Using Predictive Prefetching to Improve World Wide Web Latency

Venkata N. Padmanabhan
University of California at Berkeley
padmanab@cs.berkeley.edu

Jeffrey C. Mogul
Digital Equipment Corporation
Western Research Laboratory
mogul@pa.dec.com

Abstract

The long-term success of the World Wide Web depends on fast response time. People use the Web to access information from remote sites, but do not like to wait long for their results. The latency of retrieving a Web document depends on several factors such as the network bandwidth, propagation time and the speed of the server and client computers. Although several proposals have been made for reducing this latency, it is difficult to push it to the point where it becomes insignificant.

This motivates our work, where we investigate a scheme for reducing the latency perceived by users by predicting and prefetching files that are likely to be requested soon, while the user is browsing through the currently displayed page. In our scheme the server, which gets to see requests from several clients, makes predictions while individual clients initiate prefetching. We evaluate our scheme based on trace-driven simulations of prefetching over both high-bandwidth and low-bandwidth links. Our results indicate that prefetching is quite beneficial in both cases, resulting in a significant reduction in the average access time at the cost of an increase in network traffic by a similar fraction. We expect prefetching to be particularly profitable over non-shared (dialup) links and high-bandwidth, high-latency (satellite) links.

1 Introduction

People use the World Wide Web (WWW) because it gives quick and easy access to a tremendous variety of information in remote locations. Users do not like to wait for their results; they tend to avoid or complain about Web pages that take a long time to retrieve. That is, users care about Web latency.

Perceived latency comes from several sources. Web servers can take a long time to process a request, especially if they are overloaded or have slow disks. Web clients can add delay if they do not quickly parse the retrieved data and display it for the user. The retrieval time of Web documents also depends on network latency. The Web is useful precisely because it provides remote access, and transmission of data across a distance takes time. Some of this delay depends on bandwidth; one cannot retrieve a 1 MB file across a 1 Mbps link in less than 8 seconds. But much of the network latency comes from propagation delay. Some of these delays, such as client or server slowness or transmission time, can in principle be reduced by buying faster computers or higher bandwidth links. However, other components such as propagation delay, which is basically determined by the physical distance traversed, cannot be reduced beyond a point.

The Hypertext Transport Protocol (HTTP) version 1.0 [1], as it is currently used in the Web, is simple, but far from optimal as far as latency is concerned. Several researchers ([6],[8],[9],[11]) have analyzed the inefficiencies in use of the network by HTTP, and have proposed modifications to reduce retrieval latency significantly. However, it is difficult to push retrieval latency beyond the point where it becomes insignificant.

This motivates the investigation of ways of hiding retrieval latency from the user rather than actually reducing it. We describe a scheme in which clients, in collaboration with servers, prefetch Web pages that the user is likely to access soon, while he/she is viewing the currently displayed page. Then, if the user does request one of the prefetched pages, it will already be in the local site's cache. Thus, the *retrieval latency* (also called *retrieval time*) would be masked from the user in such cases, yielding a lower *access time*. We maintain this distinction between retrieval latency (or time) and access time through the rest of this paper.

We use a distributed prefetching scheme with distinct roles for the clients and servers. Servers, which get to see accesses from several clients, make predictions on which files are likely to be accessed in the near future. Clients initiate prefetching based on advice from servers. Clearly, the effectiveness of prefetching critically depends on how good the predictions are. We use a prediction algorithm patterned after that proposed by Griffioen and Appleton [3] in the context of file systems, though there are a few noteworthy differences.

The results from our trace-driven simulations indicate that prefetching helps significantly decrease the average access time at the cost of an increase in network traffic. The latency of retrieving Web data involves a relatively large component that is independent of the amount of data transferred. This includes network round-trip times and other overheads at the end-hosts. In such situations, it is often more effective to use prefetching to reduce latency rather than to simply increase the available bandwidth.

The rest of this paper is organized as follows. In section 2, we briefly discuss the basics of HTTP that are needed to understand the rest of this paper. In that section we also briefly describe the modifications to HTTP proposed in [8]. In section 3, we present our scheme for predictive prefetching. The methodology used for the simulation experiments is described in section 4, and the results are presented in 5. In section 6, we discuss some issues pertaining to prefetching. We present our conclusions in section 7.

2 HTTP Protocol Elements

The HTTP protocol is layered over a reliable bidirectional byte stream, normally TCP [10]. Each HTTP interaction consists of a request sent from the client to the server, followed by a response sent back from the server to the client. Requests and responses are expressed in a simple ASCII format. Most existing implementations conform to the original version of the protocol, HTTP/1.0 [1]. The next version, HTTP/1.1, is presently in draft form [2].

An HTTP request includes several elements: a method such as GET, PUT, POST, etc.; a Uniform Resource Locator (URL); a set of Hypertext Request (HTRQ) headers, with which the clients specifies things such as the kinds of documents it is willing to accept, authentication information, etc; and an optional data field, used with certain methods such as PUT.

The server parses the request, then takes action according to the specified method. It then sends a response to the client, including a status code to indicate if the request succeeded, or if not, why not; a set of object headers, meta-information about the "object" returned by the server, optionally including the "content-length" of the response; and a data field, containing the file requested, or the output generated by a server-side script.

2.1 Limitations of HTTP

We now look at the way the interaction between HTTP clients and servers appears on the network, with particular emphasis on how this affects latency. We mainly look at HTTP/1.0 since that is used by most servers and clients around today.

