1-1-2007

Prefetching in HTTP to improve performance in the delivery of web pages

Adam Serbinski
Ryerson University

b18195192

TK
5105.5
.S47
2007

PREFETCHING IN HTTP TO IMPROVE PERFORMANCE IN THE DELIVERY OF WEB PAGES

by

Adam Serbinski

B.Sc. (Ryerson University, Toronto) 2005

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Applied Science

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2007

Prefetching in HTTP to Improve Performance in the Delivery of Web Pages
Master of Applied Science, 2007
Adam Serbinski
Electrical and Computer Engineering
Ryerson University

ABSTRACT

In this work, two enhancements are presented to improve the performance of complete web page delivery. The first enhancement is the addition of disk to memory prefetching in Apache HTTP Server. This results in the number of files served from the web server cache to increase by 27%. The second enhancement is an addition to the HTTP protocol, allowing for server to client prefetching. Server to client prefetching eliminates the need for the web client to request the embedded objects of web pages. This modification results in page load times reduced to as low as 24.5% of the non-prefetching page load times.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

BACKGROUND INFORMATION AND PREVIOUS WORK

This work is on the modification of HTTP web server software and HTTP protocol to improve the delivery performance of web pages. This chapter is an introduction to HTTP, Apache HTTP Server, and a performance enhancement technique known as prefetching. We also introduce previous and related works that have been done in the area.

Background information is presented in Section 1.1. Background information on the HTML document format is discussed in Section 1.1.1. Background information on the HTTP Protocol is in Section 1.1.2. Information on Apache HTTP Server is presented in Section 1.1.3. The prefetching technique for improving data delivery is discussed in Section 1.1.4.

Section 1.2 shows previous and related work in the areas of prefetching and Internet performance. In section 1.2.1, previous work on prefetching data internally, such as from disk to memory, is discussed. Section 1.2.2 presents previous work on prefetching data across networks. Development and use of custom HTTP servers is discussed in Section 1.2.3. Previous work on improving the delivery of multimedia data on the Internet is shown in Section 1.2.4. The contributions of this thesis are shown in Section 1.3. Section 1.4 outlines the rest of this thesis.

1.1. BACKGROUND INFORMATION

1.1.1. HTML

Most web pages are organized in Hyper Text Markup Language, HTML documents [1]. HTML is the most common file type for distribution by HTTP Servers. The HTML contains text information specifying formatting, text to be displayed, and other files required to properly display the web page. The other files referenced by the HTML may include images, sounds, videos, and other types of files that the web page requires.

To display an HTML web page in a web browser, first, an HTML file is retrieved from an HTTP server. The browser then displays what it has already received, while parsing the HTML file for additional file requirements. When the browser is able to identify what other files are required, it retrieves those files from the HTTP servers on which they are located. Embedded files may or may not be from the same server as the server from where the HTML file was retrieved. When the additional files are retrieved, the web browser displays them as specified in the HTML file.

1.1.2. HTTP PROTOCOL

There are many protocols in use on the Internet, but the one most commonly recognized is HTTP. HTTP protocol is a connection based request/response protocol for serving files to client software, in particular, web browsers [2]. HTTP utilizes the lower level TCP protocol.

In an HTTP request, the client first establishes a TCP connection to the server, it then sends a request to the server, usually a GET request, and the server sends a response back to the client along the same connection. In HTTP 1.0 [3], once the server sends its response, the connection is closed, so if the client requires multiple files from that server, it must create a unique connection for each file. In HTTP 1.1 [4], the connection may remain open after the server responds, allowing the client to request additional files on the same connection.

The basic methods supported by HTTP are GET, HEAD, and POST [3]. The most commonly used, being GET, which replies to the client with a specifications header showing the details about the requested file, and with the file itself. HEAD is similar to GET, it replies to the client with the same header as with GET, but it does not attach the requested file. POST allows the client to upload data to the server.

An HTTP GET request consists of the client sending a request header to the HTTP server for a specific file that may be hosted by that server. The most basic GET request consists of the following;

```
GET / HTTP/1.0
```

The first part of the request is the method being used, in this case, GET. The second part is the path to the file on the server, if no filename is given, the server selects a default file, usually *index.html*. The third part is the protocol and version, in this case HTTP 1.0.

Additional lines may be added to the request header in case the server has to consider special characteristics for the client, such as identification of the web browser, or the client's compatibility with various special features, such as data compression. The following request header is generated by Mozilla Firefox 2.0.0.2 when requesting www.google.ca;

```
GET / HTTP/1.1
Host: www.google.ca
User-Agent: Mozilla/5.0 (X11; U; Linux i686 (x86_64); en-US; rv:1.8.1.2)
      Gecko/20070220 Firefox/2.0.0.2
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;
      q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

In HTTP protocol, the ends of lines are identified by CRLF, the carriage return ascii character followed by the new line character. The end of the header is identified by a blank line, one which only contains the two characters CRLF.

When the HTTP server receives the end of the header, it formulates a response involving both a response header, and response content. The most basic response header includes the protocol version being used, and the response code. The response code is a sequence of 3 digits which tell the client how the server was able to handle the request. For example, response code 200 means that the server was able to fulfill the GET request, and 404 means that the server could not find the file requested. Additional lines in the response can identify specific characteristics of the response, such as compression, character set, and content length.

The content length is not always required on an HTTP/1.0 connection since the end of the content can be identified by the closing of the connection. Content length is specifically required on HTTP/1.1 persistent connections. Persistent connections allow multiple requests to be fulfilled along a single connection, therefore content length is needed in order to judge where one file ends and the next header starts.

The response header provided for the above request header is the following;

