Proxy Cache Design: Algorithms, Implementation and Performance

Junho Shim[1], Peter Scheuermann[1], and Radek Vingralek[2]

[1] Department of Electrical and Computer Engineering
Northwestern University, Evanston, IL 60208, USA
shimjh, peters@ece.nwu.edu

[2] Lucent Technologies, Bell Laboratories
600 Mountain Ave., Murray Hill, NJ 07974, USA
rvingral@research.bell-labs.com

Abstract

Caching at proxy servers is one of the ways to reduce the response time perceived by Web users. Cache replacement algorithms play a central role in the response time reduction by selecting a subset of documents for caching so that a given performance metric is maximized. At the same time, the cache must take extra steps to guarantee some form of consistency of the cached documents. Cache consistency algorithms enforce appropriate guarantees about the staleness of the cached documents. We describe a unified cache maintenance algorithm, LNC-R-W3-U, which integrates both cache replacement and consistency algorithms. The LNC-R-W3-U algorithm evicts documents from the cache based on the the delay to fetch each document into the cache. Consequently, the documents which took a long time to fetch are preferentially kept in the cache. The LNC-R-W3-U algorithm also considers in the eviction consideration the validation rate of each document as provided by the cache consistency component of LNC-R-W3-U. Consequently, documents which are infrequently updated, and thus seldom require validations, are preferentially retained in the cache. We describe the implementation of LNC-R-W3-U and its integration with the Apache 1.2.6 code base. Finally, we present a trace-driven experimental study of LNC-R-W3-U performance and its comparison with other previously published algorithms for cache maintenance.

Keywords: proxy, caching, cache replacement, cache consistency, world wide web.

1 Introduction

The World Wide Web has become one of the predominant services to distribute various kinds of information. Within less than 7 years of its existence, the Web has grown to compete with well established information services such as the television and telephone networks. However, the Web delivers much poorer performance when compared to the traditional services. For example, a multi-second response time to deliver a 5 KB document is not unusual [22].

Caching is one of the ways to reduce the response time of the Web service [1, 4, 7, 11, 17, 5, 18, 24, 26, 16]. The Web documents are cached either directly by the browser or by a proxy server which is located "close" to the clients. *Cache replacement algorithms*, which dynamically select a suitable subset of documents for caching, play a central role in the design of any caching component; these algorithms have been extensively studied in the context of operating system virtual memory management and database buffer management [8, 21]. Cache replacement algorithms usually maximize the cache hit ratio by attempting to cache the data items which are most likely to be referenced in the future. Since the future data reference pattern is typically difficult to predict, a common approach is to extrapolate from the past by caching the data items which were referenced most frequently. This approach is exemplified by, e.g., the LRU cache replacement algorithm.

We argue in this paper, however, that maximizing the cache hit ratio alone does not guarantee the best client response time in the Web environment. In addition to maximizing the cache hit ratio, a cache replacement algorithm for Web documents should also minimize the cost of cache misses, i.e., the delays caused by fetching documents not found in the cache. Clearly, the documents which took a long time to fetch should be preferentially retained in the cache. For example, consider a proxy cache at Northwestern University. The cache replacement algorithm at the proxy found two possible candidates for replacement. Both documents have the same size and the same reference frequency, but one document originates from the University of Chicago while the other is from Seoul National University. The cache replacement algorithm should select for replacement the document from the University of Chicago and retain the document from Seoul National University because upon a cache miss the former can be fetched much faster than the latter.

However, the response time improvement achieved by caching documents does not come completely for free. In particular, caches must generate extra requests to maintain their content consistent with the primary copies on servers and/or sacrifice the consistency of the cached documents. Most proxy cache implementations rely on a *consistency algorithm* to ensure a suitable form of consistency for the cached documents.

Cache consistency algorithms for client/server database systems usually enforce strong consistency (i.e. no stale data returned to clients). In the context of WWW, strong consistency can be either guaranteed by having the proxy cache to poll servers each time a client hits the cache [18] or by maintaining a state on each server with callbacks for all the proxies caching the server's documents. The former approach leads to an overhead which practically eliminates the benefits of caching, while the latter approach is not supported by any of the currently available Web servers. Consequently, all caching proxies known to the authors guarantee only some form of weak consistency (i.e. clients may receive stale data) [9, 11, 17]. A typical implementation of weak consistency assigns to each document (or group of documents) a time-to-live (TTL) interval. Any client request for a document which has been cached longer than TTL units results in document validation by sending a HTTP conditional (If-Modified-Since) GET request to the server. The server responds with the requested document only if it has been updated. A common guideline for setting TTL for a document is based on the observation that documents which were not updated in the past will not be updated also in the future [6].

However, the cache consistency algorithms are not typically well integrated with the cache replacement algorithms. The published work on the topic usually considers the two algorithms as two separate mechanisms and studies one of the two in isolation [1, 4, 7, 11, 17, 5, 18, 24, 26, 16]. The proxy cache implementations either replace documents in the cache solely based on the value of TTL [2], thereby eliminating the need for a separate cache replacement algorithm, or apply the cache consistency algorithm as a filter on the contents of cache before a cache replacement algorithm is used [15, 25]. Thus, the later approach implies that cache replacement algorithm will only be applied only if the filtering process did not create enough room in the cache. Clearly, the performance of the proxy cache can be improved if the cache replacement algorithm can explicitly use the TTL estimates to select documents for eviction from its cache. For example, if the cache contains two documents of approximately the same size which are loaded from the same server and referenced by clients with the same rate, then the cache replacement algorithm should select for eviction the more frequently updated document (i.e. the one with a smaller TTL) because more references can be satisfied directly from the cache without validation and thereby provide a better response time to clients.

In this paper we describe the design and implementation of a new, unified cache replacement algorithm LNC-R-W3-U (Least Normalized Cost Replacement for the Web with Updates) which incorporates cache replacement and consistency components. We show that the LNC-R-W3-U algorithm approximates in a constrained model the optimal cache replacement algorithm, which is NP-hard. The LNC-R-W3-U algorithm is a greedy, cost-based algorithm. The cost function explicitly considers the delay to fetch each document into the cache and the cost of validation of each document as determined by its TTL. Our algorithm estimates the time-to-live intervals for cached documents using a sample of K most recent Last-Modified timestamps. A similar mechanism is also used for estimation of reference rate of each document. It has been shown that such estimates improve cache performance in presence of transient data access [21, 24, 23]. We are in the process of integrating the LNC-R-W3-U library with the Apache 1.2.6 code base [2]. We describe some of the implementation issues we have encountered. Finally, using trace-driven experiments, we compare the performance of the LNC-R-W3-U algorithm against similar algorithms published in the literature. We also use the experiments to provide guidelines for setting various fine-tuning parameters of the LNC-R-W3-U algorithm.

