RC 20636 (11/19/96)
Computer Science/Mathematics

IBM Research Report

On Disk Caching of web objects in proxy servers

Charu C. Aggarwal, Philip S. Yu

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

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

On Disk Caching of Web objects in Proxy Servers

Charu C. Aggarwal and Philip S. Yu
IBM T J Watson Research Center
Yorktown Heights, NY 10598
Email:charu@watson.ibm.com, psyu@watson.ibm.com

Abstract

With the increasing popularity of the world wide web, it may be desirable to store copies of popular documents in proxy caches and thus diminish the delay times for URL requests. Web documents have to be treated as indivisible objects for the purpose of caching. In this paper we study the problem of caching web documents on disks. In web applications, the objects are of non-homogeneous size, and this leads to a problem in physical placement in any scheme which tries to emulate LRU. One solution is to use the FIFO scheme which simplifies the disk placement issue, because the order of physical placement can be completely dictated by the order of entry of an object into the cache. In this paper, we show how to achieve a generalization of the LRU policy for the case of caching web documents in such a way that the disk placement and fragmentation issue is handled efficiently.

1 Introduction

The recent success of the world wide web has lead to a quick growth in traffic volume over the internet. This has often lead to congestions and bottlenecks in performance in more popular areas in the network. Such congestions can often lead to dramatic increases in user response times. In order to reduce access latencies, it is desirable to store copies of popular documents closer to the user, from which the access latencies are much lower. For the purpose of this paper, we shall use the term "object" and "document" interchangeably.

Caching can be implemented at various places on the network. Here our focus is on proxy server. Proxy caching is critical to Internet service providers. This is especially the case for high speed Internet access providers like cable companies which can provide huge bandwidth to user premises. Since the bandwidth to the web sites remains limited, the high bandwidth advantage can only be realized through effective proxy server caching. Similarly for corporate Intranet, proxy caching can be used to reduce the user delay to the external networks. More information on proxy servers and proxy caching may be found in [1, 4, 9, 10].

In this paper, we shall discuss the case when the web document caching is done on disks. In this case, an additional problem is created by the issue of the physical placement of the documents on the disks. In [2], we discuss a general cache replacement policy when caching is done in main memory; in that case physical placement is not the issue. There are several difficulties created by the non-uniformity of document sizes when placing them on disk. Consider, for example, a situation when the object coming into the cache is large, and several small documents may need to be removed in order to create space for it. These documents need to be contiguously located on the disk for such an interchange to occur. It may happen that objects which are adjacently placed have widely varying times of last access, making it harder to decide which *contiguous group* to choose in order to perform the replacement. Consider, for example, the case illustrated in Figure 1, where there are 5 objects A, B, C, D, E in the cache with "time-stamps" of last access as illustrated. Note that A and C are the two least recently used objects but since they are not contiguous in terms of physical placement, they may not be replaced for an entering document of size 9 units. At this stage, we would like to also make a comment on the use of the term "time-stamp" throughout this paper. A discrete global counter which is incremented by one each time an object is requested from the cache. The time-stamp of an object is defined as the value of this counter at the time of its last access.

Some work has been done for the case of main-memory caching of web documents. References to the work may be found in [1, 2, 6, 7, 8]. Both caching and fragmentation are

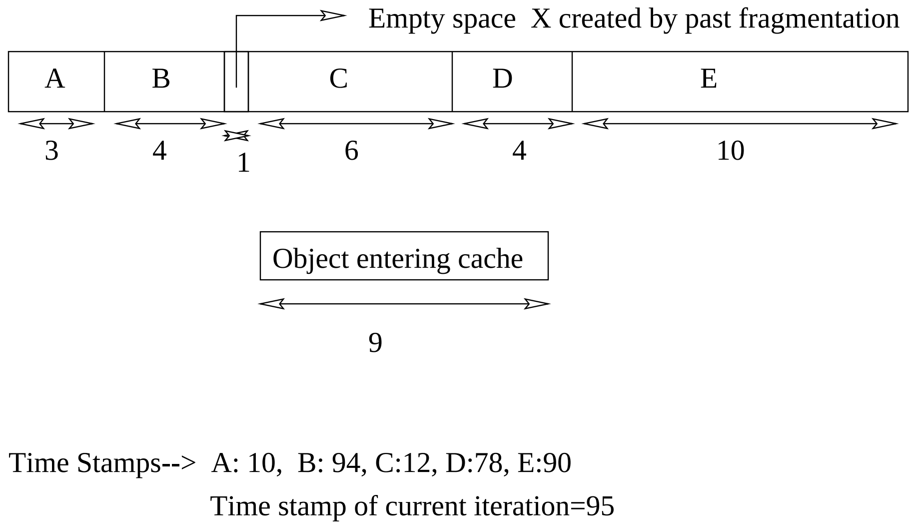

Figure 1: An example of a cache configuration

classical problems, which have been well studied; however, to the best of our knowledge this is the first paper which develops a policy for caching web objects on disks, while taking into account disk placement and configuration. One way of simplifying the complexities associated with the issue of placement and fragmentation altogether is by using a simple FIFO (First in First out) scheme. In this case, the order of placement of the objects in the disk cache is completely dictated by their order of entry into the cache; consequently the set of outgoing objects is always contiguously located (modulo edge effects).