```
HTTP/1.x 200 OK
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: GWS/2.1
Content-Length: 1602
Date: Wed, 02 May 2007 18:23:48 GMT
```

This response indicates to the client that the response is using HTTP/1.x protocol [3, 4], where x means that it is compatible with all versions of HTTP/1 protocol, including 1.0 and 1.1. It also indicates to any caching mechanism that may intercept the transmission, that the response is specific to a particular client and therefore must not be cached. The third line indicates that the response is an HTML file encoded using the UTF-8 character set. It then identifies that the response is compressed using gzip, and therefore must be decompressed before being displayed. The fifth line indicates that the HTTP server is GWS/2.1, which is Google's Web Server, version 2.1. This is a custom web server designed to fulfill their needs more precisely than other available web servers. The length of the response that would be sent following this header is 1602 Bytes. The Date header indicates when the response was generated.

1.1.3. APACHE HTTP SERVER

The Apache HTTP Server Project is a collaborative software development effort aimed at creating robust, freely-available source code for an HTTP server with quality and reliability usually expected from high priced commercial software packages [5]. The project is jointly managed by a group of volunteers located around the world using the Internet and the web to communicate and contribute ideas, and develop the source code and documentation for this project. Furthermore, the Apache HTTP Server Project is an effort to develop and maintain an open-source, free, secure, efficient and extensible HTTP server for modern operating systems including UNIX and Windows. Apache has been the most popular web server on the Internet since April 1996. In January 2007, Netcraft Web Server Survey found that more than 60% of the web sites on the Internet were using Apache HTTP Server, thus making it more widely used than all other web servers combined [6].

Apache HTTP Server is an open source web server, licensed under the Apache License.

The Apache License allows the source code to be distributed in both open and closed source software packages as long as a copy of the license is kept with the software. For this reason, the Apache License is not compatible with the GNU General Public License, GPL, which is the software license used most commonly for protecting the rights of authors of free software. The draft of GNU GPL version 3 contains a provision making it compatible with the Apache License, allowing Apache HTTP Server to be distributed under a standard license.

The version of Apache used for this project is the latest stable one, version 2.2.0, released on November 30th 2005. We chose this version because of its numerous improvements in terms of caching, and overall performance increases over the earlier versions. Hundreds of bugs and problems reported by users and developers since the previous major release (version 2.1.1 released in November of 2004) have been addressed and fixed. In addition to the fixes, numerous enhancements to the core and module source code, as well as the program itself have been made.

Version 2.2 is the first stable release of Apache source code since the release of version 2.0 in year 2003 (unstable releases end with odd numbers, *i.e.*, 2.1, 2.3, etc. while stable ones end with even numbers, *i.e.*, 2.0, 2.2, etc.). A stable release is one with fewer known bugs and complete feature implementation, suitable for deployment. An unstable release is one that has not been completed, it may contain bugs or incomplete features making it unsuitable for deployment. One of the new features in this version is stable caching. Most of the experimental caching modules in version 2.0 made disk caching possible but not very effective. In addition to changes done to main caching modules (*i.e.*, mod_cache [7], mod_disk_cache [8], and mod_mem_cache [9]), another program, htcacheclean [10], has been introduced to keep the cache size within a certain limit (a much needed feature that mod_disk_cache was lacking) and thus avoid all the pitfalls of unrestricted cache size and possible program crashes. Although htcacheclean doesn't come into play in this experiment, it makes caching within Apache feasible within a high priority environment.

1.1.4. PREFETCHING

Prefetching is a technique to improve the access time to data that is stored on a device that is slow to access. The technique of prefetching is to try to anticipate what data will be required, and to load it from the slow device in advance. When data is prefetched, then when it is required, it can be accessed immediately from a fast device instead of waiting for it to be loaded from the slow device. The slow device can be of any form, such as from a disk, or from a remote location on a network.

1.2. PREVIOUS WORK

1.2.1. INTERNAL PREFETCHING

Previous work in internal prefetching: Prefetching is used to retrieve data from a slow source to faster memory that is closer to the processor requiring the data. Internally, prefetching can be performed between disk and main memory, or from main memory to cache memory. Data access is usually not continuous. A running program may require some data now, and some data later. While a program is processing data that it has already loaded, the data source may be inactive. When that program again requires data, it will access the storage system to retrieve it.

In prefetching, during idle time between accesses, data that will be required is loaded into memory that is faster than where it is normally stored. By doing this, when the data finally is required, it can be accessed from faster memory instead of slower memory or disk. This can significantly improve the access time for that data.

There are several methods of determining what files should be prefetched. The most obvious is to prefetch files with open file handles that have been partially read by a running program. There is a high probability that files that have been partially read will be read more. The second mechanism for prefetching is by observing historical logs of which files tend to be read after which. If reading a particular file usually results in reading some other files, it may make sense to prefetch them. The third mechanism for prefetching is determined by the software being run. If a program knows in advance that it will require specific files in a particular order, then during periods of inactivity, such as while waiting for user input, it can initiate a prefetch of those files.

The prefetching mechanisms reported in [11, 12] involve logging historical data access patterns in order to model future accesses. The method used in [11] is intended for web servers hosting search engines. This approach prefetches the results of past queries from disk into memory in order to reduce server latency. The novel approach in [11] is the use of a two-level cache, including a static cache and a dynamic cache, referred to as static-dynamic cache. The static cache is read-only and caches the results of the most frequent queries. The dynamic cache uses replacement policy, such as LRU, and caches queries by the query time. They observed that clients making particular queries were more likely to make other related queries.

The mechanism reported in [12] applies to the micro architecture level. The advancement of semiconductor technology has resulted in a significant reduction in clock periods, but the trend in memory development has been towards increasing storage density. To reduce memory access time, relatively small amounts of fast cache memory are used between the microprocessor and the main memory. The problem with current cache memory application is that it uses a very simple prefetching mechanism that fetches the address line immediately following the line currently being accessed. Researchers in [12] propose a new cache level between existing cache memories and main memory, referred to as a Global History Buffer (GHB). The GHB stores the historic cache miss pattern and is used to improve the prefetching technique over what is used in conventional table based memory caches. Instead of fetching the immediate memory line on each cache miss, the GHB uses a more advanced algorithm based on cache miss addresses and can adaptively increase prefetching degree by testing successfulness of prefetching.