The remainder of this paper is structured as follows: In Section 2 we review the existing approaches to proxy cache design. In Section 3 we describe the new algorithm, LNC-R-W3-U. Section 4 contains implementation considerations of LNC-R-W3-U. In Section 5 we experimentally evaluate the performance of LNC-R-W3-U. Finally, Section 6 concludes the paper and describes the directions of future work.

2 Current State of Art

2.1 Practice

We surveyed implementations of three publicly available proxy caches: Apache 1.2.6 [2], Squid 1.1.21 [25] and Jigsaw 2.0 [15].

The Apache proxy cache uses a TTL-based consistency algorithm. The TTL is set using the Expires header as `Expires - now`. If the Expires header is absent in the response, then the TTL is set using the Last-Modified header as `fudge_factor * (now - Last-Modified)`, where `fudge_factor` is a system defined parameter. If a cached document with an expired TTL is referenced, a conditional GET request is sent to the server specified in the document's URL to validate the document. There is no separate cache replacement algorithm in Apache, because the documents are replaced directly based on their values of TTL.

The Squid cache consistency algorithm is also TTL-based and similar to that of Apache. However, whenever the number of documents in the cache exceeds a certain threshold, Squid employs a separate cache replacement algorithm to reclaim free cache space. The cache replacement algorithm works in two passes: The first pass evicts obvious victims from the cache, i.e. the documents with expired TTLs. The second pass is a standard LRU replacement.

The Jigsaw cache consistency algorithm is also TTL-based. If a cached document has an Expires header, the TTL is set in the same way as in Apache. If, however, the header is absent, then the

TTL is set to a default value of 24 hours. Jigsaw also uses standard LRU replacement to free cache space.

As with all systems implementations, the real devil is in the details. The real complexity of proxy cache implementation stems from handling of many special cases and guaranteeing caching behavior as specified by the HTTP 1.1 protocol. In this section we concentrated only on the major algorithmic steps necessary for the comparison with other published work. An interested reader should refer to the publicly available source code [2, 25, 15].

2.2 Theory

Cache replacement algorithms have been extensively studied in the context of operating system virtual memory management and database buffer pool management. Several cache replacement algorithms tailored to the proxy cache environment have been published [1, 4, 5, 26], including our own work [24].

The algorithms in [1] exploit the preference of Web clients for accessing small documents [10, 14, 20] and give preference to maintaining small documents in the cache. One of the algorithms, LRU-MIN, is used as a yardstick for performance comparison with LNC-R-W3-U in Section 5. The importance of explicitly considering the delay to fetch documents in cache replacement has been first observed in [4]. The cache replacement algorithm in [4] is cost-based. However, the choice of the cost function is not justified and uses unspecified weights. Two very similar cost-based algorithms for delay sensitive cache replacement have been independently published in [24, 26]. The LNC-R-W3-U cache replacement algorithm is directly based on our earlier published algorithm LNC-R-W3 [24]. In this paper we justify the choice of the cost function in the LNC-R-W3-U algorithm by showing that it approximates the optimal NP-hard algorithm, which maximizes the fraction of network delays saved by satisfying requests from the cache. The HYB algorithm in [26] uses an almost identical cost function to the one employed in LNC-R-W3, but makes use of different mechanisms for estimating the parameters of the cost function. An efficient implementation of an algorithm that considers cost and size of documents, called GreedyDual-Size, appeared in [5]. The authors also show that the algorithm is k-competitive, i.e. the cost achieved by the online algorithm is at most k the cost of the optimal NP-hard off-line algorithm, where k is the ratio of the cache size to the size of the smallest document. However, both [26, 5], require either modifications to the HTTP protocol or to the servers themselves. In contrast, our LNC-R-W3-U cache replacement algorithm can be applied off-shelf to any cache proxy.

Cache consistency algorithms have been extensively studied in the context of distributed file systems and client/server database systems. Most of the algorithms guarantee strong consistency and require servers to maintain a state about the data cached by clients (callbacks). "Polling-every-time" approach [18] is another example of a strong consistency method. The Alex FTP cache is an exception in that it provides only weak consistency to the clients [6]. Its validation mechanism serves as a basis for consistency algorithms found in most proxy cache implementations [9, 11, 17, 18, 2, 15, 25] and also the LNC-R-W3-U cache consistency algorithm. In [11] a number of cache consistency algorithms used by a majority of proxy caches are reviewed and a number of improvements to either their implementation or to the HTTP protocol are suggested. An experimental comparison of callback-based and TTL-based cache consistency algorithms is carried out in [11]. However, the evidence seems to be inconclusive. The authors of [11] conclude that if weak consistency is acceptable and network bandwidth is scarce, the TTL-based algorithms are more attractive; on the other hand, in [18] it is argued that both callback-based and TTL-based consistency algorithms have practically the same network overhead. However, since all caching proxies are currently using some version of weak consistency it is clear that this remains the preferred strategy to incorporate into an integrated cache replacement algorithm.

3 Algorithms

3.1 Optimal Cache Replacement

Although it has been frequently pointed out that cache replacement algorithms for proxy servers should somehow reflect document sizes and the delays to fetch documents into the cache [1, 4, 5, 24, 26], it is not a priori clear how to combine these metrics into a single cost function. In this section we provide a justification of the cost function used in LNC-R-W3-U by showing that it approximates the behavior of an optimal, off-line algorithm in a constrained model which assumes no external cache fragmentation.

For the purpose of the analysis, we assume that the references to documents in the cache are statistically independent and described by reference rates r_i, where r_i is a mean reference rate to document i. The delay to fetch document i into the cache is given by d_i and the size of document i is given by s_i. We assume that the cache employs a TTL-based cache consistency algorithm. The mean validation rate of document i is given by u_i and the delay to perform a validation check (i.e. the delay to send a conditional GET) is given by c_i.

A typical goal of a cache replacement algorithm is to minimize response time. Therefore, it should retain in cache documents which account for a large fraction of the communication delays. In other words, the cache replacement algorithm should aim to optimize the following expression

$$max \sum_{i \in I} (r_i \cdot d_i - u_i \cdot c_i) \tag{1}$$

subject to a constraint

$$\sum_{i \in I} s_i \leq S \tag{2}$$

where I describes the set of documents selected for caching and S is the total cache size.

In general, the goal of satisfying (1) is different from the goal of maximizing hit ratio (i.e. the fraction of requests satisfied from the cache), a typical objective of most cache replacement algorithms designed for database buffer or file system management. We define a similar metric called *delay savings ratio* (DSR) which is a fraction of communication delays saved by satisfying the requests from cache. The delay savings ratio is defined as

$$DSR = \frac{\sum_i (d_i \cdot h_i - c_i \cdot v_i)}{\sum_i d_i \cdot f_i} \tag{3}$$

where h_i is the number of references to document i which were satisfied from the cache, f_i is the total number of references to document i and v_i is the number of validations performed on document i.

