Replacement policies for a proxy cache

Luigi Rizzo
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
via Diotisalvi 2 – 56126 Pisa (Italy)
email: l.rizzo@iet.unipi.it

Lorenzo Vicisano
Department of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
email: l.vicisano@cs.ucl.ac.uk

Abstract

In this paper we analyse access traces to a Web proxy, looking at statistical parameters to be used in the design of a replacement policy for documents held in the cache. In the first part of the paper we present a number of properties of the lifetime and statistics of access to documents, derived from two large trace sets coming from very different proxies and spanning over time intervals of up to five months. In the second part of the paper we propose a novel replacement policy, called LRV, which selects for replacement the document with the Lowest Relative Value among those in cache. The value of a document is computed adaptively basing on information readily available to the proxy server. The algorithm has no hardwired constants, and the computations associated with the replacement policy require only a small constant time. We show how LRV outperforms LRU and other policies, and can significantly improve the performance of the cache, especially a for small one.
Keywords: Web, caching, replacement policies, communication networks

1 Introduction

The caching of Web documents is widely used to reduce both latency and network traffic in accessing data. Browsers generally implement a first-level cache, using some amount of memory and disk space to store frequently accessed documents. Being used by a single client, the browser's cache is mainly useful to store images (backgrounds, icons, etc.) which occur frequently in a set of related documents. A second-level cache is provided by caching proxy servers (*proxies*), which retrieve documents from the original site (or another proxy) on behalf of the client. Proxies serve a large set of clients and concentrate different sources of traffic, hence they are able to exploit not only the temporal locality, but also the geographical locality of the requests. In addition to this, the aggregation of requests coming from different clients allows proxies to have a large sample of references, giving useful information on documents' popularity and for implementing caching strategies. Several proxy caches have been developed in recent years. Among them, `cern_httpd` [9], `harvest` [4], and its successor `squid` [11] are popular programs which are available in source format. The latter two use a number of ingenious solutions to improve performance, and introduce the concept of cooperating proxies allowing to setup distributed scalable proxy systems.

As proxies have finite storage capacity, it is eventually necessary to replace less useful documents with more useful ones. A replacement policy is, in fact, required in every caching technique. Cache memories, placed between processors and main memory, are well known and widely studied examples from computer architecture. Other popular examples are in the fields of operating systems and distributed file systems. Web caching started borrowing the results of the studies in these fields and applying the same techniques, but recently, researchers begun to study this issue as a separate problem [3, 7, 10, 12, 13]. The caching of Web documents has actually some peculiarities which justify the development of its own techniques. The main characteristics differentiating the Web case from its predecessors are *variable object size* and *larger* *time scales*, which lead to the need/possibility of employing more sophisticated/expensive replacement algorithms than LRU, that is the one commonly used in most other contexts.

1.1 Related work

Some useful hints on Web caching algorithms can be derived from similar results in the field of file-system caching. Commonly used policies include discarding the Least Recently Used (LRU) among the objects present in cache, the Least Frequently Used (LFU), policies based on the SIZE of objects —which discard the largest objects—, or combinations of them. One of the main weaknesses of LRU (see [6]) is that the cache can be flooded by documents that are referenced only once, flushing out documents with higher probability of being reused. This situation is very likely to occur in Web caches, where references to object accessed only once account for a large fraction (2/3 in our traces) of the total. The authors of [6] observed that the probability of an object being referenced again quickly grows after the second reference, as we did in the case of Web caching (see 3). That accounts for a good behaviour of LFU policy. Nevertheless, LFU alone prevents 'dead' document, with large reference counts from being purged. This causes the so called *cache pollution* phenomenon and yields a reduction of the effective cache size. An aging policy is often used to cope with this problem.

The problem of determining an efficient replacement algorithm for a proxy cache has been studied in [12, 13]. The authors present a taxonomy of removal policies, basing on the observation of traces corresponding to five different workloads. They analyse the use of hierarchical keys to keep documents sorted in the cache, and conclude that document size outperforms all other possible keys, as far as HR (cache hit rate) is concerned, while, looking at BHR (bit hit rate), the key based on the number of previous access (LFU) is the best performing and that based on documents' size is the worse. However, they use relatively short periods of observation, a limited number of different clients, and the cache size (10% of the total size of accessed documents) used in the experiments probably exceeds by far the set of live documents[1]. In policies based on the size of a document, there is little or no chance that small documents can be discarded; as a consequence, in the long term, the cache fills up with small, old documents, and its performance decays. A similar observation holds for LFU. The phenomenon is only visible when such garbage brings the useful space in the cache well below the size of live documents, while the parameters used in [12, 13] cannot evidence this phenomenon. As also noted by the authors, a problem with hierarchical keys is that it is difficult to sort the keys by importance, and a key can be effectively used only if all the previous ones give rise to frequent ties. As a consequence, often only the first key is effective.

In a recent study [3], Cao and Irani have presented an algorithm called *GredyDual-Size*. The basic *GredyDual* algorithm sorts documents basing on their measured retrieval cost H. The document with the minimum value for H is the candidate for replacement, and when a replacement occurs all the documents get aged by the the current value, H, of the purged document. In GredyDual-Size, parameter H is slightly different being set to the ratio (retrieval cost/size) of the document, to account for the variable size of Web objects. The performance of Greedy-Dual-Size and other algorithms is evaluated in [3] on some traces, including a small subset of the DEC trace also used for our evaluation, showing their algorithm to performs slightly better than a former, non fully adaptive, version of LRV [8]. Compared to LRV, this performance comes at additional costs for taking replacement decisions. In fact, updating the cache state at each reference has cost $O(\log n)$, requiring a list search. On the contrary, as we will see, replacement decisions in LRV only require constant time[2].

[1] For a formal definition of 'live documents' see section 2.

[2] This is partly because LRV does not use the retrieval cost as a parameter, since we believe it can bias the cache's behaviour against fast servers.

1.2 Contribution of this paper

In this paper[3] we develop a novel replacement strategy, which we call LRV, explicitly intended for the World Wide Web, and based on maximising an objective function for the whole cache. The objective function is computed using a cost/benefit model that determines the relative value of each document in the cache, so as to allow the replacement algorithm to select the document with the Lowest Relative Value. The value of each document is obtained on the basis of statistical parameters collected by the server, and converted on-line into the coefficients used in the evaluation of document importance. All the computations are performed at negligible cost, allowing replacement decisions to be taken in $O(1)$ time rather than the $O(\log n)$ time required by some advanced algorithms.

The paper is organised as follows. Section 2 defines the problem, and introduces the cost/benefit model our work is based on. Section 3 analyses the probabilistic parameters which influence the value of each document, discussing how they affect already-known replacement algorithms. Section 4 presents the LRV algorithm together with a discussion of its implementation. The performance of LRV, compared to other algorithms, is finally presented basing on real traces and for a number of different cache sizes.

2 Problem's definition

Proxies aim to improve the performance offered by the World Wide Web, providing end users with faster access to its resources. A number of performance metrics can be identified for caching in the Web, which can be divided in two main classes: *user perceived* performance and *network perceived* performance. Web users perceive the goodness of the system as the time it takes to retrieve a document; that depends on the time it takes to get the document from the local cache, the time it takes to transfer the document from the original site and on whether the document is present or not in the local cache. From the network perspective, the main goal is to contain the traffic on its link, avoiding the unnecessary use of resources.

Both the user-perceived access time and the network traffic can be hardly quantified in terms of cache performance indexes. However, they are strongly dependent on two objective performance parameters, the *hit rate* (*HR*) and *byte hit rate* (*BHR*), which reasonably describe the effectiveness of a cache. HR and BHR indicate the fraction of documents and bytes, respectively, which are served from the cache instead of requesting them from the network. BHR is a direct measure of the savings in network traffic measured at the cache from/to the outside, so it also strongly influences the response time in serving documents. BHR is a significant performance index: in fact, under some realistic assumptions (see Appendix A), the *speedup* in retrieving documents, s, achievable with a cache can be expressed as

$$s = \frac{1}{1 - BHR}$$

For these reasons, in the rest of this paper we will use BHR as the main performance index, but we will also look at HR, being useful for comparing our results with previous works which mainly concentrate on HR. In evaluating these performance metrics we will discard those documents which can be identified as *uncacheable* a-priori, such those generated as a program output, or marked as uncacheable by the server.

With reference to a given time t_0, we call *history* the history of accesses to a proxy made before t_0. Accesses issued after t_0 are called *future history* or simply *future*. The history is known to proxy servers, which can keep logs recording past requests including information on the URL, size, type, requestor, transfer time for a document. Obviously, the future is not known to a proxy at runtime, so we will use this definition only to classify documents requested to the proxy:

- *Accessed* documents are all documents whose URLs appear in the history.

[3] An earlier version of this paper, describing the LRV algorithm, has been available as a DEIT Technical Report [8] since 1996, and some papers in the literature refer to that earlier version of LRV. In this paper we have extended our study to include the DEC trace set, and have modified the LRV algorithm to make it fully adaptive by removing hardwired constants in the computation of $P(t)$ (see section 3.2).

- *Dead* documents are accessed documents which do not appear in the future history. Document die because they are deleted at the source, change their content (thus effectively becoming new documents), or simply because nobody accesses them anymore.
- *Live* documents are accessed documents which also appear in the future history.

