# Cost-Aware WWW Proxy Caching Algorithms

Pei Cao
*Department of Computer Science,*
*University of Wisconsin-Madison.*
cao@cs.wisc.edu

Sandy Irani
*Information and Computer Science Department,*
*University of California-Irvine.*
irani@ics.uci.edu

## Abstract

Web caches can not only reduce network traffic and downloading latency, but can also affect the distribution of web traffic over the network through cost-aware caching. This paper introduces GreedyDual-Size, which incorporates locality with cost and size concerns in a simple and non-parameterized fashion for high performance. Trace-driven simulations show that with the appropriate cost definition, GreedyDual-Size outperforms existing web cache replacement algorithms in many aspects, including hit ratios, latency reduction and network cost reduction. In addition, GreedyDual-Size can potentially improve the performance of main-memory caching of Web documents.

## 1 Introduction

As the World Wide Web has grown in popularity in recent years, the percentage of network traffic due to HTTP requests has steadily increased. Recent reports show that Web traffic has constituted 40% of the network traffic in 1996, compared to only 19% in 1994. Since the majority of Web documents requested are static documents (i.e. home pages, audio and video files), caching at various network points provides a natural way to reduce web traffic. A common form of web caching is caching at HTTP proxies, which are intermediateries between browser processes and web servers on the Internet (for example, one can choose a proxy by setting the network preference in the Netscape Navigator[1]).

There are many benefits of proxy caching. It reduces network traffic, average latency of fetching Web documents, and the load on busy Web servers. Since documents are stored at the proxy cache, many HTTP requests can be satisfied directly from the cache instead of generating traffic to and from the Web server. Numerous studies [WASAF96] have shown that the hit ratio for Web proxy caches can be as high as over 50%. This means that if proxy caching is utilized extensively, the network traffic can be reduced significantly.

Key to the effectiveness of proxy caches is a document replacement algorithm that can yield high hit ratio. Unfortunately, techniques developed for file caching and virtual memory page replacement do not necessarily transfer to Web caching.

There are three primary differences between Web caching and conventional paging problems. First, web caching is variable-size caching: due to the restriction in HTTP protocols that support whole file transfers only, a cache hit only happens if the entire file is cached, and web documents vary dramatically in size depending on the information they carry (text, image, video, etc.). Second, web pages take different amounts of time to download, even if they are of the same size. A proxy that wishes to reduce the average latency of web accesses may want to adjust its replacement strategy based on the download latency. Third, access streams seen by the proxy cache are the union of web access streams from tens to thousands of users, instead of coming from a few programmed sources as in the case of virtual memory paging.

Proxy caches are in a unique position to affect web traffic on the Internet. Since the replacement algorithm decides which documents are cached and which documents are replaced, it affects which future requests will be cache hits. Thus, if the institution employing the proxy must pay more on some network links than others, the replacement algorithm can favor expensive documents (i.e. those travelling through the expensive links) over cheap documents. If it is known that certain network paths are heavily congested, the caching algorithm can retain more documents which must travel on congested paths. The proxy cache can reduce its contribution to the network router load by preferentially caching documents that travel more hops. Web cache replace-

[1] Navigator is a trademark of Netscape Inc.

ment algorithms can incorporate these considerations by associating an appropriate *network cost* with every document, and minimizing the total cost incurred over a particular access stream.

Today, most proxy systems use some form of the Least-Recently-Used replacement algorithm. Though some proxy systems also consider the time-to-live fields of the documents and replace expired documents first, studies have found that time-to-live fields rarely correspond exactly to the actual life time of the document and it is better to keep expired-but-recently-used documents in the cache and validate them by querying the server [LC97]. The advantage of LRU is its simplicity; the disadvantage is that it does not take into account file sizes or latency and might not give the best hit ratio.

Many Web caching algorithms have been proposed to address the size and latency concerns. We are aware of at least nine algorithms, from the simple to the very elaborate, proposed and evaluated in separate papers, some of which give conflicting conclusions. This naturally leads to a state of confusion over which algorithm should be used. In addition, none of the existing algorithms address the network cost concerns.

In this paper, we introduce a new algorithm, called GreedyDual-Size, which combines locality, size and latency/cost concerns effectively to achieve the best overall performance. GreedyDual-Size is a variation on a simple and elegant algorithm called GreedyDual [You91b], which handles uniform-size variable-cost cache replacement. Using trace-driven simulation, we show that GreedyDual-Size with appropriate cost definitions out-performs the various "champion" web caching algorithms in existing studies on a number of performance issues, including hit ratios, latency reduction, and network cost reduction.

# 2 Existing Results

The *size* and *cost* concerns make web caching a much more complicated problem than traditional caching. Below we first summarize the existing theoretical results, then take a look at a variety of web caching algorithms proposed so far.

## 2.1 Existing Theoretical Results

There are a number of results on the optimal offline replacement algorithms and online competitive algorithms on simplified versions of the Web caching problem.

The variable document sizes in web caching make it much more complicated to determine an optimal offline replacement algorithm. If one is given a sequence of requests to uniform size blocks of memory, it is well known that the simple rule of evicting the block whose next request is farthest in the future will yield the optimal performance [Bel66]. In the variable-size case, no such offline algorithm is known. In fact, it is known that determining the optimal performance is NP-hard [Ho97], although there is an algorithm which can approximate the optimal to within a logarithmic factor [Ir97]. The approximation factor is logarithmic in the maximum number of bytes that can fit in the cache, which we will call $k$.

For the cost consideration, there have been several algorithms developed for the uniform-size variable-cost paging problem. GreedyDual [You91b], is actually a range of algorithms which include a generalization of LRU and a generalization of FIFO. The name GreedyDual comes from the technique used to prove that this entire range of algorithms is optimal according to its *competitive ratio*. The competitive ratio is essentially the maximum ratio of the algorithms cost to the optimal offline algorithm's cost over all possible request sequences. (For an introduction to *competitive analysis*, see [ST85]).

We have generalized the result in [You91b] to show that our algorithm GreedyDual-Size, which handles documents of differing sizes and differing cost (described in Section 4), also has an optimal competitive ratio. Interestingly, it is also known that LRU has an optimal competitive ratio when the page size can vary and the cost of fetching a document is the same for all documents or proportional to the size of a document [FKIP96].

## 2.2 Existing Document Replacement Algorithms

We describe nine cache replacement algorithms proposed in recent studies, which attempt to minimize various cost metrics, such as miss ratio, byte miss ratio, average latency, and total cost. Below we give a brief description of all of them. In describing the various algorithms, it is convenient to view each request for a document as being satisfied in the following way: the algorithm brings the newly requested document into the cache and then evicts documents until the capacity of the cache is no longer exceeded. Algorithms are then distinguished by how they choose which documents to evict. This view allows for the possibility that the requested document itself may be evicted upon its arrival into the cache, which means it replaces no other document in the cache.

- **Least-Recently-Used** (LRU) evicts the document which was requested the least recently.

- **Least-Frequently-Used** (LFU) evicts the document which is accessed least frequently.
- **Size** [**WASAF96**] evicts the largest document.
- **LRU-Threshold** [**ASAWF95**] is the same as LRU, except documents larger than a certain threshold size are never cached;
- **Log(Size)+LRU** [**ASAWF95**] evicts the document who has the largest log(size) and is the least recently used document among all documents with the same log(size).
- **Hyper-G** [**WASAF96**] is a refinement of LFU with last access time and size considerations;
- **Pitkow/Recker** [**WASAF96**] removes the least-recently-used document, except if all documents are accessed today, in which case the largest one is removed;
- **Lowest-Latency-First** [**WA97**] tries to minimize average latency by removing the document with the lowest download latency first;
- **Hybrid**, introduced in [WA97], is aimed at reducing the total latency. A function is computed for each document which is designed to capture the utility of retaining a given document in the cache. The document with the smallest function value is then evicted. The function for a document $p$ located at server $s$ depends on the following parameters: $c_s$, the time to connect with server $s$, $b_s$ the bandwidth to server $s$, $n_p$ the number of times $p$ has been requested since it was brought into the cache, and $z_p$, the size (in bytes) of document $p$. The function is defined as:

$$\frac{\left(c_s + \frac{W_b}{b_s}\right)(n_p)^{W_n}}{z_p}$$

where $W_b$ and $W_n$ are constants. Estimates for $c_s$ and $b_s$ are based on the the times to fetch documents from server $s$ in the recent past.