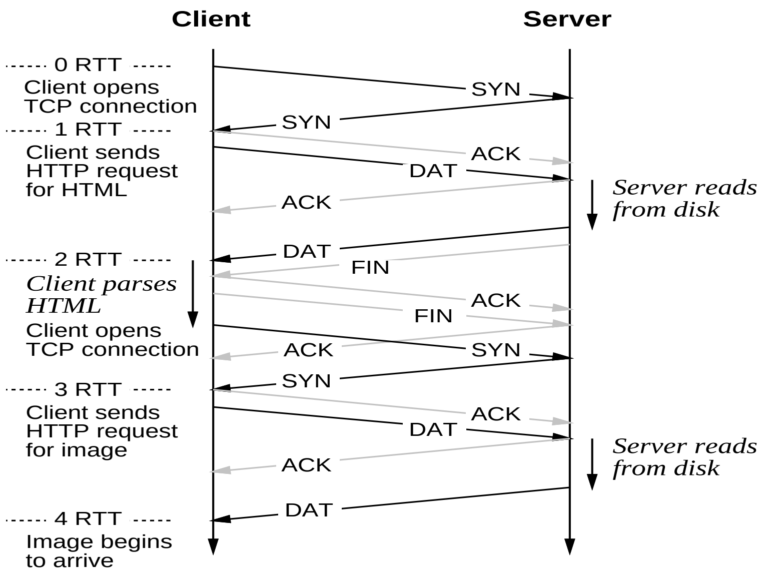

Figure 1: *This figure shows the packet exchanges between a client and a server for HTTP. Time runs down the page. DAT and ACK denote data and acknowledgement packets respectively, though most data packets also carry acknowledgements. SYN and FIN denote packets used by TCP to signal the start and end, respectively, of a connection. To the left of the Client timeline, horizontal dotted lines show the "mandatory" round trip times (RTTs) through the network, imposed by the combination of the HTTP and TCP protocols.*

Figure 1 depicts the packet-exchange between a client and a server at the beginning of a typical interaction, the retrieval of an HTML document with at least one uncached inline image. We note two obvious inefficiencies in the protocol. First, the transfer of each HTML or image file involves setting up and tearing down a new TCP connection. Second, the request-response protocol between the client and the server operates in a stop-and-go manner, with a new request being sent only after the reply to the previous one has arrived. These result in considerable delays.

2.2 Persistent Connection HTTP (P-HTTP)

We briefly discuss persistent connection HTTP (P-HTTP) proposed by Padmanabhan and Mogul [8] (the term P-HTTP is from [6]). P-HTTP uses a single, long-lived TCP connection for multiple HTTP transactions. The connection stays open for all the inline images of a single document, and across multiple HTML retrievals. This helps solve the first problem mentioned above. The HTTP/1.1 protocol [2] also defines a persistent connection mechanism to solve the same problem.

To avoid the second problem, [8] proposes two new HTTP methods (primitives), GETALL and GETLIST, that allow pipelining requests and responses between a client and a server. GETALL is a request to fetch the specified HTML file and all inline images that reside on the server. GETLIST is a request to fetch all files specified in the list that the client passes to the server. It is possible to simulate GETLIST with an asynchronous series of pipelined GETs.

Together, these modifications result in considerably reduced retrieval latency, in some cases less than half the original latency.

3 Predictive Prefetching

It is clear from section 2.1 that the retrieval of a typical Web page involves several network round trips using HTTP/1.0. P-HTTP reduces this cost considerably, but as [8] reports image-rich Web pages still suffer from multi-second retrieval latencies. In light of this, we decided to investigate techniques that do not actually reduce retrieval time, but still improve response time perceived by the user.

Users usually browse the Web by following hyperlinks from one Web page to another. Hyperlinks on a page often refer to pages stored on the same server. Typically, there is a pause after each page is loaded, while the user reads the displayed material. This time could be used by the client to prefetch files that are likely to be accessed soon, thereby avoiding retrieval latency if and when those files are actually requested. The retrieval latency has not actually been reduced; it has just been overlapped with the time the user spends reading, thereby decreasing the access time.

In our proposal, the server computes the likelihood that a particular Web page will be accessed next and conveys this information to the client. The client program then decides whether or not to actually prefetch the page. This partitioning of work between the server and the client is natural. The server has the opportunity to observe the pattern of accesses from several clients and use this information to make intelligent predictions. On the other hand, the client is in the best position to decide if it should prefetch files based on whether it already has them cached or the cost (in terms of CPU time, memory, network bandwidth, and so on) needed to prefetch data.

As an aside, we note that the server could prefetch files from disk into memory, independent of client requests. However, we believe that the benefit of this would be limited because of the dominance of network latency over disk latency, especially in a wide-area context. So in our study we only investigated prefetching from the server to clients across the network.

3.1 Architecture of the System with Prefetching

We now describe the architecture of the system with prefetching, as depicted in figure 2. On the server side, there are two types of user-level processes. One is the set of HTTP daemon processes, `httpd`, with support for persistent connections and some other features described below. One `httpd` process gets spawned to service requests from each client. Since persistent connections are supported, there is one process per client rather than one per client request. The other process is the *prediction daemon*, `predictd`, which makes prefetching-related predictions. There is only one `predictd` per server, not a new one for each client request or for each client. Furthermore, `predictd` only communicates with `httpd`, not directly with the clients. This design is based on that of the NCSA server, which invokes processes rather than threads to service client requests.