Let D_i be the set of documents accessed at least i times, and $\| D_i \|$ the size of the set. A parameter which we will often use is

$$P(i) = \frac{\| D_{i+1} \|}{\| D_i \|}$$

corresponding to the probability that a document is accessed again after the $i-th$ access. $P_{|i=1}(i)$ is a direct indication of the percentage of documents for which caching is useful at all.

In order to increase the hit rate, a proxy might try to prefetch documents, anticipating clients' requests, so that even the first request for a given document is served by the cache. In principle, this approach might have no influence on the total amount of network traffic, bring the hit rate to 100%, and reduce the latency experienced by clients in accessing documents. In practice, anticipating clients' requests is non trivial, prone to errors, and it usually has high costs in terms of additional network traffic and storage use. In [10] Padmanabhan and Mogul propose a prefetching scheme in which clients and servers cooperate, achieving a substantial reduction in latency at the cost of an increase in the network traffic. However their algorithm requires modifications both on the server and on the client and some minor additions to the HTTP protocol. In this paper we will not consider prefetching proxies.

Consistency problems always arise when using caches. Consistency in Web caching is handled using some support provided by the HTTP protocol [2] and some heuristics implemented in the caches [9, 4, 11]. An analysis of consistency issues can be found in [7]. In the following we do not deal with the consistency problem as it is orthogonal with respect to replacement policies: we view the Web as a read-only system and consider modified objects as brand new ones.

2.1 A cost/benefit model

Given the history of accesses to a proxy, we can compute HR_{max} and BHR_{max} —respectively the maximum hit rate and the maximum byte hit rate achievable, for a certain access pattern. Such hit rates can be easily achieved by a cache with sufficient storage to hold all the accessed documents (but this would probably require an unbounded amount of storage). In principle, given sufficient disk space, it suffices to hold in cache all the live documents: the replacement policy would purge dead documents and keep all live documents; clearly, such a decision could only be taken if the future history of accesses were known. Such a caching policy would still be able to achieve both BHR_{max} and HR_{max}.

If the cache is not large enough to store all live documents, optimal policies can still be defined as those maximising HR or BHR, and can be used as a reference in evaluating the goodness of a cache replacement algorithm. Note that even if the future history were known – which it is not – an optimal policy can be computationally too expensive to implement.

A suboptimal approach for replacing documents aims at optimising a given objective function, that expresses quantitatively how valuable the whole cache contents is, according to some metric. If we had an additive function to compute the value of each single document in the cache, then the whole cache value could be obtained adding the individual document values. Replacing the document with the *Lowest Relative Value* (its value over the cache space occupied) would then maximise the whole cache content value. In the following we will define such a function and will show how to compute it quickly.

Purging a documents from the cache has a benefit and a cost. The benefit B essentially comes from the amount of space freed, which is roughly proportional to the size of the document plus its metadata, possibly rounded to a multiple of the file-system's block-size.

The cost can be expressed as the cost, C, of fetching the document from the original site, multiplied by P_r, the probability that the document is accessed again in the future. Several different metrics are commonly used to compute C:

	duration	references	bytes	URLs	clients	servers	% single ref.	BHR_{max}	HR_{max}
UNIPI	5 months	1.3 M	7 GB	0.45 M	1000	20 K	0.675	0.505	0.657
DEC	25 days	22 M	66 GB	4.6 M	17000	140 K	0.624	0.669	0.789

Table 1: Major characteristics of the trace sets used in our study: duration, total number of references, total number of bytes requested, number of distinct URLs, number of distinct client hosts, number of distinct servers referenced, percentage of URLs referenced only once, BHR_{max} and HR_{max}.

connections Each retrieval of a document has the same cost, so we can assume $C = 1$. This is only appropriate when the cost of setting up a communication with the server dominates over other costs. However, it tends to overestimate small documents, and it is not a very realistic metric in many cases.

time In this case the cost of a retrieval is proportional to the time needed to fetch the document, so $C = t_{fetch}$. This number corresponds roughly to the size of the document divided by the bandwidth towards the server. This metric is useful when we want to minimise the delay in serving documents to clients. However, there are several drawbacks in the adoption of this metric. First, the transfer time depends a lot on the congestion of the network at the time of the transfer, thus presenting large variations even for the same server at different times. Second, this metric privileges documents coming from very slow (in terms of bandwidth) servers, so it might be a source of unfairness in the behaviour of the proxy. For these reasons, we do not believe this to be an appropriate metric.

traffic In this case C = size(document)+overhead, i.e. the amount of traffic needed to establish a communication, request the document and get the response. This metric is appropriate when the communication speed is independent from the source of a document, or communication costs are based on the number of bytes transferred.

Given the cost and benefit in *purging* a document, we want to determine how *valuable* is the document for the proxy. To this purpose, we can define the value V of a document as

$$V = \frac{C}{B} P_r$$

The computation of C and B is relatively easy, as it only depends on the size of the document and possibly the bandwidth towards the server; these values are readily available to the proxy. Using a metric based on traffic, which we believe to be the most appropriate in the case of Web caching, C/B is approximately constant and independent of the size of the document.

The computation of P_r is more complex. P_r is in general different for each document, and is time-dependent. In the next Sections we investigate the possibility to estimate P_r by looking at the document itself (size, type, server etc.) and at the history of previous accesses to the document, as seen by the proxy.

3 Trace evaluation

In this work we have used two different trace sets collected from two proxy servers running *squid*. The first set (*UNIPI*) comes from our departmental proxy and covers a period of five months, the second (*DEC* [5]) was collected at Digital Equipment Corporation during a period of 25 days. In Table 2.1 the characteristics of the two sets are shown; in the computation of these values, and throughout the rest of the paper, we have discarded all uncacheable documents.

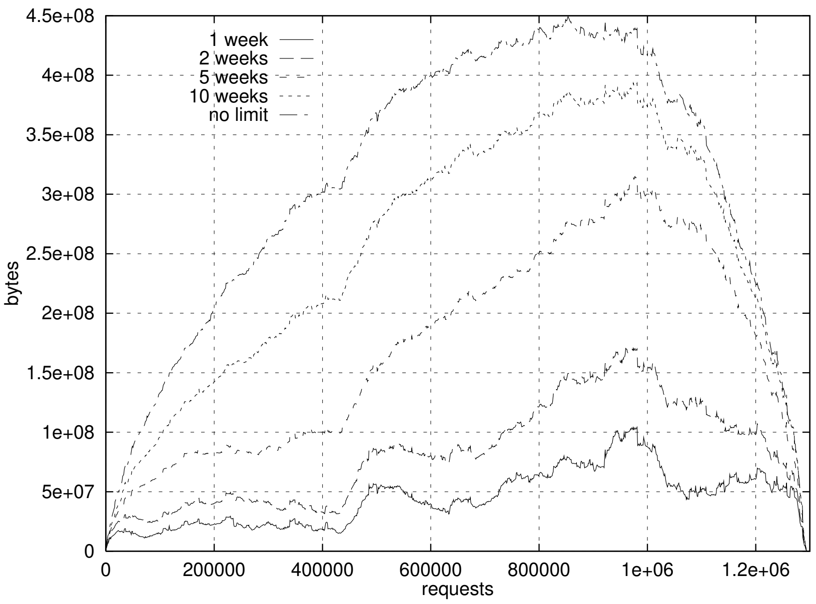

Figure 1: Total size of live documents vs. requests, UNIPI traces. The various curves show how live size changes when the maximum lifetime of a document is limited to 1..10 weeks.

In the following we will analyse the traces to extract statistical information that will be used to compute P_r. We will mainly consider the UNIPI trace set, since it covers a longer interval of time and can give more information on document history. The DEC trace set will be used in the performance evaluation part, and also to determine whether a given property applies to different trace sets.

The UNIPI set – about 1,300,000 accesses – includes about 1,000 clients, 20,000 servers, 450,000 different URLs. The set of all accessed documents amounts to about 7GB of data. Only a small fraction of the documents is accessed more than once ($P_{|i=1}(i) = 0.325$), but $\mathrm{HR}_{max} = 0.66$, meaning that documents accessed more than once feature a large number of accesses. $\mathrm{BHR}_{max} < \mathrm{HR}_{max}$ means that short documents are reaccessed more frequently. The DEC set refers to a much larger client host population – about 17,000 – and records a larger number of references – about 22,000,000 with 66GB of data provided – event though it comprises a shorter period – 25 days. It presents appreciably larger values of BHR_{max} and HR_{max}, which might be due to a better aggregation of references originated by different hosts and to a more stable client host population.

3.1 Live documents

It is unreasonable to think that a cache can store all of the accessed documents, no matter how big are its disks: in fact, the number (and total size) of accessed documents grows with time and with the number of clients. However, it is only necessary that the cache retains live documents to achieve the maximum hit rate. Live documents are a small fraction of the total, and they reach – in the UNIPI dataset – a maximum occupation of 450MB (Figure 1), when computed on the whole length of our traces. In the next section (see Figure 5) we will show that about 50% of the documents with more than one access have a reaccess time larger than one day; so, keeping documents in cache for less than one day (because of limited cache space) might dramatically increase the miss rate; on the other hand, less than 10% of the documents are reaccessed in more than 2-3 weeks, suggesting that a document can be considered dead after such a time with little influence on the cache performance.