- **Lowest Relative Value** (LRV), introduced in [LRV97], includes the cost and size of a document in the calculation of a value that estimates the utility of keeping a document in the cache. The algorithm evicts the document with the lowest value. The calculation of the value is based on extensive empirical analysis of trace data. For a given $i$, let $P_i$ denote the probability that a document is requested $i+1$ times given that it is requested $i$ times. $P_i$ is estimated in an online manner by taking the ratio $D_{i+1}/D_i$, where $D_i$ is the total number of documents seen so far which have been requested at least $i$ times in the trace. $P_i(s)$ is the same as $P_i$ except the value is determined by restricting the count only to pages of size $s$. Furthermore, let $1-D(t)$ be the probability that a page is requested again as a function of the time (in seconds) since its last request $t$; $D(t)$ is estimated as

$$D(t) = .035\log(t+1) + .45\left(1 - e^{\frac{-t}{2e6}}\right).$$

Then for a particular document $d$ of size $s$ and cost $c$, if the last request to $d$ is the $i$'th request to it, and the last request was made $t$ seconds ago, $d$'s value in LRV is calculated as:

$$V(i,t,s) = \begin{cases} P_1(s)(1-D(t)) * c/s & \text{if } i = 1 \\ P_i(1-D(t)) * c/s & \text{otherwise} \end{cases}$$

Among all documents, LRV evict the one with the lowest value. Thus, LRV takes into account locality, cost and size of a document.

Existing studies using actual Web proxy traces narrowed down the choice for proxy replacement algorithms to LRU, SIZE, Hybrid and LRV. Results in [WASAF96, ASAWF95] show that SIZE performs better than LFU, LRU-threshold, Log(size)+LRU, Hyper-G and Pitkow/Recker. Results in [WASAF96] also show that SIZE outperforms LRU in most situations. However, a different study [LRV97] shows that LRU outperforms SIZE in terms of byte hit rate. Comparing LFU and LRU, our experiments show that though LFU can outperform LRU slightly when the cache size is very small, in most cases LFU performs worse than LRU. In terms of minimizing latency, [WA97] show that Hybrid performs better than Lowest-Latency-First. Finally, [LRV97] shows that LRV outperforms both LRU and SIZE in terms of hit ratio and byte hit ratio. One disadvantage of both Hybrid and LRV is their heavy parameterization, which leaves one uncertain about their performance across access streams.

However, the studies offer no conclusion on which algorithm a proxy should use. Essentially, the problem is finding an algorithm that can combine the observed access pattern with the cost and size considerations.

### 2.2.1 Implementation Concerns

The above "champion" algorithms vary in time and space complexity. In the cases when there are a large number of documents in the cache, this can have a dramatic effect on the time required to determine which document to evict.

- LRU can be implement easily with $O(1)$ overhead per cached file and $O(1)$ time per access;

- Size can be implemented by maintaining a priority queue on the documents in memory based on their size. Since the size of a document does not change, handling a hit requires $O(1)$ time and handling an eviction requires $O(\log k)$ time, where $k$ is the number of cached documents.

- Hybrid is also implemented using a priority queue, thus requiring $O(\log k)$ time to find a replacement. Furthermore, it requires an array keeping track of the average latency and bandwidth for every Web server. It is used in estimating the downloading latency of a web page. This requires extra storage. In addition, since the estimate is updated every time a connection to the server is made, a faithful implementation requires updating many pages' latency estimation. We found this prohibitively time-consuming, and we omit the step in the implementation. We find that omitting the step does not affect our results significantly.

- LRV requires $O(1)$ storage per cached file plus some bookkeeping information. If the $Cost$ in LRV is proportional to $Size$, the authors of the algorithm suggests an efficient method that can find the replacement in $O(1)$ time, though the constants can be large. If $Cost$ is arbitrary, then $O(k)$ time is needed to find a replacement. We also found that the cost of calculating $D(t)$ are very high, since it uses *log* and *exp*.

Another concern about both Hybrid and LRV is that they employ constants which might have to be tuned to the patterns in the request stream. For Hybrid, we use the values which were used in [WA97] in our simulations. We did not experiment with tuning those constants to improve the performance of Hybrid.

Though LRV can incorporate arbitrary network costs associated with documents, the $O(k)$ computational complexity of finding a replacement can be prohibitively expensive. The problem is that $D(t)$ has to be recalculated for every document every time some document has to be replaced. The overhead makes LRV impractical for proxy caches that wish to take network costs into consideration.

# 3 Web Proxy Traces

As the conclusions from a trace-driven study inevitably depend on the traces, we tried to gather as many traces as possible. We were successful in obtaining the following traces of HTTP requests going through Web proxies:

- Digital Equipment Corporation Web Proxy server traces [DEC96](Aug-Sep 1996), servicing about 17,000 workstations, for a period of 25 days, containing a total of about 24,000,000 accesses;

- University of Virginia proxy server and client traces [WASAF96] (Feb-Oct 1995), containing four sets of traces, each servicing from 25 to 61 workstations, containing from 13,127 to 227,210 accesses;

- Boston University client traces [CBC95](Nov 1994 - May 1995), containing two sets of traces, one servicing 5 workstations (17,008 accesses), the other 32 workstations (118,105 accesses);

We are in the process of obtaining more traces from other sources.

We present the results of fourteen traces. They include all of Virginia Tech and Boston University traces, and eight subsets of the DEC traces. The subsets are Web accesses made by users 0-512, and users 1024-2048, in each week, for the three and a half weeks period from Aug. 29 to Sep. 22, 1996. The use of the subsets is partly due to our current simulator's limitation (it cannot simulate more than two million requests at a time), and partly due to our observation that a *caching* proxy server built out of a high-end workstation can only service about 512 users at a time.

We perform some necessary pre-processing over the traces. For the DEC traces, we simulated only those requests whose replies are cacheable as specified in HTTP 1.1 [HT97] (i.e. GET or HEAD requests with status 200, 203, 206, 300, or 301, and not a "cgi_bin" request). In addition, we do not include those requests that are queries (i.e. "?" appears in the URL), though such requests are a small fraction of total cacheable requests (around 3% to 5%). For Virginia Tech traces, we simulated only the "GET" requests with reply status 200 and a known reply size. Thus, our numbers differ from what are reported in [WASAF96]. The Virginia Tech traces unfortunately do not come with latency information. For Boston University traces, we simulated only those requests that are not serviced out of browser caches.

## 3.1 Locality in Web Accesses

In the search for an effective replacement algorithm, we analyzed the traces to understand the access patterns of Web requests seen by the proxies. The strik-

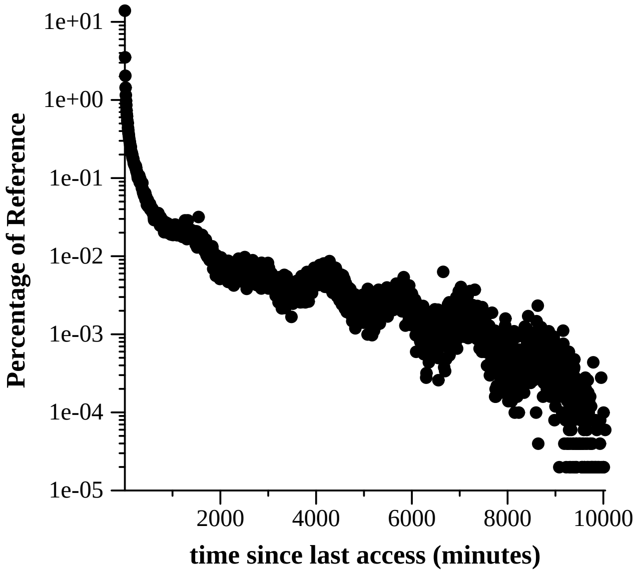



Figure 1: Percentage of references to documents whose last accesses are $t$ minutes ago, for $t$ from 5 to 10000.