On receiving a request from a client, `httpd` passes on the identity of the client and the names of the files requested to `predictd`. Since we are only concerned with file accesses, `predictd` only looks at client requests that use the GET method or its variants (such as GETALL or GETLIST in P-HTTP). `Predictd`

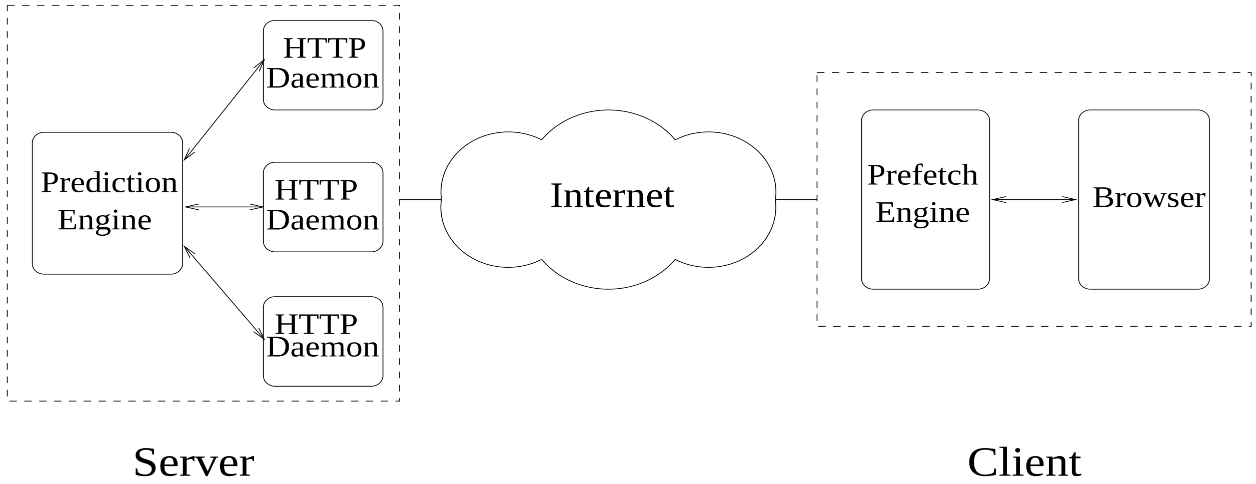

Figure 2: *This figure depicts the architecture of the system with prefetching. On the server side, the set of* `httpd` *processes (one per active client) communicate with the prediction engine, which makes prefetching-related predictions. On the client side, the prefetch engine initiates prefetching based on advice from the server and some other factors. The bidirectional arrows denote local communication between entities at the server and at the client.*

uses the prediction algorithm described in section 3.2 to determine files that are candidates for prefetching based on the likelihood of their being accessed soon, and conveys this information to the client. This information can be piggy-backed on the reply sent by `httpd` to the client, in a special field.

The client side consists of a browser, such as `Mosaic`, and a prefetch engine. The prefetch engine uses the prediction information sent by the server in its reply to decide whether or not to prefetch files. It could also make its decision based on a variety of other factors, such as the contents of the local cache (which might already contain the file), the current system load, the browser's current mode of operation (such as image loading turned off), and so on.

Once the prefetch engine has decided to prefetch a file, it sends a request to the server. In this request it also indicates that it is prefetching data, and not fetching data that the user has explicitly requested. This information can be used by the server in a variety of ways. `Predictd` could decide not to do any further prefetching-related computation based on this request since this is itself a prefetch request. Also, if multiple requests are being scheduled in any way, this request could be assigned a lower priority than explicit fetch requests.

3.2 Prediction Algorithm

Our prediction algorithm is based on that described by Griffioen and Appleton [3]. However, there are a few noteworthy differences. First, while their scheme was designed for use by the operating system to prefetch files from disk into the file system cache, our model is a distributed one with user-level processes at the server and client hosts managing prefetching across the network, into the client's cache. Thus, our scheme does not require any kernel modifications.

Second, the scheme described in [3] does not try to maintain a distinction between accesses by different processes (the clients in the context of a file system). Thus, independent accesses (by different processes), that occur close together in time, could incorrectly be considered as related. As we explain below, our

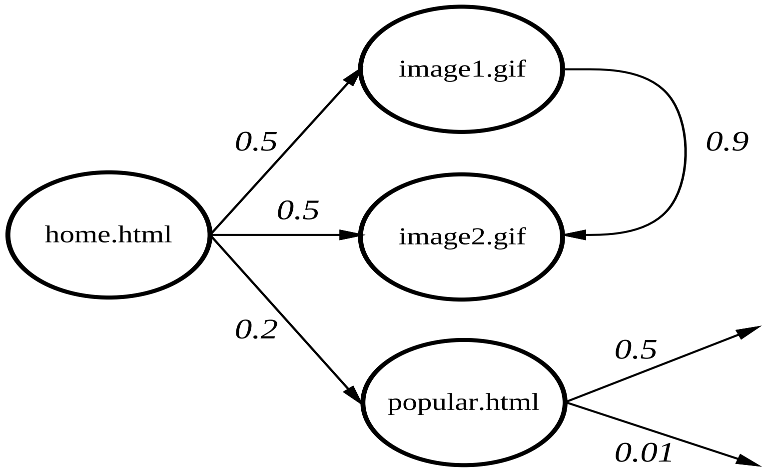

Figure 3: *This figure depicts a small portion of a hypothetical dependency graph. Based on past observations, when* `home.html` *is accessed there is a 50% chance that* `image1.gif` *will be accessed soon afterwards and also a 50% chance that* `image2.gif` *will be accessed soon afterwards. Furthermore, if* `image1.gif` *is accessed, there is a 90% chance that* `image2.gif` *will follow soon afterwards.*

scheme avoids this problem of false correlations.

The prediction algorithm constructs a *dependency graph* that depicts the pattern of accesses to different files stored at the server. The graph has a node for every file that has ever been accessed. There is an arc from node A to B if and only if at some point in time B was accessed within w accesses after A, where w is the *lookahead window* size. The weight on the arc is the ratio of the number of accesses to B within a window after A to the number of accesses to A itself. This weight is *not* actually the probability that the B will be requested immediately after A. So the weights on arcs emanating from a particular node need not add up to 1. Figure 3 depicts a portion of a hypothetical dependency graph.

The dependency graph is dynamically updated as the server receives new requests. This is done by the prediction daemon, `predictd`, which receives information about requests from each `httpd` process running on the server machine. It maintains a ring buffer of size equal to the window size w for each client that is currently connected to this server (assuming that persistent connections are used). When it receives a new request from one of the `httpd` processes, it inserts the ID of the file accessed into the corresponding ring buffer. Only the entries within the same ring buffer are considered related, so only the corresponding arcs in the dependency graph are updated. This logically separates out accesses by different clients and thereby avoids the problem of false correlations. However, in some cases, such as clients located behind a proxy cache, `predictd` will not be able to distinguish between accesses from different clients. One way of getting around this problem is to use mechanisms (such as those proposed in [5]) to pass session-state identification between clients and servers even when there is a proxy between them.

