Exploiting Statistics of Web Traces to Improve Caching Algorithms

Alexander Golynski* Alejandro López-Ortiz* Ray Sweidan*

October 20, 2003

Abstract

Web caching plays an important role in reducing network traffic, user delays, and server load. One important factor in describing the stream of requests made to a given server is the popularity of links between accessed pages. For example, the probability of requesting page A increases if a user makes a request to a page which contains a link to page A. Furthermore, this probability depends on the amount of time elapsed since the request was made to the page containing the link and on popularity of the link. In this project we (1) analyze web access logs and determine the frequency distribution of access and mean life expectancy of correlations, (2) propose two new cache replacement policies that use the above distribution, and (3) evaluate the effectiveness of the new policies by comparing them to widely-used algorithms such as GreedyDual-Size (GDS) and GreedyDual-Frequency (GDSF).

1 Introduction

There are different opinions about the importance of caching in the Web. For example, authors in [8] argue that sizes of hard drives and memory grow very fast so that the cache capacity is not a constraint. Thus, the fine tuning of caching algorithms is not of great importance. However, in [5] authors argue that

1. the byte and hit ratios grow in a log-like fashion, and thus increasing them by some small factor is equivalent to raising the cache size to some power;
2. the growth rate of Web content is much higher than the growth rate of memory sizes suitable for web caching;
3. small improvements in web caching algorithms might lead to savings in network traffic. These saving might be even greater if a hierarchy of caches is used.

*School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, {agolynski, alopez-o, rrsweida }@uwaterloo.ca

We feel that the importance of Web caching lies somewhere in the middle between these extremes. On one hand, some big commercial servers (like www.yahoo.com) cache all its contents in fast memory and seem not to experience problems in all three points mentioned in [5]. On the other hand, for proxies, such an approach is unacceptable, since the size of potential cacheable documents is large. Another area where caching may be applied is streaming media, which is known to be the next "traffic killer". In this case, the second and third points might be of great importance.

2 Background

The first greedy dual algorithm was introduced by Young [9] and dealt with pages of the same size but with varying cost. The algorithm associates with each page a value H_p that is initially set to its fetching cost. Upon a request the requested page is brought into cache and its value is set to its original value. Upon a miss, the page with the lowest H_p is evicted, and all other pages in the cache get their H_p reduced by the value of the evicted page. Thus pages get evicted because they either are "cheap" to fetch or have not been accessed recently. The algorithm is normally implemented using priority queue using H_p as key, and an inflation mechanism for keeping track of the amount of offset required upon evicting a page.

The extension proposed by Cao and Irani [2] accounts for variable document sizes by setting H equal to $\frac{cost}{size}$. The drawback of the resulting GreedyDual-Size (GDS) algorithm is that it does not account for the popularity of documents in the cache. This lead to yet another extension: the GreedyDual-Size-Frequency (GDSF) which was proposed by Cherkasova in [3]. In GDSF, the value of H is set equal to $\frac{freq \times cost}{size}$.

3 Processing and Cleaning up Web Traces

For the experimental part of this project we decided to use web traces granted to us by Information Systems and Technology (IST) of University of Waterloo. Typical entry of these traces is shown in Figure 1.

web sub-server	client ip	time/date of request
www.adm.uwaterloo.ca	X.X.X.X	[05/Jul/2002:01:27:47 -0400]

resource/protocol	code	size
"GET /infoucal/COURSE/course-ENGL.html HTTP/1.1"	200	93589

referrer
"http://www.adm.uwaterloo.ca/infoucal/MATH/mathdeg.html"

browser type
"Mozilla/4.0 (compatible; MSIE 5.01; Windows 98)"

Figure 1: Typical entry in web traces

Several issues have arose while processing these web traces

Confusing names The same object can be named differently, i.e. `http://www.quest.uwaterloo.ca`, `www.quest.uwaterloo.ca`, `quest.uwaterloo.ca` or `quest.uwaterloo.ca/index.html`. Some of these can be easily detected, however some might be very annoying, e.g. `www.quest.uwaterloo.ca` and `quest.uwaterloo.ca` in general might be referring to different pages. This become a serious problem in the logs where there are no web sub-server name field. For this type of requests we can not distinguish between `www.uwaterloo.ca` and `www.quest.uwaterloo.ca`, if the resource field is "GET /" or "GET /index.html".

Confusing sizes In many caching algorithms we know of, it is implicitly assumed that the size of response of web server (provided that its response code is OK=200) is fixed. However, in practice this size depends on the protocol used (i.e. different for "HTTP/1.0" and "HTTP/1.1"), it also might vary depending on the browser used. The other factor is that if an HTTP transfer was interrupted then the size field reflects the amount of bytes transferred before the interruption (status code is still 200). Subsequent requests might resume the download with code 206 (partial content), thus size fields of all these requests should be summed up to get the original size. This problem might become quite critical for caching algorithms which use the reciprocal value of the size, and in the worst case these algorithms will crash if the size is 0 (we encountered this situation during experiments).