This paper is organized as follows. In the next section, we discuss a replacement policy which may be considered as a generalization of the LRU rule to the case of caching web documents on disks. However, as we shall see, the time required to implement this selection rule is somewhat excessive, and in section 3, we shall show how to speed up this rule without significantly affecting performance. We also provide some theoretical results to show that the heuristic rule presented in section 3 achieves a performance close to that presented in section 2. In section 4, we shall discuss the issue of admission control and how it may be used in order to improve the hit ratio. Empirical results are provided in section 5, while section 6 provides a brief conclusion and summary.

2 The replacement policy

This section deals with the important question of choosing a set of adjacent documents which are replaced by an incoming document. As we shall see, the algorithm for performing the replacement requires a scan which is linear in terms of the number of documents cached; however in a later section, we shall show how to speed up this policy so as to make it more implementable. Consider, for example, the situation which is illustrated in Figure 1. In the Figure, we show a situation where we have a set of objects (currently in cache) $A, B, \ldots E$, and an "empty space" X. The sizes of the documents have also been illustrated in the Figure. This is one of the cases in which the incoming objects is so large that it would almost always displace more than one object from the cache. Consequently, a group of adjacent objects have to be chosen as the outgoing objects in order to make room for it. Unfortunately, the time-stamps of the adjacent objects are so disparate, that it is not immediately obvious how to choose a group of adjacent objects, so that anything similar to the LRU-criterion may be implemented. In other words, what we really need to do is to define a single objective function for a group of objects (based upon their time-stamps) which defines the desirability of their being in the cache. In order to do so, we shall define some further notations and terminology.

Let t_i be the time-stamp of the objects i and let t denote the current value of the global time counter. Then the dynamic frequency of the object i is defined by $1/(t - t_i)$. We denote the corresponding dynamic frequency by the value $DF(i, t)$.

Recall that the simple LRU policy for the case of uniform size objects simply replaces the page with the least value of the dynamic frequency. The use of the term dynamic frequency is intuitive, though somewhat misleading, because $1/(t - t_i)$ is at best a very noisy estimate of the true relative frequency of the object i.

The dynamic frequency of a set $S = \{1, \ldots, k\}$ of adjacent objects is the sum of the individual objects in the set S. In other words, if the time-stamps of the objects are $t_1, t_2 \ldots t_k$, then the dynamic frequency of the set S at time t is $\sum_{i \in S} DF(i, t) = \sum_{i=1}^{k} 1/(t - t_i)$. An important point to be noted here is that the empty space fragments which are trapped between the adjacent objects of the set S do not contribute to the dynamic frequency.

Once we have define the above objective function as an indicator for the desirability of a set of objects to be replaced from the cache, it becomes relatively easy to define the cache replacement policy. The following is the rule which may be used in order to decide which objects to eject from the cache.

Replace the set of adjacent objects with the least dynamic frequency so as to create enough space for the newly accessed object.

We shall call this scheme the Least Dynamic Frequency Rule, and it is really a direct generalization of the LRU scheme to the case of web objects. We assume that the newly accessed

Case	Outgoing documents	Dynamic Frequency
1	A, B, C	[1/(95-10)]+[1/(95-94)]+[1/(95-12)]= 1.023
2	B, C	[1/(95-94)] + [1/(95-12)] = 1.012
3	C, D	[1/(95-12)]+[1/(95-78)]= 0.071
4	D, E	[1/(95-78)] + [1/(95-90)]=0.259
5	E	[1/(95-90)] = 0.2

Table 1: Various possibilities for displacement

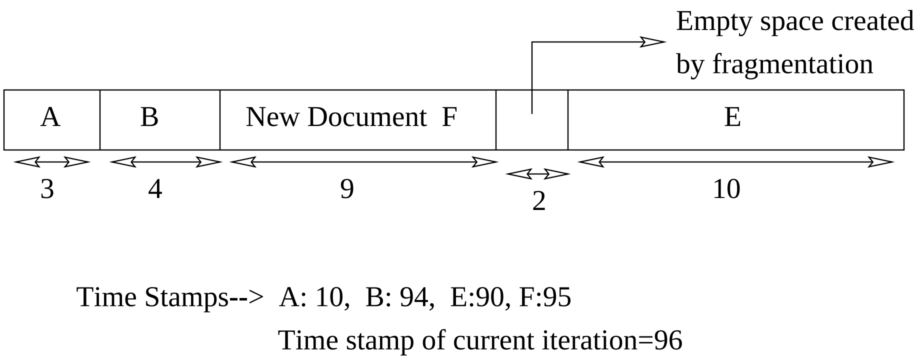

Figure 2: Configuration after interchange

object is always left adjusted in the space created by removing the outgoing objects. Note that this kind of scheme does create fragments of empty space unlike the FIFO scheme. Thus, if it is the case that one of these fragments of empty space is large enough to accommodate an object then it is certainly desirable to make use of it instead of performing a replacement. We shall refer to such an entry of an object into the cache as a *free entry*, since it does not come at the expense of other objects in the cache. For this purpose, we maintain a list of the sizes of the available empty space fragments[1] in sorted order. A quick binary search may then be used in order to determine the best fit. Otherwise, if a large enough space fragment is not available, the Least Dynamic Frequency rule is applied in order to handle cache replacement. For the example illustrated in Figure 1, if the size of the document entering the cache is one unit, then it results in a free entry, since it could be fitted into the position corresponding to the the empty space fragment X.

[1]In reality we would maintain only a list of the empty space fragments whose sizes are larger than some minimum threshold. Very small space fragments are not likely to help in free entries, and are consequently irrelevant.