Limiting the lifetime of documents is extremely useful in terms of cache storage savings. Figure 1

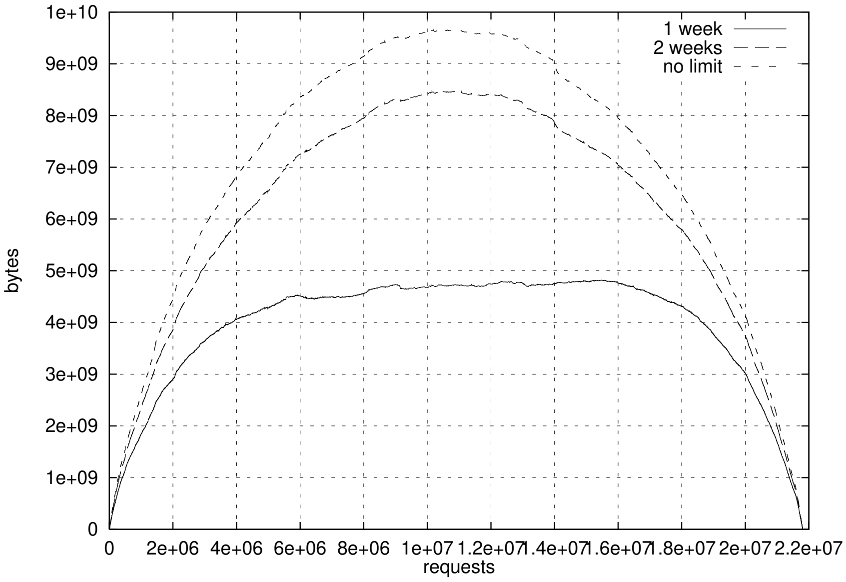

Figure 2: Total size of live documents vs. requests, DEC traces. The various curves show how live size changes when the maximum lifetime of a document is limited to 1 and 2 weeks.

---

shows the size of live documents when limiting the lifetime of a document to 1..10 weeks: reducing the lifetime of documents to 2 weeks reduces the size of live documents to 1/3; it is noticeable that HR_{max} and BHR_{max} computed with such a lifetime have a relative decrease of about 10%. Figure 2 shows the same curves for the DEC dataset. Besides the larger traffic volume, here we can note a more regular shape with a large flat area in the curve relative to 1 week lifetime. Lifetime limits larger than 2 weeks are not considered in Figure 2 because the DEC trace set is only 25 days long.

The number of active clients over time is shown by Fig. 3, for UNIPI dataset, and Fig. 4, for the DEC dataset, which also show curves computed by considering inactive a client which does not issue requests for longer than t weeks ($t = 1 \ldots 10$ in the UNIPI dataset, $t = 1 \ldots 2$ in the DEC dataset). The curves representing the size of live documents and number of clients have a similar shape; this suggests that the size of live documents is mainly a function of the number of clients. In Figure 4 we can see that, if the lifetime of documents and clients is limited, this function remains relatively constant over time for a fixed number of clients. Note that, considering the curves relative to $t = 1$ week, the storage size is roughly equal to .5MB per active client both in the DEC and UNIPI datasets. This result anyway depends on the number of users per client host and can vary over time, as the Web is rapidly evolving, and documents are becoming larger and more structured as clients' bandwidths increase and browser capabilities improve. Furthermore the amount of information available on the Web is constantly increasing. Thus, reasoning about live documents over large time intervals is extremely difficult since traffic features vary.

Another noticeable effect is that, over time, new documents are generated and old ones die with roughly the same, constant, rate. The actual rate (approximately 1/3 of all requests in our case) should depend on the type and number of clients. A low birth/death rate means either large overlaps in requests coming from different clients, or the ineffectiveness of the first-level cache implemented in clients. Our data suggest that the latter is true, since further trace analysis has shown relatively little overlap among the interests of our 1000 clients. This is different for the DEC trace set, where a larger HR_{max} suggests a better client request aggregation.

Having characterised the information contained in our log files, we can start now the analysis of the parameters which influence the probability P_r of a new access to the same document. In the next

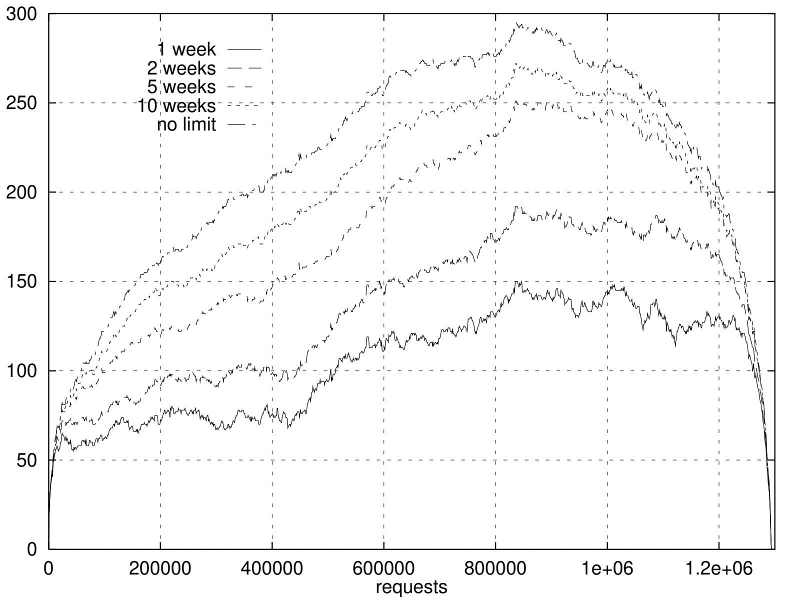

Figure 3: Number of active hosts vs. requests, UNIPI traces. The curves are computed varying the time after which a client is considered inactive.

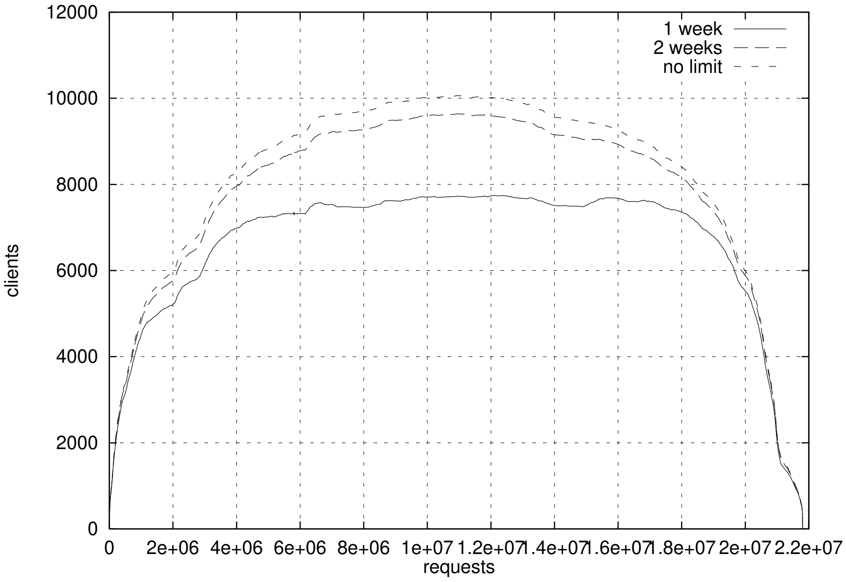

Figure 4: Number of active hosts vs. requests, DEC traces. The curves are computed varying the time after which a client is considered inactive.

section, we compute the conditioned probabilities of a document being accessed again, depending on various information (known to the proxy at runtime, thus usable in a replacement policy) such as the time from the previous access, the number of previous accesses, the server of the document, the client originating the first request, etc. Our goal is to find the parameters that show the best correlation with the probability of reaccess.

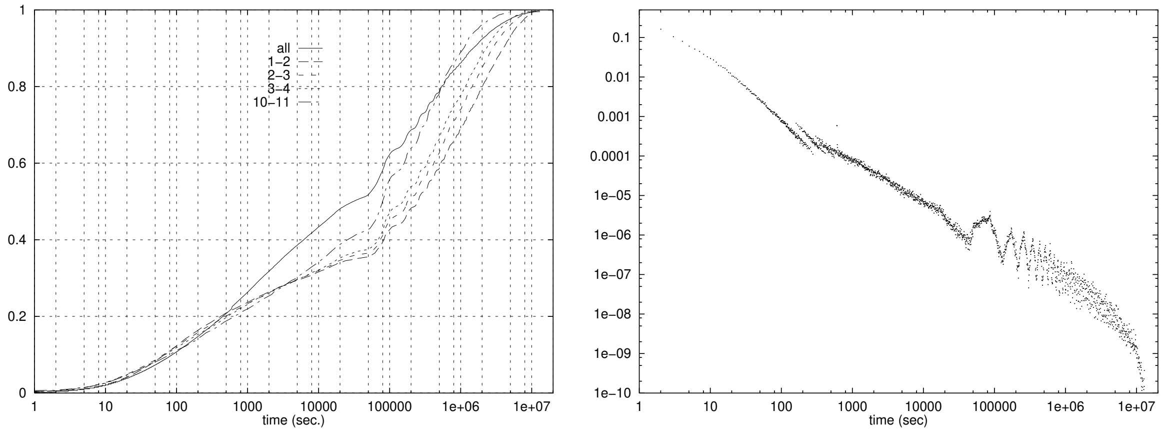

Figure 5: Distribution $D(t)$ (left) and probability density function $d(t)$ (right) of interaccess times (UNIPI trace). $D(t)$ is computed both for the i-th access and for all accesses to the same document.

