Published in the Proc. of the 1998 Computer Measurement Group Conference, Anaheim, CA, Dec. 6-11, 1998.

PREFETCHING INLINES TO IMPROVE WEB SERVER LATENCY

Ronald Dodge
US Army
7506 Irene Ct
Springfield, VA 22153
rdodge@gmu.edu

Daniel A. Menascé
Dept. of Computer Science, MS 4A5
George Mason University
Fairfax, VA 22030-4444
menasce@cne.gmu.edu

Most HTML documents contain inlines, typically image files that are automatically requested from the server after the original document is parsed by the browser. In this paper, we analyze, through simulation, the potential benefits of having the server also parse the document and pre-fetch into its main memory cache the inlines that will be requested by the remote client. The parameters for the simulation were obtained from a detailed workload characterization of a Web server available to the authors. Response time improvements of up to 48% can be achieved with even moderate cache sizes.

1. Introduction

The Web has experienced an exponential growth since its inception. Recent statistics from a 1997 Nielsen Media Research study put the number of Web users in the US at 45 million and in Canada at 5 million. Applications such as electronic commerce and digital libraries will help to further increase the growth rate. Electronic commerce is forecast to be a $6.9 billion industry by the year 2,000. Currently, 85% of America's Fortune 100 companies have a presence on the Web and spend between $0.75 and $1.25 million on their Web sites.

Web site designers and managers have two main concerns: content and performance. Content is very important to keep users interested in the site and in the products and services it has to offer. But, if the response time perceived by a user is too long, he or she will leave the site with a click of the mouse and the company will have lost a potential customer. A detailed discussion on capacity planning for Web performance can be found in [MA98].

Of the many techniques aimed at reducing this response time (or *latency*), caching is one of the most cost-effective. Caching entails keeping a copy of a recently accessed object at a location (the cache) from where it can be retrieved faster than from its original site (the originating server's disk). Caches are commonly located at the client browser and at proxy servers [ASAWF95] (located within the user's intranet). Caches located at the client browser are commonly implement by using main memory or hard disk storage where proxy server caches are implemented using local hard disk storage.

Caches can also be used in a pre-fetching mode. For example, if there is a way to predict what documents the user will request in the future, these documents can be fetched from the originating server ahead of time so that they will be already in the cache when requested.

This paper discusses the use of pre-fetching inlines from HTML documents (see section 2.0) into the server's main memory cache, reducing disk access time when these, typically large files, are requested by the web browser. Discrete event simulation using CSim (a C/C++ simulation package) was used to study the performance benefits of pre-fetching inlines.

Others have studied pre-fetching on the Web in different contexts. Padmanabhan studied the effects of prefetching files on latency, network load, and service to other users [PADM95]. Crovella and Barford suggested that by adjusting the rate at which files are prefetched, the additional network load can be minimized [CB97]. Chinen and Yamaguchi developed a prefetching proxy server, and studied the effects of retrieving the first N links within *any* Web page viewed by a user [CY96]. Foxwell and Menascé [FM98] studied the potential benefits of prefetching results of queries to search engines.

The rest of this paper is organized as follows. Section two provides the motivation for the potential benefits of prefetching inline requests. Section three describes the workload characterization carried out to obtain parameters for the simulation. Section four describes the simulation. Section five presents and analyzes the results obtained with the simulation and finally, section six provides concluding remarks.

2.0 Caching and Prefetching on the Web

Figure 1 displays the anatomy of a regular HTTP transaction. The browser sends a request for an HTML document to the server. The server retrieves the document from disk and returns it to the browser. The browser parses the HTML document and requests all inlines (typically large image files), from the server. These inlines are then retrieved from disk at the server and returned to the client. See [YEAGER96, MA98] for more details on Web transactions.

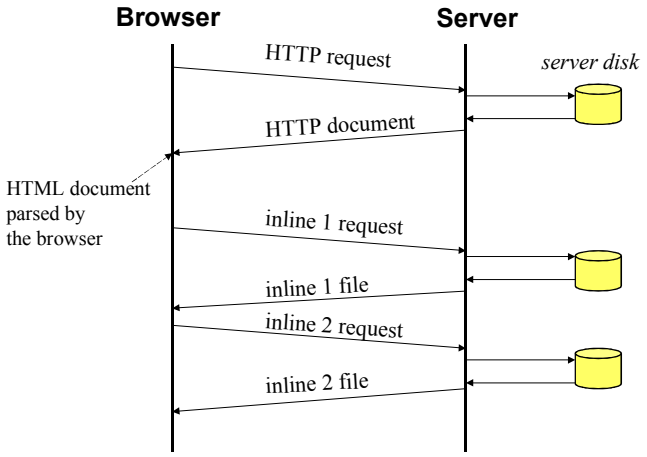

Figure 1 - Anatomy of a typical HTTP request.