Embedded objects We tried to get rid of requests to embedded objects: pictures, java scripts and stylesheets. These requests will be always generated by browser (unless this option is turned off by a user, which seems to be quite uncommon these days). This type of requests will only give us information about the latency that given user is experiencing communicating with University of Waterloo web server. We decided not to deal with this issue in our project, so we sorted out all requests to gif, jpeg, js and css files. However, there are some HTTP requests of this type as well, for example frames or automatic redirection.

These requests can be detected. Fix a user and consider a pair resource-referrer, say that the *active time* of the request is the time difference between requests to the resource and referrer by the user (i.e. the time spent by the user at the referrer page before the request to the resource has been made). We plot graphs of the active times of "computer generated" requests versus "human generated" ones, see Figure 2.

From this figure we see that shapes of the plots are different. Computer generated requests have the property that almost all of them are made within at most 2-3 seconds, while human generated seem to have a peak at 2-10 seconds.

Inaccuracy of time stamp Usually the time of request is provided with precision of one second. Imagine two requests: one arrived at the beginning of the current second and the other at the end. They will have the same time stamp, but the first user will look a little bit faster than the second one from the point of view of the server (e.g. it can make the next request in the same second, while the

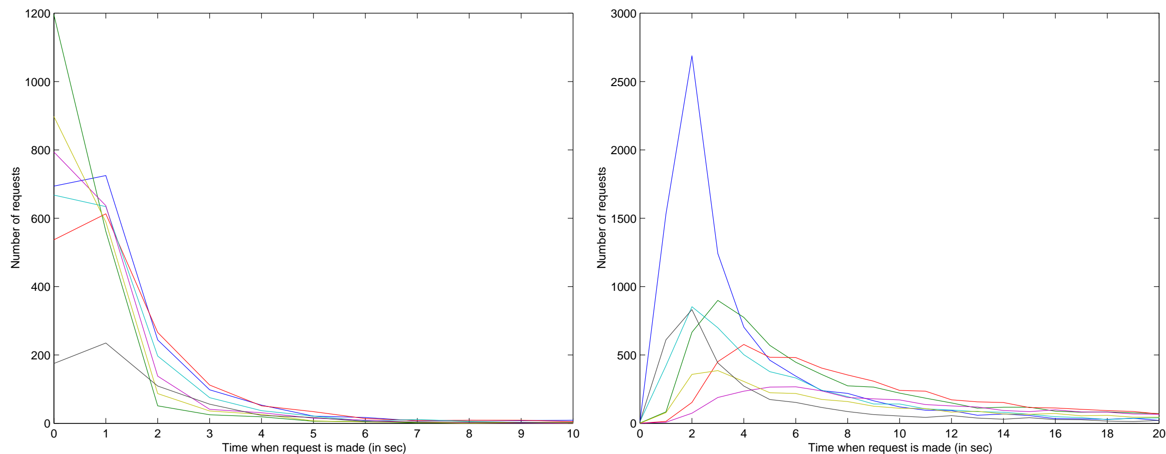

Figure 2: Active time of computer (left) versus human generated (right) requests

other user will not have time for it). This explains why the number of computer generated requests on 0-th second is sometimes smaller than on the 1-st (the effect of the first second), see Figure 2. This effect will spoil a bit the graph of computer generated requests.

Another problem related to this one is that the distant users will appear to be slower than the near ones. For example, for a "frame" link `http://www.adm.uwaterloo.ca/infosche/interface/def.htm` to `http://www.adm.uwaterloo.ca/infosche/interface/main/crse_def.html`, the active time is higher than 4 seconds for 18 percent of users (expected active time for this link is 4 seconds). This effect can spoil the estimation of actual expected active time for "human generated" links by several seconds.

Confusing users The field client ip is not precise enough to distinguish different clients or in ideal case to distinguish different sessions of the same client. For example, the ip address of `httpproxy.math.uwaterloo.ca` was used by many users and does not give any information about the user in a particular request.

Similar work on cleaning web access logs and other software related issues has been done by Krishnamurthy et al. in [7].

4 Popularity of Links

In this section, we propose a method for computing popular links. Here by a link we mean a pair of (absolute) addresses of referrer and resource. We plotted graphs showing popularity of a particular link versus its rank. Rank of a link is its place in a list of all possible links encountered in a given web trace sorted by their popularity. The graphs for five different web logs are shown in Figure 3.