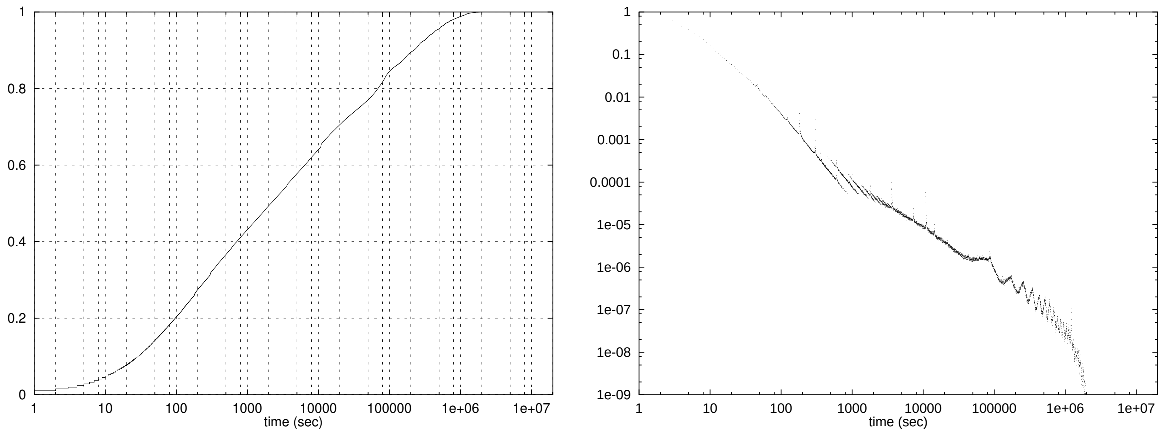

Figure 6: Distribution $D(t)$ (left) and probability density function $d(t)$ (right) of interaccess times (DEC trace).

3.2 Interaccess times

Figures 5 and 6 show distribution $D(t)$ and the probability density function $d(t)$ of times between consecutive requests to the same document. The time axis is logarithmic to ease the reading of the graphs. Although our analysis will be based on Figure 5, as it covers a longer period of time, note the similarity in the shapes of the curves with those of Figure 6; the differences between the two sets (mostly consisting in a heavier tail in the UNIPI traces) are mostly due the fact that DEC traces cover a shorter temporal interval.

We can identify at least two distinct regions: the first one covers up to one day from the previous access, while the second one covers the remaining time scale. Globally, about 60% of accesses occur within one day, with a marked peak around 24 hours, and a relatively flat area between 8 and 18 hours. This is not surprising because our users reside within the same timezone, and most of accesses occur during office hours. About 20% of accesses occur in the first 15 minutes, and 10% in the first minute. This is probably an indication of frequent reloads. In the second area (from one day to the end) there is an approximately exponential decay of interaccesses. The daily peaks are still visible (especially in $d(t)$) but have a decreasing amplitude.

Figure 5 also shows the distribution of interaccess times at the i-th access. Despite some unavoidable differences (documents with more accesses present an heavier tail in $d(t)$), the curves retain approximately the same shape; that allows us to assume that $d(t)$ is approximatively independent from the number of previous accesses.

$d(t)$ is also the p.d.f. of the next access time conditioned to the fact that the document gets requested again. Let X be the probability that a document gets reaccessed, evaluated at the time of the previous access. Assuming the independence stated above, then the p.d.f. of the next access time can be expressed as $X \cdot d(t)$, and $P_r(t)$ can be computed as:

$$P_r(t) = \int_{\tau=t}^{\infty} X \cdot d(\tau) d\tau = X \cdot (1 - D(t))$$

The above equation expresses the dependency of P_r on time. Being $D(t)$ a distribution function, $P_r(t)$ always decreases with time, independently from the shape of $D(t)$; thus, a policy such as LRU discards document with the smallest P_r. According to our model LRU is the best replacement policy, if we only consider the time from the last access.

However, we are interested in $D(t)$ because we want to use it together with other parameters which influence X. For practical purposes, an approximation $\tilde{D}(t)$ of $D(t)$ must be used, computed adaptively by the proxy basing on the available information (the access pattern). We need that $\tilde{D}(t)$ be fast to evaluate since it must be used one or more times for each document to be replaced. To have a suggestion on how to approximate $D(t)$ we can look at its derivative $d(t)$ (the p.d.f. of interacces times). It turns out that $d(t)$ can be reasonably approximated with k/t in the first part ($t < 1$ day), while in the last part it is better approximated by $e^{-\alpha t}$.

A function whose derivative has this behaviour, giving a good approximation of $D(t)$, is the following:

$$\tilde{D}(t) = c * \log\left(\frac{f(t) + \tau_1}{\tau_1}\right) \tag{1}$$

where

$$f(t) = \tau_2(1 - e^{-\frac{t}{\tau_2}})$$

τ_1 and τ_2 are in the range of 10...100s and $> 5 * 10^5$ respectively. Note that for $t \ll \tau_2$, $f(t) \simeq t$, resulting in $\tilde{D}'(t) \propto 1/t$, as we expect. Appendix B discusses this approximation and shows how the parameters c, τ_1 and τ_2 can be easily computed by the cache on the fly. In the following paragraphs, we will concentrate on finding a good estimate for the probability X (probability of reaccess evaluated at the moment of the previous access).

3.3 Number of previous accesses

If we look at $P(i)$, the probability of a document being reaccessed when it has been accessed i times, we see (Figure 7) that its value grows significantly with i, up to 0.9 and more for $i > 12$. Hence, the number of previous accesses appears to be a good indicator of the probability of a new access. However, using this parameter alone to determine P_r is a bad idea, because this does not permit the aging of documents,

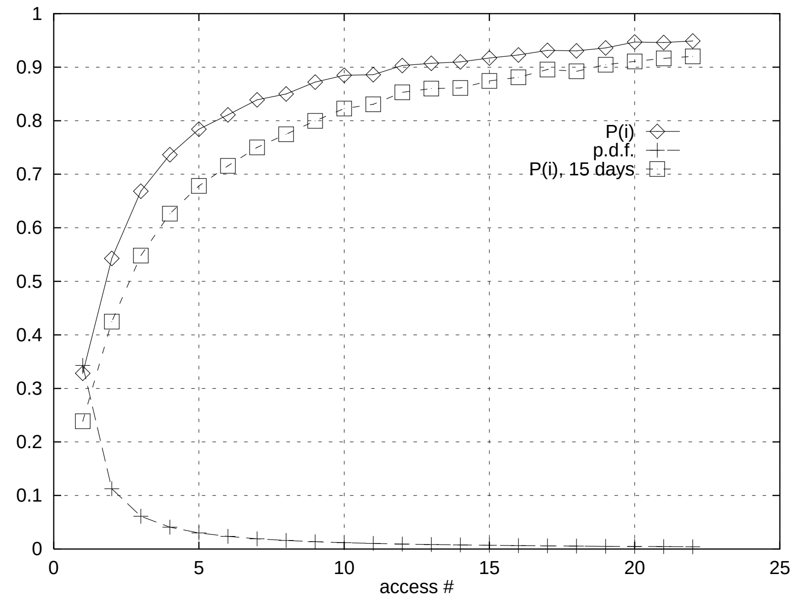

Figure 7: Top curve: the probability of more accesses after i previous ones ($P(i)$). Middle curve: $P(i)$ computed without considering new accesses issued after 15 days. Bottom curve: percentage of documents with at least i accesses.

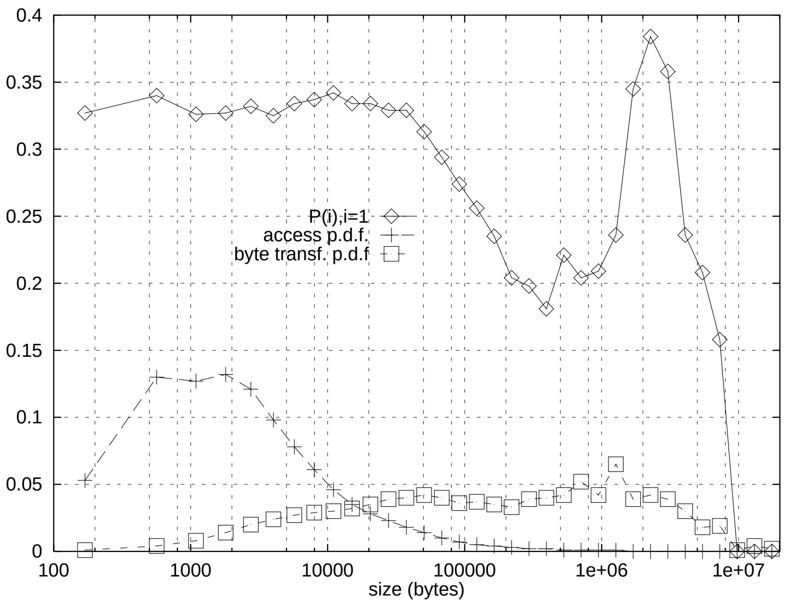

Figure 8: Top curve: percentage of documents with at least 2 accesses vs. document size. The curve with cross marks shows the access p.d.f. vs. document size. The curve with square marks shows the percentage of bytes transferred vs. size.