In the scheme depicted in figure 1, the disk at the server has to be accessed for each inline and the user-perceived latency includes the congestion at the disk as well as the access time. We suggest a different scheme shown in figure 2. In this approach, the server also parses the request while the HTML document is sent to the browser and prefetches the inlines from disk into its main memory cache. This way, when the requests for inlines arrive at the server, they can be served from the cache immediately without having to queue for the disk.

The following sections assess the potential benefits of prefetching inlines at the server. As mentioned earlier, the analysis was carried out with the help of a simulation program written in C using the simulation package CSim. We wanted to drive the simulation with parameters typical of a Web server. For that purpose, we analyzed the HTTP log of a Web server at George Mason University to characterize its workload. The results of this characterization are described in the following section.

3.0 Workload Characterization

Many studies have characterized the workload of Web servers [AAY97,AFAW97,AW96,BC94,CB96,CJ97]. These studies have concluded that many of the relevant parameters of a Web server traffic, such as the size of files requested, have a power tail. By this we mean that the tail of the cumulative distribution function decreases with a power law. Our findings support this conclusion as shown in this section.

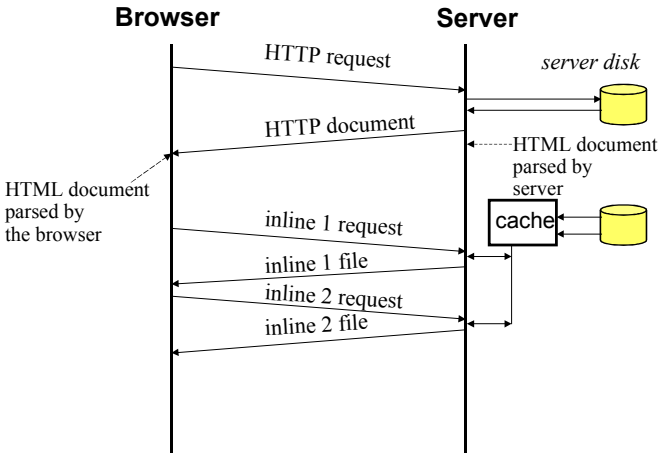

Figure 2 - Anatomy of a Web transaction with inline prefetching.

The Web server used for the workload characterization hosts mostly student/faculty HTML documents. The data was obtained from a 12 hr period on an average day. The server's HTTP logs contain information on

- the requester's address,
- the request arrival time,
- the type of request,
- the file requested (for GET requests), and
- the size of the requested file.

Additional information was obtained on the disk service parameters and the average CPU utilization due to the web server.

We describe in this section the workload characterization carried out to obtain distributions for the random variables:

- $\widetilde{T}$: inter-arrival time (in sec) of html page requests (for the purpose of statistics gathering, html pages include .html, .htm, and pages with no extension, e.g. the default index.html),
- $\widetilde{H}$: html file size (in units of 500 bytes),
- $\widetilde{N}$: the number of inline requests per HTML page,
- $\widetilde{I}$: inline file size (in units of 1,000 bytes), and
- $\widetilde{F}$: number of different files requested between consecutive requests to the same file.

The following subsections display the cumulative distributions functions (cdf) and probability density functions (PDF) for the data obtained from the

measurements. The expressions for the cdf derived through curve fitting methods are also provided.

3.1 Inter-Arrival Time

The inter-arrival times of HTML requests is derived by subtracting the request's time stamp from the previous request's time stamp. Once each request has an associated inter-arrival time, we built a histogram of data using a bin width of 1 second. The data shows an average inter-arrival time of 2 seconds and a maximum inter-arrival time of 843 seconds. The cumulative distribution function was derived similarly. The estimated cdf and PDF for the raw data are plotted in figure 3.

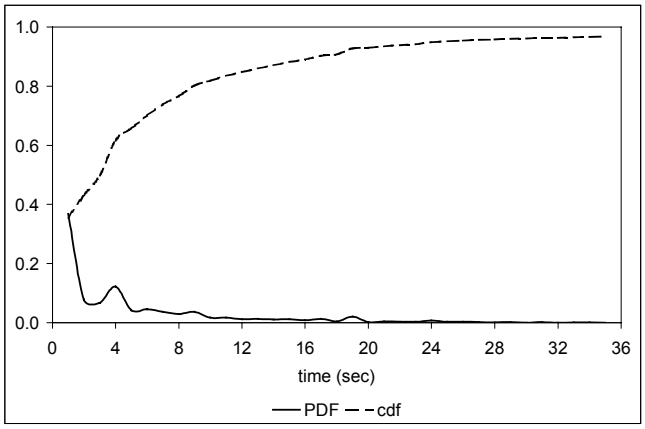

Figure 3 - Distribution of HTML request's interarrival time.