Similar results were obtained in [1], they characterized popularity of pages and popularity of *request strides*. A request stride is sequence of requests from the same

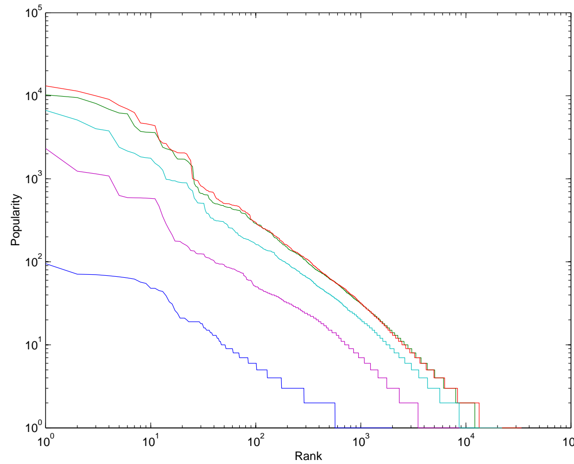

Figure 3: Popularity of link versus its rank

client where the time between successive requests in the stride is not larger than given threshold (they chose threshold of 10 seconds). However, Figure 2 shows that some "human generated" requests are made after the period of 10 seconds, moreover the expected time of the next user request is highly dependent on a particular link and usually exceeds 20 seconds. Figure 4 shows the expected active time of the 23 most popular "human generated" requests (the link from `http://www.uwaterloo.ca` to `http://www.uwaterloo.ca/weather.html` has expected active time 135 seconds)

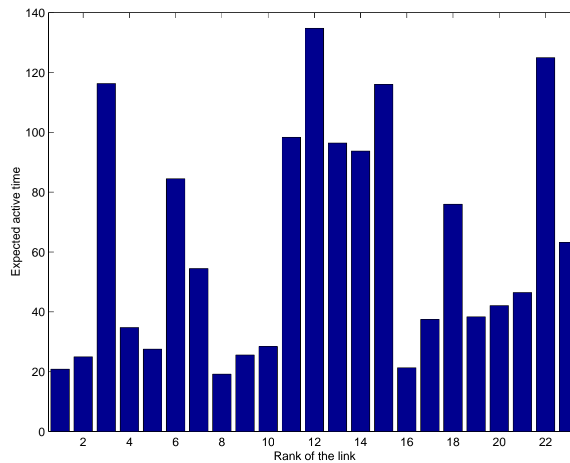

Figure 4: Expected active time for the most popular "human generated" requests

These statistics can help in constructing a good web caching algorithm, however they were computed offline using big amount of memory and computation. Here we aim to obtain a simple heuristic capable of computing the most popular links online using small amount of memory and computation resources. This can be turned into the more general problem of answering iceberg queries.

Assume that we are dealing with a stream of events (e.g. requests), which can be divided into m groups $\{g_1, g_2, \ldots, g_m\}$ (e.g. a group may represent requests which follow a given link). At each given moment of time we want to answer the question: what are the most popular k groups we have seen so far? Since the popularity of links can change over time, one might want to give more weight to the recently seen links. We studied the following heuristic: We maintain k counters each of which monitors a specific link. When the next request arrives, we check if there is a counter monitoring its link. We call the request a *hit* or a *miss* following caching terminology. If a hit occurs then we increase the counter by 1. On a miss we will try to accommodate the corresponding link by evicting links with value below a certain threshold $\theta < 1$. We try to capture recency of requests by multiplying all the counters by a fixed *aging parameter* $p < 1$ every second.
We call this algorithm `POPULAR LINK`, it is shown below.

Initialize all counters to zero
Let $L = \{l_1, l_2, \ldots l_k\}$ be the links of requests arrived at time t.
$C[i] \leftarrow C[i] * p$ // Multiply all the counters by p
for all $l \in L$ is a hit **do**
  $C[l] \leftarrow C[l] + 1$ // Increment counters for hits
end for
for all $l \in L$ is a miss **do**
  Let c be the smallest counter
  if $c < \theta$ **then**
    Release the counter c if it is busy monitoring a link
    Set c to monitor link l
    $C[l] \leftarrow 1$ // Initialize counter for l
  else
    Exit loop
  end if
end for
$t \leftarrow t + 1$

As stated the algorithm requires $\Theta(m)$ operations per second. However, this can be improved using a hybrid of a hash and a heap data structure supporting the following three operations:

Switch counter set counter to monitor another group;

Update value update value of counter;

Lookup given a group number, return the counter which is monitoring the group or NULL if there are no;

Return top return the element with minimum value and (optionally) delete it.

We construct this data structure from a heap and a hash. Let the monitored group id's to be the indexes of the hash. Value of hash at a given group id is the pair of the corresponding counter and its index in the heap. Elements of the heap are the monitored group id's. To compare two elements in the heap the heap, data structure

compares the corresponding counters in the hash. If an element changing its position in the heap the corresponding index in the hash should be updated. Using perfect hashing and Fibonacci heaps we can achieve $O(1)$ amortized time for all operations except for *return top* operation with deletion, the latter one runs in $O(\log(m))$ time.