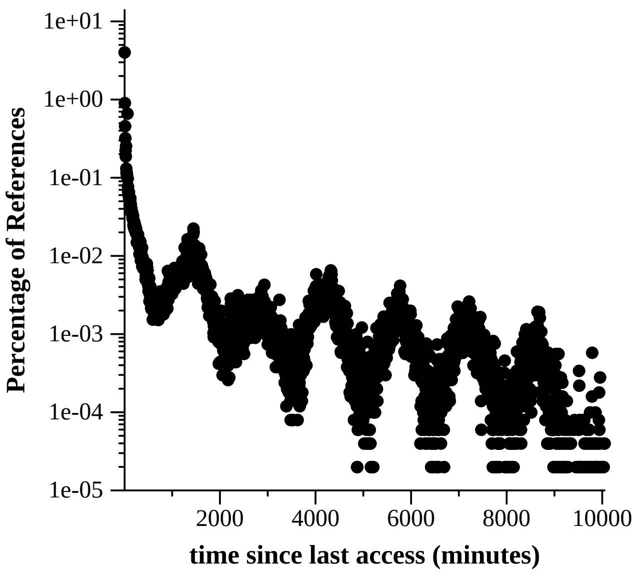



Figure 2: Percentage of references as a function of time since last access *by the same user*.

ing property we found is that all traces exhibit excellent long-term locality.

Figure 1 shows the percentage of references to a document whose last reference is $t$ minutes ago, for $t$ from 5 to 10000, in the DEC traces for the period from Sep. 12 to Sep. 18. In other words, the figure shows the probability of a document being accessed again as a function of the time since the last access to this document. The graphs for other traces are similar to the one shown here. Clearly, the probability of reference drops significantly as the time since last reference increases (note that the y-axis is in logarithmic scale), with occasional spikes around multiples of 24 hours.

Figure 3 shows the *accumulative* percentage of references to documents whose last references are less than $t$ minutes ago, for the entire DEC traces from Aug. 29 to Sep. 22. The dashed curve on the graph shows the corresponding percentage of bytes referenced. In Figure 3, which uses linear scale for the y-axis, and logarithmic scale for the x-axis, we see that the curves are nearly linear. That is, the probability of a document being referenced again within $t$ minutes is proportionally to $log(t)$, indicating that the probability of re-reference to documents referenced exactly $t$ minutes ago can be modeled as $k/t$, where $k$ is a constant.

A different study [LRV97] reached very similar conclusions on a different set of traces. Indeed, it is this observation that promoted the design of the function $D(t)$ in LRV. Since the studies find similar temporal locality patterns in the Web access traces, the probability density function of $k/t$ has been used to simulate temporal locality behavior in a recent Web proxy benchmark [WPB].

There are two reasons for the good locality in Web accesses seen by the proxy. One is that each user's accesses tend to exhibit locality — figure 2 shows the probability that a document is accessed by a user $t$ minutes after the last access by the same user, for DEC traces in the period from Sep. 12 to Sep. 18 (again, the figures for other traces are similar). Clearly, each user tends to re-access recently-read documents, and re-access documents that are read on a daily basis (note the spikes around 24 hours, 48 hours, etc. in the figure). Though one might expect that browsers' caches absorb the locality among the same user's accesses seen by the proxy, the results seems to indicate that this is not necessarily the case, and users are using proxy caches as an extension to the browser cache. [LRV97] observes the same phenomenon.

The other reason is that users' interests overlap in time — comparing figures 2 and 1, we can see that for the same $t$, the percentage in figure 1 is higher than that in figure 2. This indicates that part of the locality observed by the proxy comes from the fact that the proxy sees a merged stream of accesses from many independent users, who share a certain amount of common interests. Thus, we believe the locality observed is not particular to the traces described here, but rather a common characteristic of accesses seen by proxies with a large enough user community.

To further demonstrate the effect of inter-user sharing on hit ratios, Figure 4 shows the hit ratio

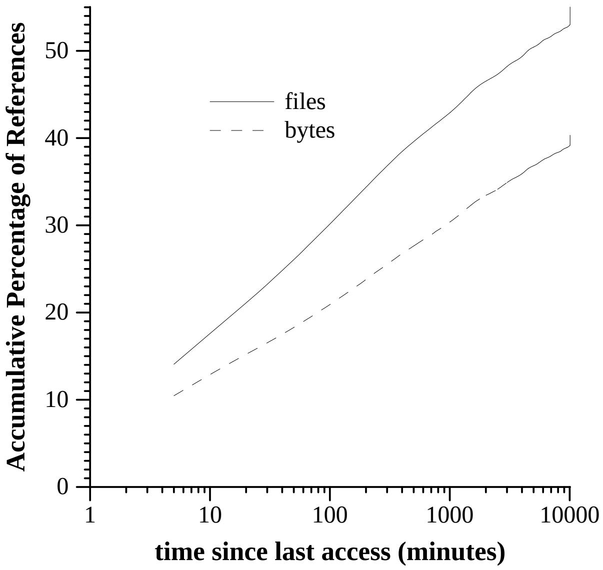



Figure 3: Percentage of references and percentage of bytes referenced to documents accessed less than $t$ minutes ago (the accumulative version of Figure 1). *Note that the y-axis is in linear scale and the x-axis is in log scale.*

and byte hit ratio as a function of the size of the user group sharing the cache. The figures are quartile graphs [Tufte], the middle curve showing the median of the hit ratios of individual groups of clients in the DEC traces, and the other four points for each group size showing the minimum, the 25% percentile, the 75% percentile, and the maximum of the hit ratios of individual groups. The median hit ratios show an almost linear increase as the group size doubles.

In the absence of cost and size concerns, LRU is the optimal online algorithm for reference streams exhibiting good locality [CD73] (strictly speaking, those conforming to the LRU-stack model). However, in the Web context, replacing a more recently used but large file can yield a higher hit ratio than replacing a less recently used but small file. Similarly, replacing a more recently used but inexpensive file may yield a lower total cost than replacing a less recently used but expensive file. Thus, we need an algorithm that combines locality, size and cost considerations in a simple, online way that does not require tuning paramters according to the particular traces, and yet maximizes the overall performance.

# 4 GreedyDual-Size Algorithm

The original GreedyDual algorithm is proposed by Young [You91b]. It is actually a range of algorithms, but we focus one particular version which is a generalization of LRU. It is concerned with the case when pages in a cache have the same size, but incur different costs to fetch from a secondary storage. The algorithm associates a value, $H$, with each cached page $p$. Initially, when a page is brought into cache, $H$ is set to be the cost of bringing the page into the cache (the cost is always non-negative). When a replacement needs to be made, the page with the lowest $H$ value, $min_H$, is replaced, and then all pages reduce their $H$ values by $min_H$. If a page is accessed, its $H$ value is restored to the cost of bringing it into the cache. Thus, the $H$ values of recently accessed pages retain a larger portion of the original cost than those of pages that have not been accessed for a long time. By reducing the $H$ values as time goes on and restoring them upon access, the algorithm integrates the locality and cost concerns in a seamless fashion.

To incorporate the difference sizes of the document, we extend the GreedyDual algorithm by setting $H$ to $cost/size$ upon an access to a document, where *cost* is the cost of bringing the document, and *size* is the size of the document in bytes. We called the extended version the GreedyDual-Size algorithm. The definition of *cost* depends on the goal of the replacement algorithm: *cost* is set to 1 if the goal is to maximize hit ratio, it is set to the downloading latency if the goal is to minimize average latency, and it is set to the network cost if the goal is to minimize the total cost.

At the first glance, GreedyDual-Size would require $k$ subtractions when a replacement is made, where $k$ is the number of documents in cache. However, a different way of recording $H$ removes these subtractions. The idea is to keep an "inflation" value L, and let all future setting of $H$ be offset by $L$. Figure 5 shows an efficient implementation of the algorithm.