The problem defined by (1) and (2) is equivalent to the knapsack problem, which is NP-hard [13]. Consequently, there is no known efficient algorithm for solving the problem. However, if we assume that sizes of cached documents are relatively small when compared with the total cache size S, and thus it is always possible to utilize almost the entire cache space, then the solution space can be restricted to sets of documents I satisfying:

$$\sum_{i \in I} s_i = S \tag{4}$$

and we will show that the optimal solution can be found by a simple greedy off-line algorithm.

The optimal algorithm, Optim, assigns to each document a cost function called *profit* defined as

$$profit_i = \frac{r_i \cdot d_i - u_i \cdot c_i}{s_i} \tag{5}$$

In order to determine (I^*), the best set of documents to cache, the Optim algorithm sorts all documents based on their profit and selects for caching the most profitable documents until the entire cache space is exhausted. We assume that the cache fragmentation is minimal, so (4) holds at that time. We show that I^*, the set of documents to cache found by Optim, satisfies the objective function defined in (1).

Theorem 1 *Among all sets of documents satisfying (4), the Optim algorithm finds the one which satisfies (1).*

Proof: Given in the Appendix. □

3.2 LNC-R-W3-U Algorithm

The LNC-R-W3-U cache replacement algorithm is a cost-based greedy algorithm. The algorithm selects for replacement documents with least cost until a sufficient space has been freed. The cost function is directly based on profit notion defined in (5). Consequently, in a steady state the on-line algorithm should approximate the off-line algorithm Optim described in the previous section. We describe below the procedures for estimating the key parameters used in the evaluation of the profit function. The LNC-R-W3-U cache consistency algorithm is a TTL-based algorithm. If a document with an expired TTL is referenced and found in the cache, its content is validated by sending a conditional GET to the server owning the document. The procedure used to estimate the TTL values for each document is also described below.

In order to evaluate the profit of document i, the cache replacement algorithm must determine estimates of the following parameters:

- r_i - mean reference rate to document i
- d_i - mean delay to fetch document i into the cache
- u_i - mean validation rate for document i
- c_i - mean validation delay for document i
- s_i - size of document i

The estimation of document size s_i is straightforward. The mean delays to fetch and validate a document, d_i and c_i, are estimated by using two sliding windows of the last K corresponding delays measured in the past. K is one of the fine-tuning "knobs" of LNC-R-W3-U. We discuss its selection in Section 5.

The mean reference rate to document i, r_i, can be estimated in two ways. First, it can be estimated from a sliding window of last K reference times. However, unlike the estimates of fetch and validation delays, the reference rate must be "aged" in the absence of references to a document. Therefore, the mean reference rate is estimated as:

$$r_i = \frac{K}{t - t_K} \tag{6}$$

where t is the current time and t_K is the time of the oldest reference in the sliding window. Consequently, the estimate decreases even when the document is not referenced. The mean reference rate can be also estimated using the time of last reference (i.e. setting $K = 1$) as in the LRU cache replacement algorithm. However, it has been pointed out in [21] that LRU performs inadequately in presence of bursty workloads or workloads consisting of multiple classes with different characteristics - a workload likely to appear on the Web. In Section 5 we experimentally evaluate the benefits of setting $K > 1$.

Whenever fewer than K reference times are available for document i, the mean reference rate r_i is estimated using the maximal number of available samples. However, such an estimate is statistically less reliable and thus the documents having fewer reference samples are more likely to be selected for replacement by LNC-R-W3-U. In particular, the LNC-R-W3-U algorithm first considers for replacement all documents having just one reference sample in increasing profit order, then all documents with two reference samples in increasing profit order, etc. until sufficient cache space has been freed (see Figure 1).

Another way to estimate the mean reference rate r_i is based on the document's size s_i. Several studies of Web reference patterns show that Web clients exhibit a strong preference for accessing small documents [10, 14, 20]. Consequently, knowing the document's size gives us some information about how frequently it will be referenced. In particular, it was shown in [10] that given a document of size s_i, the estimate of its mean reference rate can be expressed with a high confidence as

$$r_i = \frac{c}{s_i^b} \tag{7}$$

$b = 1.66$ and c is a constant (not given in [10]).

To increase the precision of reference rate estimate, we combine the two estimates into a single metric defined as

$$r_i = \frac{K}{(t - t_K) \cdot s_i^b} \tag{8}$$

Since our trace exhibited different characteristics from those reported in [10], we experimentally determine the best setting of parameter b in Section 5.

In absence of the Expires headers in the response to a request for document i, the mean validation rate u_i is calculated from a sliding window of last K distinct Last-Modified timestamps as

$$u_i = \frac{K}{t_r - tu_K} \tag{9}$$

where t_r is the time when the latest version of document i was received by the proxy cache and tu_K is the Kth most recent distinct Last-Modified timestamp of document i (i.e. the oldest available distinct Last-Modified). We assume that the update rates of documents are stable. Therefore, it should suffice to validate each document with the same rate as it was updated in the past.

If, on the other hand, the Expires header is provided, LNC-R-W3-U uses the server provided information to determine validation rate, rather than trying to estimate it from the past modification timestamps. The validation rate is calculated in a manner similar to (9), but using the Expires timestamps instead of Last-Modified timestamps whenever possible. The most recent Expires timestamp substitutes t_r in (9), the Kth most recent Expires timestamp substitutes tu_K in (9).

The LNC-R-W3-U consistency algorithm sets TTL for a newly received document i as either

$$TTL_i = \frac{1}{u_i} \tag{10}$$

if the Expires timestamp is not available, or

$$TTL_i = Expires - t_r \tag{11}$$

otherwise.

Whenever a referenced document i has been cached longer than TTL_i units, the consistency algorithm validates the document by sending a conditional GET to the server specified in the document's URL. Whenever a new version of document i is received, LNC-R-W3-U updates the sliding windows containing the last K distinct Last-Modified timestamps and the last K validation delays and recalculates c_i, u_i and TTL_i. The LNC-R-W3-U cache consistency algorithm is similar to the TTL-based algorithms used in other proxy caches, however its TTL estimates are more accurate than those based only on the most recent Last-Updated timestamp. The effects of such an estimate on the fraction of stale documents in the cache is studied in Section 5. The pseudo-code of LNC-R-W3-U is shown in Figure 1.

4 Implementation

We have integrated the LNC-R-W3-U cache management library with the Apache 1.2.6 code. We describe some of the time and space considerations that we have incorporated into our implementation.

