Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance

Li Fan, Pei Cao and Wei Lin
Department of Computer Science
University of Wisconsin-Madison
{*lfan,cao,weilin*}*@cs.wisc.edu*

Quinn Jacobson
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
qjacobso@ece.wisc.edu

Abstract

The majority of the Internet population access the World Wide Web via dial-up modem connections. Studies have shown that the limited modem bandwidth is the main contributor to latency perceived by users. In this paper, we investigate one approach to reduce latency: prefetching between caching proxies and browsers. The approach relies on the proxy to predict which cached documents a user might reference next, and takes advantage of the idle time between user requests to push or pull the documents to the user. Using traces of modem Web accesses, we evaluate the potential of the technique at reducing client latency, examine the design of prediction algorithms, and investigate their performance varying the parameters and implementation concerns.

Our results show that prefetching combined with large browser cache and delta-compression can reduce client latency up to 23.4%. The reduction is achieved using the Prediction-by-Partial-Matching (PPM) algorithm, whose accuracy ranges from 40% to 73% depending on its parameters, and which generates 1% to 15% extra traffic on the modem links. A perfect predictor can increase the latency reduction to 28.5%, whereas without prefetching, large browser cache and delta-compression can only reduce latency by 14.4%. Depending on the desired properties of the algorithm, several configurations for PPM can be best choices. Among several attractive simplifications of the scheme, some do more harm than others; in particular, it is important for the predictor to observe all accesses made by users, including browser cache hits.

1 Introduction

Ever since World-Wide Web becomes World-Wide Wait, reducing client latency has been one of the primary concerns of the Internet R&D community. For a majority of the Internet population, who access the World-Wide Web through modem connections, the low modem bandwidth is a primary contributor to client latency. Studies of commercial ISP traces show that even if a caching proxy is employed and has the best possible performance, the client latency can only be reduced by 3% to 4% due to modem connections [4]. Other studies have shown that when image distillation is used to reduce object sizes, client latency can be improved by a factor of 4 to 5, further demonstrating that slow modem links are the bottlenecks.

In this paper, we investigate one technique to reduce latency for modem users: prefetching between caching proxies and clients. The proxy, being exposed to the Web accesses of multiple users, can often predict what objects a user might access next. The modem link to the user often has idle periods as the user is reading the current Web document. If the objects are cached at the proxy, the proxy can utilize the idle periods to push them to the user, or to have the browser pull them. Since the proxy only initiates prefetches for objects in its cache, there is no extra Internet traffic. We called this technique *proxy-initiated prefetching*.

We first study the potential of proxy-initiated prefetching at reducing client latency. In addition to prefetching, there are other techniques that also reduce client latency, including large browser cache, delta compression (transferring only the differences for modified objects) [19], and application-level compression of HTML documents. Using traces of modem users' Web accesses, we study the benefits of each technique and their combinations. We find that merely increasing the browser cache size would only reduce client latency by 4%. Combining it with delta compression reduces client latency by 14.6%. Assuming perfect prediction, proxy-initiated prefetching combined with the two techniques can reduce client latency by 28.6%.

We then study the performance of realistic prediction algorithms. Unlike a Web server, the proxy does not observe accesses to a server's documents by the whole Internet population; rather, it observes accesses to the whole Web by a fixed population. Thus, the prediction capability of a proxy is different from that of Web servers. We investigate a class of prefix-based prediction algorithms called Prediction by Partial Matching (PPM). PPM essentially tries to predict the next l requests based on the past m accesses by the user, and limit the candidates by a probability threshold of t. With PPM predictors, prefetching combined with large browser cache and delta compression reduces client latency by up to 23.4%. Several (m, l, t) configurations can be best choices, depending on the criteria of the prediction algorithm.

During the implementation of the scheme we were tempted to make several simplifications; thus, we also study the impact of these "shortcuts" on the performance of the prediction algorithm. One shortcut is not notifying the proxy upon browser cache hits, that is, the predictor only sees browser cache misses. We find that for many configurations of PPM, this worsens the performance and prediction accuracy significantly and should not be adopted. Another shortcut is that the party performing the prefetching does not keep track of the content of the browser cache and thus may incur redundant transfers. We find that this shortcut increases the number of prefetch transfers significantly and should be avoided. Finally, our results show that limiting the size of the data structure of the predictor reduces the performance only moderately.

We have implemented proxy-initiated prefetching in CERN HTTPD proxy. The implementation is quite simple and the predictor increases the CPU time of the proxy by about 7%. Thus, we believe that proxy-initiated prefetching is a promising approach for alleviating World-Wide Wait for modem clients.

2 Proxy-Initiated Prefetching

Proxy-Initiated Prefetching relies on the proxy to make predictions, and either the proxy or the browser to perform the prefetch.

2.1 Overview

The basic assumptions behind proxy-initiated prefetching are:

- users have idle times between requests, because users often read some parts of one document before jumping to the next one;
- the proxy can predict which Web pages a user will access in the near future based on reference patterns observed from many users;
- the proxy has a cache that hold recently accessed Web pages.

The proxy can then either "push" the Web pages to the user's browser, or piggyback the predictions with regular responses to the browser and let the browser fetch the Web pages. There is no fundamental difference between the two schemes. Furthermore, only objects that are already in the proxy cache can be prefetched. Thus, the approach generates no wide-area network traffic.

The proxy maintains a "history structure." Every time the proxy services a request, it updates the history structure, establishing the connection between past accesses made by the same user and the current request. When the proxy detects that the connection to a user is idle, it uses the history structure to predict pages that the user might access next, checks which of the pages are in its cache, and generates a list of candidates ordered by their probabilities of access. The candidates are then pushed or fetched one by one into the browser cache. The party initiating the transfers is aware of the content of the browser cache and does not fetch objects that are already in the browser cache. The moment the user issues a new request, the prefetching is stopped and any partially fetched object is discarded at the browser end, unless the request is for the object that is being fetched. In addition, the proxy clears the list of candidate pages and recomputes a new one next time.

When the user issues a request, there are three possibilities: the requested object is in the browser cache, it is being prefetched, or it needs to be fetched through the proxy. To help proxy capture users' real accesses, the browser should notify the proxy about browser cache hits. The notification can be piggy-backed with regular requests.