In the caching schemes reported in [13, 14], hardware is used for reducing the latency of main memory. Both use a Markov model to predict future data accesses based on the current system state. In [13], a technique is used that they refer to as Predictor-Directed Stream Buffers. In [14], their technique is called Markov Predictors. A Markov model approximates the probability of accessing a reference string by using transition frequencies. Historic access frequencies are used to calculate the probabilities of particular accesses based on the current state. In [13], only the most likely transition, determined by the Markov model, is prefetched. In [14], multiple prefetching paths are followed simultaneously and are prioritized in terms of which path is more likely to be followed. Prefetching will begin by following the path with the greatest probability of being followed.

[15] applies a mechanism for prefetching data based on spacial locality. Spacial locality is a measurement of the distance between data. When the group of data is mapped nearby in memory or on disk in terms of address, it is said to have a high spacial locality. They employ a spacial buffer and monitor memory hits within the spacial buffer. When a hit is detected within the spacial buffer, neighboring blocks are prefetched into the cache.

A unique prefetching control mechanism is presented in [16]. In this, potential prefetches are assessed in terms of their usefulness. In the event that the prefetching mechanism attempts to retrieve data that is historically unlikely to be of value, the prefetch is blocked. This approach employs a feedback mechanism that records whether or not particular prefetches result in improved memory hit ratio. The results of this feedback mechanism are then used in assessing prefetching usefulness.

Comparison of internal prefetching with our work: In Chapter 2 disk to memory prefetching is implemented in Apache HTTP Server. Based on the structure of web pages, it can be reasonably known that files referenced within an HTML file will be requested shortly after that HTML file. We implement disk to memory prefetching of embedded objects within the Apache HTTP Server to reduce the access time to embedded objects upon requests for them.

This work is similar to work done in [11] in that they are both intended for decreasing the latency of an HTTP server. [11] uses spacial locality to prefetch web searches into memory before the requests are made.

This approach is similar to previous work reported in [15], in that it employs spacial locality for prefetching data, however, the scale of our prefetching is much larger. [15] presents a hardware-prefetching mechanism that acts between CPU cache memory and main memory, our approach prefetches entire files from disk into main memory.

1.2.2. PREFETCHING FROM A NETWORK

Previous work in network prefetching: Since a network is much slower than a disk, prefetching from a network resource may provide a much greater performance improvement than prefetching from a disk. In this project, we focus on the web, and more specifically, on HTTP protocol, so prefetching from HTTP servers is of great interest.

The difficulty in prefetching from HTTP is that unlike prefetching data from disk, data on HTTP servers is not random access. Though with HTTP/1.1, it is possible for a client to request parts of files by byte range, it still reads over entire ranges sequentially. Byte range requests are typically used for completion of broken downloads and are not used to implement random access. As a result, prefetching must be restricted to entire files. This makes it difficult to anticipate what data will be required. In anticipating data requirements in a random access file, it is simple to determine that if a file is open for reading, has been partially read, and is not currently being read, that the additional data from that file should be prefetched.

A web browser chooses what data to retrieve by processing data that has already been received. This makes prefetching difficult, since there is no way to know with certainty what data will be requested, until it is already time to make the request.

In [17], prefetching was added to a proxy server, Squid, to improve the delivery performance of web files to the client. A proxy server is a server situated between a client and a server. Requests generated by a client are first received by the proxy, which relays the requests to the server. A proxy also caches data retrieved from HTTP servers. By caching data, the client request may be fulfilled by the proxy directly, rather than having the request relayed to a server. For a proxy to be effective, communications between the client and the proxy must be faster than communications between the client and the HTTP server. Since the proxy server has access to the HTML file before it is delivered to the client, it may begin retrieving other files referenced by the HTML file before the requests arrive from the client, allowing the file to be served to the client immediately.

One common web prefetching mechanism is the use of web browser idle time to prefetch pages linked by the page being viewed. [18, 19] discuss this technique. This approach works mainly on non-shared networks, such as dial-up. On shared networks, such as cable or office networks, this approach may increase the network traffic, resulting in reduced performance for other users. In [18], the information associated with the current web page's hyperlinks is used to predict which link in the page is most likely to be selected, and therefore which should be prefetched. This information is referred to as semantic links, and is defined statically within the web page. During the design of a web page, semantic link information is added by using XML tags. The browser can use the semantic link information to increase prefetching performance.

In addition to using the idle time to prefetch linked documents, [19] also suggests prefetching the most important page for accesses to web page clusters using a page-rank based scheme. Page ranking has already been implemented by Google and shows how valuable a page is by the number of links in and out of that page. Web page clustering is used for assessing the proximity between pages, for example, pages of a company or pages within a book are clusters. The result of prefetching by page rank produces improved cache hit ratios when compared to random prefetching.

[20] reports on a method of prefetching that records group or individual historical access patterns to determine page popularity. It also uses a modified Markov model based on page

popularity for predicting future web file requests. They have performed a comparison between their method and other prefetching systems based on Markov models, and found that theirs resulted in an increased cache hit ratio. An increased hit ratio means that in their simulation, they are more accurately predicting future requests.

Another approach, related to prefetching, is presented in [21]. In this method, the researchers extended the concept of prefetching from traditional document prefetching, to prefetching of the entities required to establish a connection with a server storing a document. They measured the preparation time for a request by analyzing data that was collected from proxy caches. This approach performs DNS lookups, establishes connections, and "warms up" the connections by sending an HTTP "HEAD" request before the connection is required. In this manner, the connection is ready for a quick response when the client is ready to make a request. Their trace based simulations demonstrated a reduction in client wait time when receiving data that was requested.

Comparison of network prefetching with our work: In Chapter 3, we implement server to client prefetching using Apache HTTP Server and a custom lightweight proxy server. Unlike standard prefetching mechanisms, the prefetching is not initiated by the client, but by the server. Anticipating what data the client will require, the server will include that data in the response to the client. This is to minimize the number of requests that the client must make to the server, resulting in a reduction in the effects of network latency.