4.1 Time Efficiency

The applicability of the LNC-R-W3-U algorithm depends to a large extent on the time efficiency of the cache replacement procedure. In general, cost based algorithms must keep the metadata related to the cached documents sorted on the cost. In our implementation the metadata records contain for each cached document the following information: size, profit, URL of document, four sliding windows (to be explained in more detail in the next subsection) and local file name. Consequently,

```
  $t$       : time when document $i$ is requested
  $avail$   : available free space in cache

case
  document $i$ is in cache:        // cache consistency check
    $t_r$ : time when a new version of document $i$ was cached
    $TTL_i$ : time-to-live of document $i$
    if ( $TTL_i < t - t_r$ ) {          // $TTL_i$ expires
      HTTP Conditional GET request to the server
      update $c_i$ : mean validation delay to perform Conditional GET(document $i$)
      update $u_i$ : mean validation rate of document $i$
      update $r_i$ : mean reference rate of document $i$
    }
    else {                              // $TTL_i$ not expires
      update $r_i$
    }

  document $i$ is not in cache:  // cache replacement
    HTTP GET request to the server
    $s_i$ : size of document $i$, $d_i$ : mean delay to fetch document $i$ into cache
    $u_i$ : mean validation rate of document $i$

    if ( document $i$ has Expires timestamp)
      $TTL_i$ = Expires - $t_r$
    else
      $TTL_i = \frac{1}{u_i}$

    if ($avail \geq s_i$) {               // enough space
      cache document $i$ and update $r_i$
    }
    else {                              // replacement required
      for $j = 1$ to $K$
        $D_j$ = list of documents with exactly $j$ reference samples in
            increasing profit (defined in (5)) order
      $D$ = list of documents arranged in order $D_1 < D_2 < \ldots < D_K$
      $C$ = minimal prefix of $D$ such that $\sum_l s_l \geq s_i$
      evict $C$ out of cache and cache document $i$ and update $r_i$
    }
```

Figure 1: Pseudo-code of LNC-R-W3-U algorithm

a naive implementation of the algorithm would have $O(n \cdot log\ n)$ time complexity, with n being the number of cached documents, as opposed to e.g. LRU with only $O(1)$ time complexity. Obviously, the metadata does not have to be completely sorted because the cache replacement algorithm needs to find only the document with the least profit. Consequently, by organizing the metadata as a heap, we can reduce the time complexity down to $O(log\ n)$ for each application of the cache replacement[1]. However, for large cache this may still be an excessive overhead when compared to the LRU replacement.

However, we found that each time Apache invokes the replacement algorithm, it sorts all metadata based on the time when the documents expire (as given by their TTLs). We assume that such an implementation was selected due to the concurrency problems as Apache proxy spawns multiple processes. To avoid concurrency conflicts on the document metadata, each Apache process writes all metadata to disk after every replacement and the metadata is read and sorted again before every replacement invocation. Therefore, to incorporate the LNC-R-W3-U cache replacement in Apache, we only had to change the criteria which is used for sorting the metadata to the profit metric. We did not measure any observable increase in the sort time for cache sizes containing between 10,000 and 200,000 documents. Consequently, adding the LNC-R-W3-U to Apache does not slow down the Apache proxy.

Our next question was whether a heap-based cache organization would improve the performance and whether the improvement is significant when compared with the average cache access time: $(hit\_rate * hit\_time + (1 - hit\_rate) * miss\_time)$. We conducted an experiment with cache sizes 100 MB and 1GB and average file sizes 5 KB and 10 KB which are typical settings reported elsewhere [10, 12, 19]. In each experiment we measured the time to sort the entire metadata using `qsort()`, the time to build a heap on the metadata and the time to rebuild the heap after the document with minimal profit is removed. First, we found that the cost of sorting the metadata is indeed significant

[1]The heap can be reorganized in $O(log\ n)$ steps since at most a constant number of documents with minimal profit need to removed at each invocation

because it is comparable to the average time of a single cache access. Second, we found that, on average, it is approximately 6 times faster to build the heap than sort the entire cache. In our implementation the heap is built only once upon the startup of the proxy when the cache becomes full the first time. Subsequently, the heap needs to be only rebuilt after a removal of the document with minimal profit. The rebuild can be achieved in the order of 10^5 times faster than the sort! Consequently, the heap-based implementation indeed significantly improves cache performance. To resolve the aforementioned concurrency problems, we implemented the heap structure in a shared memory and protected it with latches. The experimental results can be found in Figure 2.

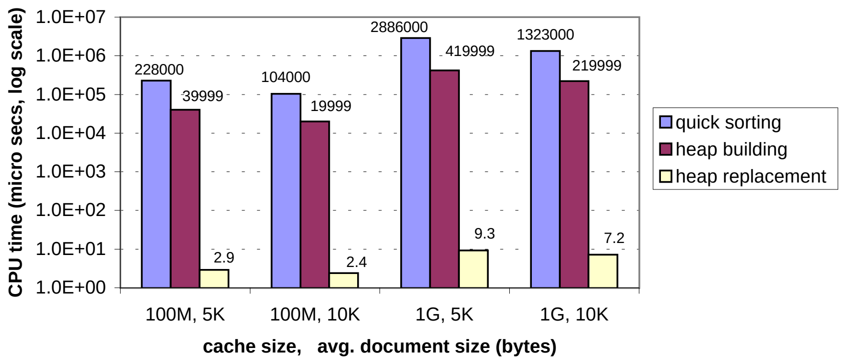


Figure 2: The performance of `qsort()` and heap-based cache replacement

It may appear that the heap structure should be built from scratch each time the cache replacement is invoked because the profit function depends on the current time. In particular, the mean reference rate estimate from (8) includes the value of current time to age documents which are not referenced. However, in our implementation, we update the reference rate of a document only if the document is accessed. In addition, we periodically use an "aging daemon" which generates dummy references to all documents to age their reference rate estimates. Dummy references are discarded once a "real" reference to the document is received. This implies that all these actions, i.e., a document access or an aging daemon, require only reorganizations operations on the heap which are cheaper than entire rebuilding operations.

4.2 Space Efficiency

The space efficiency of the LNC-R-W3-U algorithm is also an important issue because it needs to maintain substantial bookkeeping with each document in order to evaluate its profit. In particular, the LNC-R-W3-U algorithm needs to keep with every document four sliding windows: one with the last K reference times, one with the last K distinct Last-Modified times, one with the last K delays to fetch a document and one with the last K delays to perform a validation check for the document. Storage of timestamps with a precision of seconds requires 4 bytes in most UNIX implementations. Consequently, the number of bytes necessary for the bookkeeping of a single document is $16 \cdot K$. For the optimal value of K determined in Section 5, $K = 3$, the resulting overhead is 48 bytes per document.

Although 48 bytes overhead is relatively small compared to the average document size of 2.6 KB found in our trace and identical to the average URL string size which must also be cached with every document, it is possible to use standard compression techniques to the sliding window representation in order to reduce their space overhead by almost a factor of four. Since most documents do not require more then 256 seconds to fetch or validate, it is possible to use a single byte to store the document fetch and validate delays. The reference time and validation time sliding windows must keep the full timestamp in order to be able to take a difference with an absolute time in (8) and (9). However, it is possible to keep the full timestamp only for the oldest sample in each window and encode the remaining $K - 1$ samples as differences with the respect to the oldest timestamp. Therefore, we can reduce the sliding window overhead down to $2 \cdot (4 + (K - 1)) + 2 \cdot K = 4 \cdot K + 6$. For $K = 3$ we can reduce the overhead to 18 bytes per document. We are currently in the process of implementing the compression techniques described above and evaluating its impact on the cache performance.