With prefetching, the browser cache serves two purposes: caching recently accessed documents, and buffering prefetched documents. It must balance the resources devoted to the two carefully. Many schemes can be used; indeed there are many papers on this topic [5, 23]. In our simulations we assume a global Least-Recently-Used (LRU) replacement algorithm, treating prefetched objects as most-recently-used when they are put in the cache. The algorithm works fine when the browser cache is large and the accuracy of the prediction is high.

The scheme can be implemented without modification to the browser. A custom proxy located at the same machine as the browser can implement all of the interactions with the main proxy, as is described in Section 7 and in [17].

2.2 Performance Metrics

We are mainly interested in the following performance metrics:

- *Request Savings*: the number of times that a user request hits in the browser cache or the requested object is being prefetched, in percentages of the total number of user requests. We further separate the request savings into three categories:
  - prefetched: the object has been prefetched into the browser cache, and the user is accessing it for the first time.
  - cached: the object is in the browser cache, but this is not the first time that the user accesses it since it has been there.
  - partially prefetched: the object is still being sent from the proxy to the user.

  The request savings is essentially the browser cache hit ratio.
- *Latency Reduction*: the reduction in client latency, in percentages.
- *Wasted Bandwidth*: the sum of bytes that are prefetched but are not read by the client. That is, it is the extra bandwidth consumption caused by prefetching.

Request savings and latency reduction are the primary goals of proxy-initiated prefetching. Unlike in the wide-area networks, wasted bandwidth can be tolerated on modem lines (including DSL modem links), because modem lines would stay idle anyway. To most users who do not initiate other network transfers (such as ftp) while Web-surfing, wasted bandwidth has little effect.

3 Traces and the Simulator

We use the HTTP traces gathered from the University of California at Berkeley home dial-up populations from November 14, 1996 through November 19, 1996. For a detailed description of the trace see [10]. The traces are collected at a packet sniffing machine placed at the head-end of the UC Berkeley HomeIP modem bank. Thus, it corresponds to requests that would be seen by a proxy placed between the modem clients and the Internet.

The simulator uses the timing information in the traces to estimate latency seen by each modem client. The traces include, for each request, the time the client made the request (T_1), the time at which the first byte of the Web server response was seen at the head-end machine (T_2), and the time at which the last byte of the Web server response was seen at the head-end machine (T_3). For each response, the transfer from the Web server to the head-end machine and the transfer from the head-end machine to the modem client can be overlapped. Since the modem link is typically the bottleneck in the transfer [4], we estimate the client-seen latency to be $(T_2 - T_1)$ plus the time it takes to transfer the response over the modem link.

The modems in the traces are "2.4kb/s, 9.6 kb/s, 14.4 kb/s, or 28.8 kb/s wireline modems, or Metricom Ricochet (approximately 20-30 kb/s)" [10]. The traces unfortunately do not mark the modem speed for each user connection. Based on the time (late 1996) and the location (UC Berkeley) of the trace collection, we feel that 28.8kb/s modems are probably the majority. A separate study on AT&T WorldNet modem pools found that for 28.8kb/s modems, the sustained bandwidth is typically around 21kb/s [4]. Thus, *the simulator assumes that each modem link has a bandwidth of 21kb/s.* We recognize that this introduces errors in our results, and it is an ongoing effort to obtain traces that include more information.

Our simulator models contention of the modem link. If a client has multiple outstanding requests and the responses overlap each other, we do not calculate the latency for individual requests, but rather the total latency for the collection of requests. The total latency is $(T_2 - T_1)$ of the first response plus the sum of all responses divided by the modem bandwidth. The reason is two-fold. The traces do not give enough information on how the transfer of the responses are interleaved, making it hard to estimate when an individual response reaches the client. Furthermore, many concurrent requests are due to the browser using multiple connections to fetch embedded objects in an HTML document, and the total latency reflects how long the user waits for the whole document. The traces do not contain information on user aborts and thus we do not simulate them.

The simulator also models an "extended" browser cache for each client. The requests in the traces are in fact cache misses from the original browser caches, and we cannot reconstruct the user requests that those original browser caches see. Thus, for each client we simulate an extension to the original browser cache, which caches misses from the original cache (i.e. acting as a secondary cache) and buffers prefetched objects. We call this extension the browser cache in the rest of the paper.

The simulator assumes the existence of a proxy between the modem clients and the Internet. The proxy is assumed to have a disk cache size of 16GB (a moderate size for disk-based proxies), which for our traces results in no cache replacement. We do not simulate in detail disk transfer latency in the proxy, but rather assume a 10ms fixed delay in fetching a cached file from disk. We assume that the proxy has one disk and the disk transfers are serialized; in practice the proxy's disk arm is rarely a bottleneck because the modem links are so slow.

4 Reducing Client Latency

Besides prefetching, three other techniques also reduce client latency:

- Increase the size of browser caches. Browser caches typically have a default size of 5-8MB. Increasing the size of the cache increases the hit ratio and reduces modem traffic.
- Use *delta compression* [19] to transfer modified Web pages between the proxy and clients. That is, if an old copy of the modified page exists in the browser cache, the proxy only sends the difference between the latest version and the old version.
- Apply application-level compression to HTML pages. Studies [19, 20] have suggested that HTML texts can be first compressed, and then transferred from one end to another. HTTP 1.1 supports application-level compression via the "transfer-encoding" tag.

Using the traces, we first estimate the effectiveness of each technique and their combinations.

Table 1 shows the request savings and latency reductions under combinations of the four techniques. We investigate "perfect" prefetching, that is, there is an oracle that can tell us exactly which Web object a user will access next, and the object is prefetched whenever it is in the proxy's cache and the modem is idle. For delta-compression, we assume it can eliminate 88% of bytes in transfers of modified objects, according to the results in [19]. We also assume that HTML compression can eliminate 25% of bytes in transfers of HTML documents, according to the average number reported in [19]. We only count network transmission latency here, and do not model CPU overhead for compression/decompression.

The results show that prefetching and delta-compression have the biggest impacts on client latency. Merely increasing browser cache sizes does not help much — even if the browser cache is infinitely large, only 10% of the requests will be saved and 4.1% of user latency is reduced (1st row in Table 1). However, if we combine a large browser cache with delta-compression, the latency reduction jumps to 14.6%, because an additional 13.6% of requests are to modified objects whose older versions are in the browser cache (2nd row). If we further add perfect prefetching, the latency reduction doubles to 28.6%, with 50.1% of user requests hitting in the browser cache (3rd row). The effects of the two techniques change little even if we constrain the browser cache to 16MB.