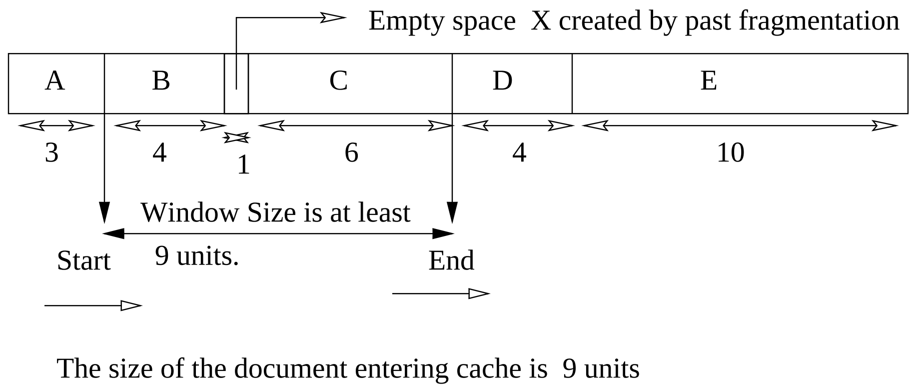

Figure 3: An example of a sliding window

Consider the example described in the Figure 1, where the document entering the cache has size 9 units. The various possibilities for the outgoing documents with corresponding dynamic frequencies are as illustrated in Table 1. As we see, the lowest dynamic frequency is for the Case 3, where the dynamic frequency of the outgoing documents is 0.071. Consequently, the documents C and D may be purged from the cache, and together with the empty space fragment X, they create a space fragment of 11 units. The incoming document is left adjusted in this space created. Since the incoming space fragment requires only 9 units of the space created this creates a new space fragment Y of 2 units of space. The resulting situation is illustrated in Figure 2.

2.1 A linear time implementation

The above algorithm can easily be implemented in such a way that it takes linear time to find the contiguous set with the least frequency of being replaced. We assume that a set of cache meta-data is maintained in main memory. This meta-data comprises the sizes and the time-stamps of the objects on the disk cache. We may assume that a linked list containing object sizes and time-stamps is maintained.

The primary idea is to use a sliding window mechanism, where the size of the window is determined by the size of the incoming document. It is here that the cache meta-data is used in order to perform the search. This is done by maintaining two pointers *Start* and *End*. The pointer *Start* points to the first object in the window, while the pointer *End* points to the last object in the window. An example is illustrated in Figure 3. The pointers are maintained in such a way, that the objects which are physically located between the two

```
Algorithm Find-Group(Required, Time);
{ Required denotes the amount of space required by incoming object }
{ Time denotes the current value of the time counter }
{ We assume that the objects are indexed 1, ... k }
{ Size[i] is the size of object i and Free[i] is the
size of the empty space fragment between objects i and (i + 1) }
begin
  { Start and End denote the ends of the sliding window; }
  Start = 1; End = 1;
  WindowSize = Free[0] + Size[1] + Free[1];
  CurrentMinimum = ∞;
  { CurrentMinimum is the dynamic frequency of the optimal window }
  while (Start ≤ k) do
    begin
    while (WindowSize < Required) and (End ≤ k) do
      begin
      { Slide the end of the window ahead }
      End = End + 1;
      WindowSize = WindowSize + Size[End] + Free[End];
      DynamicFrequency = DynamicFrequency + 1/(Time − lastaccess[End]);
      end;
    if ((DynamicFrequency < CurrentMinimum) and (WindowSize ≥ Required)) save
      the window between Start and End and reset CurrentMinimum = DynamicFrequency;
    { Slide the beginning of the window ahead }
    Start = Start + 1;
    WindowSize = WindowSize − Size[Start] − Free[Start − 1];
    DynamicFrequency = DynamicFrequency − 1/(Time − lastaccess[Start]);
    end;
end;
```

Figure 4: A linear time implementation

pointers is a minimal set of contiguous objects which has sufficient size to accommodate the entering object. The pointers slide, more or less synchronously from one end to another, always maintaining the minimal window required to accommodate the required document, while always keeping the two ends of the window at the boundaries of documents already in the cache. (Thus, in order to respect the fact that the window always contains an integral number of objects, the size of the window may be larger than the size of the incoming document.) A linear time implementation of finding the group with the least dynamic frequency is illustrated in Figure 4. The algorithm uses two parameters as the input: *Required* and *Time*. The parameter *Required* denotes the amount of space required in order to make space for a newly accessed object. In other words, *Required* is the size of the incoming object in the cache. The parameter *Time* denotes the current value of the global time counter.

At this stage, it might seem that a policy which requires linear time is somewhat excessive in terms of time-complexity; in the next section, we shall explore a scheme which speeds