Using this technique, GreedyDual-Size can be implemented by maintaining a priority queue on the documents, based on their $H$ value. Handling a hit requires $O(\log k)$ time and handling an eviction requires $O(\log k)$ time, since in both cases a single item in the queue must be updated. More efficient implementations can be designed that make the common case of handling a hit requiring only $O(1)$ time.

### Online-Optimality of GreedyDual-Size

It can be proven that GreedyDual-Size is online-optimal. For any sequence of accesses to documents with arbitrary sizes and arbitrary costs, the cost of cache misses under GreedyDual-Size is at most $k$ times that under the offline optimal replacement algorithm, where $k$ is the ratio of the cache size to the size of the smallest page. The ratio is the lowest achiev-

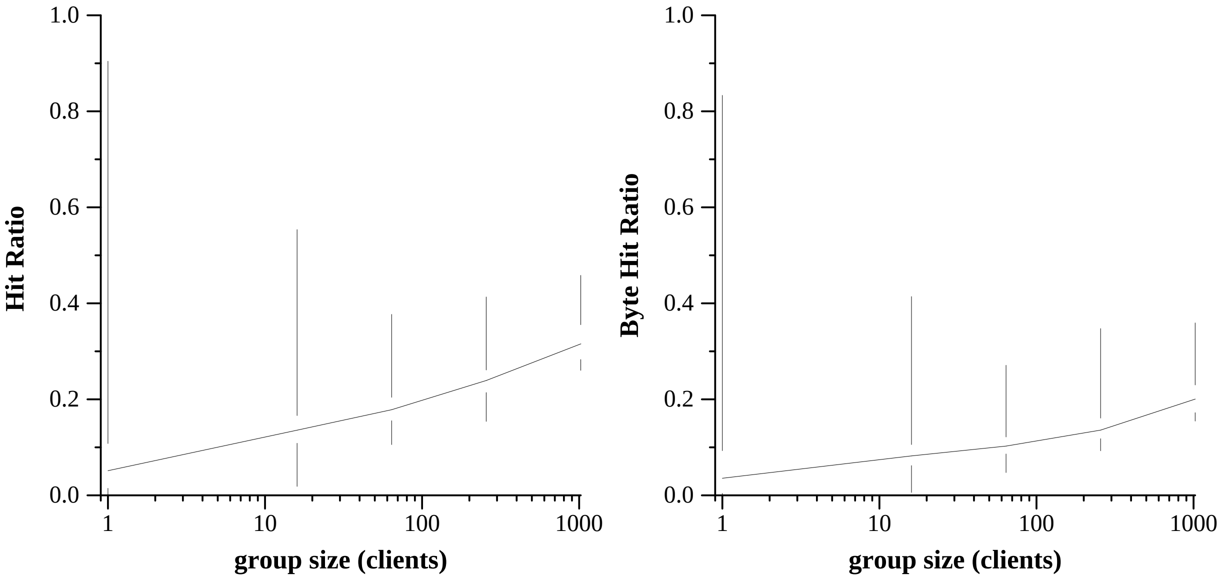



Figure 4: Hit ratio and byte hit ratio as a function of the size of the user group sharing the cache. The x-axis is in log scale.

able by any online replacement algorithm. Below is a proof of the online-optimality of GreedyDual-Size.

Neal Young proved in [You91b] that Greedy Dual for pages of uniform size is $k$-competitive. We prove here that the version which handles pages of multiple size is also $k$-competitive. (In both cases, $k$ is defined to be the ratio of the size of the cache to the size of the smallest page). The proof below is based on a proof that another algorithm called BALANCE which also solves the multi-cost uniform-size paging problem is $k$-competitive [CKPV91].

All of the above bounds are tight, since we can always assume that all pages are as small as possible and have the same cost and invoke the lower bound of $k$ on the competitive ratio for the uniform-size uniform-cost paging problem found in [ST85].

It should also be noted that the same bound can be proven for the version of the algorithm which uses $c(p)$ instead of $c(p)/s(p)$ in the description of the algorithm in Figure 5. Young has independently proven a generalization of the result below [You97]. The generalization covers the whole range of algorithms described in his original paper [You91b] instead of the particular version covered here.

Algorithm GreedyDual:
Initialize $L \leftarrow 0$.
Process each request document in turn:
The current request is for document $p$:
(1) **if** $p$ is already in memory,
(2) $H(p) \leftarrow L + c(p)/s(p)$.
(3) **if** $p$ is not in memory,
(4) **while** there is not enough room in memory for $p$,
(5) Let $L \leftarrow \min_{q \in M} H(q)$.
(6) Evict $q$ such that $H(q) = L$.
(7) Bring $p$ into memory and set $H(p) \leftarrow L + c(p)/s(p)$
**end**

Figure 5: GreedyDual Algorithm.

**Theorem 1** *GreedyDual-Size is $k$-competitive, where $k$ is the ratio of the size of the cache to the size of the smallest document.*

**Proof.** We will charge each algorithm for the documents they evict instead of the documents they bring in. The difference between the two cost measures is at most an additive constant.

Each request for a document happens in a series of steps. First the optimal algorithm serves the request. Then each of the steps of GreedyDual-Size is executed in a separate step. Each step of each request happens at a different point in time.

It is straightforward to show by induction on time that for every document $q$ in the cache

$$L \leq \min_{p \in M} H(p) \leq H(q) \leq L + \frac{c(q)}{s(q)}.$$

Let $L_{final}$ be the last value of $L$. Let $s_{min}$ denote the size of the smallest document. Let $s_{cache}$ be the total size of the cache. Note that $k = s_{cache}/s_{min}$. We will first prove that the total cost of all documents which OPT evicts is at least $s_{min} \cdot L_{final}$. Then we will show that the total cost of all documents evicted by GreedyDual-Size is at most $s_{cache} \cdot L_{final}$.

Every time $L$ increases, there is some document which GreedyDual-Size has in the cache which the optimal does not have in the cache. This is because $L$ only increases when GreedyDual-Size has exceeded the size of its cache and must evict a document. Since the optimal algorithm has already satisfied the request, it has the requested document in the cache. Since the newly requested document does not fit in GreedyDual-Size's cache, GreedyDual-Size must have some document in the cache which the optimal does not have in the cache.

Consider a period of time in which GreedyDual-Size has $p$ in its cache and the optimal does not. Such a period begins with the optimal evicting $p$ from the cache and ends when either we evict $p$ from the cache or the optimal brings $p$ back in to the cache. We will attribute any increase in $L$ which occurs during this period to the cost the optimal incurred in evicting $p$ at the beginning of the period. The cost of evicting $p$ is $c(p)$. The only thing we have to prove in establishing that the optimal cost is at least $s_{min} \cdot L_{final}$ is that $L$ increases by at most $c(p)/s(p) \leq c(p)/s_{min}$ during the period.

Let $L_1$ be the value of $L$ when the period begins. We know that at this time $H(p) \leq L_1 + c(p)/s(p)$. Furthermore, $H(p)$ does not change during this period. This is because $H(p)$ only increases when $p$ is requested. $p$ can only be requested on the last request of the period (because the period is defined to the period of time in which GreedyDual-Size has $p$ in its cache and the optimal does not). If the last request of the period is to document $p$, then the optimal brings $p$ into its cache, and hence the period ends before $H(p)$ increases. If the period ends by $p$'s eviction, $H(p)$ remains the same until $p$ is evicted. Since $H(p)$ is an upper bound for $L$, $L$ can not increase to more than $L_1 + c(p)/s(p)$ during the entire period.

Now we must establish that the total cost of all documents evicted by GreedyDual-Size is at most $s_{cache} \cdot L_{final}$. Consider an eviction that GreedyDual-Size performs. Let $p$ be the document that is evicted and let $L_1$ and $L_2$ be the values for $L$ when it is brought in and evicted from the cache, respectively. The value of $H(p)$ when $p$ is brought in to the cache is $L_1 + c(p)/s(p)$. $p$ can only be evicted if $L$ equals $H(p)$. Since $H(p)$ can only increase during the time that $p$ is in the cache, we know that $L_2 - L_1 \geq c(p)/s(p)$.