The results also show that application-level compression has limited performance benefits. This is mostly because we assume that only HTML documents can be compressed and only 25% of the HTML bytes can be saved by the compression. (An existing study [19] reports two numbers for byte savings from application-level compression for two different traces; we picked the average of the two.) Other concerns of application-level compression include the CPU overhead for compression and decompression, and the possibility of wasted efforts because modem lines also apply their own compression to packets. A comprehensive discussion of the tradeoffs in HTML compression can be found in [19]. For the rest of the results in this paper, we no longer use the technique of HTML compression.

One might notice that the percentage of latency reduction is always smaller than the percentage of request savings. The reason is that in the traces, objects that are accessed only once (not-cached) are larger and take longer to download than objects that are accessed multiple times (cached). The average size of not-cached objects is 20.8KB and the average size of cached objects is 5.2KB. This is consistent with observations in other studies [26, 15].

Browser Cache	Delta	HTML Compression	Prefetching	Request Savings	Latency Reduction
Infinite	No	No	No	10.0%	4.1%
Infinite	Yes	No	No	23.6%	14.6%
Infinite	Yes	No	Perfect	50.2%	28.6%
2MB	Yes	No	No	21.3%	12.8%
16MB	Yes	No	No	23.4%	14.4%
16MB Cache	Yes	No	Perfect	50.1%	28.5%
No Cache	No	Yes	No	0%	2.4%
16MB Cache	Yes	Yes	No	23.4%	16.5%
16MB Cache	Yes	Yes	Perfect	50.3%	30.3%

Table 1: Request savings and latency reduction under combinations of the techniques.

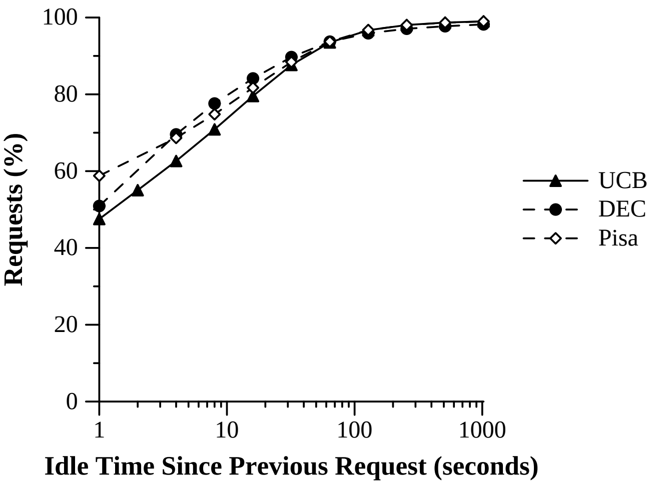

Figure 1: Cumulative distribution of idle time between user requests.

Examining the results on perfect prefetching, we find that as long as the user's next access can be predicted correctly, there is plenty of idle time on the modem link to send the object to the browser cache. In fact, the request savings of perfetch prefetching, 50.1%, is mostly limited by the requirement that only objects that are in the proxy cache can be prefetched, because the hit ratio in the proxy cache is 58.6%.

Figure 1 shows the cumulative distribution of user idle time in the UCB traces. Here we estimate the idle time by calculating the difference between the estimated end-of-transmission of one request to the start of the next request from the same user. We assume 21Kb/s modem bandwidth. The figure also includes the idle time distribution observed from two other LAN Web traces, DEC [14] and Pisa [24]. The curves are similar to each other and show that there is inherent idle time between user requests — about 40% of the requests are preceded by 2 to 128 seconds of idle time, indicating plenty of prefetching opportunities.

5 Prediction Algorithms

Seeing that prefetching under perfect prediction can reduce user latency significantly, we investigate realistic prediction algorithms.

5.1 PPM Predictors

The algorithms in this study observe patterns from past accesses from all the clients to predict the future accesses of individual clients. The patterns that we capture are in the form of "a user is likely to access URL B right after he or she accesses URL A." Clearly, only accesses from the same user should be connected. Accesses from different users are not related.

The algorithms use patterns observed from all users' references to predict a particular user's behavior. On the one hand, this may reduce the accuracy of the prediction because one user's patterns are used to predict a different user's accesses. On the other hand, this increases the number of times that the algorithms can make predictions because the sampling size is increased. We feel that in the context of proxy-initiated prefetching, the second factor is likely to be more important. A good browser cache should absorb most of the repeated accesses from a user, and it is important to predict the first-time access to a document from a modem client.

The actual prediction algorithm is based on the Prediction-by-Partial-Matching (PPM) data compressor [2, 9]. The algorithm has three parameters:

- m: the number of past accesses that are used to predict future ones. It is also called the order of the predictor, and in this paper we call it the *prefix depth*.
- l: the number of steps that the algorithm tries to predict into the future. For example, if $l = 2$, it means that the algorithm not only tries to predict the immediate next access for the user, but it also tries to predict the access after that. We call l the *search depth*.
- t: the threshold used to weed out candidates. Only candidates whose probability of access is higher than t, where $0 \leq t \leq 1$, is considered for prefetching. We call t the *threshold*.

The algorithm maintains a data structure that keeps track of the sequence of l URLs following another URL, a sequence of two URLs, and so on up to a sequence of m URLs. The data structure is typically a collection of trees. For prediction, the past reference, the past two references, up to the past m references are matched against the collection of trees to produce the set of URLs for the next l steps. Only URLs whose frequencies of accesses are larger than t are included. Finally, the URLs are sorted first by giving preferences to longer prefixes, and then by giving preferences to URLs with higher probability within the same prefix.