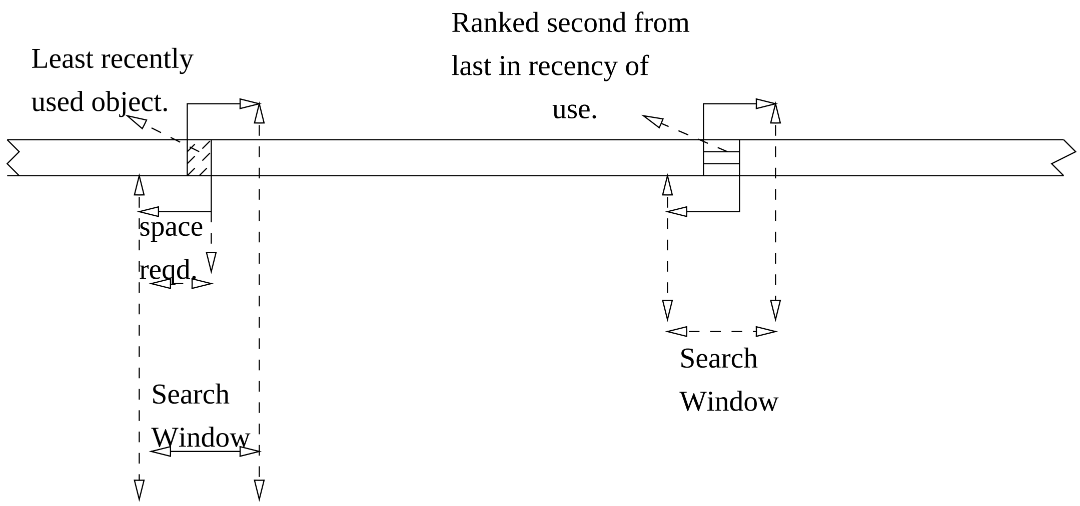

Figure 5: Localizing the search

this up by localizing the search significantly.

3 A faster implementation of the replacement rule

The primary drawback of the replacement rule discussed in the previous section is that it is time-intensive to implement a policy which requires us to scan the time stamps for each and every document in the cache. In this section, we shall discuss a remedy for this problem and discuss some heuristics which try to approximate the algorithm of the previous section in terms of performance. We shall also provide some theoretical results to give evidence that the performance of this algorithm ought to be somewhat close to the least dynamic frequency rule of the previous section.

A key observation is that when a replacement occurs, typically a small set of objects need to be displaced from the algorithm. In fact, in real simulations of the least dynamic frequency rule, it is usually the case that an incoming object does not displace more than two or three objects from the cache. This means that there is a good chance that one of this small set of two or three objects is among the least recently used objects in the cache. As a result, if we consider only the contiguous groups of adjacent objects which contain one of the r least recently used objects in the cache, then we shall see that even for very small values of r (one or two) the performance of this method closely approximates that of the least dynamic frequency rule. If the incoming document has size I, then we need to consider only at the documents which are within I units of each of the r least recently used objects. Typically, a small constant number of objects would need to be considered within the locality of each of these r points. Consequently, by localizing the search around a few of the least recently used objects, it is possible to achieve a performance which closely approximates that of the global search algorithm discussed in the previous section. We call this method the *Localized Least Dynamic Frequency* algorithm or LLDF. For the case when r least recently used objects are used, we denote this algorithm as LLDF-r. When an object which is entering the cache requires W units of space then we need to perform the search only on the objects which are within W units of physical distance from the object. An example of the search regions for the LLDF-2 algorithm is illustrated in Figure 5.

As we shall now see, the localized least dynamic frequency rule has some interesting theoretical properties. We shall show that if the degree of fragmentation is small, then the LLDF rule achieves an optimal replacement very often. Note that each change in state of the cache is either due to a replacement or a free entry. In order to do so, we make use of the fact that if the degree of fragmentation is small, the fraction of free accesses is quite small as well.

Theorem 3.1 *Let f be the steady state fraction of free entries out of the set of replacements or free entries into the cache. Then the steady-state probability that the set of outgoing objects chosen by the LLDF-r rule in the case of a replacement is as good as that chosen by the LDF rule is at least (1-2f)/(1-f).*

Proof: Note that if the set of outgoing objects has cardinality one, then the LLDF-r rule will choose the same object as the LDF rule. Thus, if we prove that the fraction of iterations in which the number of outgoing objects is one is at least $(1-2f)/(1-f)$ then we are done. Let q be the fraction of accesses in which an incoming object displaces two or more objects from the cache.

Let us consider the cache over a period of N replacements/free accesses from the cache, where $N \to \infty$. Let Q of these be replacements which displace 2 or more objects from the cache, let R be the number of replacements which displace one object from the cache, and let F be the number of free entries. Then the probability that a replacement displaces one object from the cache is $R/R+Q$. Noting that $R+F+Q = N$, it follows that $R/R+Q = (N-F-Q)/(N-F)$. Realizing that $\lim_{N\to\infty} F/N = f$ and $\lim_{N\to\infty} Q/N = q$ it follows that $\lim_{N\to\infty}(N-F-Q)/(N-F) = (1-f-q)/(1-f)$. It now remains to prove that $f \geq q$.

Then, since each fo the F free entries increases the number of objects in the cache by exactly and each of the Q replacements which displace more than one objects from the cache decreases the number by at least one, it follows that the total decrease in the number of objects in the cache over a series of N state changes is at least $Q-F$. But, this decrease