In the simulation, the random variable $\widetilde{T}$ used to represent the interarrival time of HTML requests is given by

$$F_{\widetilde{T}}(x) = 1 - e^{-0.4493x}.$$

This function was derived using the built-in regression tools in Excel and yielded an exponential curve fit with an R^2 of 0.85.

3.2 HTML File Size

The distribution function for the size of an HTML file was derived identically to the inter-arrival functions. In this instance, the bin width used is 500 bytes. The average size was 3,552 bytes, and the maximum size was 150,912 bytes. After comparing several curves to the real data, a power series curve best fits the data, with a .80 R^2. The cdf and PDF graphs for the data are shown in figure 4.

In the simulation, the random variable $\widetilde{H}$, that measures the size of a requested HTML document, in units of 500 bytes, has the following cdf:

$$F_{\widetilde{H}}(x) = 1 - .753x^{-1.724}$$

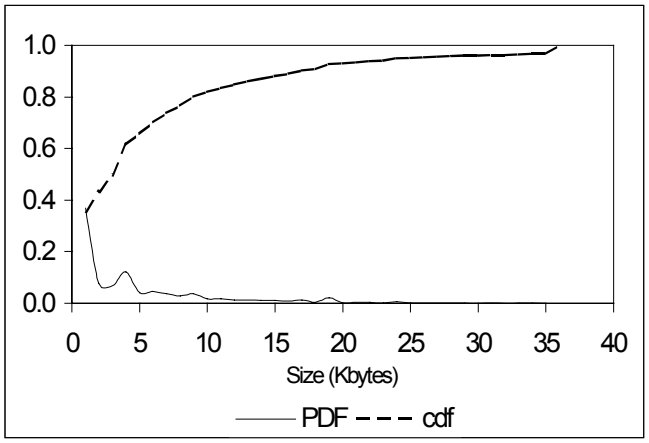

Figure 4 - Distribution of the size (in bytes) of HTML documents requested.

3.4 Number of Inline Requests per HTML Page

The distribution function for the number of inline requests per HTML page is again derived identically to the inter-arrival function. In this case, the bin width used is 1 request. The average number of requests per HTML page was 2.92 with a maximum of 84. The estimated cdf and PDF for the number of inlines per HTML page is shown in figure 5.

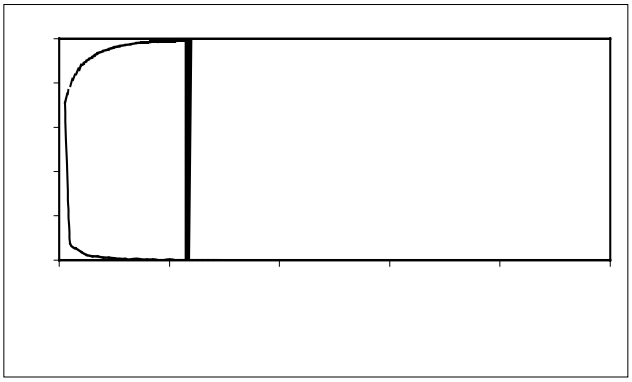

Figure 5 - Distribution of number of inlines per HTML document requested.

A power series provided the best fit to the data in figure 5 with an R^2 of 0.95. The resulting cdf used in the simulation for the random variable $\widetilde{N}$ is

$$F_{\widetilde{N}}(x) = 1 - .748x^{-2.163}$$

3.5 Inline File Size

The distribution function for inline file sizes is derived identically to the HTML file size distribution function (see subsection 3.2). However in this case, the bin

width used is 1,000 bytes. The average size is 10,392 bytes, and the maximum size is 1,507,328 bytes. After comparing several curves to the real data, again a power series curve best fits the data, with a .90 R^2. The graph for the cdf and PDF obtained from the measurement data is shown in figure 6.

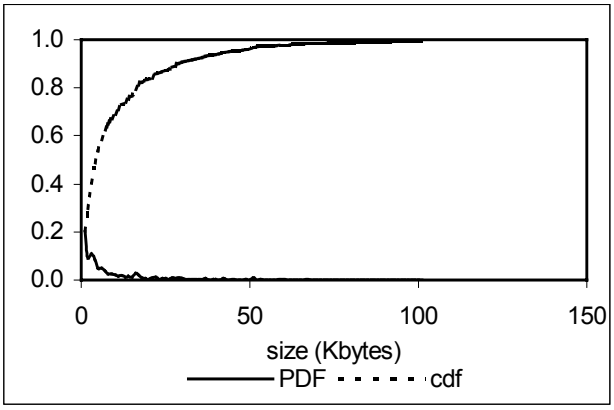

Figure 6 - Distribution of the size (in bytes) of inlines.