The algorithm is similar and yet different from previous proposed prefetching algorithms. Papadumanta and Mogul [21] studies the same algorithm, but with m always

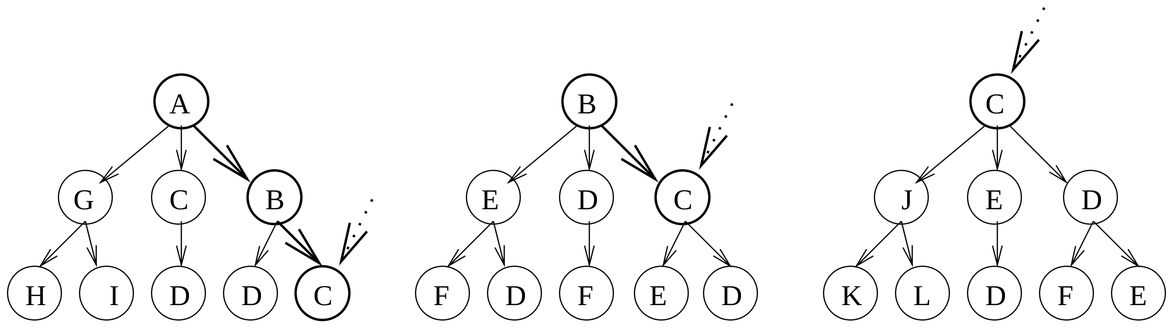

Figure 2: History Structure (being updated for user sequence A...B...C).

equal to 1. Krishna and Vitter [9] also used the PPM-based algorithm as one of their prefetching algorithms, but always set l equal to 1. The reason for considering $m > 1$ is that more contexts might improve the accuracy of the prediction. The reason for considering $l > 1$ is that sometimes, a URL is not always requested as the immediate next request after another URL, but rather within the next few requests. As we will see, one of the best performing configurations is in fact $m = 2, l = 4$.

5.2 History Structure

The implementation uses a history structure, which maintains information about the order that URLs are accessed. There is one history structure shared and updated by all the users of a given proxy.

The history structure is a forest of trees of a fixed depth K, where $K = m + l$. The history encodes all dynamic sequences of accesses (by any one user) up to a maximum length K. One root node is maintained for every page seen, and in this node is a count of how often this page was seen. Directly below each root node are all pages ever requested immediately after the root page and a count of how often the pair of requests occurred. The next level encodes all series of three pages and a count of how often this particular sequence of three pages occurred. This is continued to the K-th level.

The history structure is updated every time a user makes a request. For each user there is a list of the last K pages that the user requested. The update involves incrementing counters and possibly adding new nodes to the trees. Each update changes one node at each level of the history structure. Figure 2 shows an example of the history structure. In this example $K = 3$, and the structure is being updated after a user accesses page C following pages A and B. The sequence ABC is updated, with the counters for A and B and C incremented. The sequence BC is updated and so is the sequence C.

Encoded into the history structure is the probability of accesses for URLs following a given sequence of references. The predictor looks at a user's recent m accesses, and processes each sequence of the last n accesses, where $n = m, ..., 1$, separately. For each sequence, it first finds the corresponding tree and node in the history structure. It then follows all paths down from the node for l levels, listing all the URLs at each level along with their counts. If one URL appears in more than one path, the node counts for the URL are added together. It then divides the count of each URL by the count of the sequence, yielding the URL's relative probability of access. It then sorts the list based on the probability, and deletes those whose probabilities are less than t. Finally, the predictor generates the list of candidates by concatenating the lists from the m sequences, putting the list of the longer sequence (i.e. longer prefix) first.

Every time the modem link to a user is idle, the proxy calls the predictor for the list of candidate URLs. It then initiates prefetching of the objects in the order specified in the list. When the user makes a new request, the ongoing prefetching is stopped, and a new round of prediction and prefetching starts again next time.

The size of the history structure can be controlled using a replacement algorithm. We use LRU. All the nodes in the history structure are linked in an LRU list. When the history structure is too big, the nodes from the least-recently-used end of the list are examined one by one, and if a node is a leaf node, it is deleted from the tree.

6 Performance

In evaluating the performance of PPM predictors, we also impose two external constraints. First, files larger than 50KB are not prefetched. The reason is that prefetching large files runs the risk of being aborted, and among objects of equal probability of being accessed, it is always better to prefetch smaller ones because we can prefetch more of them. We choose a threshold of 50KB because over 95% of files are smaller than 50KB [10]. Second, no more than 8 objects can be prefetched during one idle time period. This is to avoid excessive prefetching during long idle times, and to avoid prefetched objects flooding the browser cache. The limit of 8 seems to work well in our experiments.

6.1 Performance of Proxy-Initiated Prefetching

Using the UCB traces, we study the performance of the PPM prediction algorithm varying m, l, and t. We assume that all clients have 16MB (extended) browser caches, and browser caches use the Least-Recently-Used replacement algorithm. We also assume that prefetching is used in conjunction with delta-encoding for modified objects.

Figure 3 shows the request saving, latency reduction and wasted bandwidth for the combinations of various configurations of PPM. We show the request savings as percentage of total requests and the latency reduction as percentage of total client latency. We calculate the wasted bandwidth by first taking the ratio between bandwidth consumption under prefetching and that under no prefetching, and then substracting 1 from the ratio. For request savings we show its three components: cached, prefetched and partially prefetched. We experiment with prefix depth m being 2 and

