filled. Because the auxiliary cache contains identities of objects rather than the objects themselves, its size is negligible compared to that of the main cache. (As a rule of thumb, we set X to be about twice the average number of objects in the main cache.) The auxiliary cache is maintained in strict LRU order. Figure 2 illustrates the auxiliary and main caches.

We would like to have an admission control policy which ensures that at the kth iteration the potential 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

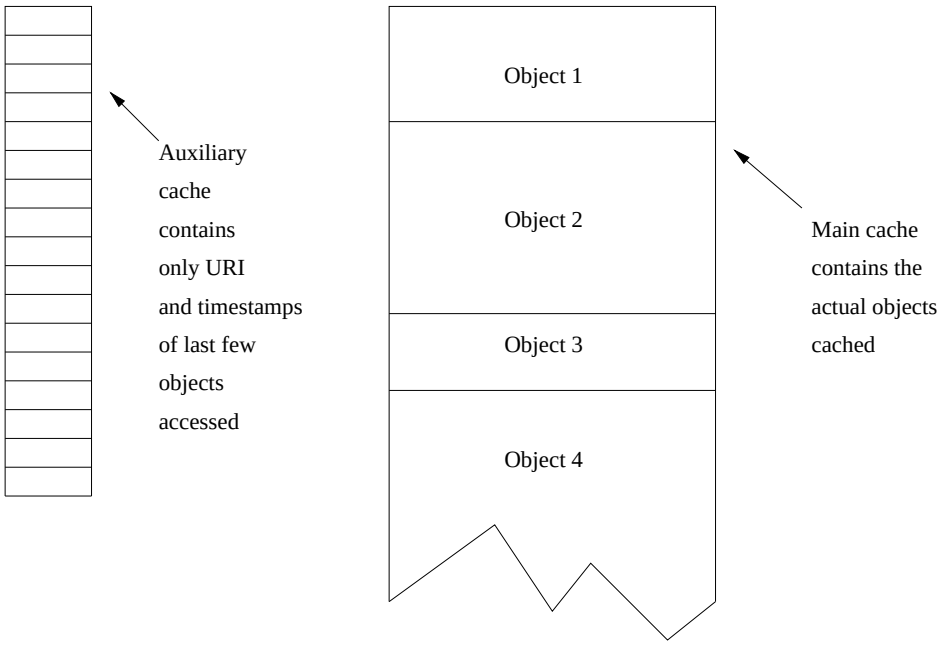


Figure 2: Performing Admission Control

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 determine if the decision which the replacement policy heuristic makes would be profitable. That is, we compare the value $(c_{i_k} \cdot (1 - r_{i_k}))/\Delta T_{i_k k}$ with the sum $\sum_i (c_i \cdot (1 - r_i))/\Delta T_{ik}$ of the set of candidate outgoing objects determined using the replacement scheme. We admit an object only if it is profitable to do so. Observe that the information needed can be obtained from the auxiliary cache. After this iteration, the time stamp of the object i_k is updated.

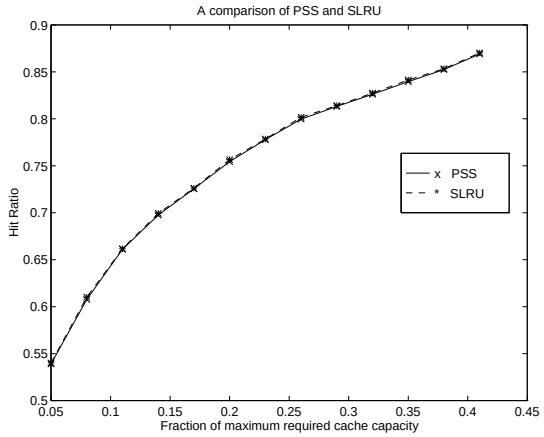
The idea of admission control bears some resemblance in spirit to the 2Q approximation of LRU-K proposed in [15, 20] for making cache more robust to workload transients. However, the method of doing so is different in this case.

6 Empirical Results

In this section we experimentally compare the performance the performance of four different caching schemes: the naive LRU policy extension, the LRUMIN policy of [1], the SIZE policy discussed in [26] and the PSS policy proposed in this paper. We employ both event and trace driven simulations.