The cdf for the random variable $\tilde{I}$ is given by

$$F_{\tilde{I}}(x) = 1 - .754x^{-1.473}$$

3.6 Duplicate Requests

The determination of a distribution function for how often a request is duplicated is critical to the simulation. After all, in order for caching to be useful, files must not only be placed into the cache, but also read from the cache!

Our approach was to count the number of HTML files requested between duplicate requests. For example if file A.html was requested, followed by 3 other HTML file requests (e.g., B.html, C.html, and D.html) before A.html was requested again, we would record an entry in bin # 3 (for 3 inter-file requests). The analysis was repeated for each html request. We then followed a similar derivation as before to produce a cumulative distribution function. A power series curve best fits the data, with a .99 R^2.

The use of the cdf in the simulation is slightly different than in the earlier sections. First, the simulator decides whether the next file to be requested will be a repeat request. Analysis of the data shows that 24% of the time a previous request is repeated. So, with probability .24, the simulator generates a duplicate request. The simulator then draws a number from the inter-file distribution to tell the client how many requests to look back to find the duplicate request to send. An assumption is made that all in-line requests are also generated. The graph for the estimated cdf and PDF for the inter-reference file distribution is shown in figure 7.

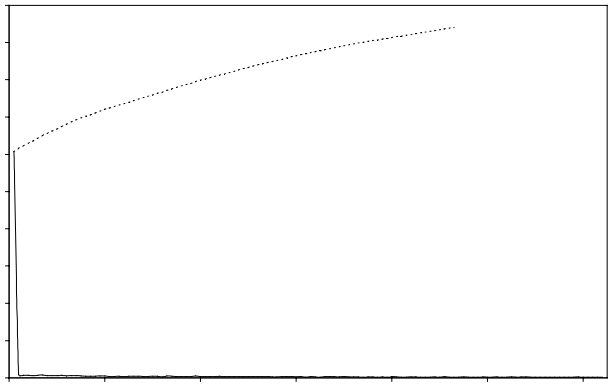

Figure 7 - File inter-reference distribution

The function used to fit the cdf for the random variable $\tilde{F}$ in the interval of interest is given by

$$F_{\tilde{F}}(x) = -.00002x^2 + .005x + .6204 .$$

Note: the above equation is only valid in the range 0 < x < 135; this covers 93% of the cases. This is equivalent to assuming that if a file is referenced again after 135 other files have been referenced, we do not consider the second reference as being a duplicate.

An interesting point to note is that an examination of the raw data reveals that duplicate requests were linked to repeat clients. In other words, when the server received a duplicate request, most of the time (85%) it came from the same client. Since the server hosts mostly personal HTML documents, this model of relatively few clients accessing a single HTML document is appropriate. This assumption would change dramatically in a business or database server model where many clients access the same HTML file. The conclusions in this paper are applicable to both cases and would theoretically provide even greater performance benefits in the later model.

3.7 Disk access

Determining the amount of time elapsed when a file is retrieved from a disk by the server is the final distribution needed to conduct the simulation. This time depends on various disk performance parameters such as seek time, rotational latency, transfer time, and block size. The disk used by GMU's web server has the following values for the performance parameters: average seek equal to 9.5ms, average rotational delay equal to 4.17ms, transfer rate equal to 15 Mbytes/sec, and block size equal to 512 bytes.

4.0 The Simulation

The goal of this simulation is to provide an environment where accurate statistics can be gathered using a variety of input parameters. We used the simulation to examine the performance characteristics, under various loads, of a Web Server that performs pre-fetching of inlines and caches them locally using a Least Recently Used (LRU) cache replacement policy.

The simulation was designed to handle a varying number of clients and servers, different cache sizes, and different values for other parameters such as the bandwidth of the network connecting clients to servers. Figure 8 shows the queuing network (QN) simulated. One or more clients submit request to a Web server through a network. The server is composed of a CPU, one disk, and an in-memory cache. Requests that find the file in the cache (a hit) do not need to use the disk.

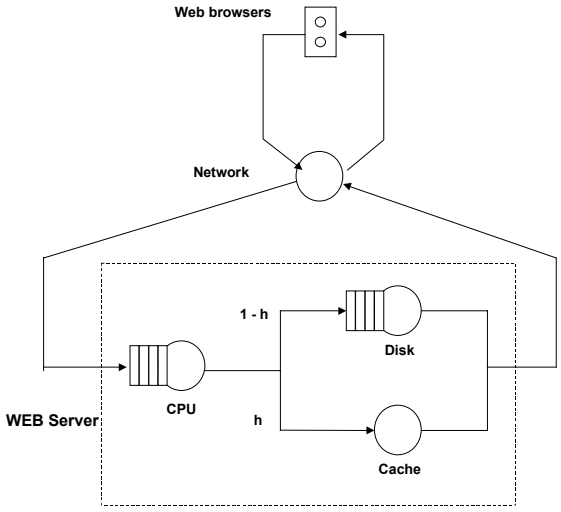

Figure 8 - QN model of clients and Web server used in the simulation.