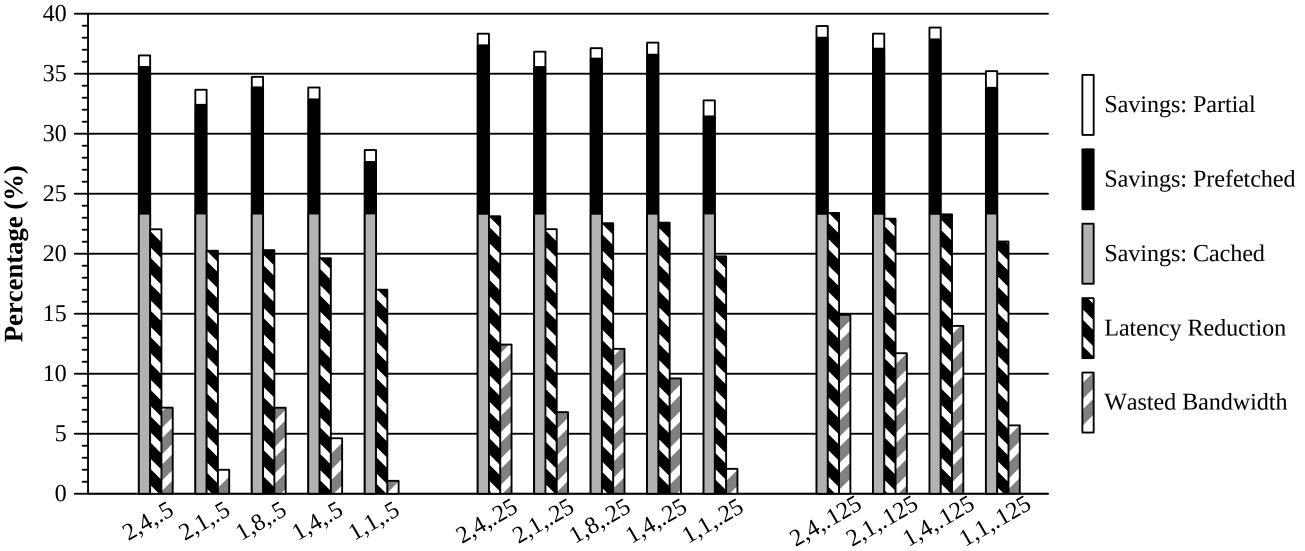

Figure 3: Request savings, latency reduction and wasted bandwidth of PPM prediction algorithms. Configurations of PPM are shown as $(prefix\_depth, search\_depth, threshold)$ tuples.

1, search depth l being 1, 4 and 8, and threshold t being 0.5, 0.25 and 0.125. We list the configuration as (m, l, t) tuples. We omit some results of $l = 8$ because they are similar to $l = 4$.

Prefetching combined with delta compression reduces user latency from 17% to 23.4%. The request savings (browser cache hit ratio) ranges from 27% to 39%. The extra traffic generated by prefetching is moderate, in the range of 1% to 15%. About 60% of the request savings comes from the caching effect of the browser cache, and about 40% comes from prefetching. This is reflected in the latency reduction as well; without prefetching, 16MB browser cache and delta compression can reduce user latency by 14.4%, whereas prefetching adds another 9% to the latency reduction. The results show that there are clear benefits in proxy-initiated prefetching, and that the PPM predictor works well.

Examining the effects of the parameters, we see that decreasing the threshold t increases the wasted bandwidth. For configurations with $l = 1$, lower t also increases the latency reduction. However, for $l > 1$, $t = 0.25$ is the best choice. The reason is that for fixed m and l, lower threshold merely makes more URLs eligible for prefetching and adds them to the end of the candidate list. When $l = 1$, there are not many choices of URLs (since only the first level nodes are considered) and lower t helps to generate enough candidates. When $l > 1$, there are many choices of URLs and t should be ≥ 0.25.

Increasing the search depth l increases both the latency reduction and the wasted bandwidth. $l = 4$ appears the best choice, as larger l makes little difference. Increasing the prefix depth m also increases the latency reduction and wasted bandwidth, but the effect is more pronounced when the threshold is high. In general, using more contexts helps.

In order to examine the accuracy of the prediction algorithm, we show in Figure 4 the total number of candidates suggested by the predictor (*attempted*), the actual number of objects that are prefetched (*prefetched*), and the number of objects that are prefetched and actually accessed by the user (*used*). The ratio between *used* and *prefetched* is the accuracy of the prediction algorithm. The accuracy depends on the configuration, ranging from 40% under $(2, 4, 0.125)$ to 73% under $(1, 1, 0.5)$. Lowering the threshold always decreases the accuracy. There is no direct correlation between the accuracy of a configuration and its latency reduction. Some configurations, such as $(2, 4, 0.5)$, achieve both high accuracy (52%) and high latency reduction (23.3%), while the low threshold configurations appear to sacrifice accuracy for more prefetches.

The large differences between *attempted* and *prefetched* indicate that many candidates are not prefetched due to the lack of modem idle time and our imposed limit of 8 objects per prefetch session. (Some *attempted* numbers are too large to show in the figure.) Luckily, those candidates also have lower probabilities of reference, since higher probability ones are fetched first.

Summarizing the two figures, we give the following recommendations for the configuration of PPM. If the highest latency reduction is the goal and some amount of wasted bandwidth can be tolerated, then $(2, 4, 0.125)$ is the best choice. If both high latency reduction and low wasted bandwidth are desired, then $(2, 4, 0.5)$ is the best choice. If limits on storage requirements make smaller m and l desirable, then both $(2, 1, 0.25)$ and $(1, 1, 0.125)$ are good choices.

Though not shown here, we have also decomposed the *prefetched* objects and the *used* objects based on their types, including "html", "gif", "jpg" and "other." About 46% to 50% of the prefetched objects are "gif" files, and 18% to 24% are "jpg" files. This suggests that some of the prefetched objects are probably embedded objects of the previous HTML file, and can perhaps be discovered using alternative methods such as scanning the HTML files. However, the traces lack sufficient information to estimate the percentage of embedded objects in prefetched ones. About 13% to 18% of the prefetched objects are HTML files, and the rest are unknown types. The accuracy of the prediction is higher for "jpg" objects (around 65%) and "gif" objects (around 58%), and lower for "html" and "other" objects (around 35%).

The results under PPM are not as good as those under the perfect predictor. Section 4 shows that perfect predictor