4.3 Metadata Garbage Collection

A straightforward implementation of the LNC-R-W3-U algorithm may, however, lead to starvation of documents having fewer than K reference samples. Since the reference rate estimates based on fewer than K samples are less reliable, the LNC-R-W3-U cache replacement algorithm preferentially evicts documents with fewer references. However, if the reference samples are discarded when the corresponding document is evicted from the cache, then they must be collected again from scratch the next time the document is cached. However, since the document is again likely to be selected for replacement, it may be impossible to collect the necessary K samples to cache it permanently, irrespective of its reference rate. To prevent the starvation, LNC-R-W3-U retains all metadata associated with a document (including all sliding windows) even after the document has been evicted from the cache.

The metadata is garbage collected using the following rule:

> The metadata associated with an evicted document is garbage collected from the cache whenever the profit calculated using the metadata is smaller than the least profit among all cached documents.

Clearly, retaining metadata related to documents with profits smaller than the least profit among all cached documents does not lead to any performance improvement because such documents would immediately become candidates for replacement, should they be cached.

5 Experimental Evaluation

5.1 Experimental Setup

5.1.1 Trace Characteristics

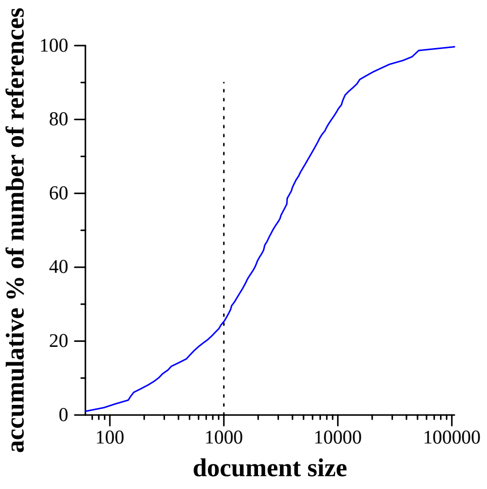
We evaluated the performance of LNC-R-W3-U on a client trace collected at Northwestern University. The trace represents a seven day snapshot (November 96) of requests generated by clients on approximately 60 PC's in a public lab at Northwestern University. The trace contains about 20K requests. Browser cache hits and non-cacheable requests (e.g. URL's containing "bin" or "cgi-bin") were filtered out from the trace. The browsers were temporarily adjusted so that all requests were re-directed to a proxy cache where for each referenced URL, we recorded the time when the request for the document arrives at the proxy, the time when the response to the request is received, the time when proxy issued conditional GET for the document, the time when response for the conditional GET was received, the size of the document and the values of Expires and Last-Modified headers in the received response (if available).

To gain more insight into the nature of the client requests' characteristics, we captured some of the statistical properties of our traces. We concentrated on four aspects:

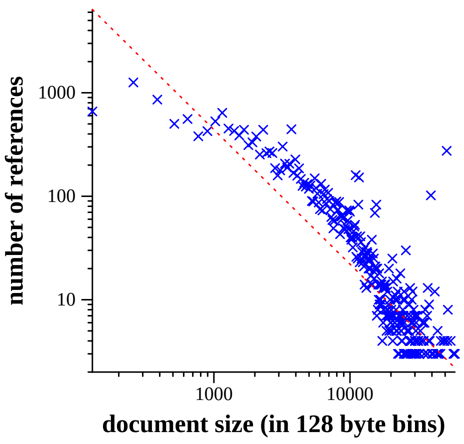
- the dependence of reference rate on document size
- the correlation between the delay to fetch a document to cache and the size of the document
- the fraction of requests received with Expires and Last-Modified headers
- the fraction of updated documents

Previously published trace analyses [10, 14, 20] show that small files are much more frequently referenced than large files. Our traces exhibit the same characteristic as Figure 3a indicates. For example, 25% of all client requests are for documents smaller than 1KB.

As discussed in Section 3, other researchers have observed a hyperbolic dependence between document size and reference rate. In order to determine the skew of the hyperbola given by parameter b in equation (7) we found the least-squares fit for a hyperbola on the trace data, which determines the best value of b as 1.30. The comparison between the best-fit line and the real trace data is shown in Figure 3b.



a. Request size distribution(log scale)



b. Least squares fit(log scale)

Figure 3: Trace characteristics

The correlation between the document size and the delay to fetch the document is defined as:

$$Cor(s,d) = \frac{Cov(s,d)}{\sqrt{Var(s) \cdot Var(d)}} \tag{12}$$

where $Cov(s,d)$ is the covariance between size and delay, $Var(s)$ is the variance of size and $Var(d)$ is the variance of delay. $Cor(s,d)$ shows whether the delay to fetch a document to cache varies across documents of similar size. $Cor(s,d) = 1$ indicates that delay is linearly dependent on size. Consequently, there is no need for delay-sensitive caching as delay can be always determined from size. On the other hand, $Cor(s,d) = 0$ indicates that there is no relationship between size and delay and thus delay-sensitive caching is necessary. If most of the clients access primarily local documents, we expect $Cor(s,d)$ to be close to 1. On the other hand, if the clients access a significant fraction of remote documents, we expect $Cor(s,d)$ to be close to 0. We measured the value $Cor(s,d)$ on our trace as 0.2145, which is relatively low. Therefore, delay-sensitive caching is indeed important.

We found that approximately 89% of documents in the trace contain Last-Modified timestamp, approximately 7% of documents contain Expires timestamp and 90% of documents contain either Last-Modified or Expires timestamp.

We regard a document as updated if its Last-Modified timestamp changed at least once within the 7 day period studied in our trace. If a document was updated within the 7 day period, but was not subsequently referenced, we have no means of ascertaining that the update occurred and thus such updates are not captured in our trace. We found that approximately 6% of all documents were updated and less than 2% out of these changed every day. (Figure 4).

Our results confirm the generally held belief that WWW is read-mostly environment and they are in accord with the analysis of server-based traces reporting daily update rates in the range 0.5% - 2% [3, 17]. It is possible that a larger fraction of documents is updated if one considers also dynamic CGI documents. However, since dynamic documents are not cacheable, such updates are irrelevant for the results presented here.

documents with Last-Modified	89%
documents with Expires	7%
documents with Last-Modified or Expires	90%
updated documents	6%

Figure 4: Update Trace Characteristics

5.1.2 Performance Metrics and Yardsticks

The *delay saving ratio* (DSR) defined in Section 3 is the primary cache performance metric in all experiments. We also use *hit ratio* (HR) as a secondary performance metric. The hit ratio is defined as a fraction of requests which were satisfied from cache.