---

something that is absolutely necessary in order to avoid pollution of the cache. From Figure 5 we see that $D(t)$ depends weakly on the number of previous accesses, so P_r can be approximated as

$$P_r(i,t) = P(i) \cdot (1 - D(t))$$

As a result, the number of previous accesses only modifies the LRU ordering for documents with a comparable time from the last access. In some cases, a document with many past accesses can have a higher probability of being reaccessed than a much more recent document with just one or two accesses. The use of $P(i)$ does not change the LRU ordering within classes of documents with the same number of accesses.

As the largest set of documents – those with one access – belongs to the same class, we would like to make further distinction among them basing on some other parameter. Documents with only a single access carry the following information: size, document's URL (which can be decomposed in protocol, server, document type), and the client originating the request. In the following sections we will look at the dependency of $P_{|i=1}(i)$ on these parameters.

3.4 Document size

Figure 8 shows the dependency of $P_{|i=1}(i)$ on the size of a document ($P_{|i=1}(i,s)$). It is well known that short documents tend to be preferred over long ones, especially by clients working over slow connections. This explains why a large fraction of requests refers to documents with a size of 5KB or less (Figure 8, access p.d.f), for which the delays in transferring the document over a slow modem line are still bearable. Below this threshold, differences in size do not affect significantly the behaviour of clients. Large documents, taking much longer to download, might discourage clients from accessing them, but in some cases they are still popular (e.g. this is the case of software distributed over the net). This also influences the shape of $P_{|i=1}(i,s)$: as shown by the graph, $P_{|i=1}(i,s)$ is slightly larger for $s < 30KB$, while documents in the 30KB-1.5MB range are definitely less popular. The peak around 2MB corresponds to large software packages distributed through the Web.

The dependency on the size is a useful parameters to make a selection among documents with one access for at least two reasons. First, small documents (those with a relatively higher $P_{|i=1}(i,s)$) take less space in the cache, so that we have more room for documents with two or more accesses, for which the probability of reaccess is generally higher. Second, this parameter partitions documents in groups of comparable sizes with different $P_{|i=1}(i)$, so that its use has more influence on the performance of a replacement algorithm.

3.5 Other parameters

We have investigated the dependency of P_r on other parameters, as described in the next paragraphs, but none of them has proven to be a significant indicator for our purposes. As a consequence, they are not used in the LRV algorithm.

Document's source

The dependency of P_r on the source of the document has been evaluated using the following technique. For each server S, we keep the value $P^S_{|i=1}(i) = \frac{\|D_2\|}{\|D_1\|}$, i.e. the value $P_{|i=1}(i)$ computed on documents coming from S. Documents are then partitioned in 10 groups, depending on the $P^S_{|i=1}(i)$ of their server *at the time of the first access to the document.* Finally, the $P_{|i=1}(i)$ is computed for each group.

Figure 9 shows the dependency of $P_{|i=1}(i)$ on the $P^S_{|i=1}(i)$. If documents were allocated to a group after the second access to the document, the graph would be a straight line between (0,0) and (1,1). In practice, the graph deviates from the linear behaviour since the value of $P^S_{|i=1}(i)$ for a given document is computed at the time of the first access. In general, a high $P^S_{|i=1}(i)$ for a server means a high chance that a document coming from that server is accessed again. Note that we have used $P^S_{|i=1}(i)$ rather than

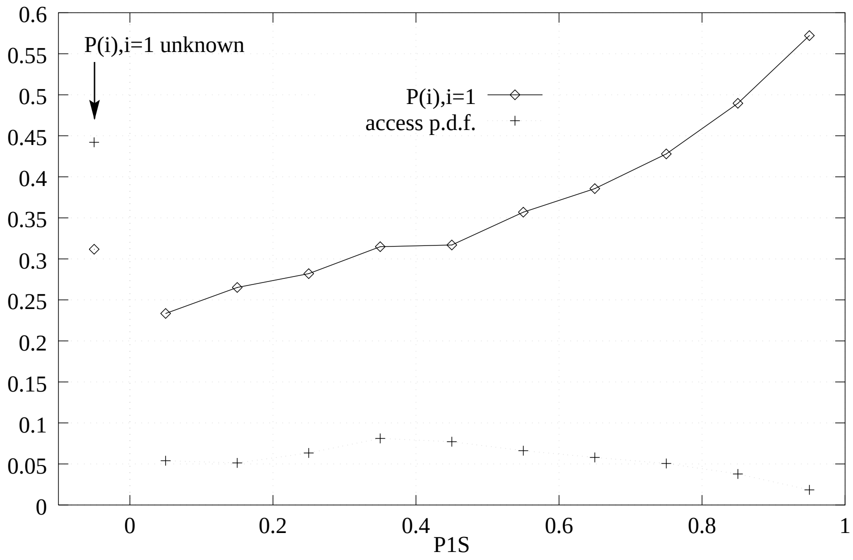

Figure 9: Percentage of documents with at least 2 accesses ($P_{|i=1}(i)$) computed on a per-server-class basis. Documents are grouped basing on the average $P_{|i=1}(i)$ computed on the set of documents coming from the same server ($P^S_{|i=1}(i)$). Below, the percentage of documents in each class. The marks on the left refer to non-classifiable documents.

---

the hit rate to classify servers: with the latter, even a single, very popular, document could bring the parameter arbitrarily close to 1, whereas parameter $P^S_{|i=1}(i)$ gives the same weight to all documents.

While there appears to be some correlation between $P_{|i=1}(i)$ and $P^S_{|i=1}(i)$, the problem with this parameter is that a large number of documents comes from servers with too little information to get reliable statistics (Figure 9, "access p.d.f." shows the percentage of documents belonging to each class), thus making this parameter not very useful for practical purposes.

Client requesting the document

The dependency of P_r on the client which first requests a document has been evaluated with a technique similar to the one used for servers. Figure 10 shows the dependence of $P_{|i=1}(i)$ on the $P^C_{|i=1}(i)$ computed for documents requested by the same client. As in the case of $P^S_{|i=1}(i)$, the graph has been computed by only considering clients with a sufficiently large number of requests.

Here, a low $P^C_{|i=1}(i)$ means a properly working cache on the client (which requests most documents only once), while a large value is a clear indication that the cache on the client is ineffective. Documents requested by clients with a low $P^C_{|i=1}(i)$ appear to have a marginally higher $P_{|i=1}(i)$, but the difference is too small to be useful. On the other hand, clients with a large $P^C_{|i=1}(i)$ are simply not reliable because what they try to do is to use the proxy as their first level cache, an approach which is not scalable and thus should not be used to influence the behaviour of the proxy.

Type of document

We have computed the value of $P_{|i=1}(i)$ per type of document. Almost 40% of the requested documents are graphic files, and for them $P_{|i=1}(i) = 0.36$, slightly higher than the average. Overall, the vast majority of documents have a similar $P_{|i=1}(i)$. These results are consistent with those found in [1] and [12]. This parameter is not useful in building a cache replacement policy since the only documents which have a

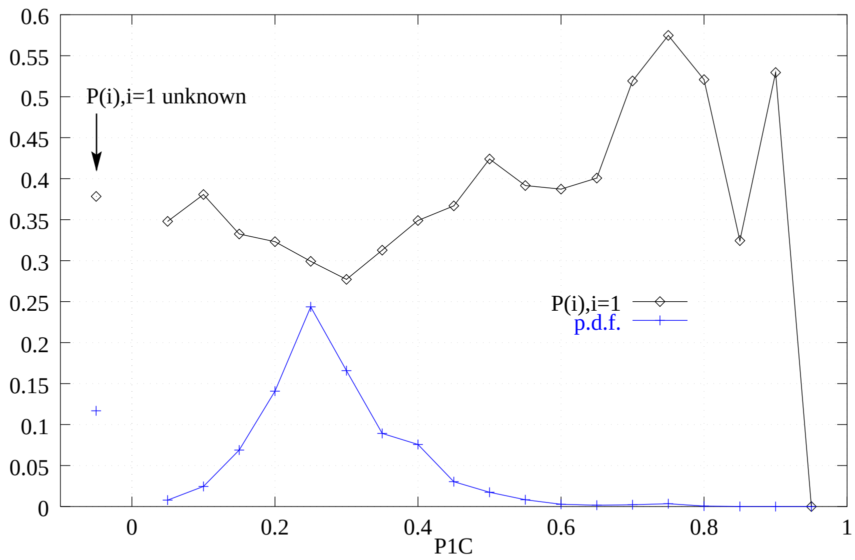

Figure 10: Percentage of documents with at least 2 accesses ($P_{|i=1}(i)$) computed on a per-client-class basis. Documents are grouped basing on the average $P_{|i=1}(i)$ computed on the set of documents requested by the same client ($P^C_{|i=1}(i)$). Below, the percentage of documents in each class. The marks on the left refer to non-classifiable documents.