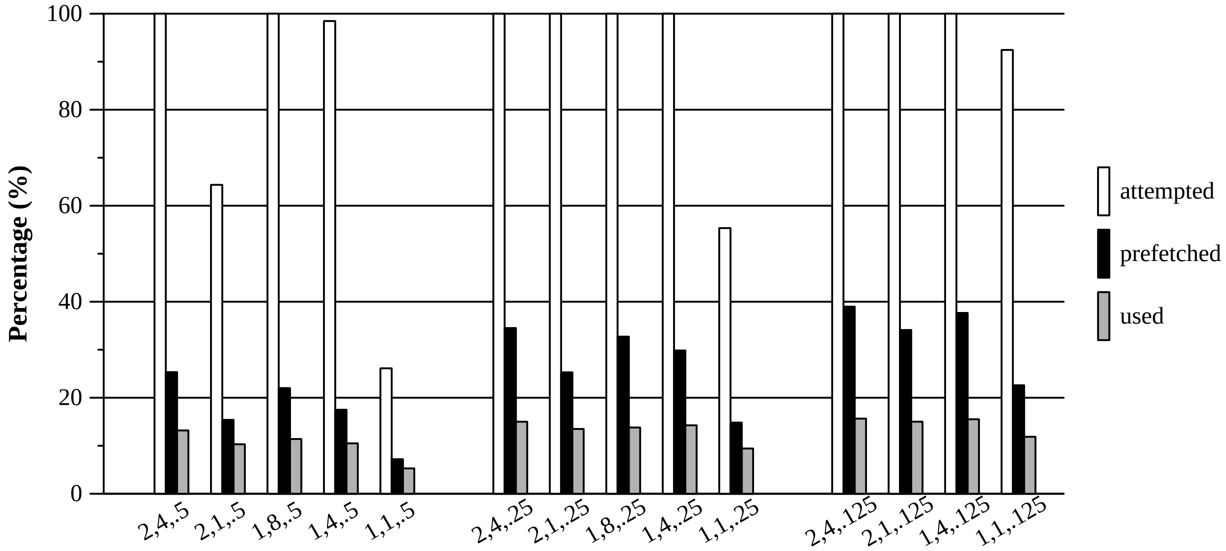

Figure 4: The total number of candidates suggested by the predictor, the number of objects that are actually prefetched, and the number of objects that are prefetched and are accessed by the user, all expressed in percentage of the total number of requests in the trace.

increases the request savings by 27%, whereas PPM predictors only increases the request savings by about 14%. Besides accuracy, the amount of history that a predictor sees also affects its performance. If the predictor has never seen a sequence before, then it cannot predict it. For example, if the predictor has never seen AB appearing before, then it cannot predict B when it sees A. To measure the effect of limited history on our results, we examine the traces to find the percentage of repeated pairs of accesses. This serves as an upper bound on the performance of predictors with $m = 1$ and $l = 1$. We found that only 22% of access pairs are repeated. Thus, the PPM predictor is largely limited by the short 5-day duration of our trace. In practice, when the predictor is able to observe user accesses over weeks or months, the performance of PPM predictor can be improved.

6.2 Effects of Implementation Variations

The previous results assume that the implementation strictly follows the description in Section 2.1. In practice, however, engineers are tempted to make various "shortcuts" to reduce the overhead and complexity of the implementation. Here we look at the effects of some of the shortcuts on the prediction algorithm.

No proxy notification upon browser cache hits: The previous simulations assume that the browser will send a notification to the proxy whenever a user request hit in the browser cache, so that the proxy will observe all requests made by user. In practice, however, it would be simpler if the browser does not send such notifications. In this case, the prediction algorithm is observing the cache misses from the browser cache. In addition, there is no feedback on whether a prediction is correct or not.

We simulate this scenario (called "no-notice") for three configurations of the PPM predictor, $(2, 4, 0.5)$, $(1, 4, 0.25)$ and $(1, 1, 0.125)$. Figure 5 shows the latency reduction and the percentage of prefetch transfers of the original scheme and "no-notice". Without viewing the user's full access stream and without feedbacks from the browser, the proxy initiates fewer prefetches. Further analysis shows that "no-notice" reduces the accuracy of the prediction from about 50% to about 46%. Thus, this shortcut should be avoided unless it significantly simplifies the implementation.

Prefetching without knowledge of the content of browser caches: The simulations so far assume that the party performing the prefetching knows the content of the browser cache and skips the URLs that are already in the browser cache. This is the case if the proxy sends the list of candidates to the browser and lets the browser initiate the prefetching of each URL. However, if the proxy pushes the URLs itself, it has to maintain a record of the content of every browser cache in order to avoid unnecessary transfers. What happens if the proxy simply does not keep such records?

We simulate this scenario, called "oblivious", for the three configurations of PPM predictor. The results are also shown in figure 5. The performance is worse compared to the original scheme, though not as bad as "no-notice." The main reason seems to be the significant increase in the number of prefetch transfers. Thus, if wasted bandwidth is a concern, it is important for the prefetcher to keep track of browser cache contents.

Limiting the size of history structure: Our results so far assume that no replacement occurs for the history structure. The storage requirement for the history structure is typically $O((m + l) * N)$, where m is the prefix depth, l is the search depth, and N is the number of requests in the history. Thus, setting $m = 1$ and $l = 1$ has the advantage of smaller storage requirements. In our previous simulations, for $(1, 1, t)$ configurations, the history structure eventually grows to about 32MB; the trace consists of 1.9 million requests, and each node in the history structure takes 8 bytes (4-byte unique identifier for the URL, and 4-byte count).

We also experimented with reducing the storage of the history structure by about 25% and triggering LRU replacements of the nodes when necessary. The latency reduc-

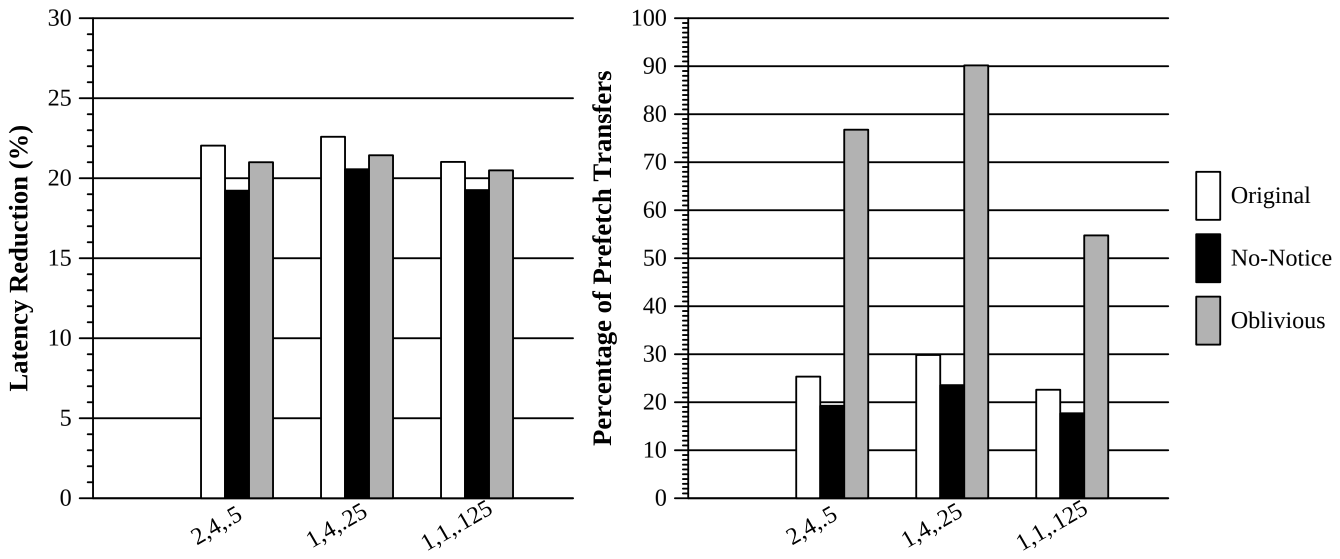

Figure 5: Comparison of the original prefetching scheme and the implementation short cuts.

tion from the algorithm is slightly decreased, dropping from 23.4% to 22.4% under $(2, 4, 0.125)$, and from 21.0% to 20.2% under $(1, 1, 0.125)$. Thus, smaller history structures worsen the performance.