Note that the PSS scheme was obtained as a more easily implementable variant of the SLRU algorithm described in Section 2. Both of these schemes were implemented in conjunction with the admission control of Section 5. We compare SLRU to PSS in order to show that the two schemes are virtually identical in terms of performance.

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.



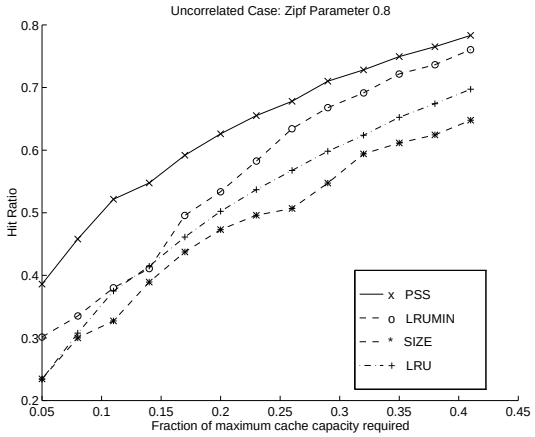


Figure 3: Figure 3: PSS and SLRU.....Figure 4: Hit Ratio, No Correlation

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

(2) Cost savings, as noted in Section 4, can also be important.

(3) A final measure is the *robustness* of the various schemes. It is well known that the performance of caching policies for web objects often depends upon whether smaller objects have higher frequency or vice versa. The LRU scheme is very robust for uniform size objects and varying distributions of relative frequencies. All the schemes that we compare are in fact generalizations of LRU in one fashion or another, and consequently it is useful to see how the correlation of size and frequency factors into the robustness of the proposed schemes.

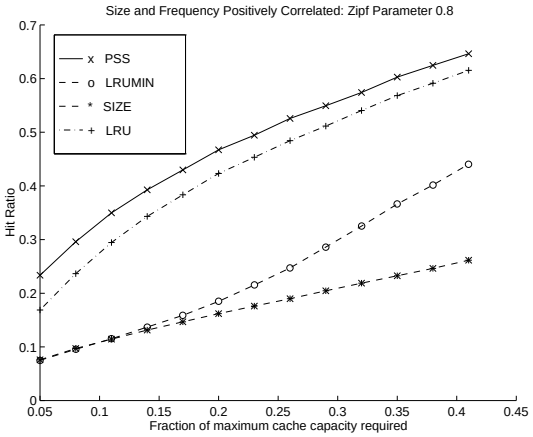
6.1 Event Driven Simulation

The primary motivation for performing event driven simulation is to understand the effect of the varying parameters on the performance of the schemes. 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 [16]. Thus the frequency of object i is proportional to $1/\pi(i)^\theta$, where θ is the Zipf parameter, and π is a permutation vector. By varying π 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 case where object size and frequency are uncorrelated. That is, we choose π randomly. In this case, we chose a Zipf parameter of 1, which yields a fully Zipf distribution. 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 next tested three different relationships between the frequency of an object and its size. For each of these cases, we considered a Zipf parameter equal to 0.8, which is somewhat less skewed.

(1) There is no correlation between the object size and the frequency. In this case, Figure 4



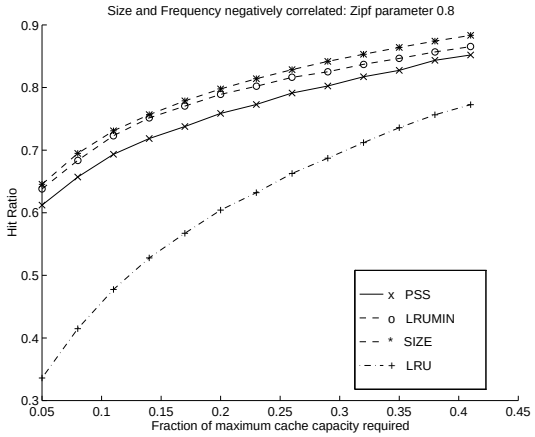


Figure 4: Figure 5: Hit Ratio, Positive Correlation.....Figure 6: Hit Ratio, Negative Correlation

shows that the LRUMIN scheme outperforms LRU most of the time, though for relatively small cache sizes (and also for very skewed frequency distributions) the situation is reversed. The SIZE policy is outperformed by both the LRU and 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 each of the other schemes here.