The following idea helps to eliminate the need to multiply counters by p each second. Let t_0 be the time of the first request. Let $D[i] = C[i]p^{-(t-t_0)}$ be the new counters and call $p^{-(t-t_0)}$ the *aging factor* at time t. Counters $D[i]$ form the same heap as $C[i]$ (since multiplying each value of the heap by a constant preserves the order). It is easier to maintain $D[i]$, since they do not require multiplication step ($C[i] \leftarrow pC[i]$) each second. The step $C[l] \leftarrow C[l]+1$ becomes $D[l] \leftarrow D[l]+p^{-(t-t_0)}$. For the purposes of implementation, we do periodic "clean up", i.e. set all $D[i] \leftarrow D[i]p^{t-t_0}$ and reset $t_0 \leftarrow t$ whenever aging factor $p^{-(t-t_0)}$ exceeds a certain threshold. Clean up is needed to make sure we never overflow floating point counters.

To estimate the aging parameter p, we take the following approach. First, fix some time parameter t'. Then we pretend that we are no longer interested in requests which took place more than t' seconds ago and the value of the link is proportional to number of requests to it within t' seconds. However, in our algorithm the value (=counter) of the link at time t is given by the formula $c_t = pc_{t-1} + \beta r_{t-1}$, where r_t is number of requests to that link on t-th second, β is some multiplicative parameter which does not affect the algorithm's behavior. This gives

$$c_t = b_1 r_{t-1} + b_2 r_{t-2} + \ldots + b_t r_0 \tag{1}$$

where $\{b_t\}$ are defined by $b_1 = \beta$ and $b_k = pb_{k-1}$, or simply $b_k = \beta p^{k-1}$. Our goal is to approximate the following equation (sliding window)

$$c_t = 1 * r_{t-1} + 1 * r_{t-2} \ldots + 1 * r_{t-t'} + 0 * r_{t-t'-1} + 0 * r_{t-t'-2} + \ldots \tag{2}$$

with (1). We choose the following measure of badness of approximation to minimize.

$$\begin{aligned} \mu(p,\beta) &= (b_1 - 1)^2 + \ldots + (b_{t'} - 1)^2 + b_{t'+1}^2 + b_{t'+2}^2 + \ldots \\ &= \sum_{i=1}^{t'} (b_i - 1)^2 + \sum_{i>t'} b_i^2 = \sum_i b_i^2 - 2\sum_{i=1}^{t'} b_i + t' \end{aligned}$$

Substituting $b_k = \beta p^{k-1}$ gives

$$\mu(p,\beta) = \frac{\beta^2}{(1-p^2)} - \frac{2\beta(1-p^{t'+1})}{1-p} + t'$$

minimizing μ over β gives $\beta = A^{-1}v$, where $A = 1/(1-p^2)$ and $v = (1-p^{t'+1})/(1-p)$. So the expression to minimize over p is

$$\mu(p) = -A^{-1}v^2$$

The graph of μ for $t' = 10$ is shown in the Figure 5 This function has exactly one minimum if $t' > 1$ and the argument of this minimum (optimal p) asymptotically behaves like

$$p_{\min} \sim 1 - \frac{c}{t'} + O\left(\frac{1}{t'^2}\right)$$

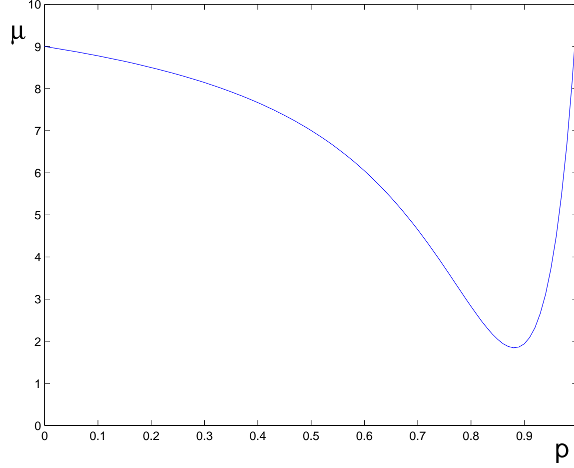

Figure 5: Plot of μ over p for $t' = 10$

if $t' \to \infty$. The constant c is the positive root of the equation $\exp(c) - 1 - 2c = 0$, $c \approx 1.256$

These considerations can be easily extended to the case where counters are updated by n-th order linear recurrence.

$$c_t = \lambda_1 c_{t-1} + \lambda_2 c_{t-2} + \ldots + \lambda_n c_{t-n} + \xi_1 r_{t-1} + \xi_2 r_{t-2} + \ldots + \xi_n r_{t-n}$$