7 Implementation Experience

We have implemented proxy-initiated prefetching in the CERN httpd proxy software. CERN httpd uses a process-based structure and forks a new process each time a new request arrives. A separate predictor process is introduced, which communicates with other processes via UDP messages. The predictor runs in an infinite loop, waiting to receive two types of messages: "updates" which say "user i just accessed URL u," and "queries" which asks for the list of prefetching candidates for a user. It sends lists of URLs via UDP to other processes. Due to memory constraints, the predictor remembers the last few requests for each client, but does not keep track of client browser cache contents. The ordinary httpd process is modified to send update and query messages to the predictor before processing a user request, and wait for the list of candidates after processing the request. The process then checks a shared global array of flags to see whether the modem link is idle, and if it is, starts pushing the URL objects on the existing connection (we assume that the connection stays open until the proxy closes it). The separation between the predictor process and regular httpd processes makes it easy to port the implementation to other proxy software, such as Squid.

At the client side, instead of modifying browsers, we set up a copy of the CERN httpd proxy. The browser requests are first sent to the local proxy. The local proxy manages its own cache, issues requests to the main proxy, and receives pushed objects. A separate HTTP entity header is used to mark the URL for each pushed object. The current implementation uses the "proxy-push" approach, and the main proxy has no knowledge of the cache contents at the client side.

Rigorous measurements of the performance of the implementation are difficult. Not only do we need to emulate modem connections on the LAN, but we also need to generate workloads that reflect typical browser behavior and Internet latencies. We have instrumented the Linux kernel to simulate modem connections on our Ethernet LAN, and are working on the workload issues. Simple experiments do show that the prediction engine increases the CPU overhead at the proxy by about 7%.

8 Related Work

Prefetching can be applied in three ways in the Web contexts:

- between Web servers and browser clients;
- between Web servers and proxies;
- and between proxies and browser clients.

The first two approaches run the risk of increased Wide-Area Network traffic, while the last one only affects the traffic over the modems or the LAN.

Prefetching between Web servers and browser clients: Early studies have mostly focused on prefetching between Web servers and browser clients, since relatively few proxies were deployed then. Padmanabhan and Mogul [21] analyze the latency reduction and network traffic of prefetching using Web server traces and trace-driven simulation. The prediction algorithm they used is also based on the PPM data compressor, but with prefix depth of 1. The study shows that prefetching from Web servers to individual clients can reduce client latency by as high as 45%, at the expense of doubling the network traffic. Bestavros and Cunha [3] presents a model for speculative dissemination of World Wide Web documents. The work shows that reference patterns observed at a Web server can be used as an effective source of information to drive prefetching, and reaches similar conclusions as [21]. Finally, Cuhan and Bestavros [8] use a collection of Web client traces and study how effectively a user's future Web accesses can be predicted from his or her past Web accesses. They show that a number of models work well and can be used in prefetching.

We found that these early studies do not answer our questions about the performance of prefetching between clients and proxies. First, the studies investigate prefetching between user clients and Web servers; caching proxies are not considered or modelled. There are reasons to believe that a proxy's capability to predict users' Web accesses is different from the server's. Since a proxy only sees the Web requests to its users, its ability to predict is limited by the smaller sampling size. On the other hand, a proxy can predict documents across Web servers, increasing its chances at reducing client latency. Second, most of the studies use Web server traces which do not accurately capture user idle time. From a particular Web server's point of view, a user's accesses to other Web servers appear as idle time. Thus, it is difficult to estimate from Web server traces the true user idle time that can be exploited to push documents to users over the modem lines. Lastly, none of the studies investigate modem connections.

Prefetching between Web servers and caching proxies: As proxies become more widespread, a number of studies have investigated prefetching between Web servers and proxies [15, 18, 12, 13]. Kroeger and Mogul [15] investigates the performance limits of prefetching between Web servers and proxies, and shows that combining perfect caching and perfect prefetching at the proxies can at best reduce the client latency by 60% for high-bandwidth clients (i.e. those that accesses the Web through a LAN instead of modems). Markatos and Chronaki [18] propose that Web servers regularly push their most popular documents to Web proxies, which then push those documents to the clients. They evaluate the performance of the strategy using several Web server traces and find that the technique can anticipate more than 40% of a client's request. The technique requires cooperation from the Web servers. The study did not evaluate client latency reduction from the technique. Cohen *et al* [7] also investigates similar techniques. Wcol [12] is a proxy software that prefetches documents from the Web servers. It parses HTML files and prefetches links and embedded images. The proxy, however, does not push the documents to the clients. Finally, Gwertzman and Seltzer [13] discuss a technique called Geographical Push-Caching where a Web server selectively sends its documents to the caches that are closest to its clients. The focus of the study is on deriving reasonably accurate network topology information and using the information to select caches.

Prefetching between proxies and browser clients: Loon and Bharghavan [17] have proposed similar ideas as we described here. Their study presents a design and an implementation of a proxy system that performs the prefetching, image filtering, and hoarding for mobile clients. However, the study does not fully address the potential and limits of the prefetching scheme, the design of the prediction algorithms, and which algorithm performs the best.

Other Related Work on Prefetching Outside the Web contexts, prefetching has been studied extensively in file systems and memory systems. Several studies investigate application-controlled prefetching in the file system contexts [6, 23], where an application gives prediction of what it might access next. Other studies also investigate speculative approaches to file prefetching [11, 22, 16]. The prediction algorithms used there are also derived from data compressors, while the compressor and the parameters vary.