1.2.3. HTTP SERVERS FOR CUSTOM SERVICES

Custom http servers: There are many different general purpose web servers available.

Of all the web servers that exist, Apache HTTP Server is the most popular, being used by 60% of all Internet web servers [6]. As detailed in section 1.1.3, Apache HTTP Server is a flexible and extensible web server. It can be adjusted to fulfill the needs of most web sites.

Despite the availability of general purpose web servers, there is a need for custom HTTP servers. One such custom HTTP server, mentioned in section 1.1.2, is GWS, which is responsible for coordinating queries across the more than 450,000 servers used by Google [22]. Google's infrastructure is based on the use of a great number of low cost servers in parallel environments. Queries that are generated by particular clients may run across multiple machines distributed within a single cluster of servers, or across multiple server clusters distributed globally. The results of these queries are then joined by the initiating server and sent back to the client. The scale of the job alone makes use of a general purpose web server, like Apache HTTP Server, impractical.

Other custom web servers are built in as components of other software. The Common Unix Printing System (CUPS) [23] uses a custom HTTP web server running on the non-standard TCP port 631. Most HTTP servers run on port 80. The CUPS HTTP server has two purposes; print server management and configuration, and network printing. The management and configuration interface is accessed using a normal web browser, and can be used to add and remove printers, and manage printers and print jobs. For network printing, a print client will generate a postscript file for a print job, and send it to the CUPS server using the HTTP "POST" command. Specifications for the print job, such as printer and print settings, are transmitted in the HTTP header.

Another example for custom HTTP servers are those used to configure network devices, such as IP Cameras and broadband routers. In these systems, the HTTP server must be very small in order to fit in the devices memory.

The advantage of using web based configuration interfaces for software or devices is in their compatibility with multiple platforms. Any computer with a web browser can work with these interfaces.

Comparison with our custom http server: In Chapter 4, we develop a custom HTTP server. The purpose of our custom HTTP server is to efficiently implement the server to client prefetching that we originally added to Apache HTTP Server. Unlike GWS, our server focuses on low-traffic web sites and implementing server to client prefetching in a manner efficient for low traffic requirements and minimal system resources. Most HTTP servers focus on the possibility of handling extremely high volumes and pay little attention to the performance of smaller web sites.

1.2.4. SERVING MULTIMEDIA ON THE WEB

Previous work: Multimedia data is much larger than text. As available network bandwidth increases, so does the ability to transfer very large files, such as multimedia files. The number of multimedia objects available on the Internet has significantly increased since its inception. Today, there are many websites devoted to the distribution of multimedia files.

With the increasing quantity of multimedia content on the Internet, efficiently distributing it has become increasingly important. Most work done to improve the distribution of multimedia data on the Internet has been to increase the transfer speed or to reduce the server bandwidth requirements. One approach is the use of multicasting to distribute data to several clients simultaneously. The UDP protocol transmits data faster than TCP by not guaranteeing complete data delivery [24].

Proxy servers can cache popular multimedia objects to reduce the number of clients retrieving multimedia files directly from their originating servers, and therefore reduce bandwidth consumption. The distribution of multimedia through caching proxy servers is problematic. Though this approach has the potential to reduce the traffic on the originating servers, caching of multimedia objects consumes a great deal of disk space. Partial caching is proposed in [25, 26, and 27] to reduce bandwidth consumption and to reduce the disk space requirements of proxy servers.

In [25], a network aware caching scheme is proposed, using client side proxy caches. In addition to being aware of network bandwidth availability, this scheme is also aware of media popularity and bit-rate requirements. This approach takes network condition, popularity of multimedia object, and cache size into consideration to determine the size of cache segments for multimedia files. This scheme also can deliver media to the client jointly from both the originating server and from the proxy. With awareness of the media and the network, partial caching is implemented to maximize media delivery performance without over burdening the cache server.

In [26], a partial caching mechanism is suggested for implementation in the Squid caching proxy server, for interactive video. In this implementations, video data is segmented into non-overlapping segments and each is cached as individual files. When cached video is requested, the proxy delivers the initial segment to the client immediately to mask startup delay.

The authors recognize several challenges with this approach, including what they refer to as "dirty-initial-segments", in which the data delivered by the originating server doesn't match the data stored in the cache. They propose an algorithm called "validating upon partial replacement" to synchronize the cache and minimize disturbance to the user.

The approach to partial caching of multimedia objects presented in [27] is called fragmental proxy caching and focuses on modifying the cache replacement scheme to maximize quality of service for interactive video functions. In streaming multimedia, users often use interactive functions, such as fast forward, rewind, and pause, in addition to regular playback. This approach uses what they refer to as a sliding history window to track dynamic user requests, and a tunable-victimization mechanism to control which cached objects are selected for replacement based on the requirements of the multimedia application. Their simulation showed a greater performance in terms of byte-hit ratio and number of delayed video starts. Their proposed caching scheme and replacement policy produced finer granularity caching with better performance.

Another problem with proxy caching of multimedia objects, is that the proxy itself has limited bandwidth to serve multimedia requests. To compensate for this problem, as well as for the limitations of client bandwidth, [28] suggests that transcoding mechanisms be applied to proxies to reduce the volume of the data being requested based on the device specifications of the client. Transcoding means the re-compression of media objects and allows the size of a media object to be reduced at the expense of quality. They proposed a new model for caching multimedia objects in transcoding proxies. In this model, caching decisions are made on all of the caches between the client and the server, to find optimal locations.

Comparison with our work: Our work, presented in Chapter 5, is dissimilar to all previous work targeting multimedia distribution on the Internet. In our approach, we improve the delivery of the initial portion of multimedia data embedded in webpages in order to start playback on the client earlier. Again, prefetching is used by applying the same mechanism that we present in Chapters 3 and 4.

1.3. CONTRIBUTIONS OF THIS THESIS

This thesis details modifications made to the functioning of HTTP servers and HTTP clients. Disk to memory prefetching of embedded objects improves the cache hit ratio for Apache HTTP Server by 27%. Server to client preloading reduces web page load times to as low as 24.5% of normal. A custom HTTP server is introduced, implementing server to client prefetching while using less system resources than Apache HTTP Server.