Draw an interval on the real line from $L_1$ to $L_2$ that is closed on the left end and open on the right end. Assign the interval a weight of $s(p)$. If we draw an interval for every such eviction, the cost of GreedyDual-Size is upper bounded by the sum over all intervals of their length times their weight. By definition, all intervals lie in the range from 0 to $L_{final}$.

The final observation is that the total weight of all the intervals which contain any single point on the real line is at most $s_{cache}$. Consider a point $L'$ on the line where an interval begins or ends. The total weight of the intervals that cover this point is the sum of the sizes of the documents which are in the cache when $L$ reaches a value of $L'$. Since the size of the cache is at most $s_{cache}$, the sum of the weights of the intervals which cover $L'$ is at most $s_{cache}$.

□

# 5 Performance Comparison

Using trace driven simulation, we compare the performance of GreedyDual-Size with LRU, Size, Hybrid, and LRV. Size, Hybrid, and LRV are all "champion" algorithms from previously published studies [WASAF96, LRV97, WA97]. In addition, for LRV, we first go through the whole trace to obtain the necessary parameters, thus giving it the advantage of perfect statistical information. In contrast, GreedyDual-Size takes into account cost, size and locality in a more natural manner and does not require tuning to a particular set of traces.

## 5.1 Performance Metrics

We consider five aspects of web caching benefits: hit ratio, byte hit ratio, latency reduction, hop reduction, and weighted-hop reduction. By hit ratio, we mean the number of requests that hit in the proxy cache as a percentage of total requests. By byte hit ratio, we mean the number of bytes that hit in the proxy cache as the percentage of the total number of bytes requested. By latency reduction, we mean the percentage of the sum of downloading latency for the pages that hit in cache over the sum of all downloading latencies. Hop reduction and weighted-hop reduction are used to measure the effectiveness of the algorithm at reducing network costs, as explained below.

To investigate the regulatory role that can be played by proxy caches, we model the network cost associated with each document as "hops". The "hops" value can be the number of network hops travelled by a document, to model the case when the proxy tries to reduce the overall load on Internet routers, or it can be the monetary cost associated with fetching the document, to model the case when the proxy has to pay for documents travelling through different network carriers.

We evaluate the algorithms in a situation where there is a skew in the distribution of "hops" values among the documents. The skewed distribution models the case when a particular part of the network is congested, or the proxy has to pay a different amount of money for documents travelling through different networks (for example, if the proxy is at an Internet Service Provider). In our particular simulation, we assign each Web server a hop value equal to 1 or 32 [2], so that 1/8 of the servers have hop value 32 and 7/8 of the servers have hop value 1. This simulates the scenario, for example, that 1/8 of the web servers contacted are located in Asia, or can only be reached through an expensive or congested link.

Associated with the "hop" value are two metrics: hop reduction and weighted-hop reduction. Hop reduction is the ratio between the total number of the hops of cache hits and the total number of the hops of all accesses; weighted-hop reduction is the corresponding ratio for the total number of hops times "packet savings" on cache hits. A cache hit's packet saving is $2 + file\_size/536$, as an estimate of the actual number of network packets required if the request is a cache miss (1 packet for the request, 1 packet for the reply, and $size/536$ for extra data packets, assuming a 536-byte TCP segment size).

For each trace, we first calculate the benefit obtained if the cache size is infinite. The values for all traces are shown in Table 1. In the table, BU-272 and BU-B19 are two sets of traces from Boston University [CBC95], VT-BL, VT-C, VT-G, VT-U are four sets of traces from Virginia Tech [WASAF96], DEC-U1:8/29-9/4 through DEC-U1:9/19-9/22 are the requests made by users 0-512 (user group 1) for each week in the three and half week period, and DEC-U2:8/29-9/4 through DEC-U2:9/19-9/22 are the traces for users 1024-2048 (user group 2). We experimented with other subsets of DEC traces and the results are quite similar to those obtained from these subsets.

[2] These numbers are chosen partly because, at one time, the maximum number of hops along a packet's route was 32.

Below, we divide our results into three subsections. In Section 5.2, we measure the hit rate and byte hit rate of different algorithms. In Section 5.3 we compare the latency reduction. In Section 5.4 we compare the hop reduction and weighted hop reduction. The corresponding value under the infinite cache are listed in Table 1. Since these simulations assume limited cache storage, their ratios cannot be higher than the infinite cache ratios.

The cache sizes investigated in the simulation were chosen by taking a fixed percentage of the total sizes of all distinct documents requested in the sequence. The percentages are 0.05%, 0.5%, 5%, 10% and 20%. For example, for trace DEC-U1:8/29-9/4, which includes the requests made by users 0-512 for the week of 8/29 to 9/4 and has a total data set size of 9.21GB, the cache sizes experimented are 4.6MB, 46.1MB, 461MB, 921MB and 1.84GB.

Due to space limitation, we organize the traces into four groups: Boston University traces, Virginia Tech traces, DEC-U1 traces, and DEC-U2 traces, and present the averaged results per trace group. The results for individual traces are similar.

## 5.2 Hit Rate and Byte Hit Rate

We introduce two versions of the GreedyDual-Size algorithm, GD-Size(1) and GD-Size(packets). GD-Size(1) sets the cost for each document to 1, and GD-Size(packets) sets the cost for each document to $2+size/536$ (that is, the estimated number of network packets sent and received if a miss to the document happens). In other words, GD-Size(1) tries to minimize miss ratio, and GD-Size(packets) tries to minimize the network traffic resulting from the misses.

Figure 6(a) shows the average hit ratio of the four groups of traces under LRU, Size, LRV, GD-Size(1), and GD-Size(packets). The graphs from left to right show the results for Boston University traces, Virginia Tech traces, DEC-U1 traces and DEC-U2 traces, respectively. Figure 6(b) is a simplified version of 6(a) showing only the curves of LRU and GD-Size(1), highlighting the differences between the two. Graph (c) shows the average byte hit ratio for the four trace groups under the different algorithms.

The results show that clearly, GD-Size(1) achieves the best hit ratio among all algorithms across traces and cache sizes. It approaches the maximal achievable hit ratio very fast, being able to achieve over 95% of the maximal hit ratio when the cache size is only 5% of the total data set size. It performs particularly well for small caches, suggesting that it would

| Trace | Clients | Total Requests | Total GBytes | Hit Rate | Byte HR | Reduced Latency | Reduced Hops | Reduced WeightedHops |
|---|---|---|---|---|---|---|---|---|
| BU-272 | 5 | 17007 | 0.39 | 0.25 | 0.15 | 0.13 | 0.16 | 0.09 |
| BU-B19 | 32 | 118104 | 1.59 | 0.47 | 0.27 | 0.20 | 0.48 | 0.25 |
| VT-BL | 59 | 53844 | 0.674 | 0.43 | 0.33 | - | 0.35 | 0.16 |
| VT-C | 26 | 11250 | 0.159 | 0.45 | 0.38 | - | 0.33 | 0.15 |
| VT-G | 26 | 47802 | 0.630 | 0.50 | 0.30 | - | 0.49 | 0.31 |
| VT-U | 74 | 164160 | 2.30 | 0.46 | 0.33 | - | 0.40 | 0.25 |
| DEC-U1:8/29-9/4 | 512 | 633881 | 9.21 | 0.42 | 0.35 | 0.24 | 0.34 | 0.25 |
| DEC-U1:9/5-9/11 | 512 | 691211 | 9.32 | 0.40 | 0.31 | 0.23 | 0.32 | 0.23 |
| DEC-U1:9/12-9/18 | 512 | 658166 | 9.23 | 0.39 | 0.31 | 0.19 | 0.39 | 0.32 |
| DEC-U1:9/19-9/22 | 512 | 280087 | 3.86 | 0.38 | 0.31 | 0.16 | 0.25 | 0.21 |
| DEC-U2:8/29-9/4 | 1024 | 455858 | 5.57 | 0.33 | 0.22 | 0.20 | 0.27 | 0.19 |
| DEC-U2:9/5-9/11 | 1024 | 428719 | 5.13 | 0.30 | 0.21 | 0.18 | 0.25 | 0.16 |
| DEC-U2:9/12-9/18 | 1024 | 408503 | 4.94 | 0.29 | 0.19 | 0.15 | 0.24 | 0.17 |
| DEC-U2:9/19-9/22 | 1024 | 170397 | 2.00 | 0.26 | 0.19 | 0.15 | 0.17 | 0.11 |

Table 1: Benefits under a cache of infinite size for each trace, measured as hit ratio, byte hit ratio, latency reduction, hop reduction, and weighted-hop reduction.

be a good replacement algorithm for main memory caching of web pages.