(2) There is complete positive correlation between object size and frequency. Thus the value of $\pi(i)$ is chosen to be $501 - i$, so that the ith object has size i and relative frequency $1/(501 - i)^{\theta}$. So the largest object has the highest frequency, and the smallest object has the lowest frequency. The corresponding hit ratio curve is shown in Figure 5. As we can see, in this case the LRUMIN policy and the SIZE policy perform very poorly since they are biased too strongly towards objects of smaller size. The SIZE and LRUMIN schemes result in more frequently accessed objects being displaced for the sake of less frequently accessed objects. 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 LRUMIN 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 ith object has size i and relative frequency $1/i^{\theta}$. Thus, the largest object has the lowest frequency and the smallest object has the highest frequency. Figure 6 shows the corresponding hit ratio curves. This is the most favorable case for the LRUMIN policy: The small objects have high frequency, and the LRUMIN 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 objects may be at varying distances from

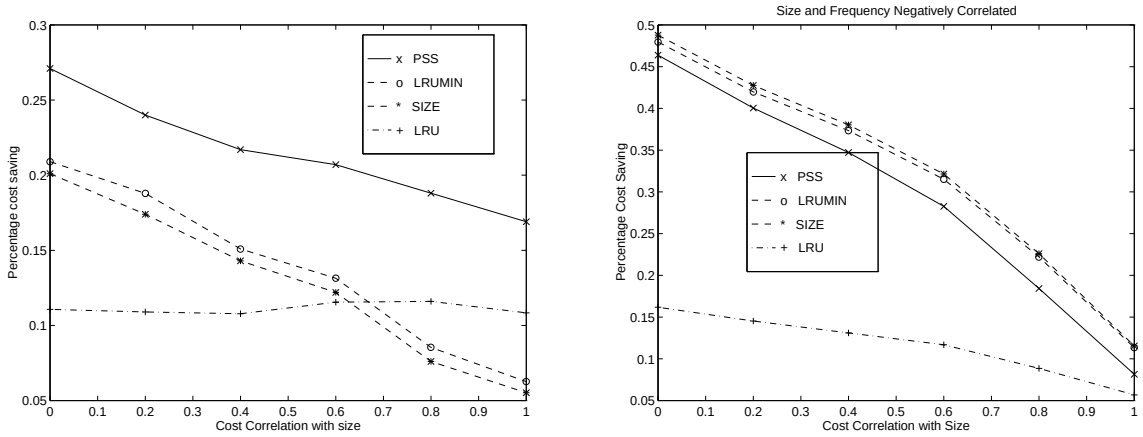


Figure 5: Figure 7: Cost, No Correlation.....Figure 8: Cost, Negative Correlation

the cache. In order to test the effects of different relationships between the size of an object and its access cost we used the following weighting scheme.

$$\text{Cost(Object)} = K_1 \cdot [\text{Size Component}] + K_2 \cdot [\text{Noise Component}].$$

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 an object. In other words, $A = \sum_{i=1}^{500} S_i \cdot f_i$, where f_i is the relative frequency of object i. Then, the cost C_i of accessing an object i is as follows:

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

Here 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 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, and vice-versa.

We tested the four 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 three different cases, depending upon whether size and frequency were uncorrelated, negatively correlated or positively correlated. These cases are plotted in Figures 7, 8 and 9, respectively. On the X-axis we show the value of the parameter q, while the Y-axis shows the percentage cost savings by caching. The following observations can be 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 the greatest effect on the performance of the LRUMIN and SIZE policy and the least effect on the LRU scheme. For example, in the

case of 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 either smaller or larger objects, and a differing structure of access cost does not affect the performance of the policy. All the other schemes (PSS, LRUMIN 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 LRUMIN and the SIZE schemes are most aggressive in discriminating against larger objects, this effect is felt most strongly in these. In the uncorrelated case for example, the LRU policy is actually better than the LRUMIN 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 with or better than the best of the other three schemes (LRUMIN, SIZE, and LRU). Even in the negatively correlated case, the PSS scheme is only marginally worse than the LRUMIN 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 of the scheme corresponding to the negatively correlated case to be much higher than the hit ratio for the positively 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. We define the robustness of a policy A as follows:

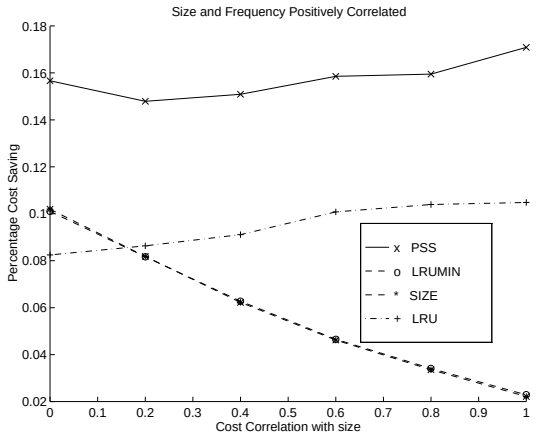
$$\text{Robustness(Policy A)} = \frac{\text{Hit Ratio(Policy A, Negatively Correlated Case)}}{\text{Hit Ratio(Policy A, Positively Correlated Case)}}. \quad (1)$$

So, 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 LRUMIN 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. 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.

6.2 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: The performance is good for both kinds of workloads. For each of the logs, we had two traces whose lengths were between 100000 to 150000 user accesses each. Most of the frequently accessed pages had relatively smaller sizes. We ran the simulation for varying values of the cache capacity.

The performance curves for the case of the two server traces are illustrated in Figures 11 and 12. Again, the PSS scheme performs significantly better than the LRU, LRUMIN and SIZE schemes.



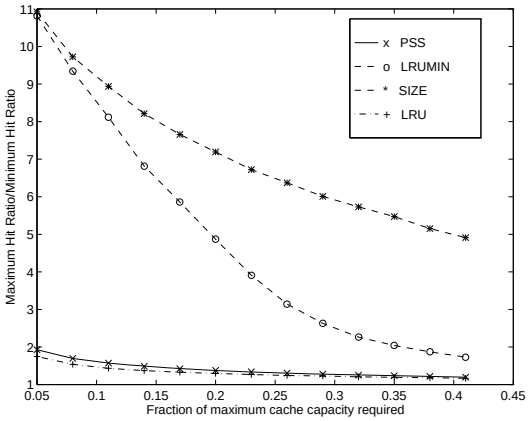
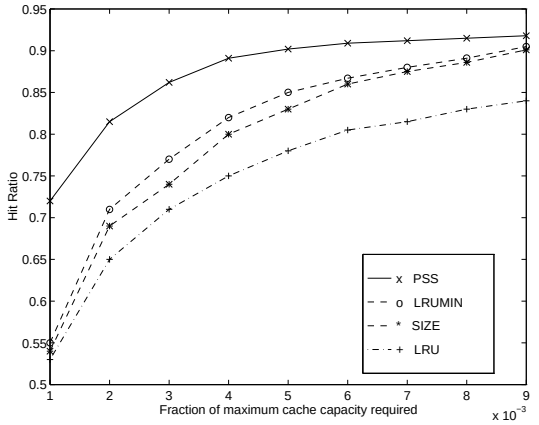


Figure 6: Figure 9: Cost, Positive Correlation.....Figure 10: Robustness.....



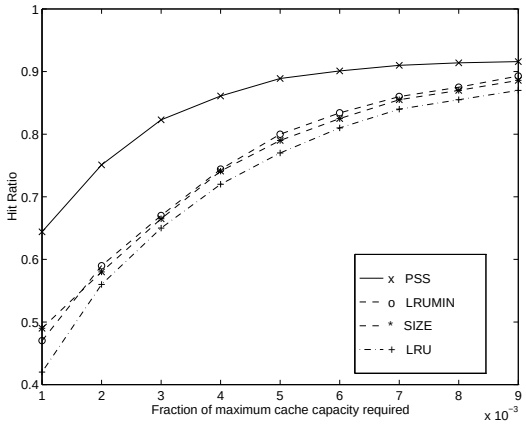
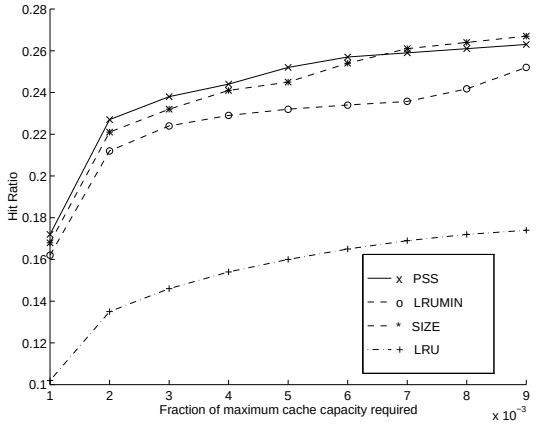