Most of the algorithms, including the one studied here, are inspired by an important early study by Krishnan and Vitter demonstrating the relationship between data compression and prediction [25, 9]. Their study shows that if the source behind a reference string can be modelled as an m-order Markov source, then the compressor-driven prediction algorithm will converge to the optimal predictor given a long enough reference string. Our algorithms extend the algorithms described in [25] and [9] by considering search depth of more than 1, that is, not only predicting which object will be used next, but also which objects will be used after that. Our results show that for best performance, the search depth needs to be larger than 1.

9 Conclusions and Future Work

Today, a large percentage of the Internet population access the World Wide Web via low bandwidth modem connections. Users spend a lot of time impatiently waiting for web pages to come up on screen. Currently, new infrastructure to lessen the latency is slow in deployment. In the mean time, any technique that reduces the latency over the existing infrastructure should be looked into. We have investigated the potential and performance of one technique, prefetching between the low-bandwidth clients and caching proxies, and found that combined with delta compression, it can reduce user-visible latency by over 23%. Prediction algorithms based on the Prediction-by-Partial-Matching compressor perform well. The technique is easy to implement and can have a considerable effect on user's Web surfing experience.

There are several limitations in our study. First, we assume fixed user request arrival times in our simulation. The assumption is reasonable if the arrival rate is mainly dominated by users' thinking time. However, if the arrival rate is dominated by the latency to service requests, then as requests are serviced faster, they also arrive faster. We have run simulations assuming fixed durations between the end of transfer of one request to the beginning of the next request. The results are quite similar to those reported before, with the latency reduction slightly lower. Second, our calculation of client latency is merely an estimate based on the timestamps recorded in the traces and the modem bandwidth. It does not include protocol overhead and is far from accurate. We also does not model the proxy in detail. However, we feel that the estimate is good enough to give us an indication of how well the prefetching scheme might perform. Finally, we have not investigated the implementation of delta compression. Other research groups are actively working on it [1, 19] and combining the implementations of the two techniques is straightforward.

Much future work remains. We are working on proxy workload development and realistic measurement of the performance of our prefetching implementation. We are also looking into prediction algorithms other than PPM. Finally, we are investigating how the scheme might perform in high-latency high-bandwidth environments such as satellite connections.

Finally, more detailed information on the simulation and other results can be found in our technical report at *http://www.cs.wisc.edu/wisweb/prefetch.html*. The source codes of the simulator and the implementation are available there as well.

References

[1] Gaurav Banga, Fred Douglis, and Michael Robinovich. Optimistic deltas for www latency reduction. In *Proceedings of the 1997 USENIX Technical Conference*, 1997. Available from http://www.research.att.com/~douglis/.

[2] T. C. Bell, J. C. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall Advanced Reference Series, 1990.

[3] Azer Bestavros and Carlos Cunha. Server-initiated document dissemination for the www. *IEEE Data Engineering Bulletin*, September 1996.

[4] Ramon Caceres, Fred Douglis, Anja Feldmann, Michael Rabinovich, and Gideon Glass. Web proxy caching: The devil is in the details. *Proceedings of the 1998 SIGMETRICS Workshop on Internet Server Performance*, 1997. available at http://www.cs.wisc.edu/~cao/WISP98/html-versions/anja/proxim_wisp/index.html.

[5] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Proc. 1995 ACM SIGMETRICS*, pages 188–197, May 1995.

[6] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling. In *TOCS*, November 1996.

[7] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Improving end-to-end performance of the web using server volumes and proxy filters. In *Proceedings of ACM SIGCOMM'98*, September 1998.

[8] Carlos Cunha and Carlos F. B. Jaccoud. Determining www user's next access and its application to prefetching. In *Proceedings of ISCC'97: The Second IEEE Symposium on Computers and Communications*, July 1997. URL: http://www.cs.bu.edu/students/alumni/carro/Home.html.

[9] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proceedings of SIGMOD'93*, pages 257–266, May 1993.

[10] Steven Gribble and Eric Brewer. UCB home IP HTTP traces. *Available at http://www.cs.berkeley.edu/~gribble/traces/index.html*, June 1997.

[11] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Conference Proceedings of the USENIX Summer 1994 Technical Conference*, pages 197–208, June 1994.

[12] Wcol Group. Www collector - the prefetching proxy server for www. *http://shika.aist-nara.ac.jp/products/wcol/wcol.html*, 1997.

[13] James Gwertzman and Margo Seltzer. The case for geographical push-caching. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, 1995.

[14] T. M. Kroeger, J. Mogul, and C. Maltzahn. Digital's web proxy traces. *Available at URL: ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html*, August 1996.

[15] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proceedings of USENIX Symposium on Internet Technology and Systems*, December 1997.

[16] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 275–288. USENIX, 1997.

[17] Tong Sau Loon and Vaduvur Bharghavan. Alleviating the latency and bandwidth problems in www browsing. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, December 1997. URL: http://timely.crhc.uiuc.edu/.

[18] Evangelos P. Markatos and Catherine E. Chronaki. A top-10 approach to prefetching on the web. Technical report, Technical Report No. 173, ICS-FORTH, Heraklion, Crete, Greece, August 1996. URL http://www.ics.forth.gr/proj/arch-vlsi/www.html.

[19] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *Proceedings of ACM SIGCOMM'97*, August 1997. Available from http://www.research.att.com/~douglis/.

[20] Henrik Frystyk Nielsen, Jim Gettys, Anselm Baird-Smith, Eric Prudhommeaux, Hakon Wium Lie, and Chris Lilley. Network performance effects of http/1.1. In *Proceedings of ACM SIGCOMM'97*, August 1997. Available from http://www.w3.org/Protocols/HTTP/Performance/Pipeline.

[21] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, July 1996.

[22] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, September 1991.

[23] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, December 1995.

[24] Luigi Rizzo. Web proxy traces. *Available at URL: http://info.iet.unipi.it/~luigi/proxy-traces/*, May 1997.

[25] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 121–130, also appears as Brown Univ. Tech. Rep. No. CS–91–46, October 1991.

[26] S. Williams, M. Abrams, C.R. Stanbridge, G. Abdulla, and E.A. Fox. Removal policies in network caches for world-wide web documents. In *Proceedings of the ACM SIGCOMM'96*, August 1996. URL http://ei.cs.vt.edu/~succeed/96sigcomm/.