However, Figure 6(c) reveals that GD-Size(1) achieves its high hit ratio at the price of lower byte hit ratio. This is because GD-Size(1) considers the saving for each cache hit as 1, regardless of the size of document. GD-Size(packets), on the other hand, achieves the overall highest byte hit ratio and the second highest hit ratio (only moderately lower than GD-Size(1)). GD-Size(packets) seeks to minimize (estimated) network traffic, in which both hit ratio and byte hit ratio play a role.

For the Virginia Tech traces, LRV outperforms GD-Size(packets) in terms of hit ratio and byte hit ratio. This is due to the fact that those traces have significant skews in the probability of references to different sized files, and LRV knows the distribution before-hand and includes it in the calculation. However, for all other traces where the skew is less significant, LRV performs worse than GD-Size(packets) in terms of both hit ratio and byte hit ratio, despite its heavy parameterization and foreknowledge.

LRU performs better than SIZE in terms of hit ratio when the cache size is small (less or equal than 5% of the total date set size), but performs slightly worse when the cache size is large. The relative comparison of LRU and Size differs from the results in [WASAF96], but agrees with those in [LRV97].

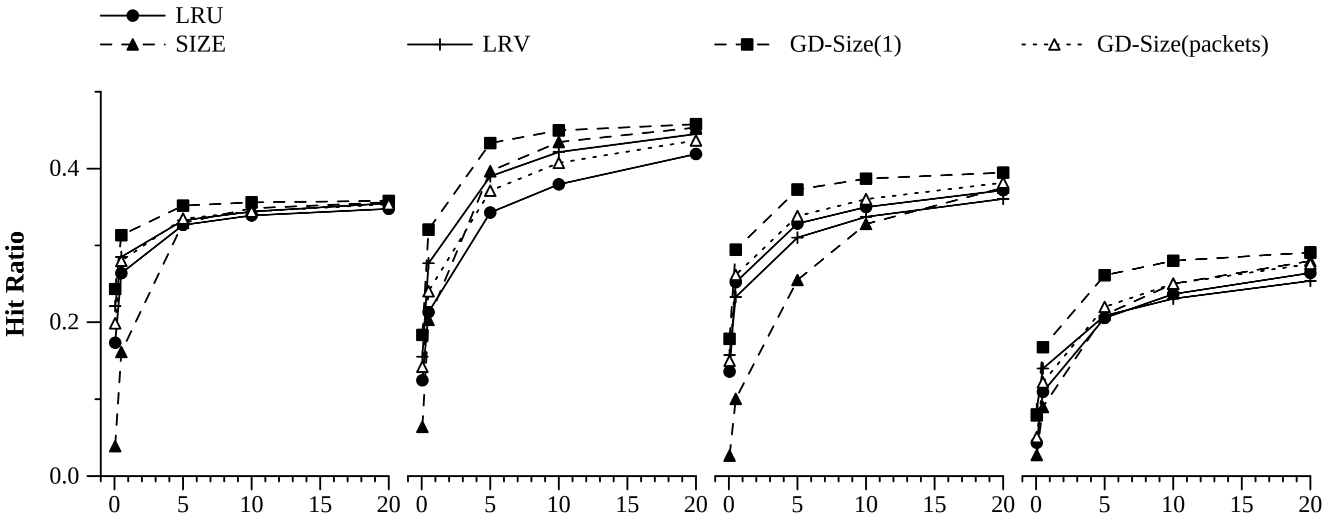
In summary, for proxy designers that seek to maximize hit ratio, GD-Size(1) is the appropriate algorithm. If both high hit ratio and high byte hit ratio are desired, GD-Size(packets) is the appropriate algorithm.

## 5.3 Reduced Latency

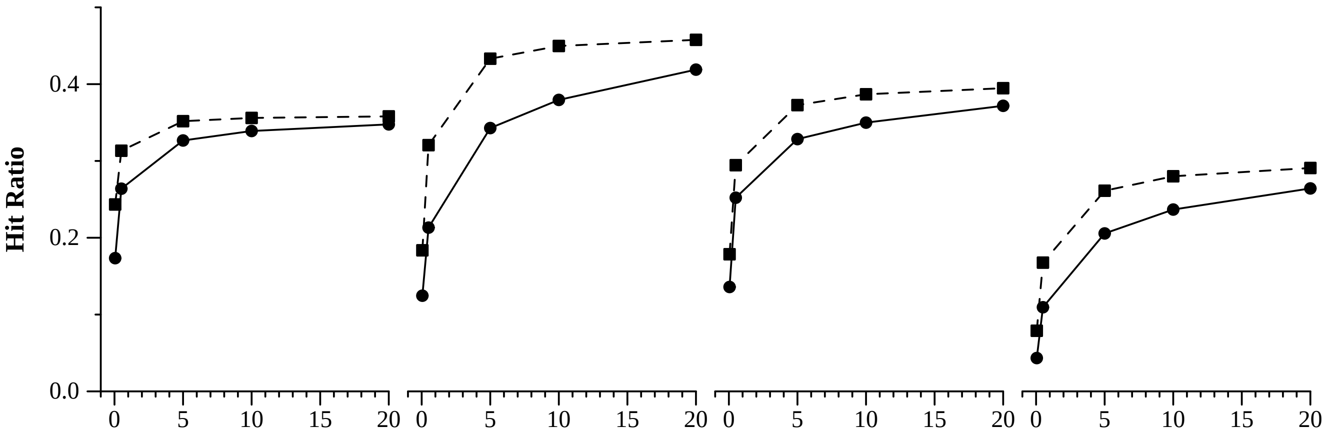
Another major concern for proxies is to reduce the latency of HTTP requests through caching, as numerous studies have shown that the waiting time has become the primary concern of Web users. One study [WA97] introduced a proxy replacement algorithm called *Hybrid*, which takes into account the different latencies incurred to load different web pages, and attempts to minimize the average latency. The study [WA97] further showed that in general the algorithm has a lower average latency than LRU, LFU and SIZE.

We also designed two versions of GreedyDual-Size that take latency into account. One, called GD-Size(latency), sets the cost of a document to the latency that was required to download the document. The other, called GD-Size(avg_latency), sets the cost to the estimated download latency of a document, using the same method of estimating latency as in Hybrid [WA97].

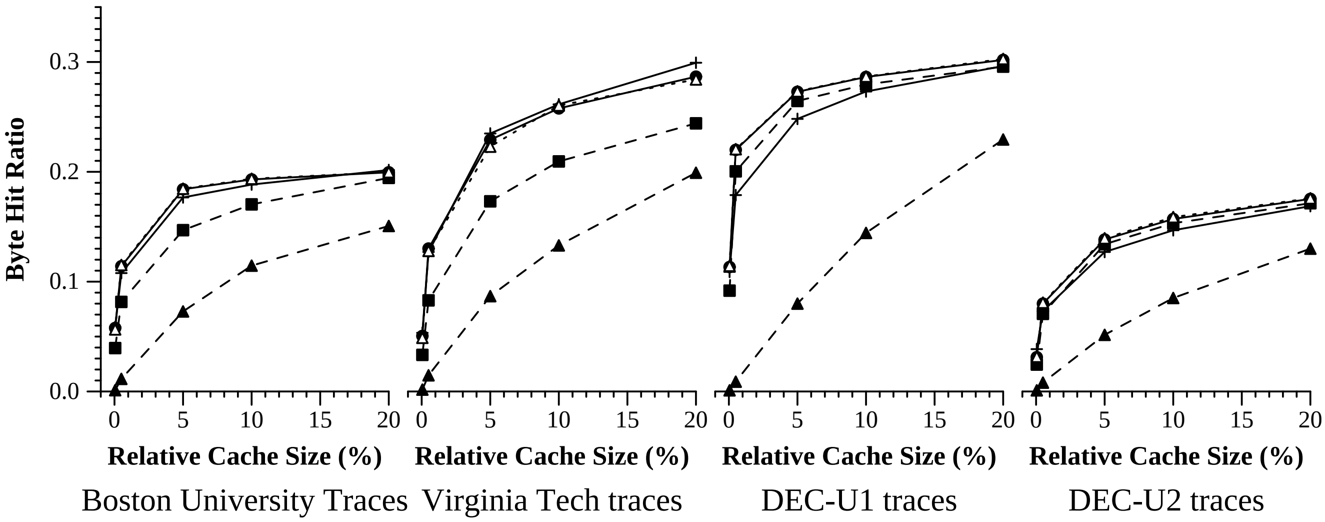
Figure 7(a) shows the latency reductions for LRU, Hybrid, GD-Size(1), GD-Size(latency) and GD-Size(avg_latency). The graphs, from left to right, show the results for Boston University traces, DEC-U1 traces and DEC-U2 traces. The figure unfortunately does not include results for Virginia Tech traces because they do not have latency information for each HTTP request. Clearly, GD-Size(1) performs the best, yielding the highest latency re-