In this case, general solution is $b_k = \beta_1 {p_1}^k + \beta_2 {p_2}^k \ldots \beta_n {p_n}^k$ for some constants $\{\beta_i, p_i\}$. Define matrix A and vector v to be

$$A = \begin{bmatrix} \frac{1}{p_1^2} & \frac{1}{p_1 p_2} & \cdots & \frac{1}{p_1 p_n} \\ \frac{1}{p_2 p_1} & \frac{1}{p_2^2} & \cdots & \frac{1}{p_2 p_n} \\ \vdots & \ddots & \ddots & \vdots \\ \frac{1}{p_n p_1} & \frac{1}{p_n p_2} & \cdots & \frac{1}{p_n^2} \end{bmatrix}; \qquad v = \begin{bmatrix} \frac{(1-p_1)^{t'+1}}{1-p_1} & \frac{(1-p_2)^{t'+1}}{1-p_2} & \ldots & \frac{(1-p_n)^{t'+1}}{1-p_n} \end{bmatrix}^T$$

Then the measure $\mu(p, \beta)$ becomes

$$\mu(p, \beta) = \beta^T A \beta - 2\beta^T v + t'$$

Minimizing over β gives $A\beta = v$ and

$$\mu(p) = -v^T A^{-1} v$$

which is hard to minimize over p numerically. Experiments (for $n = 2, 3$) show that the minimum is achieved when $p_1 = p_2 = \ldots = p_n$ which probably can be proved for general n. This problem can be also generalized for any "shape" of sliding window, i.e. if we wish to approximate

$$c_t = w_1 r_{t-1} + w_2 r_{t-2} + \ldots + w_{t'} r_{t-t'} + 0 * r_{t-t'-1} + 0 * r_{t-t'-2} + \ldots \tag{3}$$

for some (positive) constants w_i.

The above considerations suggest the use of a simple 1-dimensional recurrence. We implemented this idea in Perl using the efficient data structures described above. A similar approach (but for a different purpose) was proposed in [6], their idea was to halve the value every two days, which corresponds to our value of $p = (1/2)^{1/(2*24*60*60)} \approx 0.9999959887$ and the value of $t' = 313223$ seconds (3.5 days). This value of quite large, as it implies that we remember the entire history for 3.5 days. To compare different algorithms we use the competitive ratio defined as follows. Define *popularity* of a group l, $n(l)$ to be the number of occurrences of events of group l in the whole data stream. At time t the competitive ratio $f(t)$ of the algorithm is the sum of popularities of the groups that have a counter at time t divided by the sum of popularities of top m groups (maximum possible popularity algorithm can achieve). More formally,

$$f(t) = \frac{\sum_{l \in C_t} n(l)}{\sum_{l \in T_m} n(l)} \tag{4}$$

where C_t is the set of groups monitored at time t and T_m is the set of m most popular groups.

We made a number of experiments for different traces and values of m, looking for the optimal t'. For example, if we try $t' = 11250$ seconds and compare it with $t' = 313223$ mentioned above, we get the plots shown in Figure 6. We found out that

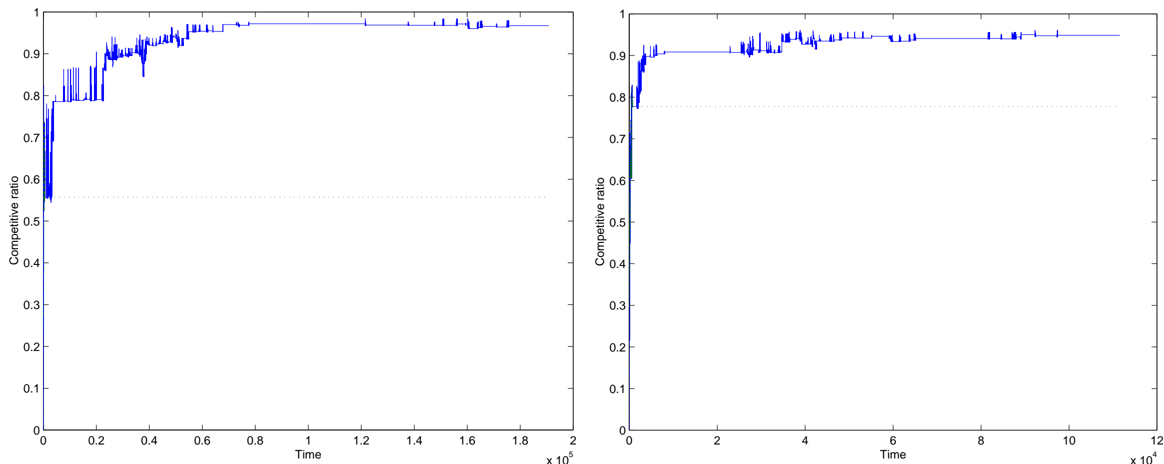

Figure 6: Measuring performance of `POPULAR LINKS` ($t' = 11250$ - solid line, $t' = 313223$ - dashed line