1.4. THESIS OUTLINE

The remainder of this document is organized as follows; Chapter 2 presents a technique whereby the Apache HTTP Server will use prefetching internally to reduce its response time by reducing the effects of disk latency. In Chapter 3, the prefetching technique is applied to client-server network communications in the form of a modified Apache HTTP Server to reduce the effects of network latency. In Chapter 4, we present a custom built HTTP server to implement network prefetching more efficiently than Apache. We discuss the impact that network prefetching has on the delivery of embedded multimedia objects in Chapter 5.

CHAPTER 2

DISK TO MEMORY PREFETCHING APACHE HTTP SERVER

2.1. INTRODUCTION

In this work, we have developed an enhancement to the Apache HTTP Server to predict future file accesses, based on an analysis of the response data. Traditionally, Apache HTTP Server serves web files directly from disk, but it includes a caching module which can be activated as needed. The Apache caching module can be configured to cache to disk or memory. All experiments we have done involving caching use strictly memory caching and not using disk caching. With the caching module activated and configured to use memory, subsequent requests for the same file may be served from memory. We have developed a modification to the Apache HTTP Server, which allows objects embedded in HTML documents to be fetched into memory before they are requested. Serving files from memory is much faster than serving them from disk, reducing the time that it takes for the server to generate its response, and allowing the response to be sent sooner. The faster the HTTP server is able to generate a response for the client, the lower the client latency. We use the term client latency to mean the total amount of time between the moment when a client issues a request for a web page, and when that web page has been fully received, including all embedded objects, such as images and sounds.

We use prefetching from disk to cache memory to reduce the web server internal latency. The internal latency is the amount of time that the HTTP server software must wait idle for the hardware to deliver required data. This modification does not improve the rate at which data can be transmitted over the network, but allows the server to begin transmitting data earlier than it would be able to if the file were located on disk. One metric we use for measuring the performance of this modification is hit ratio. Hit ratio is the proportion of all files served that are served from memory. Improvements to the hit ratio are improvements to the speed and responsiveness of the web server. The results of testing hit ratio improvement were published in [29].

Our approach to prefetching does not use a probabilistic model, but rather prefetches the embedded objects of a web page based on references to them in HTML files that have already been requested. Our strategy uses the structure of HTML files to detect embedded objects stored on the local server. By parsing the HTML file, we can find the additional objects that the client will require to properly display the web page. After parsing the HTML file for embedded objects, the objects are then loaded into the cache used by Apache HTTP Server. In this way, the existing caching module can be used to deliver prefetched data without the need to define an additional cache specifically for prefetched data. This minimizies the degree of modification required, and maximizes the potential performance by only needing to scan through a single cache. This also eliminates the need to synchronize two caches, and cuts down on the amount of memory required.

We implement the technique of prefetching embedded objects into memory on Apache HTTP Server and show the performance improvement through trace driven simulations based on external access logs. The external access logs are used to demonstrate the overall improvement that would be experienced if all web servers implemented this technique.

2.2. DATA COLLECTION AND METHODOLOGY

Our experiment is based on log files collected by IRCache, which was begun as a project of the National Laboratory for Applied Network Research (NLANR), University of California, San Diego (USCD), and the National Science Foundation. The files used were all dated January 12, 2007. These logs were filtered to include only lines with response codes of "200 OK". The logs were also filtered to include only the URL, since the additional data in the log is of no importance to this experiment. Additionally, the logs were filtered to include only static files. Script generated files, such as PHP and CGI, were cut from the logs as their contents, being unpredictable, can interfere with the test results. We detect unsuitable files in the logs by their file extensions.

Upon complete filtering of the logs, each URL was downloaded to disk on our test server to create a mirror of the data in these logs that we can use for testing. The files were downloaded using GNU Wget. Because data on the Internet changes very frequently, in order to minimize the differences between the data referenced in the logs and what was actually available, we downloaded the data on January 13, 2007, as soon as the logs were made available.

The filtered logs used are as follows;

1) bo2 ("Boulder, CO"). This log contained a total of 65404 URLs, including 4311 HTML files and 41734 images of the types we scan for.

2) ny ("New York, NY"). This log contained a total of 117296 URLs, including 4124 HTML files and 89996 images.

3) pa ("Palo Alto, CA"). This log contained a total of 104457 URLs, including 6869 HTML files and 69856 images.

The files in the logs that are neither HTML nor images, are other types of files, such as archives and documents. These files are neither scanned for prefetching, nor are they prefetched.

Upon completion of each segment of the experiment, the Apache HTTP Server error log was collected for analysis. A minor modification was made to the Apache caching module in order for it to more clearly report cache hits and cache misses in the error log. The error log is a file generated by Apache HTTP Server where it records debugging information. This modification does not in any way impact the function of Apache. Because of the way that we are using the Apache caching module, all cache hits are memory hits. All cache misses are disk accesses. The disk and operating system caches have no effect on our testing since the first access to a file is always before the disk or operating system has a chance to cache the file. Subsequent access to the file are directed to Apache's cache. Upon a successful test, the Apache error log was filtered to contain only the lines reporting cache hits and cache misses. The lines reporting hits were counted and used to determine the average hit ratio up to each line in the log. In total, 6 sets of logs were processed and collected during simulation, two for each of the three input logs, one being for prefetching and the other for non-prefetching Apache HTTP Server.

Our tests for hit ratio do not directly imply a change in client latency. The hit ratio is only a way of determining if and how many additional files will be served from the cache memory as opposed to being served from slow disk.

An additional experiment we performed, following the hit ratio test, was a test to measure the actual performance difference as would be experienced by a web client. This test measures the change in client latency resulting from prefetching objects from disk to memory in the server. For this experiment, we mirrored 5 popular web pages bearing characteristics compatible with prefetching, and having a wide range of number of embedded objects. The web sites were chosen off lists provided by IRCache and SeekingAlpha, and include the homepages for www.amazon.com, www.aol.com, www.google.ca, www.mapquest.com, and www.wikipedia.org.

2.3. PROPOSED STRATEGY