We use *staleness ratio* (SR) as the primary cache consistency metric. The staleness ratio is defined as a fraction of cache hits which return stale documents. We say that a document returned by the cache to client is stale if the trace record corresponding to the request has a later Last-Modified timestamp than the time when the document was brought to the cache.

We compared the performance of LNC-R-W3-U against plain LRU, LRU-MIN [1], and LNC-R-W3 [24]. LNC-R-W3 corresponds to our algorithm without the cache consistency component. Both LRU and LRU-MIN use a simple TTL-based cache consistency algorithm which sets TTL to either `Expires - now` if the Expires header is available or `now - Last-Modified` otherwise. These algorithms correspond to the algorithms implemented in most proxy caches.

Similarly to LNC-R-W3-U, LRU-MIN also exploits the preference of Web clients for accessing small documents. However, unlike LNC-R-W3-U, LRU-MIN does not consider the delays to fetch documents to the cache and estimates reference rate to each document using only the time of last reference. Whenever LRU-MIN needs to free space for a new document i of size s_i, it first attempts to select for replacement any document larger than s_i (the documents are searched in the LRU order). If no document has been found, LRU-MIN considers the lists of documents larger than $s_i/2$, $s_i/4$, etc., in LRU order until enough space has been freed.

5.2 Experimental results

5.2.1 Infinite cache

We ran experiments with unlimited cache size in order to study the potential of caching on the workloads in our trace. As the results in Figure 5 show, our trace exhibits reference locality (given by hit ratios) similar to other previously published results [1, 14, 12]. The results in Figure 5 also reveal that local documents exhibit a higher degree of reference locality than remote documents, because the delay-saving ratios are lower than the corresponding hit ratios. Again, this is consistent with similar findings in [10].

DSR	HR	cache size
0.427	0.464	18MB

Figure 5: Performance with infinite cache

5.2.2 Fine-tuning LNC-R-W3-U

A common problem with many algorithms (including many commercial systems) is the existence of "magic" fine-tuning knobs. When set properly, the algorithms perform better than simpler alternatives with fewer or no knobs, but poor setting may lead to a disastrous performance. Typically, the user is granted the "privilege" to determine the optimal setting of such parameters given the characteristics of his/her environment. Ideally, such parameters should be completely eliminated from system design by allowing the systems to self-tune the parameters.

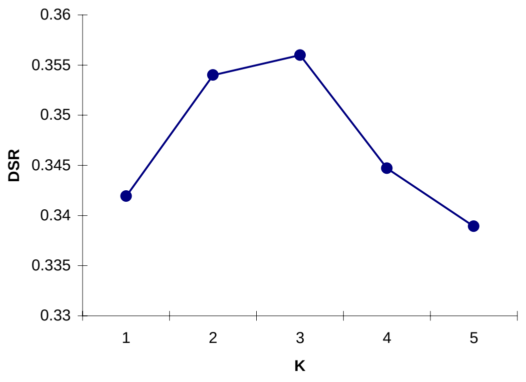
The LNC-R-W3-U algorithm has two such parameters: the size of sliding window K and the skew of the dependence between reference rate and document size b. Although our implementation of LNC-R-W3-U is not self-tuning, we at least provide criteria for the users to set the two parameters.

Selection of K

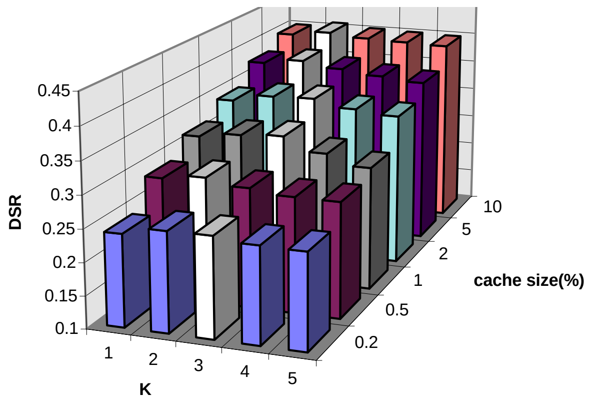
Increasing the value of K improves the reliability of reference rate estimates. Clearly, the burstier the workload the larger value of K should be selected. On the other hand, large values of K result in higher spatial overhead to store the reference samples. The transition from $K = 1$ to $K = 2$ is particularly sharp, since LNC-R-W3-U with $K = 1$ does not need to retain any reference samples after eviction of corresponding documents. Thus we expect the best performance for small values of K, larger than 1. The results (Figure 6) confirm our expectations. With cache size 2% of the total size of all documents, setting K=3 leads to best performance as shown in Figure 6a. The improvement of delay saving ratio relative to K=1 is 4.1%. Figure 6b confirms that also for other cache sizes the best performance is achieved with small values of K (marked by white bar). Therefore, we conjecture that K should be set to 2 or 3 to obtain the best performance.

Selection of b

As explained in Section 3, parameter b determines the skew of dependence of reference rate on document size. The higher the value of b, the stronger the preference of clients to access small documents. In Section 5.1.1 we determined $b = 1.30$ as the best fit for the data in our trace. However, since we found the best fit of Section 5.1.1 relatively noisy, we validated the prediction also experimentally. Figure 7a confirms the prediction for cache size 2% of the total size of all documents. White bars in Figure 7b show the optimal values of parameter b for other cache sizes. In most cases the measured optimal values are close to the predicted optimum $b = 1.30$. Because other Web client traces exhibit similar skew of dependence of reference rate on document size [10], we conjecture that for best performance on most web workloads b should be set between 1 and 2.



a. K settings on 2.0 % cache size



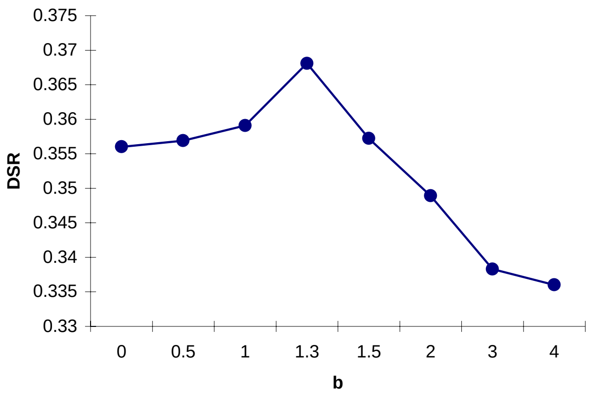
b. K on all cache sizes

Figure 6: Impact of K on DSR

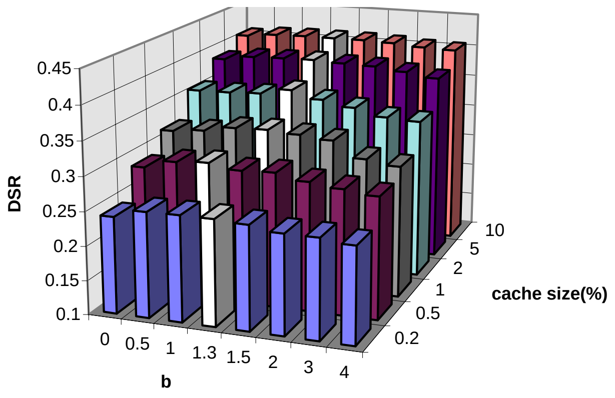
5.2.3 Cache Performance Comparison

We compared the performance of LNC-R-W3-U with LRU, LRU-MIN and LNC-R-W3. For both LNC-R-W3-U and LNC-R-W3 we used the optimal setting of b =1.3 and K=3. As Figure 8 indicates, LNC-R-W3-U provides consistently better performance than LRU and LRU-MIN for all cache sizes. And its performance is very close to that of LNC-R-W3 which does not make any effort to enforce consistency.