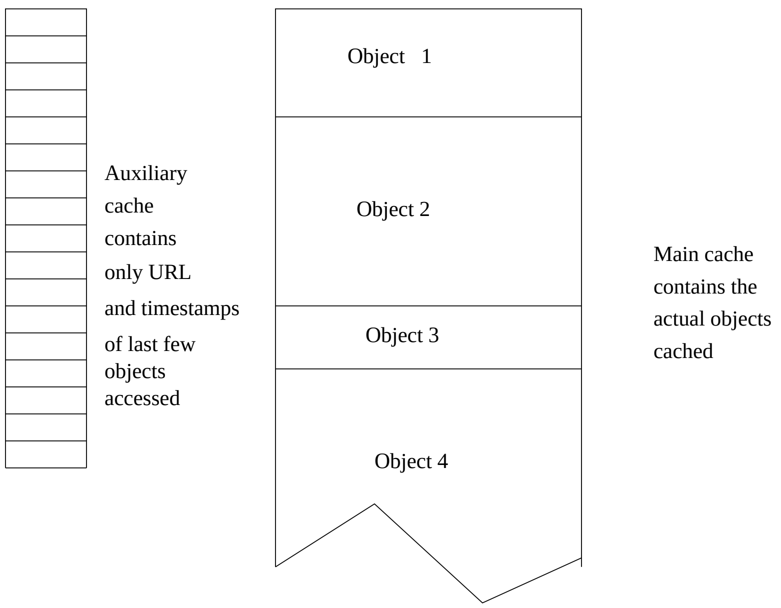

Figure 6: Admission Control

must be bounded above for all finite cache capacities, and consequently, we must have: $\lim_{N \to \infty}(Q - F)/N \leq 0$. It follows immediately that $f \geq q$. ∎

4 Admission Control

An *admission control policy* decides whether or not it is worthwhile to introduce a new object into the cache. An admission control policy is used only when bringing in a new object requires purging a few old objects. Thus, if a free entry can be achieved in bringing in a newly accessed object to the cache, admission control is not required to be used.

The idea of admission control has also been discussed in the context of main memory caching in [2]. Having a good admission control policy is especially important for the case of caching non-uniform size objects on disk, because the non-homogeneity of the object sizes causes a considerable amount of disruption and fragmentation when an object is added and others are purged from the cache. Having too frequent replacements may lead to many wasted fragments of space and storing objects which are never hit at all. We wish to bring in a new object only if it is worthwhile to do so. In order to do so, we maintain a small auxiliary cache which maintains only the *identities* of some fixed number of (say Z) objects. (As a thumb rule, we set Z such that it contains the identities of about twice the number of objects as the main cache.) Note that since the auxiliary cache contains only the identities of the objects rather than the objects themselves, the amount of memory required for it is negligible compared to the main cache.

In the auxiliary cache, we also maintain the time-stamp of the last access of an object. This auxiliary cache is maintained in strict LRU order. Thus, in a sense, this cache contains a list of the identities of the more popularly accessed objects though the number of objects for which it contains meta-data is approximately twice as much as the average number of objects in the main cache.

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. In other words, whenever there is a request for an object, whether or not that object is available in the main cache, a replacement occurs on the auxiliary cache using a pure LRU replacement policy.

We would like to have an admission control policy which ensures that at the tth iteration the incoming object i_t is a popular enough to account for the objects that it displaces. In the event that there is enough free space available for the object i_t, then we do not worry about admission control, we simply bring i_t into the cache. Otherwise, we check if i_t occurs on the popular list. If it does not, then the object i_t 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_t does occur on the list, then we read up its last access (other than the current one) and make a comparison between $DF(i_t, t)$ (dynamic frequency of accessed object) and the sum of the dynamic frequency values of the set of possible outgoing objects, which are determined using the replacement rule of the previous section. Observe that in the calculation of the dynamic frequency of the recently accessed object i_t, the time-stamp of its last access (excluding the current access) can be obtained from the auxiliary cache. Of course, after this iteration, the time-stamp of the object i_t on the auxiliary cache is changed to t. Thus, the set of possible objects to be purged from the cache are determined even before it is determined whether or not an object ought to enter the cache. Of course, if it is determined that the dynamic frequency of the incoming object is larger than that of the tentative replacements, then the main cache remains unchanged. Thus, to summarize, the steps of the algorithm which specifies the replacement policy in the even that an accessed object is not found in the cache are as follows:

(1) Let i_t be the object which is desired to be brought into the cache at the tth iteration. If a space fragment of large enough size is available, bring i_t into the cache (free entry).

(2) Determine the set of possible outgoing objects $o_1, o_2, \ldots o_k$, using the least dynamic frequency rule.

(3) If the incoming object does not occur on the auxiliary list, then the cache remains unchanged.

(4) Otherwise, compare the dynamic frequency of the object which was just accessed (incoming object), and the sum of the dynamic frequencies of the set of possible outgoing objects determined by the selection rule. The dynamic frequency of the incoming may be obtained by using the time-stamps in the auxiliary cache.

(5) If the dynamic frequency of the object which was just accessed is larger, then we bring it into the cache.

The algorithm for maintaining the auxiliary cache is very simple and can be stated in one step only:
Whenever an object is accessed, whether or not it is found in the main cache, modify the auxiliary cache in LRU fashion in order to add/change its time-stamp. Any modifications to the auxiliary cache are performed only *after* modifications to the main cache if any.

5 Empirical Results

As a benchmark to compare performance, we implemented the FIFO policy, which avoids the fragmentation issue altogether because of ease of physical placement in a FIFO policy. Thus, the FIFO policy has an inherent advantage over the LDF policy, because of the fact that much lesser space is wasted due to fragmentation. However, as we shall see, the LDF policy more than makes up for this wasted space because of its clever choice of pages which should be cached. Further, the amount of fragmentation is significantly reduced by implementing admission control in conjunction with the LDF policy. Unless explicitly mentioned, all implementations of the LDF policy and its variants include admission control. We also implemented the LLDF-r policy for the cases when $r = 1, 2$.

The primary issue which we have analyzed in this paper, is the variation in performance of the depending upon size or relative frequency or a combination of the two factors. The primary measure of performance that was used in order to test the scheme was the hit ratio.

We tested the scheme both for trace-driven and event-driven simulations.