4.1 Implementation

We used CSim [CSIM97] as the simulation package for the experiments. CSim offers a wealth of C and C++ functions for discrete event simulation and statistics gathering. The simulation is broken down into three sections:

- Simulation driver
- Client simulation
- Server simulation

The pseudo code for each of these sections is given in the appendix.

4.1.1 Simulation Driver

The simulation driver initializes the simulation. The driver is responsible for:

- initializing all global variables,
- creating all tables and other statistic gathering tools,
- starting the indicated number of clients and servers,
- and generating a report upon completion of the simulation.

4.1.2 Client

The client section of the simulation required handling many different scenarios, from a duplicate request, to a request with any number of inline requests, to accommodating several bandwidths. These requirements are accomplished by a series of functions that the client uses to build its request. The client uses two linked lists to maintain information about the requests it has sent. The first list maintains all pertinent HTML file information and the second maintains all inline file information.

4.1.3 Server

The server section immediately enters into a receive mode where it waits for requests from the clients. When it receives a request, the server notes the time (to be used in utilization computations) and then determines what type of request was received. Except for the final request (sent by a generic process that comes alive when the simulation time has expired), all requests are of type GET. The server then checks to see if the requested file is in memory (by traversing a linked list); if so, it sends the file to the client and updates the files' position in the cache list (for LRU policy). If the file is not in memory, the server retrieves the file from disk (making note of the disk service time) and forwards the file to the client. The server then, if the file is smaller than the cache size, places the HTML request in cache (if there is not enough available memory the least recently used file is removed until there is enough room). Any inline files that are identified by parsing the HTML document likewise have their position in the cache updated or are retrieved from disk and placed into the cache. The server continues to accept incoming requests until it receives a quit message from the timer function. The time required to parse the HTML requests was not included in the simulation because it was deemed negligible when compared with disk and network times.

4.2 Simulation Experiment

We ran the simulation for all combinations of number of clients and cache size indicated in Table 1 with a single server. As workload input, we used the previously discussed distribution functions to replicate the real data observed from the Mason web server. All clients were assumed to be equivalent, i.e., they drew

their request parameters from the same distribution. The simulation time was set to produce a similar number of requests (under the lowest client load) as found in the real data (70,000 requests).

Table 1 - Range of Parameters for the Simulation.

Server Cache Size (Kbytes):							
0	16	32	64	128	256	512	1,024
Number of clients:							
5	10	15	20	25			

The following subsection discusses the simulation validation. Results are presented and discussed in section five.

4.3 Simulation Output Analysis

Batch means analysis [MACD87] was used to compute confidence intervals for the simulation. The batch size used was 1,000 requests. The total number of requests simulated was on the order of 60,000 including HTML and inline requests. Warmup effects were dealt with by deleting the first 100 requests of the first batch. The specified accuracy was 10% and the confidence level was 95%. The half-width interval for the response time of inline requests was about 1.79% of the batch mean for the last batch. For HTML requests, the half-width interval for the response time was 2.46% of the batch mean for the last batch.

4.4 Simulation Validation

To validate the simulation we observed the resulting distributions of the input parameters from a no-caching run. These results were then compared to the baseline data. The resulting graphs were in very close agreement with the input distributions for all parameters.

5.0 Results

The first set of results to be considered is the response time seen by the client. The response time is broken down into the HTML request response time, inline file request response time, and the total response time for a request (time to receive the HTML file plus all its inlines).

Figure 9 shows the average response times for an HTML file request. As expected, as the cache size increases, the response time for all client populations decreases. Note that the performance benefit of additional cache begins to lessen at 128Kbytes. The 25-client population data set shows a 48% reduction in response time between a 0 (no-caching case) and a 128-Kbyte cache. Between 128-Kbyte and 1,024-Kbyte caches, the response time dropped by 15%.