In terms of delay-savings ratio, LNC-R-W3-U gives on average 38.3% improvement over LRU and 9.8% improvement over LRU-MIN. The maximal improvement over LRU and LRU-MIN is



a. b settings on 2% cache size

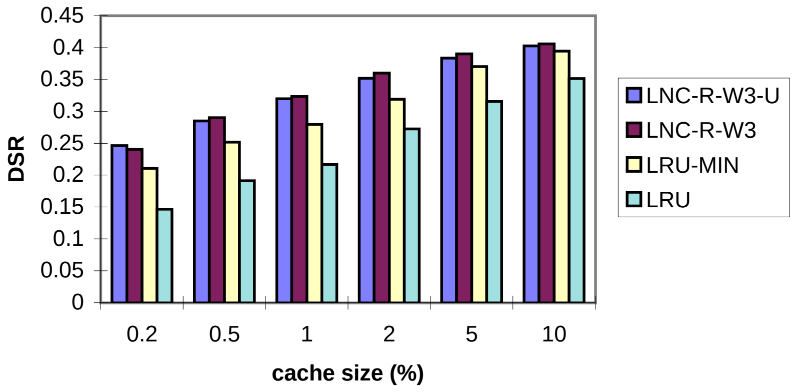


b. b on different cache size

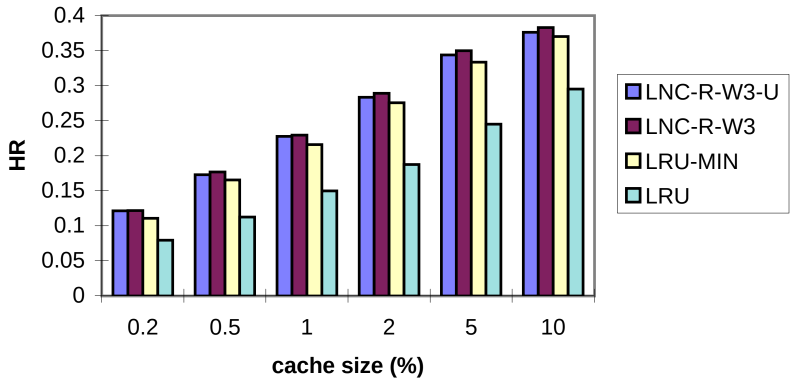
Figure 7: Impact of b settings on DSR

67.8% and 17.4% for cache size 2.0% and 1.0% respectively. On average, the delay-savings ratio of LNC-R-W3-U is only 1.0% below the DSR of LNC-R-W3. In the worst case, DSR of LNC-R-W3-U is 2.5% below the DSR of LNC-R-W3 for cache size 10.0%. The delay-savings ratio comparison can be found in Figure 8a.

Although LNC-R-W3-U is not designed to maximize the cache hit ratio, it still provides an improvement over LRU and LRU-MIN as shown in Figure 8b. In particular, the average improvement is 43.4% over LRU and 4.5% over LRU-MIN. The hit ratio of LNC-R-W3-U provides even closer to the hit ratio of LNC-R-W3; on average 0.4% and no more than 0.6% below the hit ratio of LNC-R-W3.



a. Delay-Savings Ratio



b. Hit Ratio

Figure 8: Performance comparison

In addition to improving performance of the cache, the LNC-R-W3-U algorithm also significantly improves its consistency. On average, LNC-R-W3-U achieves a staleness ratio which is by factor of 3.2 better than the SR of LNC-R-W3, in the worst case it improves SR of LNC-R-W3 by factor of 1.9 when cache size is 0.5%. LNC-R-W3-U also improves the stale ratios of both LRU and LRU-MIN. On average, LNC-R-W3-U achieves a staleness ratio which is 47.8% better than the SR of LRU and 54% better than the SR of LRU-MIN. In the worst case, it improves SR of LRU by 10.2% when cache size is 0.5% and improves SR of LRU-MIN by 8% when cache size is 2.0%. The staleness ratio comparison of all four algorithms can be found in Figure 9.

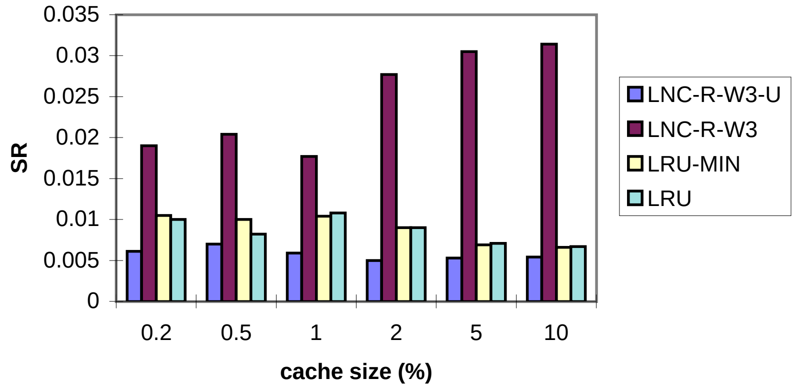


Figure 9: Staleness Ratio

6 Conclusion

We have described the design and implementation of a new unified procedure for cache maintenance, LNC-R-W3-U, which incorporates components for cache replacement and consistency maintenance for Web proxies. We see three main contributions of our work:

- We demonstrated that it is indeed important to consider the communication delays in cache replacement. We show that the LNC-R-W3-U algorithm improves the performance (delay saving ratio) on average by 38.3% when compared to LRU and 9.8% when compared to LRU-MIN.
- We showed the importance of considering cache replacement and consistency algorithms which cooperate. The integrated solution improves cache staleness on average by 47.8% when compared to LRU and 54% when compared to LRU-MIN.
- We demonstrated that cost-based cache replacement algorithms can be implemented in an industrial-strength cache proxy with no slowdown.

In addition, we justify the choice of our cost function theoretically and, in contrast to other algorithms, we introduce only two fine-tuning knobs for which we provide default setting criteria.