---

significantly different (and lower) $P_{|i=1}(i)$ are archive files, which have a large average size and are already acted upon by size-based decisions.

3.6 Summary

At this point it is worth summarising what we believe are general enough features of accesses to a proxy server.

- the distribution of interaccess times for the same document decays rapidly with time. A significant fraction of reaccesses is concentrated in a few minutes' interval. Also, some daily periodicity appears to exist in access patterns.
- the probability of reaccess appears to depend heavily on the number of previous accesses, and more weakly on the size of the document;
- some parameters, such as document's source, or the client requesting the document, appear to influence somehow the probability of reaccess. However, using these parameters is unsafe because they apply only to a small set of documents, and dangerous because of the risk of biasing the policy to the advantage of some client/server. As an example, we might favour clients without properly working first level cache, with obvious fairness and scalability problems.
- the probability of reaccess depends on the file type, but this parameter gives approximately the same information as the file size.

4 The LRV algorithm

In Section 2.1 we have introduced the relative value of a document (V) as a function of P_r, the probability that a document is accessed again. In Section 3 we have discussed how various parameters can be used to compute P_r.

In this section we present the LRV algorithm, based on the relative value (V) of a document, and we show how to obtain an approximation of V suitable for our purposes. The LRV algorithm simply selects the document with the Lowest Relative Value as the most suitable candidate for the replacement. As already seen, the relative value, V, of a document is proportional to P_r, thus, in the end, the issue is evaluating this probability.

Section 3 has evidenced some of the features of the accesses to a proxy server. It has been shown how P_r is strongly influenced by the time from the previous access, but also other parameters have some impact on this probability. A class of documents which is particularly difficult to handle is the one containing documents which have been accessed only once: it comprises many documents which generally have a low probability of being accessed again.

To simplify the problem of computing P_r, we have made the assumption that the variables defining the document state are independent; this way we can express P_r as the product of the various probabilities. Moreover, in order to keep the algorithm simple, we have tried to use as few parameters as possible, and we have neglected the dependence of P_r on some of the chosen parameters in certain circumstances. We have only used those parameters which, in the trace evaluation, have shown a significant influence in P_r, and are not correlated – thus not violating our assumption of independent variables. Finally we have tried to make the algorithm fast, and to be able to compute its parameters adaptively, rather than using precomputed probabilities biased on our traces, which might not suit all situations.

The parameters that we have selected are the following:

- *time from the last access, t*, for its large influence on the probability of a new access. As shown in Section 3.2, the dependence of P_r on the time from the last access can be expressed as $1 - \tilde{D}(t)$. Appendix B shows how to compute the parameters which influence $\tilde{D}(t)$ adaptively basing on the history of previous accesses.

- *number of previous accesses, i*. From Figure 7 we see that this parameter allows the proxy to select a relatively small number of documents with a much higher probability of being accessed again. The computation of $P(i)$ can be done very easily by the server, and only small number of distinct classes is necessary as $P(i)$ tends to saturate for small values of i. Simulations have shown that the use of this parameter has a great benefit on the performance of the cache.

- *document size, s*. This seems to be the most effective parameter to make a selection among documents with only one access. Thus we use it only for these documents. As the policy based on document size tends to discard large documents, this also allows us to employ less storage for document with one access, saving space for the other documents (which generally have a higher probability of being reaccessed). As in the case of $P(i)$, $P_{|i=1}(i, s)$ can be computed adaptively by the proxy at little cost.

We compute $P_r(i, t, s)$ as follows:

$$P_r(i,t,s) = \begin{cases} P_{|i=1}(i,s)(1-\tilde{D}(t)) & \text{if } i = 1 \\ P(i)(1-\tilde{D}(t)) & \text{otherwise} \end{cases}$$

Note that we neglect the dependence of $P_r(i, t, s)$ on s when $i > 1$. For a proper implementation of LRV, especially for the computation of $P(i)$, it is required that metadata for documents are retained longer than the associated documents. Since metadata take on average about 1-2% of the space taken by documents, this does not constitute a problem.

4.1 Performance

The performance of LRV, compared to other algorithms, such as LRU, LFU, size and FIFO, has been simulated basing on both the UNIPI and the DEC trace set. A random replacement policy has been considered as well. Initial evaluations have been done using a general purpose simulator for networks of cooperative proxies. Subsequent experiments have been done with specialised C programs, much more efficient in processing the large traces we have used. One of these programs contains a full implementation of LRV and all the other policies that have been evaluated. The various programs process sanitised traces where strings (e.g. URL, client names etc.) are replaced by unique numbers, thus allowing a very fast processing.

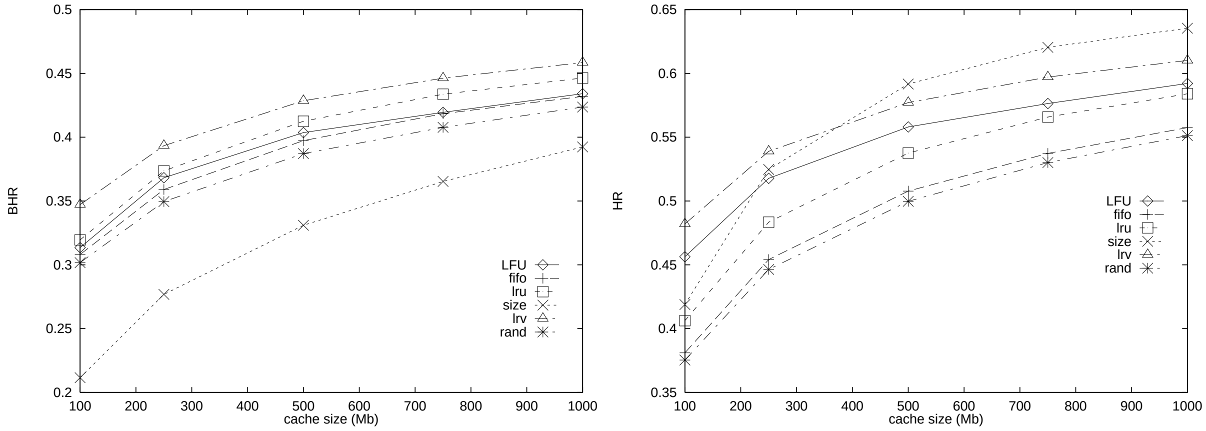

Figure 11: The values of BHR and HR for different policies vs. the cache size, UNIPI trace set.

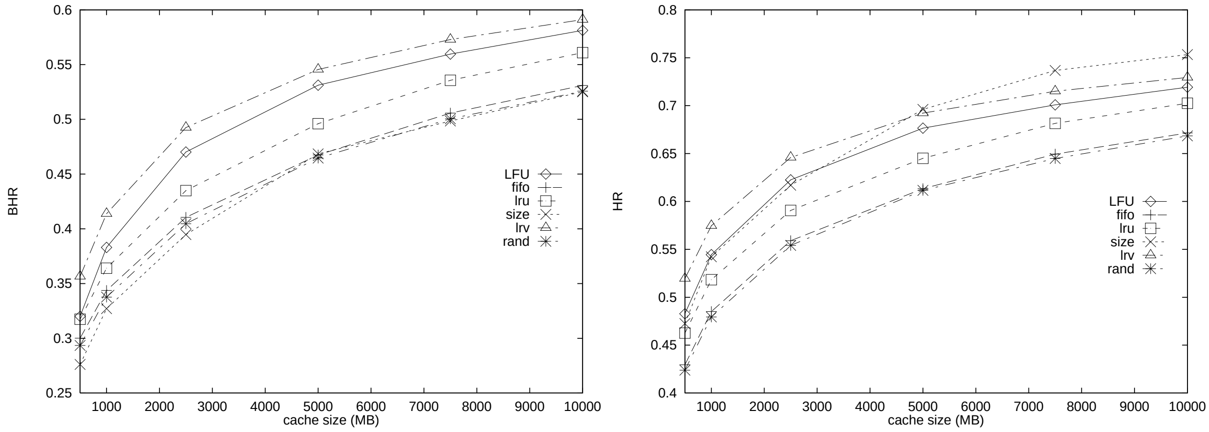

Figure 12: The values of BHR and HR for different policies vs. the cache size, DEC trace set.