Figure 7: Figure 11: Server Cache Case 1.....Figure 12: Server Cache Case 2



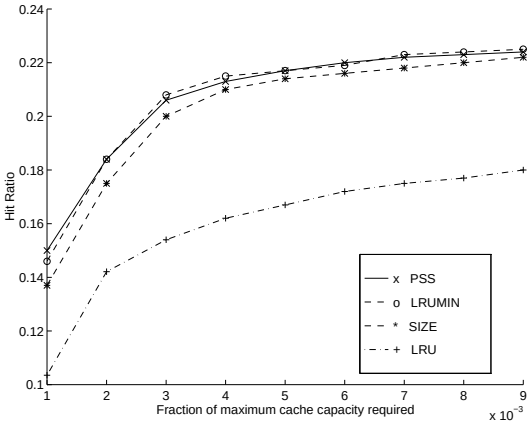


Figure 8: Figure 13: Proxy Cache Case 1.....Figure 14: Proxy Cache Case 2

In the case of the two proxy traces (Figures 13 and 14), the performance difference is less stark, though in both 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 determining 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, LRUMIN and the SIZE schemes.

7 Summary and Conclusion

In this paper we have discussed various schemes for handling the caching of web objects. Such schemes typically involve both a cache replacement and a cache admission policy. The non-uniform size of web objects causes standard LRU to perform less well than one would normally anticipate. We devised a new scheme called PSS specifically oriented towards web caching. The PSS scheme is the more implementable variant of its theoretical ancestor SLRU, also developed in this paper.

We compared the hit ratios and robustness of PSS and other web replacement policy algorithms, using both event and trace driven simulations. Based on these experimental results, we conclude that PSS is a practical and viable caching algorithm. PSS has good hit ratio performance, and is also robust to varying workload characteristics.

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, MA, 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] C. C. Aggarwal, J. L. Wolf, P. S. Yu, and M. Epelman. "On Caching Policies for Web Objects." *IBM Research Report*, 1997.

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

[5] M. F. Arlitt. "A Performance Study of Internet Web Servers." Master's thesis, Computer Science Department, University of Saskatchewan, Saskatoon, Saskatchewan, May 1996.

[6] M. F. Arlitt and C. L. Williamson. "Web Server Workload Characterization: The Search for Invariants." In *Proceedings of SIGMETRICS*, Philadelphia, PA, April 1996.

[7] J-C Bolot, and P. Hoschka. "Performance Engineering of the World Wide Web." *WWW Journal 1,* (3), 185-195, Summer 1996.

[8] 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, IL, 1994.

[9] S. J. Caughey, D. B. Ingham, and M. C. Little. "Flexible Open Caching for the Web", *Proceedings of the Sixth International World Wide Web Conference.*

[10] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. "A Hierarchical Internet Object Cache." *Technical Report 95-611*, Computer Science Department, University of Southern California, Los Angeles, CA.

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

[12] 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.

[13] 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.

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

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

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

[17] 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.*

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

[19] 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, MA, 1995.

[20] 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,* Washington DC, 1993.

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

[22] 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, IL, 1994.

[23] P. Scheuermann, J. Shim, and R. Vingralek. "A Case for Delay-Conscious Caching of Web Documents." *Proceedings of the Sixth International World Wide Web Conference.*

[24] A. Silberschatz, and P. B. Galvin. *Operating Systems Concepts.* Addison Wesley, Reading, MA, fourth edition, 1994.

[25] N. Smith "The UK National World Wide Web Proxy Cache at HENSA Unix, 1995," http://www.hensa.ac.uk/wwwcache/intro.html

[26] 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.

[27] R. P. Wooster, and M. Abrams, "Proxy Caching that Estimates Page Load Delays," *Proceedings of the Sixth International World Wide Web Conference.*