small values of t', (e.g. $t' < 10$) lead to a very unstable competitive ratio, while for big values of t' the function f stabilizes too fast, i.e. the monitored counters will be chosen among the popular groups in the beginning of the stream (usually stabilizes no later than 5000-th event), which are not necessary the most popular in the entire stream. These groups are popular enough to be capable of maintaining sufficient values, which allows them to hold counters and not being evicted. This prevent all the other groups (including the most popular ones) from entering the cache. Figure 7 shows the behavior of `POPULAR LINKS` with $p = 1$ for the first 3000 requests and with $p = 0.9$ for 500 requests taken from the middle of the stream.

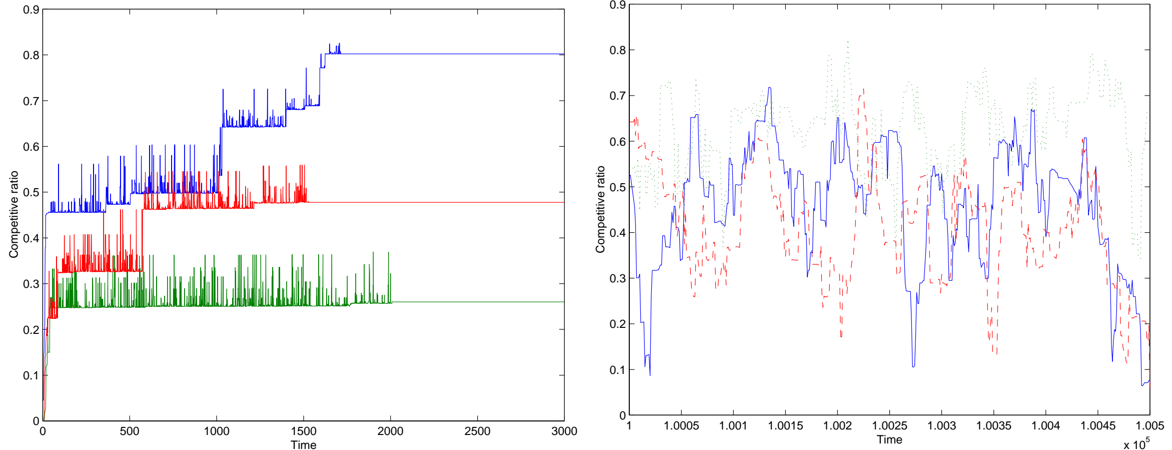

Figure 7: Measuring performance of POPULAR LINKS for the margin cases of the aging parameter ($p = 1$ on the left and $p = 0.9$ on the right)

5 Page Dependency

Based on the concept of functional dependencies in database applications, we define a *page dependency* model which captures the time-dependent transition probability matrices. An entry p_{ij} in a given probability matrix $P^{(t_a)}$ represents the probability that page j is requested after page i within time interval t_a. The active time interval t_a is a measure of the period during which the correlation between pages i and j is considered active. The main role of t_a is to divide a given set of $P^{(t)}$ matrices into two groups: an active group if $t < t_a$, and a dormant group otherwise. This idea can be extended to a larger number of groups in future work. The connection between page dependency and P^{t_a} is made by considering a column j in P^{t_a} and finding the row i_{max} which produces the highest probability of requesting page j. We define $page_{i_{max}}$ to be the *determining page* of page j, and $dependency(i, t_a)$ to be the set of pages determined by page i within an active period t_a.

5.1 Approximate Page Dependency

In Section 6 we show how to use the concepts of page dependency and active time interval in designing a new extension of the greedy-dual family of algorithms. In this section, we describe two approaches to computing page dependency sets. The first approach uses brute force to keep a hash table of the requested pages. For each requested page, an inner hash is used to maintain the count of referring pages and the identity of the page with the highest count. This approach was used in our experiments on the proposed heuristic. Although the memory requirements of this method makes it infeasible in practice, the use of brute force for testing and "proof of concept" purposes was acceptable. We illustrate the method with an example (Figure 8), which shows how hash tables (right) are used to maintain the determining pages for the requests shown on left (assume $t_a = 40$). The resulting page dependencies are: $1 \rightarrow \{3\}, 3 \rightarrow \{1, 2\}$;

requested page	referrer page
1	3
2	1
2	3
1	3
3	2
2	4
3	1
2	3

Requested page	requests (pid, count)	max	argmax
1	(3, 2)	2	3
2	(1, 1); (3, 2); (4, 1);	2	3
3	(1, 1); (2, 1)	1	1

Figure 8: Computing page dependency: link sequence (left), counting referrer pages (right)

both $dependency(2, 40)$ and $dependency(4, 40)$ are empty.