The performance, for different cache sizes, is shown in Figure 11 and 12. For UNIPI trace set, it must be kept in mind that the set of documents accessed at least once amounts to 7Gb of data, while the set of live documents is always lower than 450MB (see Figure 1), hence we have used comparable cache sizes (100MB..1GB); we have done the same for DEC trace set, considering that the data volume is roughly ten times larger. For simplicity, and compatibility with existing proxy caches, we have run the replacement algorithm when the cache occupation reached a high-water mark (100% in our case) and continued to purge documents until occupation reached a low-water mark (90% in our case). This watermarking technique somewhat reduces the exploitation of the available storage, but it is used in some algorithms where finding the candidates for replacement is expensive, e.g. requires sorting documents. Increasing the gap between high and low-water marks reduces the overhead for the above computations, but also

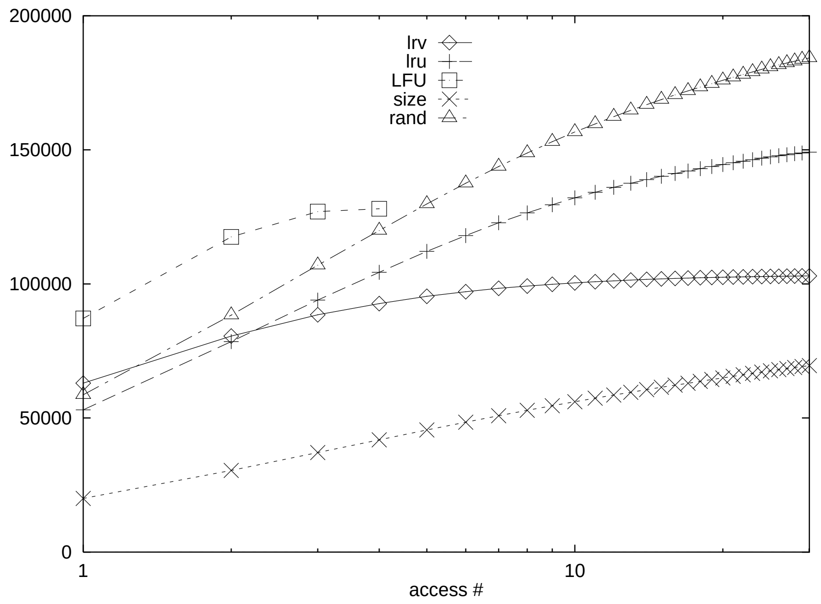

Figure 13: The cumulative number of wrong choices in discarding documents vs. number of accesses to the document; cache size is 500MB (UNIPI traces).

reduces the effectiveness of use of the available storage. LRV can easily determine which document(s) to purge on demand in a small constant time.

Figure 11 and 12 show the values of HR and BHR for different policies and cache sizes. As it can be seen, LRV features a consistently higher BHR than other policies in all conditions. The same happens for the HR, except in the case of the SIZE policy with large caches (but this is not accompanied by a comparable BHR). Reducing the cache size causes the SIZE policy to worsen because of the pollution of the cache with small documents which are never replaced. This phenomenon does not appear with larger cache sizes just because filling the cache with dead documents requires more time.

4.2 Wrong decisions

Not knowing the future, any replacement algorithm is subject to make errors and discard documents which will be accessed in the future. These errors are unavoidable if the size of live documents exceeds the cache size, while in principle they might be avoided if the cache size is even marginally larger than live documents. Thus, it is interesting to evaluate the number of errors made by the various replacement policy in discarding documents which will be accessed again, when the cache size is slightly larger than the set of live documents. We have run this experiment on the UNIPI trace set (whose maximum live document set amounts to 450MB) using a cache size of 500MB.

In Figure 13 we show the cumulative number of errors made by various policy. As expected, the LFU policy is never wrong about documents with a large number of accesses, but does many errors on documents with fewer accesses. It is also expected that the largest number of errors is made by the RANDOM policy, which makes no use of the available information. The best performing policies in this graph are LRV and SIZE. It should be noted that the absolute number of errors is not directly comparable among different policies, since the total number of replacement might also vary. Additionally, the effect of each error on the overall cache performance depends on the size of the document which has been discarded. In our case, the SIZE policy is significantly different from others, since it tends to discard large documents. As a consequence, SIZE makes fewer replacements than other policies, thus explaining

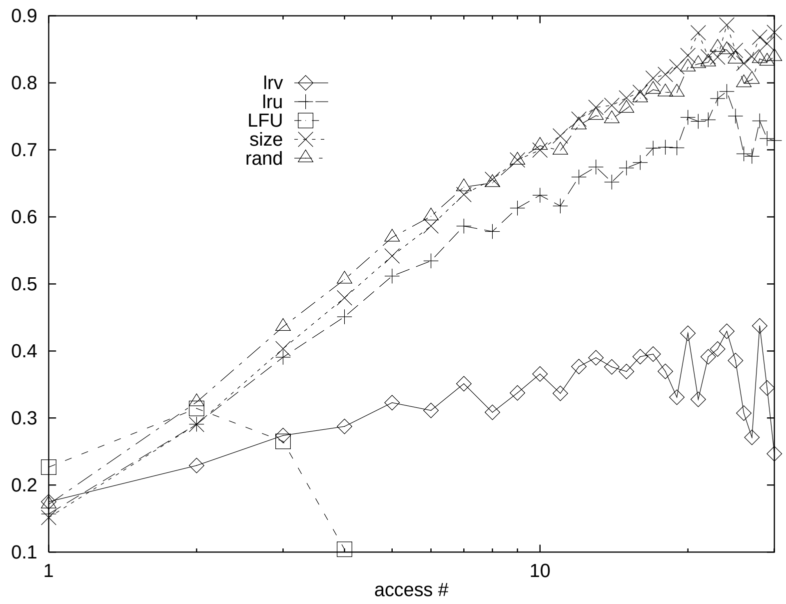

Figure 14: The percentage of wrong choices in discarding documents vs. number of accesses to the document; cache size is 500MB (UNIPI traces).

a smaller number of errors.

In Figure 14, the errors on each document class (with the same number of accesses) are weighted on the total number of replacement for that class, giving more insight on the actual behaviour of each policy. In fact, we believe that this is an extremely useful indicator for tuning an algorithm. Other parameters, such as BHR or HR, often present very little sensitivity to the parameters of the algorithm, and, especially, do not help in identifying the situations where the algorithm performs poorly. From the graph, we see that all policies but LFU and LRV tend to make a large percentage of errors on documents with many accesses. On the other hand, LFU does not perform well on documents with one access – which is a significant set of documents – because the lack of an aging mechanism effectively reduces the cache size for new documents.

4.3 Implementation details

Proxy implementors are often concerned by the computational complexity of a replacement policy, because of the high load on the proxy. In this section we discuss how LRV can be implemented very efficiently so that its overhead poses no concerns.

Documents with one access are classified in a small number of groups, say 10, basing on their size. Documents with two or more accesses are classified basing on the number of previous accesses, with the last group containing those documents with more than about 10 accesses (because $P(i)$ saturates). The metadata associated to documents are held in a doubly-linked FIFO queues, one for each group.

Since $(1 - \tilde{D}(t))$ is a monotonic function of t, the document to be replaced can be selected by simply computing P_r for the first document of each queue, and selecting the one with the smallest value. Being the number of queues small, we see immediately that the decision only requires a small constant time, not dependent on the number of documents in cache. The queue management related to each new request reduces to a simple queue extraction and tail insertion, both constant-time operations[4]. On document

[4]We neglect the time necessary to locate the metadata associated to the newly requested document, since this operation is always necessary independently of the cache replacement algorithm being used.

removals, it is simply necessary to store the record associated to the document in a pool of storage holding metadata for documents not in cache anymore.

Removal costs can be further reduced as follows. First, because of the granularity of the measurements, it is certainly not necessary to compute the P_r for each queue (and sort the computed values) more than once per second. Second, if more documents must be purged at once to make room in the cache (e.g. when a watermarking technique is used), it is only necessary to recompute one P_r for every additional deletion. So, in practice, we can assume that the implementation of LRV will have a maximum overhead of one computation of P_r and one scan of a short (20..30 entries) array for each replacement. As a practical indication, on a Pentium-class CPU, a straightforward computation of P_r requires about 3μs per document, resulting in about 50..100 μs per second (a negligible fraction of CPU time) plus 5 μs per deletion.

We remark the fact that all parameters involved in the computation of LRV are computed adaptively from the cache. $P(i)$ and $P_{|i=1}(i,s)$ can be derived by keeping a pair of counters per group, with the relevant pair being updated at each access. $\tilde{D}(t)$ coefficients get computed dynamically too, as shown in Appendix B: the overhead associated to their computation is negligible as it occurs infrequently.

5 Conclusions

We have presented a detailed analysis of traces of accesses to Web proxies, in search of statistical parameters which could be used in the design of a cache replacement policy. The traces we have analysed cover a period of up to five months, and include a very large number of accesses, thus allowing us to draw significant conclusions even on the long-term behaviour of proxy accesses and document lifetime. The main goal of our analysis was to relate the probability of a document being reaccessed to the history of previous accesses, and discriminate among useful and useless dependences.

Based on these dependences, and on a cost/benefit model for Web documents, we have designed a replacement policy called *Lowest Relative Value* (*LRV*) to achieve a better selection of documents to purge. LRV computes documents' values using a fully adaptive, easy-to-compute formula. We have shown how LRV overcomes some limitations of LRU and other policies, and that it can outperform them in all cases. LRV proves to be particularly useful in the presence of small caches. LRV can be easily inserted in existing proxy servers, and we have shown how its implementation can be highly efficient. With a proper organisation of data, keeping documents sorted by value requires constant-time operations on data held in main memory, thus allowing replacement to be performed on demand rather than using watermarking techniques, which empty the cache below a low-watermark level whenever its occupancy exceeds the high-watermark level.