(a) Hit ratios of LRU, Size, LRV, GD-Size(1) and GD-Size(packets) for each trace group.




(b) A simplified version of (a) showing only the curves for LRU and GD-Size(1).




(c) Byte hit ratios of LRU, Size, LRV, GD-Size(1), and GD-Size(packets) for each trace group.

Figure 6: Hit ratio and byte hit ratio comparison of the algorithms.

duction. GD-Size(latency) and GD-Size(packets) finish the second, with LRU following close behind. GD-Size(avg_latency) performs badly for small cache sizes, but performs very well for relatively large cache sizes. Finally, Hybrid performs the worst.

Examination of the results shows that the reason for Hybrid's poor performance is its low hit ratio. In the Boston University traces, Hybrid's hit ratio is much lower than LRU's for cache sizes $\leq 5\%$ of the total data set sizes, and only slightly higher for larger cache sizes. For all DEC traces, Hybrid's hit ratio is much lower than LRU's, under all cache sizes. Hybrid has a low hit ratio because it does not consider how recently a document has been accessed during replacement.

Since [WA97] reports that Hybrid performs well, our results here seem to suggest that Hybrid's performance is perhaps trace-dependent. In our simulation of Hybrid we used the same constants in [WA97], without tuning them to our traces. Unfortunately we were not able to obtain the traces used in [WA97].

It is a surprise to us that GD-Size(1), which does not take latency into account, performs better than GD-Size(latency) and GD-Size(avg_latency). Detailed examination of the traces shows that the latency of loading the same document varies significantly. In fact, for each of the DEC traces, variance among latencies of the same document ranges from 5% to over 500%, with an average around 71%. Thus, a document that was considered cheap (taking less time to download) may turn out expensive at the next miss, while a document that was considered expensive may actually take less time to download. The best bet for the replacement algorithm, it seems, is to maximize hit ratio.

In summary, GD-Size(1) is the best algorithm to reduce average latency. The high variance among loading latencies for the same document reduces the effectiveness of latency-conscious algorithms.

### 5.4 Network Costs

To incorporate network cost considerations, GD-Size(hops) sets the cost of each document to the hop value associated with the Web server of the document, and GD-Size(weightedhops) sets the cost to be $hops \times (2 + file\_size/536)$. Figure 7(b) and 7(c) show the hop reduction and weighted-hop reduction for LRU, GD-Size(1), GD-Size(hops), and GD-Size(weightedhops).

The results show that algorithms that consider network costs do perform better than algorithms that are oblivious to them. The results here are different from the latency results because the network cost associated with a document does not change during our simulation. The results also show that the specifically designed algorithms achieve their effect. For hop reduction, GD-Size(hops) performs the best, and for weighted-hop reduction, GD-Size(weightedhops) performs the best. This shows that GreedyDual-Size not only can combine cost concerns nicely with size and locality, but is also very flexible and can accommodate a variety of performance goals.

Thus, we recommend GD-Size(hops) as the replacement algorithm for the regulatory role of proxy caches. If the network cost is proportional to the number of bytes or packets, then GD-Size(weightedhops) is the appropriate algorithm.

### 5.5 Summary

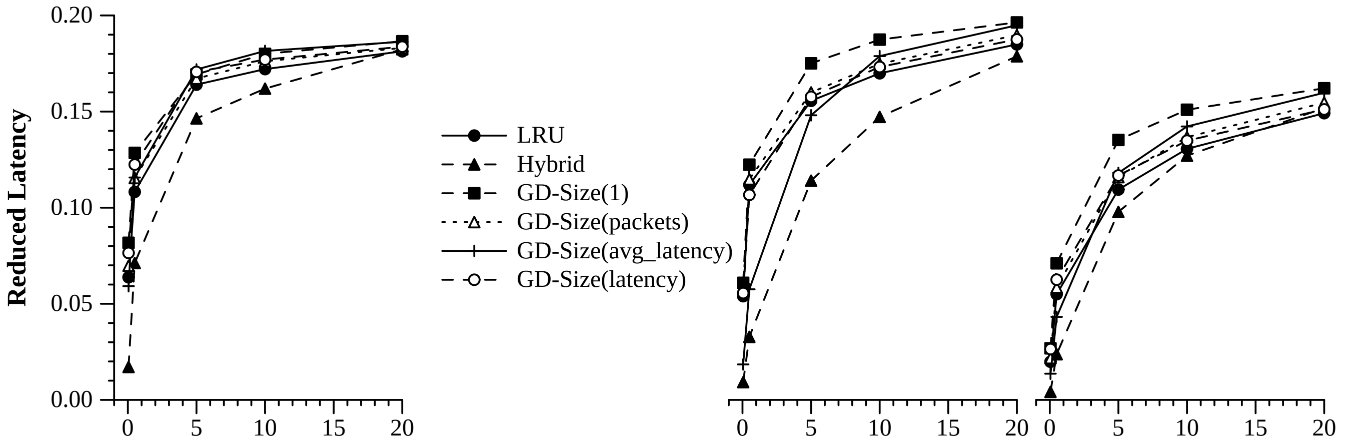
Based on the above results, we have the following recommendation. If the proxy wants high hit ratio or low average latency, GD-Size(1) is the appropriate algorithm. If the proxy desires high byte hit ratio as well, then GD-Size(packets) achieves a good balance among the different goals. If the documents have associated network or monetary costs that do not change over time, or change slowly over time, then GD-Size(hops) or GD-Size(weightedhops) is the appropriate algorithm. Finally, in the case of main memory caching of web documents, GD-Size(1) should be used because of its superior performance under small cache sizes.