5.1 Event Driven Simulations

We tested the caching scheme on objects of sizes uniformly distributed between 1 and 500. We assumed that there were 500 different objects, where the sizes of the different objects were equally spaced apart. For the purpose of our experiments, we used one object of each size i. with one object of each size i. Further we assumed that there were several different

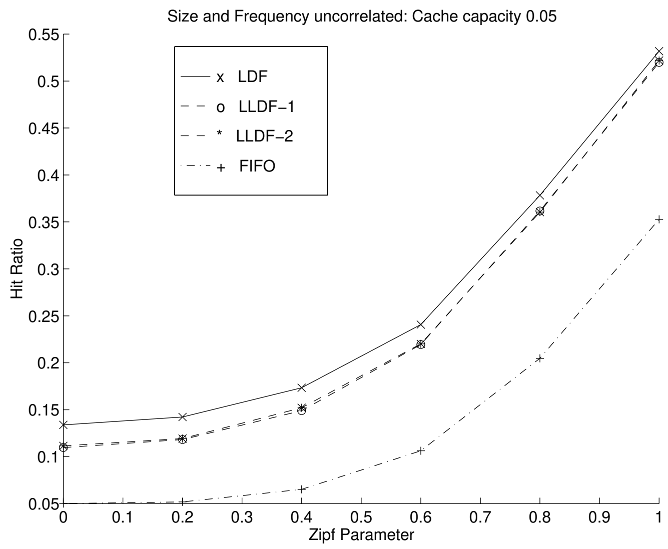

Figure 7: The uncorrelated case: Varying Zipf Parameter

frequencies for the objects. The objects had a zipf-like frequency distribution, where the frequency of object i is proportional to $1/\pi(i)^{\theta}$, where θ is the zipf parameter, and π is a permutation vector. Depending upon the application, the frequency of an object might or might not be related to its size. Taking this into account, we tested for several different cases. The two primary control parameters that we varied were the cache capacity and the degree of skewness in the relative frequency distribution (or, in other words, the frequency zipf parameter).

We noted that the average amount of storage wastage because of fragmentation was between 6 and 10% of the cache size for the case of the LDF scheme. However, since this policy was LRU-like scheme, it tended to keep only frequently accessed objects in the cache. We tested the scheme for three different cases:

(1) There was no correlation between the object size and the frequency. In the Figure 7, we have illustrated the performance of the scheme for varying skew in the relative frequency and fixed cache capacity. The performance for varying cache capacity and fixed skew has been illustrated in Figure 8. As we see from both the figures, the LDF scheme consistently outperforms the FIFO scheme by a reasonably large margin in terms of the hit ratio. The LLDF-1 and LLDF-2 rules were also implemented and their performance was almost as good as the LDF rule. One interesting feature of the performance is that the difference in performance between the LLDF-1 and LLDF-2 rules is so small that it lies within the margin of statistical error. Since the LLDF-1 rule is faster to implement in practice, it seems to be the method of choice. Another