We believe the characterisation of proxy accesses presented in Section 3 to be reasonably general, because it has been derived from relatively large traces covering almost 20,000 clients and 200,000 servers. Of course, the features of Web documents and Web usage are rapidly changing over time, and monitoring the evolution of these features over time will be necessary. We can certainly expect an increase on average document size (because of the availability of faster networks and richer documents, including streaming media), and perhaps a reduction in the hit rate because of the increasing use of personalised documents. Although we do not expect such changes to have significant impact on the behaviour of the LRV algorithm because of its full adaptivity, it might well be possible that specific Web access patterns arise which could provide more useful parameters to be used in cache-replacement decisions.

Acknowledgements

The authors wish to thank Paolo Lorenzetti for his initial work on the LRV algorithm and Jon Crowcroft for his comments. We are also grateful to DEC for making the traces of their Web proxy available for simulation purposes.

References

[1] M.F.Arlitt, C.L.Williamson, "Web Server Workload Characterization: The Search for Invariants", Proc. of SIGMETRICS 96, May 1996, Philadelphia, PA, USA

[2] T.Berners-Lee, R.Fielding and H.Frystyk, "Hypertext Transfer Protocol – HTTP/1.0", *RFC 1945*, May 1996.

[3] Pei Cao, Sandy Irani, "GreedyDual-Size: A Cost-Aware WWW Proxy Caching Algorithm", 2nd Web Caching Workshop, Boulder, Colorado, June 1997.

[4] A.Chankhunthod, P.Danzig, C.Neerdaels, M.Schwartz and K.Worrell, "A Hierarchical Internet Object Cache", Proc. of the 1996 USENIX Technical Conference, January 1996, San Diego, CA, USA.

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

[6] R.Karedla, J.S.Love, B.G.Wherry, "Caching Strategies to Improve Disk System Performance", *IEEE Computer*, pp.38-46, v.27, n.3, March 1994.

[7] J.Gwertzman, M.Seltzer, "World Wide Web Cache Consistency ", Proc. of the 1996 USENIX Technical Conference, January 1996, San Diego, CA, USA.

[8] P.Lorenzetti, L.Rizzo, L.Vicisano, "Replacement policies for a proxy cache", Technical report LR-960731, DEIT, Univ. di Pisa, July 1996
`http://www.iet.unipi.it/~luigi/caching.ps`

[9] A.Luotonen, H.Frystyk, T.Berners-Lee, "W3C httpd,"
http://www.w3.org/hypertext/WWW/Daemon/Status.html.

[10] V.N.Padmanabhan, J.C.Mogul, "Using Predictive Prefetching to improve World Wide Web Latency", *ACM Computer Communication Review*, pp.22-36, v.26, n.3, July 1996.

[11] "Squid Internet Object Cache", http://www.nlanr.net/Squid/.

[12] S.Williams, M.Abrams, C.R.Standridge, G.Abdulla, and E.A.Fox, "Removal Policies in Network Caches for World-Wide Web Documents", Proc. of ACM Sigcomm96, August 1996, Stanford University, CA, USA.

[13] S.Williams, M.Abrams, C.R.Standridge, G.Abdulla, and E.A.Fox, "Errata for 'Removal Policies in Network Caches for World-Wide Web Documents' ", *ACM Computer Communication Review*, pp.118-119, v.27, n3, July 1997.

Appendix

A Average Retrieval Time

The retrieval time of a given document i of size s_i depends on its presence in cache. For cached documents, the retrieval time is s_i/B_i, where B_i is the actual bandwidth between the cache and the client; if the document must be fetched from the original site, the retrieval time is $s_i/\min(b_i, B_i)$, where b_i is the actual bandwidth to the server providing the document. In these expressions we have neglected the overheads needed to establish the connection and to issue the request.

In many cases, clients have a high bandwidth to the proxy, which is often connected to the rest of the network by a slower link. So, we can safely assume that $b_i \ll B_i$. The expected retrieval time through the cache can be expressed as $t_i = (s_i/B_i)p_i + (s_i/b_i)(1-p_i)$, being p_i the probability of finding document i in cache. If we now consider a large sample of N documents and assume that bandwidth to the source is independent from the document size, we can compute the average speedup achievable by the cache, S, as the ratio between the retrieval times with and without cache:

$$S = \frac{\left(\sum_{i=0}^{N} s_i\right)/\overline{b}}{\left(\sum_{i=0}^{N} s_i p_i\right)/\overline{B} + \left(\sum_{i=0}^{N} s_i(1-p_i)\right)/\overline{b}}$$

Recalling that, by definition, $BHR = \frac{\sum s_i p_i}{\sum s_i}$, we have:

$$S = \frac{1}{(\overline{b}/\overline{B})\, BHR + 1 - BHR} \simeq \frac{1}{1 - BHR}$$

where the last approximation holds when $\overline{b} \ll \overline{B}$.

B Adaptive computation of $D(t)$ approximation

In Section 3.2 we introduced the approximation of $D(t)$ as:

$$\tilde{D}(t) = c * \log\left(\frac{f(t) + \tau_1}{\tau_1}\right) \quad \text{where } f(t) = \tau_2(1 - e^{-\frac{t}{\tau_2}})$$

τ_1 accounts for the periodicity of frequent references to popular documents, while τ_2 accounts for the long term decay. As we noticed experimentally, τ_1 and τ_2 are far apart from each other, so the following approximation holds for $\tilde{D}(t)$ when $t \ll \tau_2$:

$$\tilde{D}(t) \approx X(t) = c * \log\left(\frac{t + \tau_1}{\tau_1}\right) \tag{2}$$

(recall from Section 3.2 that τ_2 is very large, say $\tau_2 > 5 \cdot 10^5$). This allows us to compute τ_1 and c by evaluating $D(t)$ in two points t_a and t_b ($t_a \ll t_b \ll \tau_2$), and solving the non linear system:

$$\begin{cases} X(t_a) \approx D(t_a) \\ X(t_b) \approx D(t_b) \end{cases} \tag{3}$$

By rewriting Equations 3 as follows:

$$\begin{cases} \tau_1 = t_1/\left(e^{\frac{D(t_a)}{c}} - 1\right) \\ c = D(t_b)/\left(\log(\frac{t_2+\tau_1}{\tau_1})\right) \end{cases} \tag{4}$$

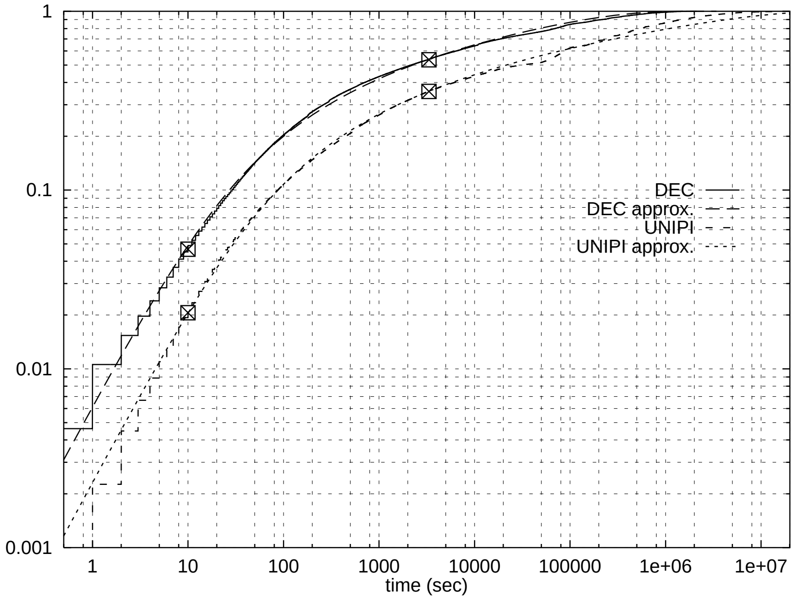

Figure 15: $D(t)$ Approximation for both UNIPI and DEC trace set.

the system can be solved iteratively by starting from a given $c = c_0$ and substituting the value obtained from the first equation in the other. This way of proceeding always converges to the correct values of c, τ_1 after few iterations. Finally, we can compute τ_2 remembering that:

$$\lim_{t\to\infty} \tilde{D}(t) = 1$$

which gives

$$\tau_2 = \tau_1(e^{\frac{1}{c}} - 1)$$

The speed of convergence and the goodness of the approximation of $D(t)$ with $\tilde{D}(t)$ depend on the choice of t_a and t_b. Although we could devise some sophisticated procedure to find optimum values, our experiments show that the values of $t_a = 10$s and $t_b = 3000$s, empirically chosen, permit to have a good approximation, and allow a fast convergence in the computation of c and τ_1, which is in the order of 10 iterations. Figure 15 shows the results obtained in both our trace sets.

Finally it is worth mentioning that $D(t_a)$ and $D(t_b)$ can be computed as $D(t_a) = ref_a/ref_{all}$ and $D(t_b) = ref_b/ref_{all}$. ref_{all}, ref_a and ref_b are three accumulators: ref_{all} is updated each time a document gets referenced again, while ref_a and ref_b are updated when a document is referenced again within t_a or t_b seconds, respectively, from the previous access.