The second approach to determining page dependency belongs to the well researched area of finding fast algorithms for frequent itemsets/hotlists. The paper [4] describes in more detail different algorithms which can be applied to finding frequent items in a data stream. Our implementation of finding frequent itemsets is described in section 4.

6 GreedyDual-Active

In this section we describe two methods for using the popularity of *page links* in caching replacement. The new methods are extensions to GreedyDual family of algorithms.

6.1 Extending GreedyDual-Size

To incorporate the effect of page links and active time t_a, we extend the GreedyDual-Size proposed by Cao and Irani in [2] as follows. When a page p is requested, we identify the pages in cache which are determined by page p. Then for each such page, we increase its current H value to the maximum possible at this time and store the increase amount in Δ. The increase amount represents a *credit* that is granted to each determined page for a limited amount of time. When the credit period expires, the value of the corresponding page is restored to its original value, provided that it is still above the minimum value L. The idea behind the credit and expiry period is to give each page determined by the currently requested page p a chance to prove that it is indeed going to be requested "soon". Therefore, it would make sense to increase the chances for these pages to stay in the cache. Here is the pseudocode for the above extension, called GreedyDual-Active-1.

$L \leftarrow 0$
Let p be the current request
for all page s in cache **do**

if page s is determined by p **then**
$\Delta(s) \leftarrow [L + cost(s)/size(s)] - H(s)$
$H(s) \leftarrow L + cost(s)/size(s)$
$expiryTime(s) \leftarrow currentTime + t_a$
else
if $0 < expiryTime(s) < currentTime$ **then**
$expiryTime \leftarrow 0$
$H(s) \leftarrow max\{L, H(s) - \Delta(s)\}$
end if
end if
end for
Run GreedyDual-Size(p)

A similar extension can be made to Cherkasova's greedy-dual-size-frequency algorithm [3].

6.2 Extending GreedyDual-Frequency

As an alternative to using page dependency, we use the list of top frequent links to decide which page(s) in cache is likely to be requested next (given that the current request is for page p). The top frequent links are maintained using the method described in section 4. Let Top be the set of such top links, and let S_p be the set formed by extracting from Top all the links from p to some page in cache. For each link $l = (from, to)$ in S_p, we give page to a credit equal to $\frac{cost}{size} \times \frac{freq}{sum}$, where $cost$ and $size$ are as defined before, $freq$ is the value of counter $count(l)$ monitoring link l (as described in section 4), and sum is the sum of $count(l)$ over all links in S_p. The variable sum measures the degree of competition for credit between pages predicted by the $from$ page. Credits are maintained in a queue for a period determined by the active time, and, upon expiry, all credits are withdrawn. Here is pseudocode for the above method, called GreedyDual-Active-2.

$sum \leftarrow 0$
Let p be the current request
for all link $l = (p, q)$ in Top and q in cache **do**
$sum \leftarrow sum + count(l)$
$S_p \leftarrow S_p \cup q$
end for
for all page s in S_p **do**
$credit \leftarrow (freq/sum)(cost(s)/size(s))$
$H(s) \leftarrow H(s) + credit$
$expiryTime(s) \leftarrow currentTime + t_a$
add $credit, expiryTime, s$ to queue
end for
for all credit in queue, $expiryTime < currentTime$ **do**
delete credit's record from queue
$H(s) \leftarrow H(s) - credit$
end for

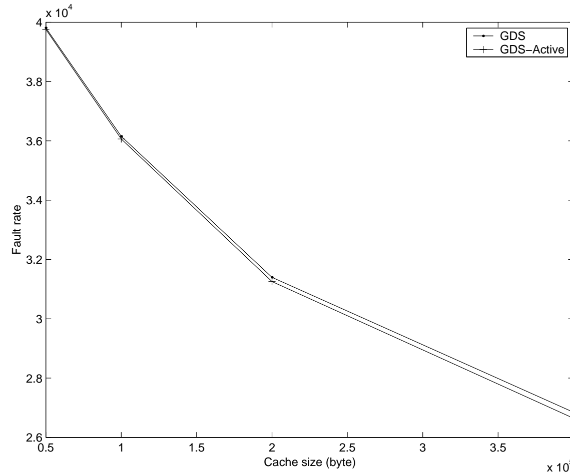

Figure 9: Performance of GreedyDual-Active-1 vs. GreedyDual-Size

Run GreedyDual-Frequency(p)

7 Performance Evaluation

We used a trace-driven simulation to compare the performance of GreedyDual-Active(-1 & -2) with GreedyDual-Size and GreedyDual-Frequency, respectively.