`Predictd` bases its predictions on the dependency graph. When A is accessed, it would make sense to prefetch B if the arc from A to B has a large weight (which implies that there is a good chance of B being accessed soon afterwards). In general, `predictd` would declare B as a candidate for prefetching if the arc from A to B has a weight higher than the *prefetch threshold*, p. It is possible to set this threshold differently for each client and also vary it dynamically.

3.3 Some Issues

We have implemented the prediction daemon, and have made the necessary changes to `httpd` for it to communicate information on accesses to `predictd` through a UNIX pipe. In case of a GETALL or GETLIST request, the modified `httpd` conveys this fact to `predictd` so that the latter is aware that *all* the files corresponding to the GETALL or GETLIST have already been sent to the client and hence need not be considered as candidates for being prefetched at this time. We have not yet implemented the client-server communications interface and the client-side support for prefetching.

There is the issue of how the lookahead window is managed when there are multiple accesses to the same file within a window. As an example, consider a window size of 10 and the sequence of accesses $ABB \cdots AC \cdots AD \cdots ABB$, where $\cdots$ denotes gaps much larger than the window size. If we counted the multiple occurrences of B within a window, then the weight of the arc from A to B would be $4/4 = 1$. However, this does not reflect the dependency between accesses to A and B correctly because, in fact, B does *not* follow A within a window 50% of the time. Caching at the clients should eliminate such multiple accesses, but they happen sometimes, for instance, when the data pointed to by a URL (B in this case) is updated frequently. We ignored such multiple accesses to the same file within a window while computing the weights on arcs.

Dependencies between accesses to different files may vary with time. For instance, certain pages at a Web site might be very popular for a few days, so it would make sense to prefetch them whenever a client accesses the "home page" for that site. As the popularity of these pages wanes, prefetching them would be less beneficial. The weights on arcs in the dependency graph should be adjusted accordingly using some form of aging. Furthermore, nodes in the dependency graph of files that have not been accessed for a long time could be pruned to limit the size of the graph. In our implementation, we have ignored these issues.

Finally, we note that some items are inherently non-prefetchable (such as the result of filling out a form). Other items might have an "intermediate" prefetchability; for example, a "live" camera shot might be worth prefetching 1 second before the actual reference, but not 10 minutes early. Ideally, the prediction algorithm and the prefetching scheme should take these into account.

4 Experimental Methodology

We evaluate the usefulness of our prefetching scheme using simulations. We use the access logs of Digital Equipment Corporation's main Web server (`http://www.digital.com`) to drive the simulations. This is a regular `httpd` server from NCSA, so there are no GETALL or GETLIST accesses. In each run, the simulator uses the first 50000 access log entries to prime its dependency graph, without simulating prefetching. It uses the next 150000 entries to simulate the working of a real system with prefetching predictions and updates to the dependency graph. It also simulates a 100 MB LRU cache at each client. In our simulations, a clients always prefetch files that the server advises it to, except when the the file is already in the client's cache.

The following parameters are varied in the experiments:

1. The prefetch threshold, p, which is varied from 0.0 through 1.1 in steps of 0.1. The value 1.1 corresponds to no prefetching, since the weight on an arc in the dependency graph cannot exceed 1.0.

2. The lookahead window size, w, which takes on values between 2 and 10. A window size of 2 corresponds to the minimal, one-step lookahead.

3. The maximum number of URLs that `predictd` can advise a client to prefetch at any one time, i (standing for the amount of prediction "information"). This is assigned integer values from 1 through 3, and is also set to infinity, which corresponds to there being no limit.

The following performance metrics are computed in each simulation run:

1. The *average access time per file*, computed assuming a zero retrieval time on a hit in the client cache, and a retrieval time based on the models described in sections 4.1 and 4.2 on a miss.

2. The *fractional increase in network traffic*, computed as the ratio of the increase in the total number of data bytes transferred from the server to the client with prefetching, to the total without prefetching.

4.1 Network model

We need a way of estimating the time for retrieving files across the network in order to evaluate the benefits of prefetching. For this purpose, we construct a simple model of the network.

The data-pipe between the client and the server is modeled using a linear regression. Retrieving a file of size s bytes is assumed to incur a startup cost, b_0, and a per-byte cost, b_1, yielding a total time of $b_0 + s * b_1$. The startup cost includes the round-trip times for setting up a new connection, the time for sending the HTTP request, etc. The per-byte cost reflects the share of the network bandwidth available for communication between a client and a server. From the set of data points, $\{(x, y)\}$, the parameters of the linear regression can be computed as $b_1 = \frac{n \sum xy - \sum x \sum y}{n \sum x^2 - (\sum x)^2}$ and $b_0 = \frac{\sum y - b_1 \sum x}{n}$. For simplicity, we use the same network model for all clients.

In order to obtain data points for constructing the model, we instrumented a Web browser to record the retrieval time for files. We ran the browser on a host connected to an ethernet segment at UC Berkeley, and made 230 random retrievals of various sizes from Digital Equipment Corporation's main Web server, in Palo Alto, California (not far from Berkeley). Figure 4 shows the data-points and the line corresponding to a linear regression model with parameters $b_0 = 1.13$ seconds and $b_1 = 5.36 \times 10^{-5}$ seconds per byte (equivalent to a bandwidth of 149 Kbps). Our network model does not attempt to model the progress of transport (TCP) connections in detail (e.g. slow start, congestion control, etc.). From the figure, we see that this simple model fits the data quite well.