## 6 Conclusion

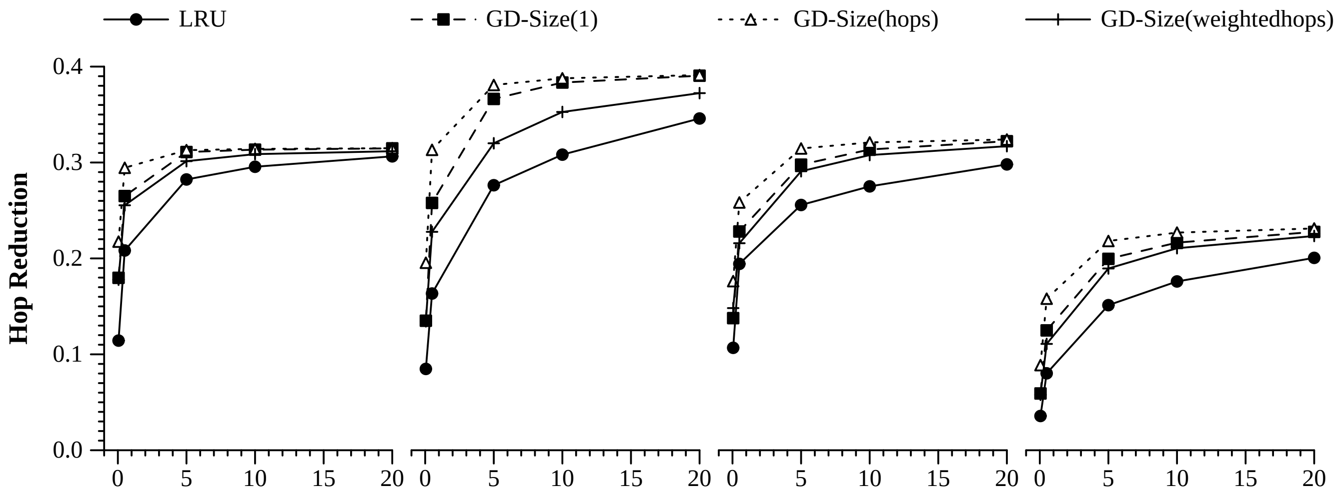
This paper introduces a simple web cache replacement algorithm: GreedyDual-Size, and shows that it outperforms existing replacement algorithms in many performance aspects, including hit ratios, latency reduction, and network cost reduction. GreedyDual-Size combines locality, cost and size considerations in a unified way without using any weighting function or parameter. It is simple to implement and accommodates a variety of performance goals. Through trace-driven simulations, we identify the cost definitions for GreedyDual-Size that maximize different performance gains. GreedyDual-Size can also be applied to main memory caching of Web documents to further improve performance.

The GreedyDual-Size algorithms shown so far can only optimize one performance measure at a time. We are looking into how to adjust the algorithm when the goal is to optimize more than one performance measures (for example, both hit ratio and byte hit ratio). We also plan to study the integration of hint-based prefetching with the cache replacement algorithm.

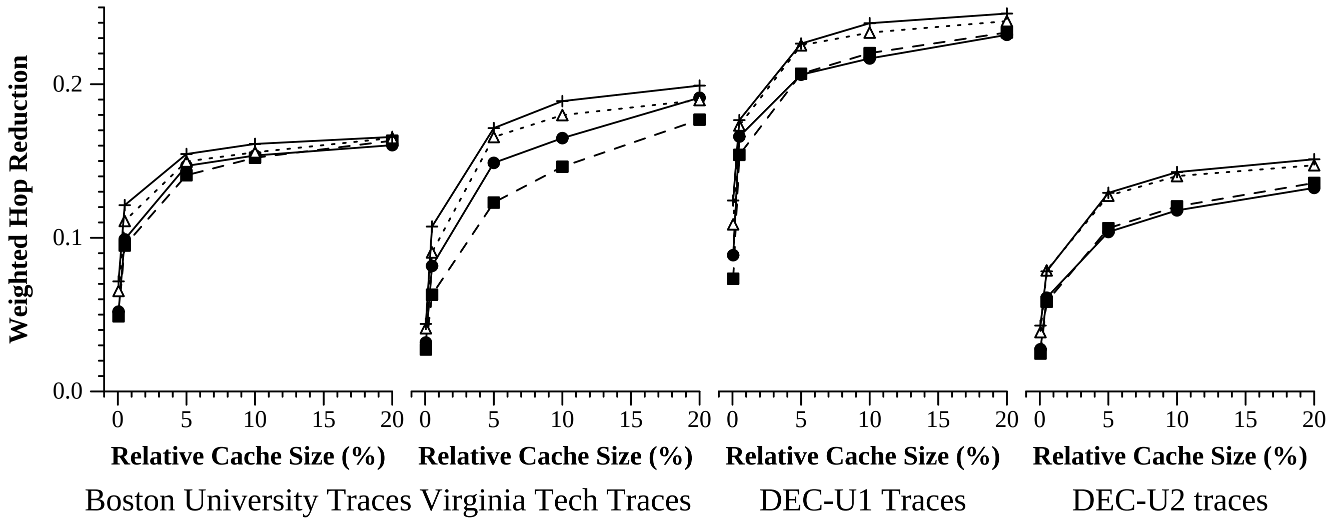
Finally, we have shown that if an appropriate




(a) Latency reduction under LRU, Hybrid, GD-Size(1), GD-Size(packets), GD-Size(latency) and GD-Size(avg_latency) for Boston University, DEC-U1, and DEC-U2 traces.




(b) Hop reduction under LRU, GD-Size(1), GD-Size(hops) and GD-Size(weightedhops).




(c) Weighted hop reduction under LRU, GD-Size(1), GD-Size(hops) and GD-Size(weightedhops).

Figure 7: Latency, hops, and weighted hops reductions under various algorithms.

network cost can be associated with a document, GreedyDual-Size algorithm can be used to adjust the caching of different documents to affect the Web traffic. In other words, if proxy caches use the GreedyDual-Size algorithm, and they can be informed of the congestion on the network, the caches can cooperate to reduce the traffic over the congested links. However, how to detect congested path on the network and how to assign appropriate cost values for the affected documents are topics beyond the scope of this paper, and remain our future work.


## Acknowledgement

The research is not possible without the support from people who make their proxy traces available. Sandy Irani is supported in part by NSF Grant CCR-9309456. Our shepherd, Carl Staelin, contributed the accumulative percentage graphs and the quartile graphs and greatly improved the paper. Finally, the anonymous referees provided very helpful comments.


## References


[ASAWF95] M. Abrams, C.R. Standbridge, G.Abdulla, S. Williams and E.A. Fox. Caching Proxies: Limitations and Potentials. WWW-4, Boston Conference, December, 1995.

[Bel66] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[CD73] G. Coffman, Jr., Edward and Peter J. Denning, Operating Systems Theory, *Prentice-Hall, Inc.* 1973.

[CKPV91] M. Chrobak, H. Karloff, T. H. Payne and S. Vishwanathan. New results on server problems. newblock *SIAM Journal on Discrete Mathematics*, 4:172–181, 1991.

[DEC96] Digital Equipment Cooperation, Digital's Web Proxy Traces ftp://ftp.digital.com/pub/DEC/traces/proxy /webtraces.html.

[FKIP96] A. Feldman, A. Karlin, S. Irani, S. Phillips. Private Communication.

[Ho97] Hosseini, Saied, Private Communication.

[LC97] Chengjie Liu, Pei Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the 1997 International Conferences on Distributed Computing Systems*, May, 1997.

[Ir97] S. Irani. Page replacement with multi-size pages and applications to web caching. In the *Proceedings for the 29th Symposium on the Theory of Computing*, 1997, pages 701-710.

[LRV97] P. Lorenzetti, L. Rizzo and L. Vicisano. Replacement Policies for a Proxy Cache. http://www.iet.unipi.it/ luigi/research.html.

[CBC95] Carlos R. Cunba, Azer Bestavros, Mark E. Crovella Characteristics of WWW Client-based Traces BU-CS-96-010, Boston University.

[LM96] Paul Leach and Jeff Mogul. The Hit Metering Protocol. Manuscript.

[HT97] IETF The HTTP 1.1 Protocol - Draft. http://www.ietf.org.

[ST85] D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[Tufte] Edward Tufte The Visual Display of Quantitative Information. Graphics Printers, Feburary 1992.

[W3C] The Notification Protocol. http://www.w3c.org.

[WASAF96] S. Williams, M. Abrams, C.R. Standbridge, G.Abdulla and E.A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the ACM Sigcomm96*, August, 1996, Stanford University.

[WA97] R. Wooster and M. Abrams. Proxy Caching the Estimates Page Load Delays. In the *6th International World Wide Web Conference,* April 7-11, 1997, Santa Clara, CA. http://www6.nttlabs.com/HyperNews/get/ PAPER250.html.

[WPB] Jussara Almeida and Pei Cao. The Wisconsin Proxy Benchmark (WPB). http://www.cs.wisc.edu/∼cao/wpb1.0.html.

[You91b] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*,June 1994, vol. 11,(no.6):525-41. Rewritten version of "Online caching as cache size varies", in The 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, 241-250, 1991.

[You97] N. Young. Online file caching. To appear in *the Proceedings for the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.