7.1 Traces Used in Experiments

Initially we had to decide whether to go with proxy or server traces. Proxy traces offer a client-centric view of the web while server traces are site-centric. The focus on site makes server traces easier to analyze and simplifies the task of detecting pattern and correlations. Thus we used traces from `uwaterloo.ca` web server. The logs cover the most recent 4 weeks, and include requests made to such servers as adm., co-op, human resources.

Both algorithms were implemented in perl and their fault rate was determined on a trace of $\approx 1,600,000$ requests. The cache size was varied between 1% to 10% of the traces' working set (i.e. total unique file size). Figure 9 shows the fault rate of both GreedyDual-Size the GreedyDual-Active1 for different cache sizes. Although the magnitude of improvement obtained from GD-Active-1 is small, the results show that the underlying structure of page links can be exploited beneficially in caching policies.

The results of varying the active time parameter (t_a = 4, 40, and 400) showed that the fault rate of GD-Active-1 is *insensitive* to changes in the value of t_a. This result is consistent with the fact that the current implementation of GD-Active-1 makes the following simplifying assumption:

- Active time t_a is uniform over all pages. We could extend this by making t_a a function of one or more property of each page (e.g., type and popularity).

- Each page is determined by exactly one referring page. We could extend this to include a set of *most frequent* referrers.
- The maximum H value of each determined page is constant. Again, we could relax this restriction so that H_max can become a function of the cached pages.
- When a page is requested repeatedly within the active time interval, the old credit data is overwritten. The result of being determined twice is to merely extend the $expiryTime$ and reset the H value; old “debt” is forgiven. In a sense, GreedyDual-Active-1 is too generous. We could fix this by adding a data structure for all previously awarded credits, which could improve the sensitivity of the algorithm to changes in t_a.

We suspect that a more elaborate definition of t_a and $dependency(i, t_a)$ may result in a more pronounced improvement in the relative performance of GD-active.

Figure 10 shows the fault rate of both GreedyDual-Frequency the GreedyDual-Active-2 for different cache sizes. The magnitude of improvement obtained from GD-Active-2 is more significant than those obtained from GD-Active-1. This is due to two main reasons:

1. The use of link frequency in Active-2 allows each page in cache to be ”predicted” by more than one page during an active time interval; Active-1, on the other hand, uses only one predicting page.
2. Experimentation with GD-Active-2 has resulted in a number of fine-tuning features, such as multiplying the base H value (i.e. $cost/size$) by a factor of 1/6, which boosted the overall performance of the algorithm.

We also experimented with various values for t_a and the number of links kept in the top frequent set. The results showed that the fault rate is insensitive to changes in these two parameters.

References

[1] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS)*, 1996.

[2] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.

[3] Ludmila Cherkasova. Improving www proxies performance with greedy-dual-size-frequency caching policy. Technical Report HPL-98-69R1, Hewlett-Packard Laboratories, November 1998.

[4] Erik D. Demaine, Alejandro Lopez-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *The 10th European Symposium on Algorithms*, 2002.

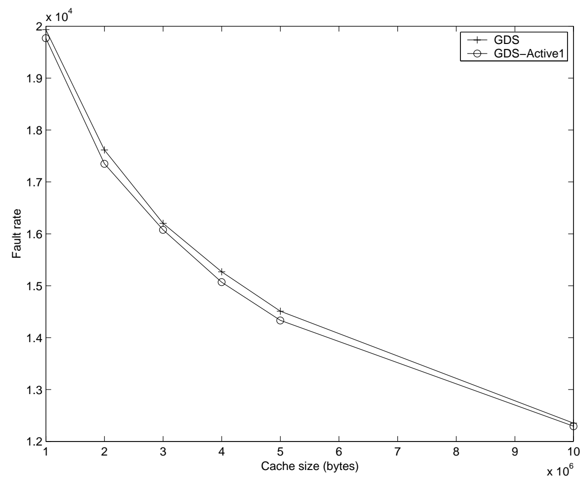

Figure 10: Performance of GreedyDual-Active-2 vs. GreedyDual-Frequency

[5] Shudong Jin and Azer Bestavros. GreedyDual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams. In *The 5th International Web Caching and Content Delivery Workshop*, 2000.

[6] Shudong Jin and Azer Bestavros. Popularity-aware greedy dual-size web proxy caching algorithms. In *The 20th International Conference on Distributed Computing Systems*, 2000.

[7] B. Krishnamurthy and J. Rexford. Software issues in characterizing web server logs, 1998.

[8] Michael Rabinovich and Oliver Spatscheck. *Web Caching and Replication.* Addison Wesley, 2002.

[9] Neal Young. Online caching as cache size varies. In *Proceedings of the 2nd Annual ACM-SLAM Symposium on Discrete Algorithms*, pages 241–250, 1991.