We extrapolate the above model to the case where the client host is connected via a 28.8 Kbps modem line rather than an ethernet. Based on the larger latency of the modem link and the increased transmission time for HTTP requests over this link, we estimate the startup cost b_0 to be about 1.5 seconds. Assuming the 28.8 Kbps modem line to be the bottleneck link, the per-byte cost is about 2.7×10^{-4} seconds per byte.

4.2 Retrieval Model

File retrievals, both demand-fetches (in response to explicit user requests) as well as prefetches, share the bandwidth of the client-server data-pipe. Demand-fetches are given priority over prefetches; on-going prefetches, if any, are suspended when a new user request is issued, and resumed only after all such fetches

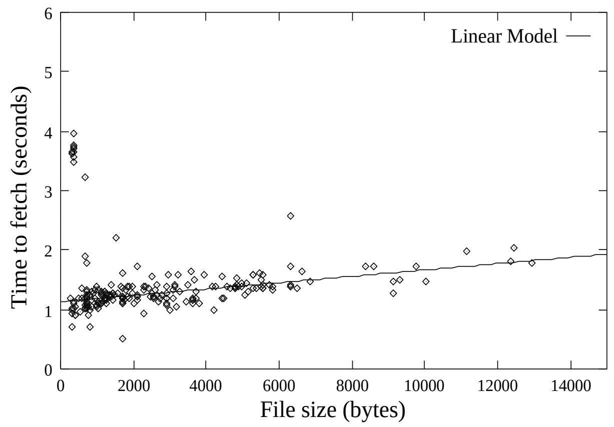

Figure 4: *A scatter plot of the time taken by a client at UC Berkeley to fetch files (Web pages, inline images, etc.) of different sizes from Digital's Web server, and the line corresponding to the linear regression model.*

have completed. In practice, the client could use separate TCP connections for the demand-fetches and the prefetches. When required, the client could throttle the prefetch connection by shrinking the TCP receiver window. However, the algorithms employed by TCP will still allow the server to send *min(congestion window, receiver window)* bytes of data before the connection is fully throttled.

We consider two different models of file retrieval over the data-pipe between a client and a server. The first is the *no-overlap* model which assumes that file retrievals happen sequentially (except for prefetches which can be suspended in the middle and resumed later). The second is the *overlap* model which allows a client to issue new retrieval requests before earlier ones have completed. The fixed startup latency of a file retrieval, which largely arises due to network round-trip delays, could be overlapped with on-going transfers. This models the effect of multiple parallel connections used by `Netscape Navigator` [7] or pipelined requests described in [8].

Finally, for simplicity we ignore the interaction between data transfers to different clients. While this could introduce inaccuracies, we believe that this is a plausible assumption for the following reason. Our measurements of the network connectivity between UC Berkeley and Digital were done in the presence of competing traffic to other clients (and, in general, other Internet traffic). Consequently, the model we developed reflects the share of the total network bandwidth that is available for the data-pipe between UC Berkeley and Digital. If there is a fair distribution of network resources, it might be reasonable to assume each client-server data-path is guaranteed this share of the network bandwidth.

5 Results

In this section, we discuss the results obtained from simulation experiments. Most of the discussion focuses on results obtained using the network model for the connectivity between UC Berkeley and Digital. However, we also present some results for prefetching across a slower, modem-speed link. Unless otherwise mentioned, the upper bound on the number of URLs that the server can advise the client to prefetch, i, is set to 3. In the discussion below, we justify this choice.

Figure 5 shows the variation of average file access time with the prefetch threshold and the lookahead window size. This is shown both for the overlap and the no-overlap network models. Increasing the

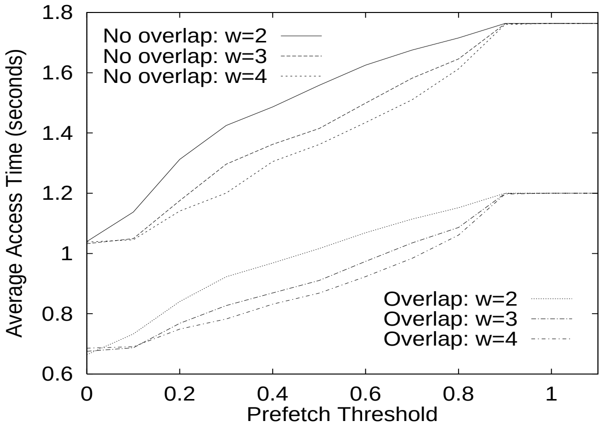

Figure 5: *The average file access time versus the prefetch threshold, p. The upper set of curves are for the no-overlap model described in Section 4.2 while the lower set is for the overlap model. Within each set, the lookahead window size, w, is also varied.*

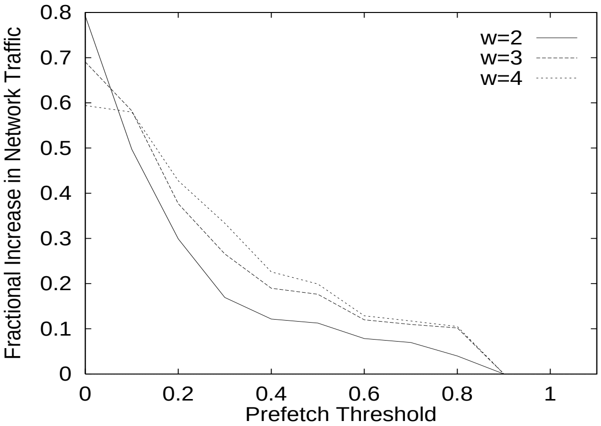

Figure 6: *The fractional increase in network traffic versus the prefetch threshold, for different values of the lookahead window size, w. The curves shown are for the overlap model; those for the no-overlap model are essentially the same.*

prefetch threshold results in less aggressive prefetching and, consequently, a larger average file access time. The access time is maximum when the prefetch threshold is larger than 1, resulting in no prefetching. Increasing the lookahead window size decreases the average access time. This is because a larger window is better able to capture dependencies between accesses to different files, including those not accessed consecutively.