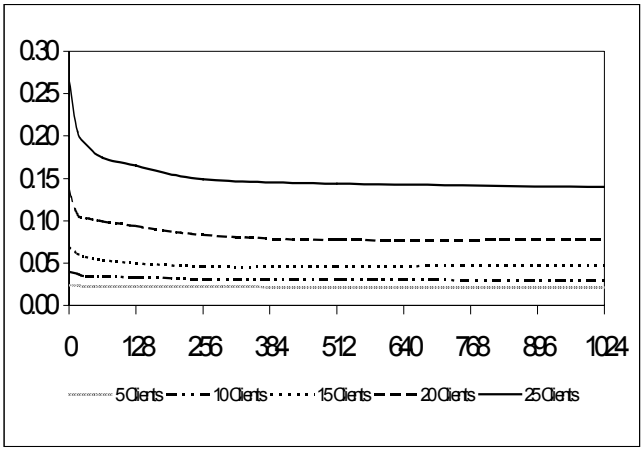

Figure 9 - Response time (in sec) of HTML document requests vs. cache size (in Kbytes)

The observed response times for in-line file requests, shown in figure 10, behave similarly to the HTML file requests. In this case, the reduction in response time between a 0 and 128-Kbyte cache is 46%, while the response time drops only 20% between 128 and 1024 Kbytes.

The total response time, depicted in figure 11, again behaves as expected. Note that the total response time is not simply the sum of the response time of HTML requests and inline requests due to the fact that several in-line requests may be processed for each HTML request. As before, the performance improvement decreases significantly for caches larger than 128 Kbytes.

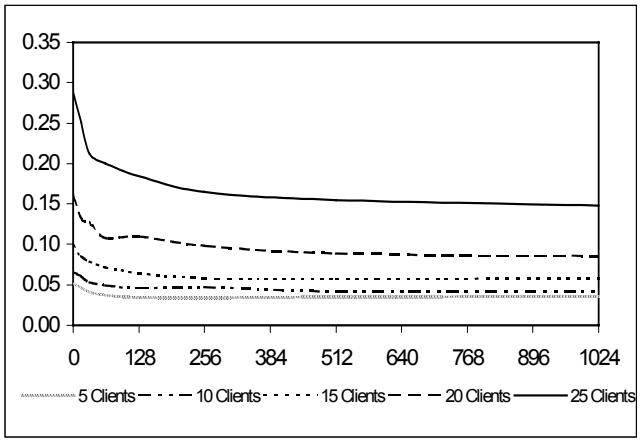

Figure 10 - Response time (in sec) of inline files vs. cache size (in Kbytes)

The following graphs (figures 12 and 13) display the probability of finding a file in the cache (file cache hit ratio). Intuitively, one would expect that as the cache size increases, the probability of finding a duplicate request in the cache would likewise increase. While this is the case, there is a point of diminishing returns for caches larger than 128 Kbytes. Similar results are shown with the byte hit ratio, defined as the number of

bytes found in the cache divided by the total number of bytes retrieved from the server.

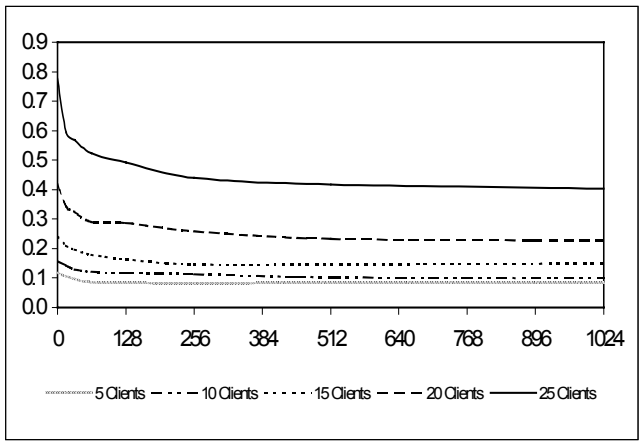

Figure 11 - Total Response time (in sec) vs. cache size (in Kbytes)

Finally, figure 14 examines the effect that additional cache memory has on the utilization of the disk at the server. At the beginning, as the cache size increases, the utilization decreases as the cache hit ratio increases (see figures 12 and 13). Then, when the cache hit ratio stabilizes, the disk utilization does not vary any longer with the cache size.

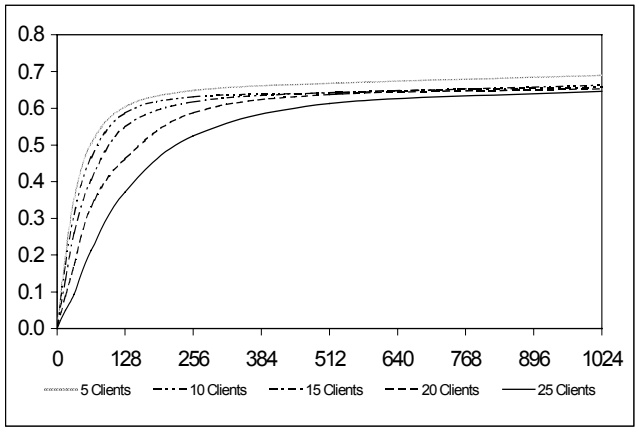

Figure 12 - File Hit Ratio vs. cache size (in Kbytes).