In the near future we plan to finalize testing and full integration of LNC-R-W3-U library with the Apache 1.2.6 code base and make the source code available to the public. Our procedure can be easily integrated any commercial cache proxy since it does not require any extensions to the HTML protocol or to any changes to the servers.

Our experiments have indicated that the hit ratio cannot be improved over 47% even with an infinite cache, an observation which is consistent with that of other researchers in this field. On alternative way to increase the hit ratio is by using replicated servers [22]. Replicated servers are more complex to manage and they do require changes to the HTML or the server code; on the other hand they also bring additional advantages for reliability purposes. We are currently studying the various tradeoffs involved between proxy caches and replicated servers.

References

[1] M. Abrams, C. Standridge, G. Abdulla, S. Williams, E. Fox, "Caching proxies: Limitations and potentials", *Proc. 4th International World Wide Web Conference*, 1995.

[2] Apache 1.2.6 HTTP server documentation, available at `http://www.apache.org/`, 1998.

[3] A. Bestavros, "Speculative Data Dissemination and Service", *Proc. 12th International Conference on Data Engineering*, 1996.

[4] J. Bolot and P. Hoschka, "Performance Engineering of the World Wide Web: Application to Dimensioning and Cache Design", *Proc. 5th International World Wide Web Conference*, 1996.

[5] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms", *Proc. USENIX Symposium on Internet Technologies and Systems*, 1997.

[6] V. Cate, "Alex- a global file system", *Proc. 1992 USENIX File System Workshop*, 1992.

[7] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, K. Worrell, "A hierarchical Internet object cache", *Proc. USENIX 1996 Annual Technical Conference*, also available at `http://excalibur.usc.edu/cache-html/cache.html`.

[8] E. Coffman and P. Denning, *Operating Systems Theory*, Prentice-Hall, 1973.

[9] A. Cormack, "Web Caching", available at `http://www.nisc.ac.uk/education/jisc/acn/caching.html`, 1996.

[10] C. Cunha, A. Bestavros, M. Crovella, "Characteristics of WWW Client-based Traces", *Technical Report TR-95-010*, Boston University, Apr. 1995.

[11] A. Dingle and T. Partl, "Web Cache Coherence", *Proc. 5th International World Wide Web Conference*, 1996.

[12] B. Duska, D. Marwood, M. Feeley, "The Measured Access Characteristics of World-Wide-Web Client Proxy Caches", *Proc. USENIX Symposium on Internet Technologies and Systems*, 1997.

[13] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[14] S. Glassman, "A caching relay for the World Wide Web", *Computer Networks and ISDN system*, Vol 27, 1994.

[15] Jigsaw 2.0 HTTP server documentation, available at `http://www.w3c.org/Jigsaw/`, 1998.

[16] B. Krishnamurthy and C. Wills, "Piggyback Server Invalidation for Proxy Cache Coherency", *Proc. the 7th International World Wide Web Conference*, 1998.

[17] J. Gwertzman and M. Seltzer, "World-Wide Cache Consistency", *Proc. USENIX 1996 Annual Technical Conference*, 1996.

[18] C. Liu and P. Cao, "Maintaining Strong Cache Consistency in the World-Wide Web", *Proc. 17th International Conference on Distributed Computing Systems*, 1997.

[19] S. Manley and M. Seltzer, "Web Facts and Fantasy", *Proc. USENIX Symposium on Internet Technologies and Systems*, 1997.

[20] Daniel O'Callaghan, "A Central Caching Proxy Server for WWW users at the University of Melbourne", available at `http://www.its.unimelb.edu.au:801/papers/AW12-02/`.

[21] E. O'Neil, P. O'Neil, G. Weikum, "The LRU-K page replacement algorithm for database disk buffering", *Proc. ACM SIGMOD International Conference on Management of Data*, 1993.

[22] M. Sayal, Y. Breitbart, P. Scheuermann, R. Vingralek, "Selection Algorithms for Replicated Web Servers", *Proc. Workshop on Internet Server Performance*, 1998.

[23] P. Scheuermann, J. Shim, R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager", *Proc. 22nd International Conference on Very Large DataBases*, 1996.

[24] P. Scheuermann, J. Shim, R. Vingralek, "A Case for Delay-Conscious Caching of Web Documents", *Proc. 6th International World Wide Web Conference*, 1997.

[25] Squid 1.1.21 Internet Object Cache Documentation, available at `http://squid.nlanr.net/Squid/`, 1998.

[26] R. Wooster and M. Abrams, "Proxy Caching That Estimates Page Load Delays", *Proc. 6th International World Wide Web Conference*, 1997.

Appendix

We provide a proof of theorem 1 from Section 3.

Theorem 2 *Among all sets of documents satisfying (4), the Optim algorithm finds the one which satisfies (1).*

Proof: Let $I \neq I^*$ be an arbitrary subset of documents satisfying (4). We will show that $\sum_{i \in I} r_i \cdot d_i - u_i \cdot c_i \leq \sum_{i \in I^*} r_i \cdot d_i - u_i \cdot c_i$. Since Optim selects retrieved sets with maximal profit, it follows that

$$\sum_{i \in I} \frac{r_i \cdot d_i - u_i \cdot c_i}{s_i} \leq \sum_{i \in I^*} \frac{r_i \cdot d_i - u_i \cdot c_i}{s_i} \tag{13}$$

We can assume that $I^* \cap I = \emptyset$. If not, then the intersecting elements can be eliminated from both sets while preserving (13). We define

$$\frac{r_{min} \cdot d_{min} - u_{min} \cdot c_{min}}{s_{min}} = min_{i \in I^*} \frac{r_i \cdot d_i - u_i \cdot c_i}{s_i} \tag{14}$$

$$\frac{r_{max} \cdot d_{max} - u_{max} \cdot c_{max}}{s_{max}} = max_{i \in I} \frac{r_i \cdot d_i - u_i \cdot c_i}{s_i} \tag{15}$$

Since I^* contains retrieved set references with maximal $\frac{r_i \cdot d_i - u_i \cdot c_i}{s_i}$ and $I^* \cap I = \emptyset$, it must be true that $\frac{r_{min} \cdot d_{min} - u_{min} \cdot c_{min}}{s_{min}} \geq \frac{r_{max} \cdot d_{max} - u_{max} \cdot c_{max}}{s_{max}}$. Consequently,

$$\sum_{i \in I^*} r_i \cdot d_i - u_i \cdot c_i \geq \frac{r_{min} \cdot d_{min} - u_{min} \cdot c_{min}}{s_{min}} \cdot S \geq \frac{r_{max} \cdot d_{max} - u_{max} \cdot c_{max}}{s_{max}} \cdot S \geq \sum_{i \in I} r_i \cdot d_i - u_i \cdot c_i \tag{16}$$

Therefore, we have shown that the document set I^* selected by Optim indeed satisfies (1). □