The benefit of reduced access time due to prefetching comes at the cost of an increase in the amount of data transferred from the server to the clients, which we quantify in terms of the fractional increase in network traffic. As shown in figure 6, an increase in the prefetch threshold decreases this quantity whereas an increase in the lookahead window size increases it.

Assuming a no-overlap network model instead of an overlap one results in an increase in the estimated file access times. However, the relative improvement in access times is quite similar for the two models. Furthermore, the amount of network traffic is not affected by the choice of one model versus the other. In the remainder of this section, we focus on the overlap model, which is likely to be closer to reality than the no-overlap model.

It is clear that a balance needs to be struck between the improved access time and the increase in traffic. The inverse relationship between these quantities is clear from figure 7. We also note that, in general, a larger lookahead window size results in a smaller access time for a given increase in traffic. For instance, in figure 7, a window size of 4 results in better performance than the other values shown. The performance improvement derived from increasing the window size beyond 4 is limited, so we use this setting for all the other experiments.

As discussed in section 4, retrieving a file from a Web server involves a significant startup cost, which is largely independent of the network bandwidth. So just increasing the bandwidth will not reduce the access time beyond a point. Figure 8 illustrates this graphically for the UC Berkeley–Digital network model. The horizontal, dotted lines show the simulated mean access times for non-prefetching systems with available bandwidths of 100%, 120%, and 200% of that used when simulating the prefetching system. The solid curve

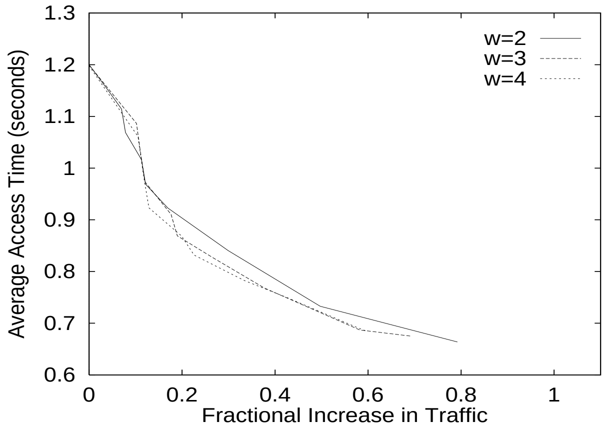

Figure 7: *The average file access time versus the fractional increase in network traffic. Curves are shown for lookahead window sizes of 2, 3 and 4.*

corresponds to prefetching with a lookahead window size of 4. It is clear from the figure that prefetching can result in lower access times compared to just increasing the available bandwidth. For instance, the figure shows that prefetching can reduce the average access time to about 0.8 seconds with a 25% increase in network traffic. In contrast, even a doubling of the bandwidth only reduces the access time to about 1 second, in the absence of prefetching.

To investigate the benefit of prefetching when the bandwidth is low, we consider the case where a 28.8 Kbps modem link is the bottleneck on the path between the client and the server. Figure 9 is the analogue of figure 8 for this case. Since the bandwidth is low, the contribution of data transmission time to the total

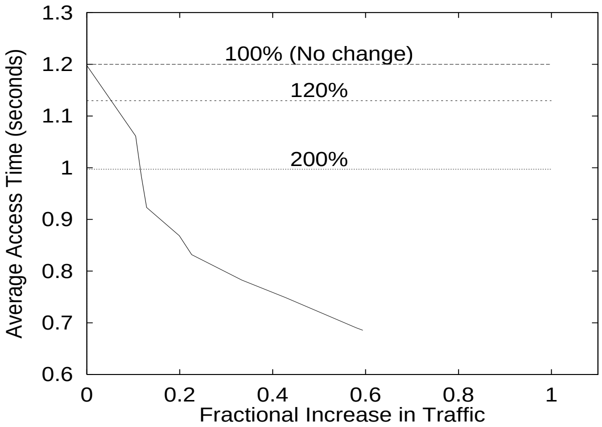

Figure 8:

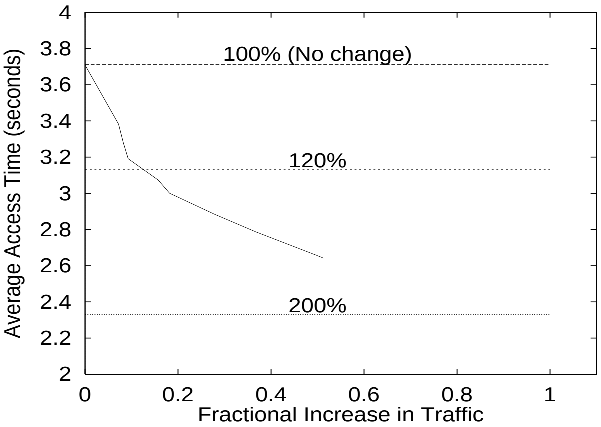

Figure 9:

The average file access time with and without prefetching, both with UC Berkeley–Digital network model (figure 8) and a 28.8 Kbps modem link (figure 9). The solid curve in each figure corresponds to the case of prefetching with a lookahead window size of 4. The horizontal, dotted lines correspond to non-prefetching systems with available bandwidths of 100%, 120%, and 200% of that used when simulating the prefetching system.

file retrieval time is significant. This explains why an increase in bandwidth reduces the average file access time more drastically than before. However, prefetching is still quite beneficial – when the prefetching threshold is set to a point that requires a 20% increase in network traffic, the resulting access time is lower