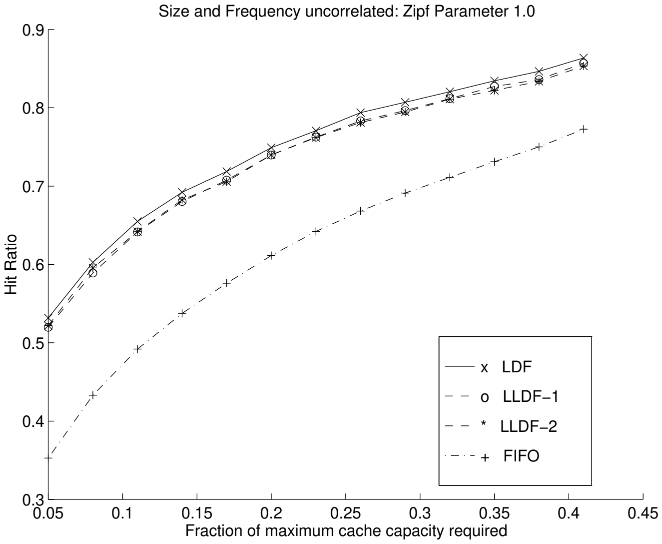

Figure 8: The uncorrelated case: Varying Cache Capacity

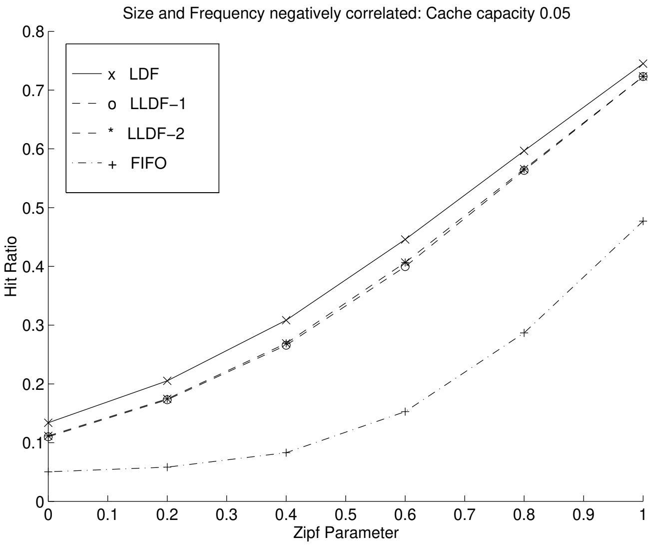

Figure 9: The negatively correlated case: Varying Zipf Parameter

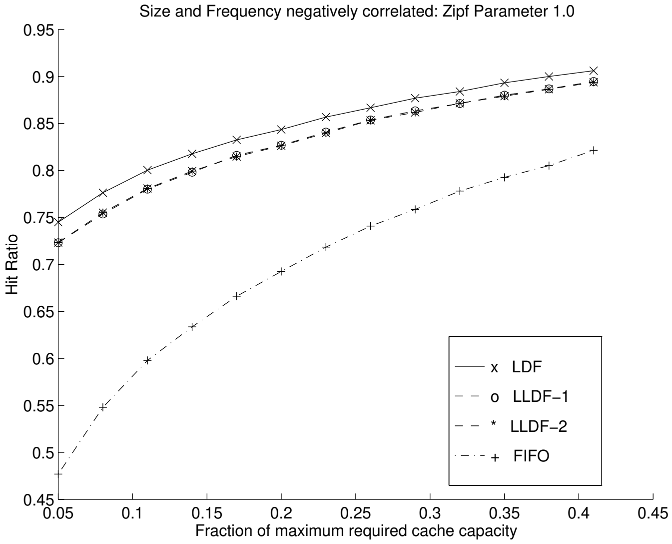

Figure 10: The negatively correlated case: Varying Cache Capacity

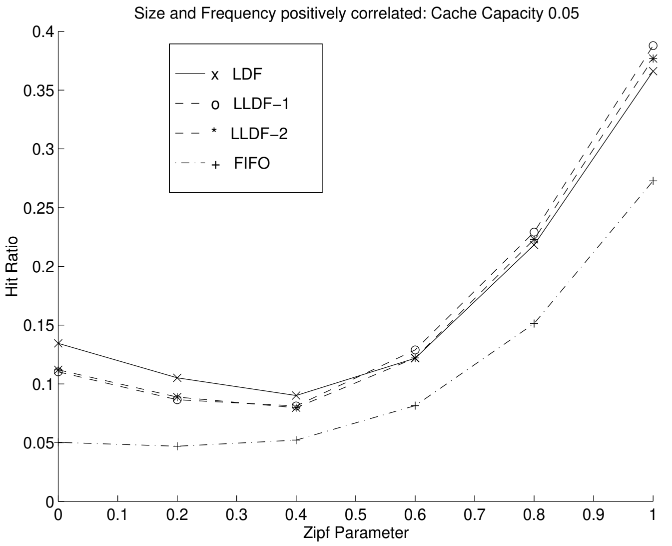

Figure 11: The positively correlated case: Varying Zipf Parameter

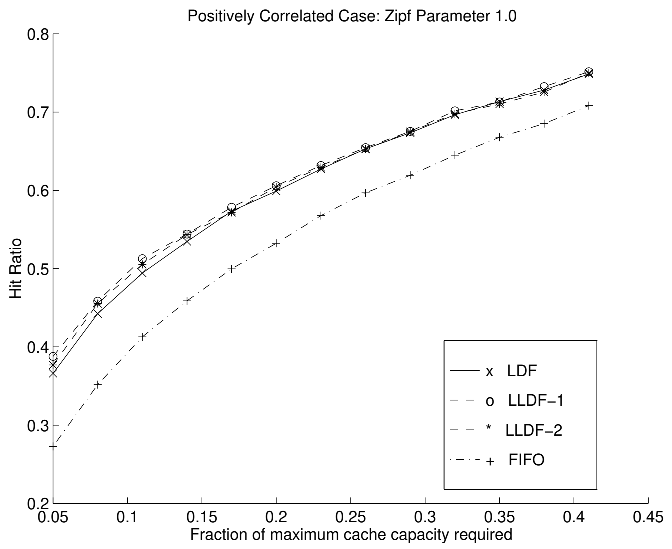

Figure 12: The positively correlated case: Varying Cache Capacity

feature which is evident from the Figure 7 is that as the zipf frequency parameter (or skew) increases, the performance difference between the LDF/LLDF-r and the FIFO policy widens. This is to be expected, since the former policies try to emulate LRU in their bias towards documents of high relative frequency.

(2) There is complete negative correlation between object size and frequency. Thus, the largest object has the least frequency and vice-versa. In this case the FIFO policy does not do well at all. This is because the big objects which enter the cache once in a while to displace many small objects (which also happen to be frequently accessed) cause sufficient disruption to result in a drastic worsening in the performance of the policy. This is one of the situations in which one can appreciate the importance of admission control of the LDF policy. The large objects which are not so frequently accessed and are liable to be space-hoggers rarely ever come into the cache. The corresponding curves for varying skew in the relative frequencies and cache capacities are illustrated in Figures 9 and 10 respectively. Again, the LLDF-1 and the LLDF-2 rules perform almost as well as the LDF policy.

(3) There is complete positive correlation between object size and frequency. Thus the largest object has the highest frequency. The corresponding performance curves are illustrated in Figures 11 and 12. One very interesting feature fo the performance of the LDF scheme as seen from Figure 11 is that with the increase in the skew factor the performance of the LDF scheme first worsens and then finally improves. Further, both the LLDF-1 and LLDF-2 schemes actually outperform the LDF scheme slightly for