Our strategy for improving the performance of Apache HTTP Server is to load embedded objects of HTML documents into memory as the HTML document is being served. After an HTML file is requested by the client, it is assumed that the client will also request objects referenced by that file. As the file is served to the client, it is scanned for embedded objects. When particular types of embedded objects are located in the HTML file, they are loaded from disk into the cache memory allocated to the Apache caching module. As the client web browser receives the HTML file, it will scan through it to find embedded objects. If objects are found, it will generate secondary requests for those objects to the servers hosting those files. By the time the secondary requests arrive at their destinations, the server hosting the HTML file will have loaded the embedded objects it stores from disk into cache memory, allowing the caching module to serve the object from memory, rather than from slow disk. The difference in the time between serving a file from disk and serving a file from memory is the amount of time saved by the client between making the request for the file and actually receiving the file.

The proposed solution to improve delivery of a web page would follow these steps:

1. Apache serves an HTML file to the client

2. Apache scans through the HTML to see what images (and other objects) are embedded in this page that the client may request.

3. Based on those scans, Apache loads objects from disk into memory cache (preloading them into memory for faster delivery)

4. Some time later, when the client requests an image or embedded object, Apache serves it from the memory cache, thus saving us milliseconds in disk access time.

A custom-made filtering module was created to scan through HTML files being served by the Apache HTTP Server, and load them into memory for faster delivery as they are requested by the client. Apache HTTP Server is written in C language. Apache modules are also written in C.

Our custom module is referred to as mod_pre_cache. It was written to parse html files for finding embedded objects. It is compiled into Apache at runtime (every time the server is started or restarted), run independently of any other modules, and is automatically executed by Apache when an HTML page is served to a client.

Here are the two major functions contained in the mod_pre_cache module:

1) pre_cache_scan_filter() function of this module scans through the html page being served to the client and finds the embedded objects of predefined types (*e.g.*, jpg, jpeg, gif). It then prompts the caching module to load the embedded objects into memory by generating requests for those files.

2) fix() is a helper function that modifies the requested URL such that it is minimized. For example, it processes "/path/../" into "/", "/./" into "/", and "//" into "/" as needed in order to be able to identify multiple requests for the same file with different URLs.

2.4. EXPERIMENTAL DESIGN

We performed two experiments. The first experiment tested the change in the proportion of files that were served from memory as opposed to disk. The second experiment tested the change in the client latency.

2.4.1. HIT RATIO TEST

A log test was designed to test our modified Apache with a load, which is as close to real traffic as possible in a repeatable environment. Although a log contains real user patterns in accessing the objects, it does not simulate the behavior of a real web browser, such as fetching objects in parallel.

A Log trace from the IRCache project [30] provides a wide sampling of web activity on a day-by-day basis. Our testing is designed to reenact one day's worth of activity in serial, and to measure the hit ratio and byte hit ratio over the course of the test. For each object fetched, any improvement in memory hit ratio should be due to the effect of prefetching.

The IRCache log files were filtered to allow only requests using the GET method, the HTTP protocol, and 200 OK response codes. Also, requests for dynamic pages indicated by a ? query part, and objects that do not exist anymore, were excluded from the trace.

We used an IRCache Proxy log to simulate a large scale deployment of modified Apache by serving the resources accessed by the real client request patterns recorded in the log. The URLs referenced by the IRCache logs were downloaded to our test server.

We used GNU Wget [31] as our client to fetch each object from the filtered IRCache log. Wget was instructed to discard the object after receiving it to avoid writing to disk, as well as to only try each fetch once even in the event of an error such as partial download. The test was executed by running Wget on each URL from the IRCache log in sequence.

The URLs in the logs were first downloaded to our local server from their originating servers using Wget. This was performed immediately after the logs were generated in order to maintain consistency between the logs and the data we were processing. Wget was instructed to create a directory hierarchy representative of the URLs it downloaded, with the host names at the root. Each host's directory tree was stored on our local Apache HTTP Server, where it could be accessed using a minor modification to the original URL. The original URL must be modified to reflect our server being as the sole server for all objects, so an URL such as;

http://www.somedomain.com/path/file.html

becomes;

http://ourserver/www.somedomain.com/path/file.html

The IRCache logs were modified to add our local server's host name to the root of each URL. With this modification, every file referenced by the original IRCache logs would be referenced by the modified log on our local Apache server. Following this modification, Wget was instructed to download each URL again, from our local Apache HTTP Server. With normal use, Wget outputs the file downloaded into the filesystem where it is run, but during our testing, the output files were not needed, and so were discarded. For each log used, this was performed once for modified Apache, and once for unmodified Apache. Before each time this experiment was run, the Apache logs were erased, the cache cleared, and the server restarted. Following each run of the experiment, the error_log was collected for later processing.

With our modification to the Apache caching module, it records access as a cache hit or miss in its error log, a file called error_log. Following the experiment, this log was filtered to contain only reports of cache hits and cache misses, which were then used to calculate the precise overall hit ratios and byte hit ratios for each of the three IRCache logs, up to every line in the log for both regular and for modified Apache.

2.4.2. CLIENT LATENCY TEST

For our second experiment, we measured page load times for several popular websites. In this experiment, we located a server on a residential cable network with maximum upload speed of 800 Kbps, and a client on a separate high speed network. We tested all websites under 4 different conditions; we tested for prefetching enabled and prefetching disabled for CPU speed of 1.0 GHz, and 2.4 GHz. The test server has a dual core processor and is running a symmetric multiprocessing kernel. We tested at two different CPU speeds in order to determine if the addition of prefetching caused any significant processing overhead. From our client machine, we downloaded the target web pages from our modified server 100 times each and measured the total amount of time that elapsed between issuing the request for the HTML file, and having received all files necessary to display the page.

2.5. RESULTS

Our results cover both of the two experiments separately. Section 2.5.1 details the results for testing hit tatio and byte hit ratio. Section 2.5.2 details the results for testing client latency.

2.5.1. HIT RATIO RESULTS

The metrics used to measure the performance of a caching system are hit ratio and byte hit ratio. Hit ratio, R_h, is the proportion of files served from cache, and is obtained by dividing the total number of files served from memory, S_m, by the total number of files served from cache and from disk, S_d. Byte hit ratio, R_{bh}, is the proportion of bytes served from cache, and is obtained by dividing the sum of all bytes retrieved from cache, S_{bm}, by the total number of bytes retrieved from cache and the total number of bytes retrieved from disk, S_{bd}.