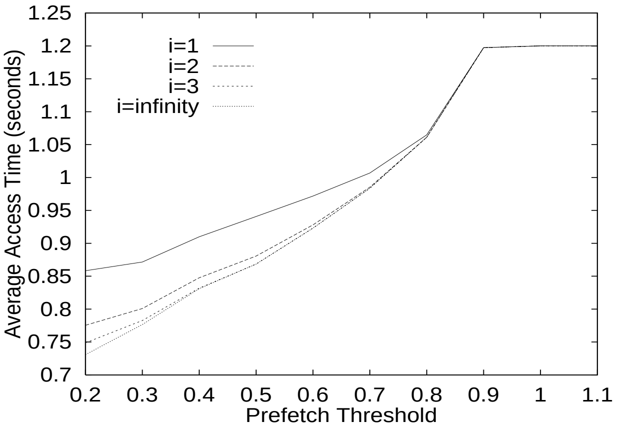

Figure 10: *The average file access time versus the prefetch threshold, p, for different values of i.*

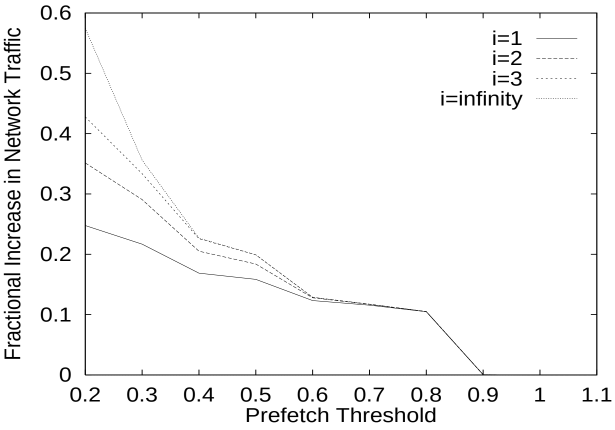

Figure 11: *The fractional increase in network traffic versus prefetch threshold, for different values of i.*

than the non-prefetching system would obtain with a 20% increase in available bandwidth.

In the discussion so far, we have set the parameter i, which determines the amount of prefetching-related information that the server can piggyback on replies to clients, to a constant value, 3. We now provide justification for this choice. Figures 10 and 11, respectively, show the average file access time and the increase in network traffic for various values of i. The ideal case is when i is set to infinity, implying that there is no limit on the amount of prediction information that the server can convey to the clients. From the figures, we see that when i is set to 3, both the traffic and the access time curves are close to those for i equal to infinity, especially when the prefetch threshold is larger than about 0.3. This indicates that setting i to a relatively small value (and, consequently, having the server send only a small amount of prefetching information to the clients) is sufficient for good performance.

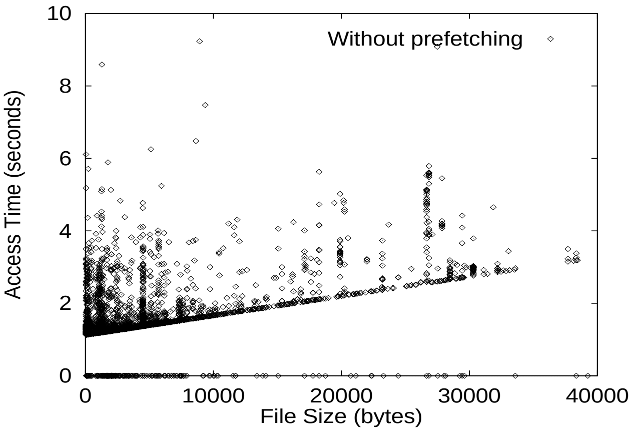

Figure 12: *A scatter plot of access time versus file size without prefetching.*

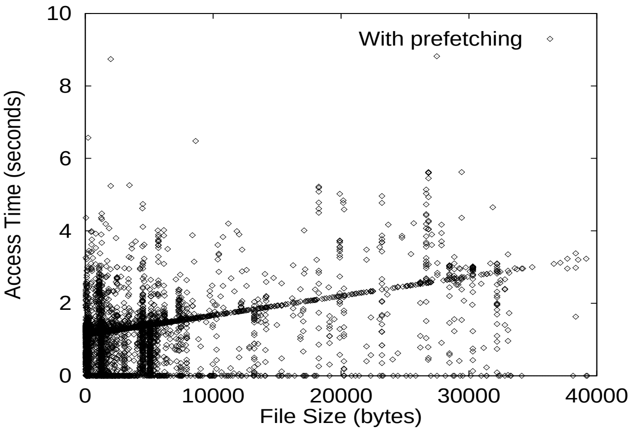

Figure 13: *A scatter plot of access time versus file size with the prefetch threshold set to 0.4 and the lookahead window size set to 4.*

Finally, we consider the effect of prefetching on the variability of file access times. From a user's perspective, it might be desirable to reduce this variability while also decreasing the average access time. Figures 12 and 13 show scatter plots of file access time versus file size without prefetching and with prefetching, respectively. Both the plots show a distinctive line corresponding to the linear model used for estimating file retrieval times. In the absence of prefetching, most of the data points either lie on this line or above it (due to queuing delays). There are also many points with zero access times corresponding to cache hits. When prefetching is done, there are numerous points that lie below the line corresponding to the linear model, because prefetching masks a part or the whole of the retrieval time. The prefetching system also yields more points with an access time of zero. These trends are evident from the distribution of access times shown in Table 1.

We quantify the variability in file access times in terms of the *standard deviation of errors*, s_e^2 [4]. Given a collection of n data-points, $\{(x, y)\}$, a simple linear regression with parameters b_0 and b_1 can be constructed (as described in section 4.1). The sum of squared errors, SSE, is then defined to be $\sum y^2 - b_0 \sum y - b_1 \sum xy$. The standard deviation of errors, s_e^2, is computed as $\sqrt{\frac{SSE}{n-2}}$. This is a measure of the deviation of the data-points from line corresponding to the linear regression. Table 1 shows the regression parameters and the standard deviation of errors corresponding to figures 12 and 13.