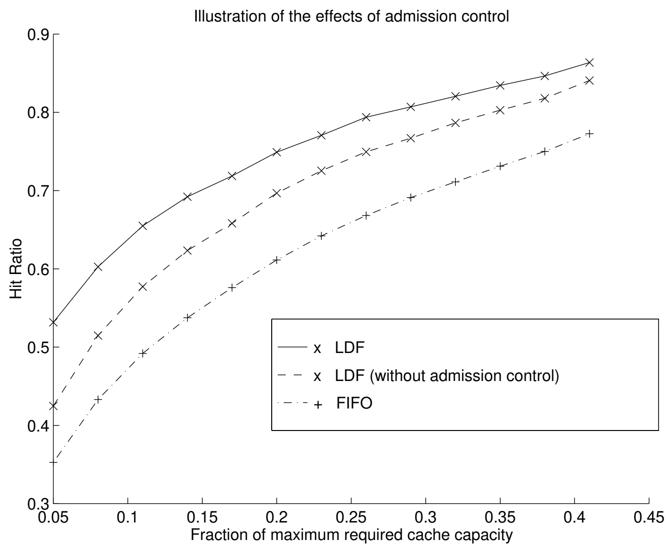

Figure 13: Illustration of the effect of admission control

high skew parameters. The reason for both these apparent anomalies is attributed to the fact that when the object size and frequency are positively correlated, increasing the zipf skew parameter tends to cause two forces to work in opposition to each other: higher skew parameters improve the performance of the LDF scheme because of the affinity of the scheme for more frequently accessed objects, but at the same time the fact that objects with high frequency tend to have larger size and hence occupy more space in the cache works to the detriment of the scheme.

We tested the effect of admission control on the performance of the two policies. The corresponding performance curve is illustrated in Figure 13. As we see, admission control plays an important role in improving the performance of the scheme significantly. The LDF rule without using admission control performs better than the FIFO rule because of a better replacement policy. However, the addition of admission control improves the performance of the scheme even further. This is because of two reasons:

(1) The amount of fragmentation is significantly reduced and consequently there is less wastage of space.

(2) If admission control is not used some part of the cache is often clogged with objects which are accessed once or twice. Such objects get screened off.

5.2 Trace Driven Simulations

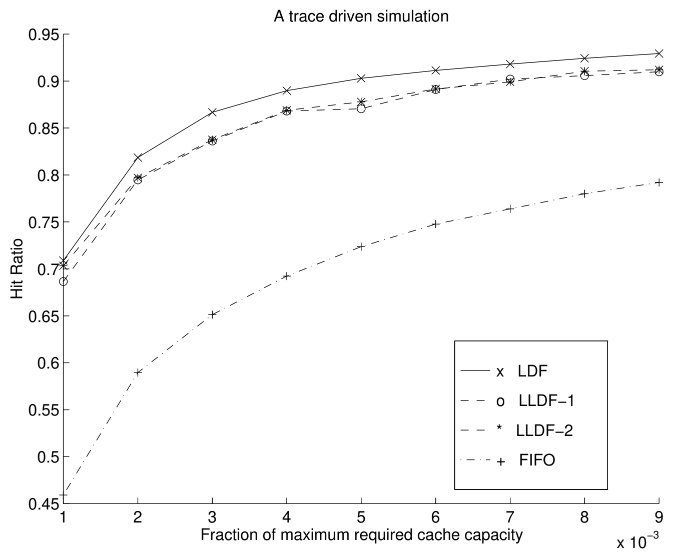

Figure 14: A trace driven simulation

We tested the scheme for a trace driven simulation based on the access-log for the IBM server on January 3, 1996. Altogether, there were 251496 distinct accesses spanning over 2389 distinct pages. The relative frequencies of access were quite skewed and the performance results as illustrated in Figure 14 exhibit a similar pattern to those obtained in the event driven simulations.

6 Conclusions and Summary

In this paper, we have studied the issues of proxy caching on disks. To address the issue of non-uniform objects, the concept of dynamic frequency is first introduced as a measure on the desirability for replacement of a group of consecutively located objects. We proposed a new approach for managing disk caching which consists of both a cache replacement and a cache admission policy. A new replacement policy, referred to as the Least Dynamic Frequency (LDF) Rule, is devised and evaluated. A variation (LLDF) of the LDF based on localized search space for replacement candidates is also considered to provide fast execution time. The admission control provides an effective way of preventing caching less used objects and reducing fragmentation.

References

[1] Abrams M., Standridge C. R., Abdulla G., Williams S., and Fox E. A. Caching Prox-

ies: Limitations and Potentials. *Fourth International World Wide Web Conference Proceedings.*

[2] Aggarwal C. C., Epelman M. A., Wolf J. L., and Yu P. S. On caching policies for web objects. *IBM Research Report, 1996.*

[3] Braun H., and Claffy K. Web Traffic Characterization: An Assessment of the Impact of Caching Documents from NCSA's Web Server. *Proceedings of the 2nd international WWW Conference.* Chicago, October 1994.

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

[5] Danzig P. B., Schwartz M. F., and Hall R. S. A case for caching file objects outside internetworks. *ACM SIGCOMM 93 Conference,* pages 239-248, September 1993.

[6] Glassman S. A caching relay for the World Wide Web. *First International World Wide Web Conference.* Pages 69-76, May 1994.

[7] Markatos E. P. Main memory caching of Web documents. *Computer Networks and ISDN Systems.* 28 (1996) 893-905.

[8] Pitkow J. E. and Recker M. M. A Simple Yet Robust Caching Algorithm Based on Dynamic Access Patterns. *GVU Technical Report: VU-GIT-94-39.* Also in: *Proceedings of the Second International WWW Conference.*

[9] Luotonen A., and Altis K. World Wide Web Proxies. *First International Conference on the World Wide Web.* Geneva, May 1994; (also) *ISDN Systems 27, No 2, 1994.*

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