$$R_h = S_m / (S_m + S_d)$$

$$R_{bh} = S_{bm} / (S_{bm} + S_{bd})$$

Our proposed mechanism causes an improvement in memory hit ratio, which in turn drives an improvement in object fetch times. The object fetch times are lower because objects are fetched from memory instead of being served from very slow disk. In this work we report the memory hit ratio and byte hit ratio generated by modified Apache and regular Apache.

Our testing shows an improvement to both hit ratio and byte hit ratio, and as such, this mechanism provides an improvement to the caching performance of Apache HTTP Server.

From the error_log file generated by the servers during the test, we can show that our proposed method has improved the memory hit ratio as we predicted. Figure 2.1 and Table 2.1 show the memory hit ratio and byte hit ratio over the filtered log, for modified and regular Apache. Please note that hits generated from memory are faster as opposed to slower disk hits.

With the improved hit ratio, we can reduce the time needed to download the objects in the source log. For each log entry tested, we present the hit ratio throughout the test in Tables 2.1 through 2.3. Our tests yielded overall file hit ratio improvements for the "bo2", "ny", and "pa" traces respectively of 27.83%, 23.52% and 24.34%. We also measured overall byte hit ratio improvements respectively of 3.00%, 2.30%, and 0.67%.

Immediately noticeable is the difference between the improvements in file hit ratio and byte hit ratio. The much lower improvement in the byte hit is due to the sizes of the files being prefetched. The byte hit ratio is not nearly as significant as the hit ratio for the client, since the addition of prefetching to a web server reduces the server internal latency without doing anything to improve the overall data transfer rate between the server and the client. In other words, the size of the file being served from memory makes less of an impact on the client latency since the disk is much faster than the network. Additionally, the byte hit ratio is influenced very strongly by very large files that are not related to the display of web pages. Some files accessed on the Internet are of types such as iso9660, which are copies of filesystems for optical disks such as CDs and DVDs. These files are extremely large, up to 8.4GB for a DVD, and they can not prefetched. The lower byte hit ratio is due to the prefetching being performed on smaller, embedded images, but the logs also reference some very large files that are not prefetched, such as iso9660 filesystems.

The file and byte hit ratios for prefetching approach the values for standard caching Apache HTTP Server with a large number of requests. Prefetching is only effective for the first access to an embedded object. Subsequent accesses to the same files will already be cached regardless of whether prefetching is used or not. For this reason, the difference between the hit ratios for prefetching and standard Apache decreases as the number of requests increases.

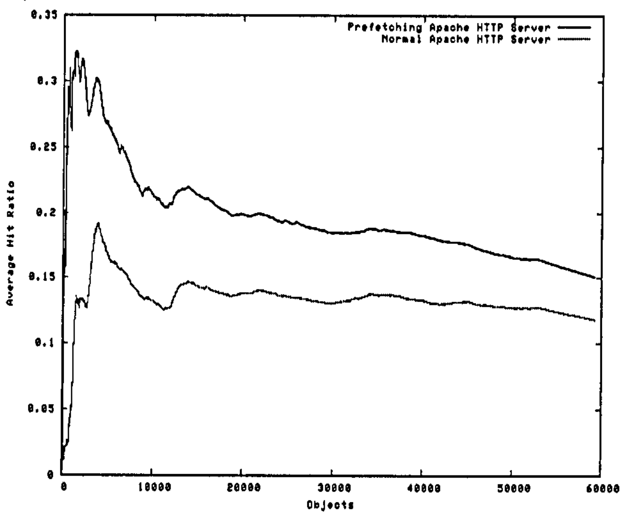

Figure 2.1(a) – bo2 Hit Ratio

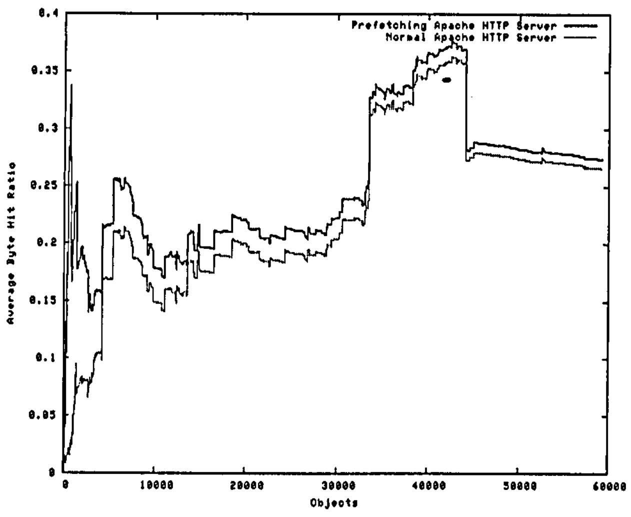

Figure 2.1(b) – bo2 Byte Hit Ratio

Table 2.1. Summary of log test data for bo2. Average hit and byte hit ratio at the end of the test.

	Regular Apache HTTP Server	Prefetching Apache HTTP Server
Hit Ratio, R_h	11.82%	15.11%
Byte Hit Ratio, R_{bh}	26.64%	27.44%

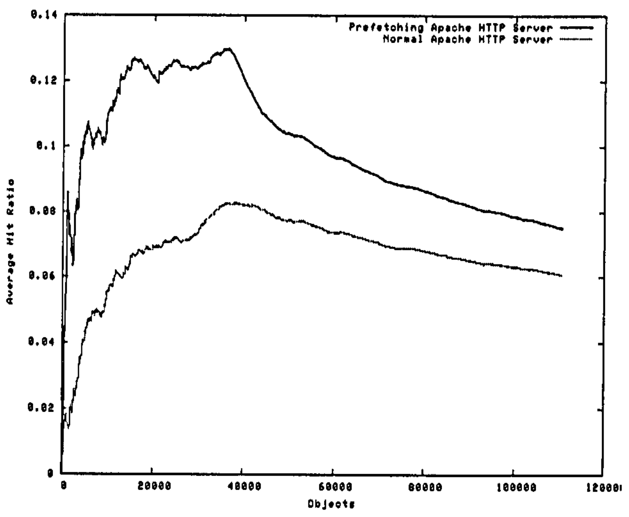

Figure 2.2(a) – ny Hit Ratio

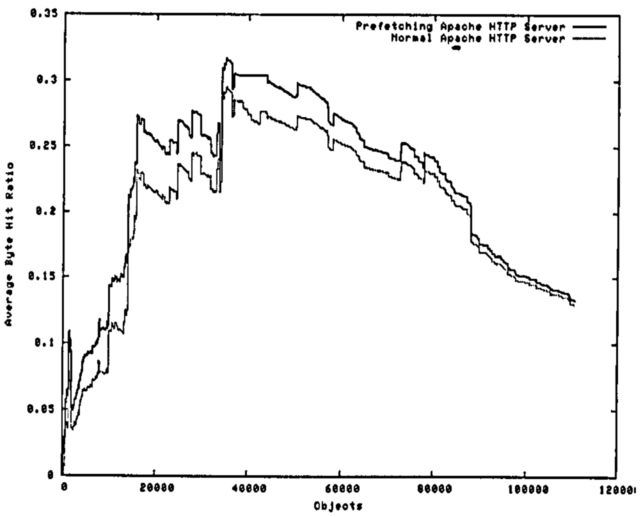

Figure 2.2(b) – ny Byte Hit Ratio

Table 2.2. Summary of log test data for ny. Average hit and byte hit ratio at the end of the test.

	Regular Apache HTTP Server	Prefetching Apache HTTP Server
Hit Ratio, R_h	6.08%	7.51%
Byte Hit Ratio, R_{bh}	13.02%	13.32%

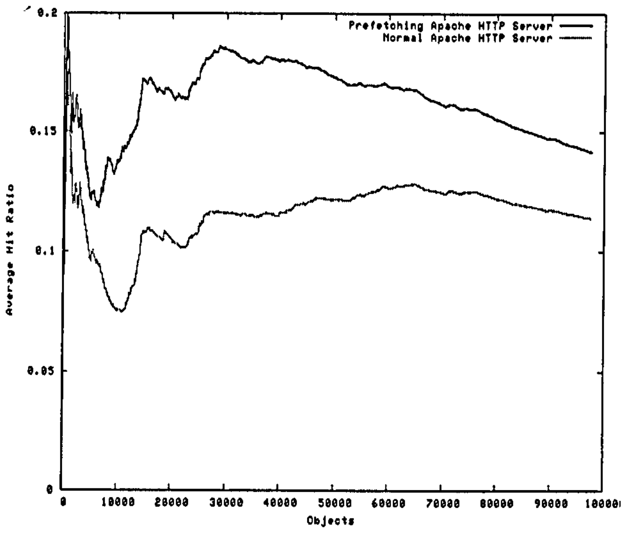

Figure 2.3(a) – pa Hit Ratio

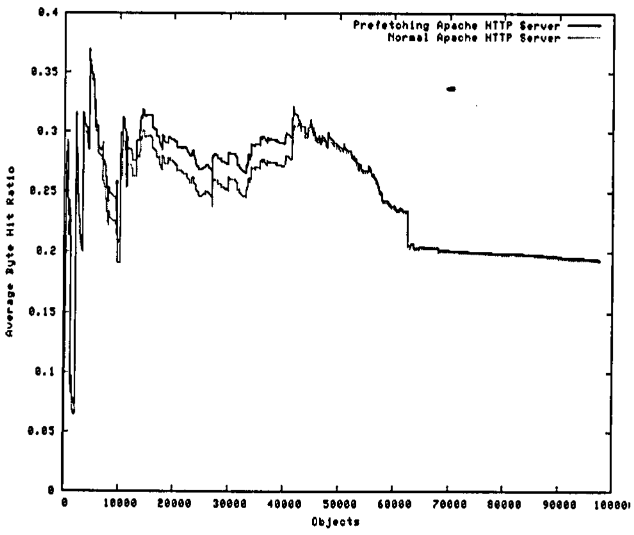

Figure 2.3(b) – pa Byte Hit Ratio

Table 2.3. Summary of log test data for pa. Average hit and byte hit ratio at the end of the test.

	Regular Apache HTTP Server	Prefetching Apache HTTP Server
Hit Ratio, R_h	11.42%	14.20%
Byte Hit Ratio, R_{bh}	19.27%	19.40%

2.5.2. CLIENT LATENCY RESULTS

For our second experiment, we recorded a performance decrease for all but 1 of the sites tested when running the CPU at 1.0 GHz. The site showing improvement at lower CPU speed had fewer, larger embedded objects than other sites tested with similar page load times. With the CPU running at 2.4 GHz, we measured a marginal performance improvement for all sites except www.wikipedia.org, which still performed worse with prefetching enabled than disabled. The results of this experiment are summarized in Figure 2.4(a), Figure 2.4(b), and Table 2.4. One site, www.mapquest.com, showed a significant improvement in performance, of over 2 seconds, for both prefetching enabled and disabled.

Our results in this test are averages from performing page loads 100 times for each web page tested, and are the actual page load times for the sites tested when served with our equipment. The results of testing each of the sites is different due to several factors; the number of embedded objects impacts the number of requests that the client will have to make to the server, the sizes of the embedded objects impacts the effect of caching on that file, and the structure of the HTML file influences how much time it takes to parse the file for embedded objects.

Our testing is performed at two CPU speeds to determine whether or not hardware performance can influence the effects of prefetching. What we found is that on the same server with nothing more than the CPU frequency changed, that the performance of the hardware can control whether prefetching from disk to cache memory will cause a decrease or an increase in client latency. What this means is that prefetching adds enough overhead to the server to impact how fast Apache serves files.

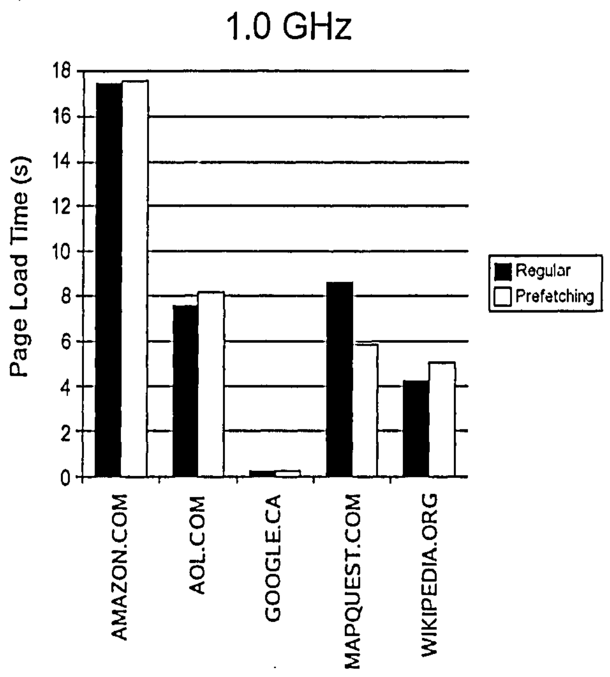

Figure 2.4(a): Prefetching 1.0 GHz

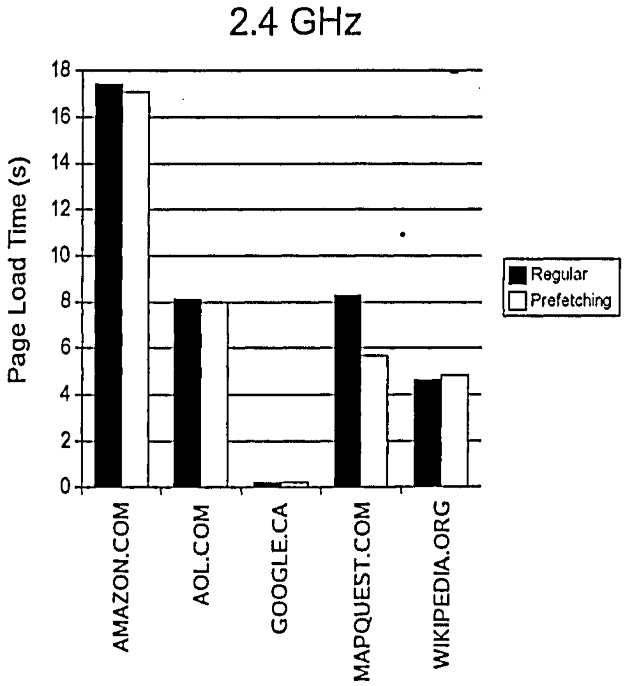

Figure 2.4(b): Prefetching 2.4 GHz

Table 2.4. Average Client Page Load Time

	Prefetching	*Regular*	*Improvement*	*Prefetching*	*Regular*	*Improvement*
	1.0 GHz	***1.0 GHz***	***1.0 GHz***	***2.4 GHz***	***2.4 GHz***	***2.4 GHz***
amazon.com	17.579249	17.478133	-0.6%	17.058943	17.424097	2.1%
aol.com	8.137085	7.535382	-8.0%	7.969212	8.135689	2.0%
google.ca	0.212823	0.209752	-1.5%	0.215615	0.216995	0.6%
mapquest.com	5.785428	8.595447	32.7%	5.680260	8.265422	31.3%
wikipedia.org	5.024183	4.202554	-19.6%	4.829026	4.649538	-3.9%

2.6. CONCLUSION

In this work, we have improved the performance of the caching system of Apache HTTP Server, to be able to provide more memory hits instead of disk hits when serving web pages. Memory hits for the web documents result in elimination of disk latency for these documents when they are requesting by clients.

The log test was performed to measure the improvement in memory cache hit ratio and byte hit ratio when using modified Apache. This test shows improvements of 27.75%, 23.44% and 27.8% for the “bo2”, “ny”, and “pa” traces respectively for hit ratio. Byte hit ratio improvements are 2.99%, 2.33%, and 0.69% for "bo2", "ny", and "pa".

For our page load time performance measurements, it is clear that prefetching files from disk to memory does offer some performance improvements under some circumstances. Some web pages benefit more than others from prefetching, specifically, www.mapquest.com yielded significant performance improvement at both CPU speeds tested and greater performance improvement than all other web pages tested. Other sites showed no improvement, specifically www.wikipedia.org, which performed worse with prefetching at both CPU speeds tested. With the remaining three sites tested, prefetching offered very marginal performance improvements, and this improvement depended on the speed of the CPU in the server.

With disk to memory prefetching at 2.4 GHz, our greatest improvement is a reduction in client latency of 31.3%. Our worst improvement is an increase in latency of 3.9% due to increased server overhead. The remainder of the sites exhibited improvements in the range 0.6% to 2.1%. At 1.0 GHz, only one site yielded a performance improvement.