Additional consideration was given to distinguishing the performance advantages gained by prefetching over the performance gains of caching alone. To accomplish this we ran two more sets of simulations, increasing the client population to 100 (to increase the load on the server). The first simulation set only performed caching of requested files. The second set conducted both prefetching and caching. As before, the cache size was increased from 0 to 1024 Kbytes on each simulation run.

The results (Figure 15) showed a consistent 15% performance advantage by performing both prefetching and caching. The high response time is due a high server CPU utilization (80%) caused by the increased client population.

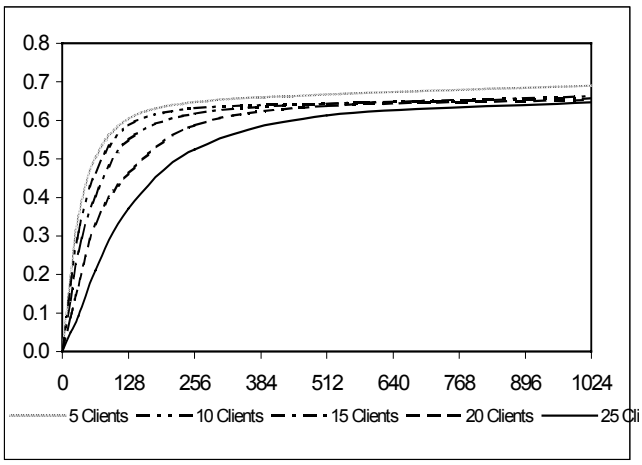

Figure 13 - Byte Hit Ratio vs. cache size (in Kbytes).

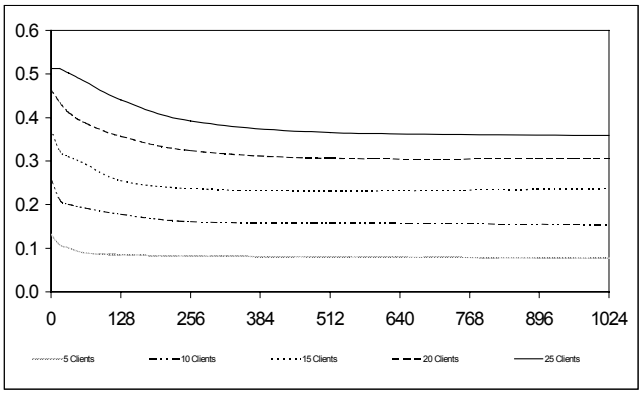

Figure 14 - Server disk utilization vs. cache size (in Kbytes).

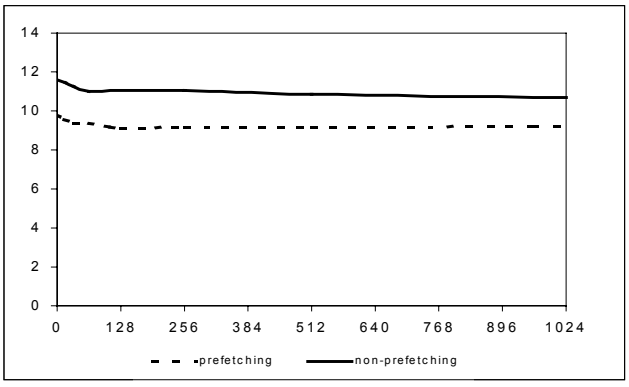

Figure 15 - Total Response time (in sec) vs. cache size (in Kbytes).

6.0 Conclusions and Future Research

Many techniques have been explored to improve the performance of Web servers at a time when traffic is

increasing at exponential rates. Caching is one of these techniques. Prefetching can be used in conjunction with caching to decrease user perceived latency.

This paper analyzed the potential benefits of prefetching inlines into the server's main memory cache. Simulation was used to model this situation. The simulation was driven by parameters that mimic the workload of a real web server. Results showed that, in the model used, a 48% improvement in response time can be obtained through prefetching of inlines. The proposed modification would not require changing the HTTP protocol nor the browsers, therefore could be easily adopted.

This paper only examined the potential benefits in response time to the client. Other areas impacted by the prefetching scheme presented such as, the cost of additional memory cache at the server or the effect on network resources, due to higher throughput are potential areas for further study.

References

[AAY97] J. Almeida, V. Almeida, D. Yates, "Measuring the Behavior of a World Wide Web server, IFIP, New York, April 1997.

[AFAW97] Abdulla, G., Fox, E., Abrams, M., and Williams, S. "WWW Proxy Traffic Characterization with Application to Caching", 1997.