	Distribution of access times			Linear regression parameters		
	Zero	Small	Large	b_0 (sec)	b_1 (sec/byte)	s_e^2 (sec)
Without prefetching	20%	0%	80%	0.95	4.27×10^{-5}	1.52
With prefetching	42%	6%	52%	0.53	3.90×10^{-5}	1.60

Table 1: *The three columns to the left show the distribution of file access times in terms of three categories: zero (cache hit); non-zero but still smaller than that implied by the linear network model; and larger than that implied by the linear model. The three columns to the right show the parameters of the linear regression: the fixed cost (b_0); the per-byte cost (b_1); and the standard deviation of errors (s_e^2). Note that this linear regression is only computed for the purpose of quantifying the variation in files access times. It is not used to model access times.*

The standard deviation of errors with prefetching is 1.6 seconds, which is only slightly higher than that without prefetching (1.52 seconds). Thus prefetching can reduce the average file access time significantly without increasing the variability by much.

6 Discussion

Our results indicate that predictive prefetching of Web data can lead a significant reduction in perceived latency, but at the cost of an increase in the network traffic. Here we discuss some other issues related to prefetching.

As explained in section 3, prefetching-related predictions are made by servers which can observe the pattern of accesses from several clients. For this purpose, a server needs to maintain a dependency graph that reflects these patterns. On each client access, the server consults this data structure to make its predictions. If it is necessary to minimize the additional load imposed on the server, the construction of the dependency graph can be scheduled for off-peak hours (such as late at night). Since it is reasonable to expect client access patterns to remain stable at least for the duration of a day, we believe that such scheduling will not adversely impact the effectiveness of prefetching.

For clients that access the Web via proxy caches, prefetching can happen in two ways: between Web servers and the proxy cache, and between the proxy cache and the clients. In the latter case, the proxy cache makes the prefetching-related predictions and conveys them to the clients. One advantage that proxy caches have relative to Web servers is that they can observe client access patterns across servers.

We consider two situations where the presence of a proxy cache is advantageous from the point of view of prefetching. The first is the case where each client is connected directly to the proxy via a non-shared link, such as a modem or ISDN line. In such a situation, it would be optimal for all the idle time on the link to be filled up with prefetch traffic. However, a mechanism is needed to rapidly throttle the prefetch traffic when needed, to avoid affecting the flow of other traffic across the link.

The second case is where each client receives data via a high-bandwidth, high-latency link, such as a satellite downlink with a bandwidth of several Mbps and a latency hundreds of milliseconds. The reverse connection may be via a slow, telephone line. In such a scenario, the availability of spare bandwidth on the downlink and the large startup latency of fetching data on demand make prefetching an attractive proposition. By placing a proxy cache near the satellite ground station, throughputs close to the downlink bandwidth can be achieved between the cache and the client without increasing the load on any other part of the network.

7 Conclusions

We have presented a prefetching scheme for the World Wide Web aimed at reducing the latency perceived by users. In this scheme, the servers tell the clients which files are likely to be requested next by the user, and the clients decide whether or not to prefetch the files based on local considerations (such as the contents of the local cache).

Our simulation results show that a substantial reduction in latency perceived by a client (quantified in terms of the average time to access a file) can be achieved at the cost of a similar increase in the network traffic. Since the retrieval time of a file includes a substantial startup latency, prefetching is often more effective in reducing the access time than just increasing the bandwidth.

We conclude that prefetching might be worthwhile, especially when increasing bandwidth demands do not significantly degrade service for other users nor increase the cost for service. Two scenarios in which prefetching might be especially useful involve clients connected to a proxy cache via a non-shared modem or ISDN line, or via a high-bandwidth and high-latency satellite downlink.

To support prefetching, the HTTP protocol could be enhanced to allow servers to piggyback prefetching hints on replies to clients. Also, it would help scheduling at a server if prefetches could be distinguished from demand-fetches, for instance to give them a lower priority.

Acknowledgements

Much of this work was done using the facilities of Digital's Western Research Lab, Network Systems Lab and Cambridge Research Lab. Thomas Kroeger and Carlos Maltzahn provided useful comments on a draft of this paper.

References

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

[2] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee. "Hypertext Transfer Protocol – HTTP/1.1", *Internet Draft draft-ietf-http-v11-spec-01.txt, IETF,*, June, 1996. This is a working draft.

[3] James Griffioen and Randy Appleton. "Reducing File System Latency using a Predictive Approach", *Proceedings of the 1994 Summer USENIX Technical Conference*, Cambridge MA, June, 1994.

[4] Raj Jain. "The Art of Computer Systems Performance Analysis", *John Wiley & Sons, Inc.*, 1991.

[5] David M. Kristol. "Proposed HTTP State-Info Mechanism", *Internet Draft draft-kristol-http-state-info-01.txt, IETF,* September, 1995. This is a working draft.

[6] Jeffrey C. Mogul. "The Case for Persistent-Connection HTTP", *Proceedings of the ACM SIGCOMM Conference*, Boston, MA, August, 1995.

[7] Netscape Communications Corporations, *http://www.netscape.com*, 1996.

[8] Venkata N. Padmanabhan and Jeffrey C. Mogul. "Improving HTTP Latency", *Proceedings of the Second International World Wide Web Conference, Chicago, IL*, pages 995-1005, October, 1994. (An updated version appeared in *Computer Networks and ISDN Systems*, v.28, nos.1&2, December 1995, pp. 25-35.)

[9] Venkata N. Padmanabhan. "Improving World Wide Web Latency", *Technical Report UCB/CSD-95-875, Computer Science Division, University of California, Berkeley, CA*, May, 1995

[10] J.Postel. "Transmission Control Protocol", *RFC 793, Network Information Center, SRI International*, September, 1981.

[11] Simon E. Spero. "Analysis of HTTP Performance Problems", *URL http://sunsite.unc.edu/mdma-release/http-prob.html*, July, 1994.