[ASAWF95] Abrams, M., Standridge, C., Abdulla, G., Williams, S., and Fox, E. "Caching Proxies: Limitations and Potentials", Computer Science Department, Virginia Tech, 1995.

[AW96] M. Arlitt and G. Williamson, "Web Workload Characterization", *Proc. of the 1996 SIGMETRICS Conf. Measurement Comput. Syst*., ACM, Philadelphia, May 1996.

[BC94] Braun, H., and Claffy, K. "Web Traffic Characterization: An Assessment of the Impact of Caching Documents from NCSA's Web Server", *Proc. of the 2nd International WWW Conference*, Chicago, 1994.

[CY96] Chinen, K., and Yamaguchi, S. "An Interactive Prefetching Proxy Server for Improvements of WWW Latency", Nara Institute of Science and Technology, 1996.

[CB96] M. Crovella and A. Bestravos, Self-Similarity in World Wide Web Traffic: evidence and possible causes", *Proc. of the 1996 SIGMETRICS Conf. Measurement Comput. Syst.,* ACM, Philadelphia, May 1996.

[CB97] Crovella, M. and Barford, P. "The Network Effects of Prefetching", Computer Science Department, Boston University, 1997.

[CJ97] Cunha, C. and Jaccoud, C. "Determining WWW User's Next Access and Its Application to Prefetching", Computer Science Department, Boston University, 1997.

[CSIM97] Schwetman, Herb. "CSIM Primer", www.mesquite.com.

[FM98] Foxwell, H. and D. A. Menascé, Prefetching Results of Web Searches, *Proc. of the 1998 Computer Measurement Group Conference*, Anaheim, CA, Dec. 6-11, 1998.

[MACD87] MacDougall, M. H., *Simulating Computer Systems*, MIT Press, 1987.

[MAD94] Menascé, D. A., V. A. F. Almeida, and L. W. Dowdy, *Capacity Planning and Performance Modeling: from mainframes to client-server systems*," Prentice Hall, Upper Saddle River, NJ, 1994.

[MA98] Menascé, D. A. and V. A. F. Almeida, *Capacity Planning for Web Performance: metrics, models, and methods*," Prentice Hall, Upper Saddle River, NJ, 1998.

[PADM95] Padmanabhan, V., "Improving World Wide Web Latency", Computer Science Division, University of California at Berkeley, 1995.

[YEAGER96] N. Yeager and R. McGrath, *Web Server Technology,* Morgan Kaufmann, San Francisco. 1996.

Appendix: Pseudo-code for the simulator

The pseudo code for the simulation driver is:

```
Initialize variables
Create tables and processes
Initialize tables
Start servers
Start clients
Wait for completion
Submit reports
```

Below is the pseudo code for the client:

```
determine the HTML file to be requested
if (requested file is not a duplicate)
    make file
    add to tail of list
    send request for file
else
    draw file from inter-file distribution /* determine
          how many requests ago the file was
          requested */
     traverse the file list backwards the number of
          files indicated by the distribution
     /* if NULL is encountered a new file is
       generated */
    add file to the end of the list
receive response from server
```

```
InLine = First_Inline
while (InLine <> null) {
      request InLine
      wait for server response
      InLine = next_inline
wait inter-arrival time
```

The sequence above is repeated for the duration of the simulation.

```
if (request is HTML) {
    if (file is in memory) {
        send file;
        move file to head of list
        parse the file
        InLine = First_InLine
        while (InLine <> null)  {
            if (InLine in memory)
                move to head of list
            else {
                read InLine from disk
                while (InLine size > available_memory) {
                  available memory =  available_memory -
                                size of InLine to be removed
                    remove file at the tail of the list
                }
                put new file at head of list
            }
        }
    }
    else { /* HTML file not in memory */
        read HTML file from disk
        send file
        while (HTML size > available_memory) {
                  available memory =  available_memory -
                                size of HTML to be removed
                    remove file at the tail of the list
                }
        put new file at head of list
        parse the HTML file
        /* traverse linked list of inlines */
        InLine = First_InLine
        while (InLine <> null)  {
            if (InLine in memory)
                move to head of list
            else {
                read InLine from disk
                while (InLine size > available_memory) {
                  available memory =  available_memory -
                                size of InLine to be removed
                    remove file at the tail of the liSt
                }
                put new file at head of list
            }
        }
else  {/* file is an inline */
    if inline is in memory
        send file
        move file to head of list
    else {
        read file from disk
        send file to client
        while (InLine size > available_memory) {
                  available memory =  available_memory -
                                size of InLine to be removed
                    remove file at the tail of the list
                }
        put new file at head of list
        }
}
```