CHAPTER 3

NETWORK PREFETCHING APACHE HTTP SERVER

3.1. INTRODUCTION

From the results obtained from Apache HTTP Server modified to prefetch objects from disk to memory, it is clear that there are a great number of requests for files that can be anticipated. Loading objects from disk to memory, while improving the cache hit ratio and reducing the effect of disk access latency, does not provide the performance improvement desired since the greatest influences on web page load time are network latency and network speed.

To extend the concept of prefetching data to improve web page load time, we have developed our own custom module for Apache HTTP Server to prefetch data directly to the client before the data is requested.

In HTTP/1.0 protocol, every file being retrieved from the server requires its own connection to be established between the client and the server [3]. This places additional resource strain on the server and on the client than what would be experienced using a single connection. Since embedded objects are referenced within HTML, these files cannot be requested until after that part of the HTML file has already been received. In the worst case scenario, where an embedded object is referenced at the end of an HTML document, this may mean that the final, or only embedded object of a web page cannot even be requested until after the entire HTML document has been received.

In HTTP/1.1, this situation has been marginally improved. This version of HTTP supports persistent connections and request pipelining [32]. What this means is that while receiving the HTML file, the client can submit requests along the same connection, rather than having to establish new connections. This can result in a reduction in server and client workload and reduced file transfer times. A weakness common to HTTP/1.0 and HTTP/1.1 is the need for the client to wait for the HTML file before it can issue requests for embedded objects. HTTP/1.1 can in most cases improve on the page load times of HTTP/1.0, however, the worst case performance of HTTP/1.1 for HTML files that do contain at least one embedded object, is no better than HTTP/1.0

We have developed an additional enhancement to the HTTP protocol. This modification eliminates the need for the client to wait for the delivery of the HTML file before being able to request the embedded objects. In our solution, the server delivers the embedded objects to the client without being explicitly requested. In essence, the requests for the embedded objects are generated by the server instead of by the client.

In addition to solving the problem of network latency, our modification is also able to overcome TCP slow start. In TCP slow start [33], the server sends a small amount of data to the client before waiting for an acknowledgment that the data has been received. As it receives acknowledgments, the server sends larger amounts of data until it detects that data is lost. Because TCP communications use slow start for congestion avoidance, the transfer of small files may not provide adequate opportunity for full bandwidth utilization. When grouping all the files of a website in a single connection, TCP is able to build up to a greater speed before the connection is terminated.

3.2. DATA COLLECTION AND METHODOLOGY

This experiment is based on web pages presented by several popular websites. There are several lists of the most popular websites available, including the ones published by IRCache [34] and SeekingAlpha [35]. Our choices in web pages were determined by the varying number of embedded objects. Since our testing is to determine the performance improvement for loading full web pages, including all embedded objects, it is necessary to consider a variety of sites with different characteristics. Additionally, our choice was influenced by the distribution of the embedded objects and how they are referenced within the HTML. Since our prefetching is based on HTML tags, it is necessary for all embedded objects for the site to be referenced by the HTML. Having images referenced by HTML tags is important since websites that reference images by other means, such as in scripts, would not have all of their embedded objects identified, and as a result, would provide incorrect results when testing.

The web pages selected for testing were downloaded to disk using GNU Wget. Wget is able to follow the structure of HTML files, download all embedded objects to a local filesystem heirarchy, and modify the HTML file to reference the objects that it stored in the local filesystem. These websites were mirrored to our test server on March 16, 2007.

Our choice of websites include;

1) amazon.com. This site had 60 embedded objects.

2) aol.com. This site had 34 embedded objects.

3) mapquest.com. This site had 28 embedded objects.

4) wikipedia.org. This site had 15 embedded objects.

5) google.ca. This site had 1 embedded object.

Each website was tested 100 times for prefetching enabled and for standard HTTP. The page load times were recorded for each test. The tests for modified HTTP protocol were performed immediately following the tests for standard HTTP protocol in order to minimize the variations due to network conditions.

3.3. PROPOSED STRATEGY

Our strategy for improving the load time for pages served by using HTTP protocol is to forward embedded objects of HTML documents directly to the client along with the HTML file. The forwarding of the objects is done without the client explicitly requesting the file, therefore it reduces the time to receive the embedded files by the time that it takes for a message to be relayed to the server and back.

In using standard HTTP protocol, every file must explicitly be requested by the client in order for the server to send it. Before the client can request any embedded object, it must receive and parse the HTML file. It then creates new requests to the server for those objects, which add to the network traffic, the server's queue, and force the server to spawn multiple new processes to handle those requests. This adds a significant load to the network and to the server.

With our modified HTTP protocol, implemented in our custom module for Apache

HTTP Server, the client issues a request only for the HTML file, and the server appends the embedded objects in the same response as with the HTML file. It is not necessary for the server to spawn multiple processes to service one web page, there are fewer requests in the queue, the network traffic is decreased, and due to the size of the transmission, the network bandwidth is more fully utilized.

Our custom module for Apache HTTP Server implements a content generator to replace the standard content generator included with Apache. The standard Apache content generator is capable of serving standard HTTP/1.0 and HTTP/1.1 requests, its response can be modified only so far as the contents of the header.

In a standard retrieval of a web page using a single connection from the client to the server, the time, t, to begin receiving a document from a web server, can be expressed as the following for HTTP/1.0;

$t = h + r + p + s$, where;

h is the time to negotiate a connection,

r is the time to send a request from client to server,

p is the time for the server to process the request,

s is the time for the first byte of the response to be transferred from the server to the client.

Further, the time R_i, to receive the i^{th} document of a web page, can be expressed as;

$R_i = t + size_i \div rate$, where *rate* is the transfer rate of the connection.

The time to receive all of the documents related to a web page is, therefore;

$T = \Sigma R_i$, for i from 1 to n files embedded in the web page.

This can also be expressed as;

$T = R_1 + t \times (n-1) + (\Sigma size_i) \div rate$, for i from 2 to n.

In prefetching, T is reduced by t for each embedded object;

$t \times (n-1)$,

So in prefetching, T becomes;
$T = R_1 + \Sigma size_i \div rate$, for i from 2 to n

For HTTP/1.1, T is reduced only by an additional h for every additional file requested along an established connection. h is a small component of t.

Our solution to improve the delivery time of a web page follows these steps;

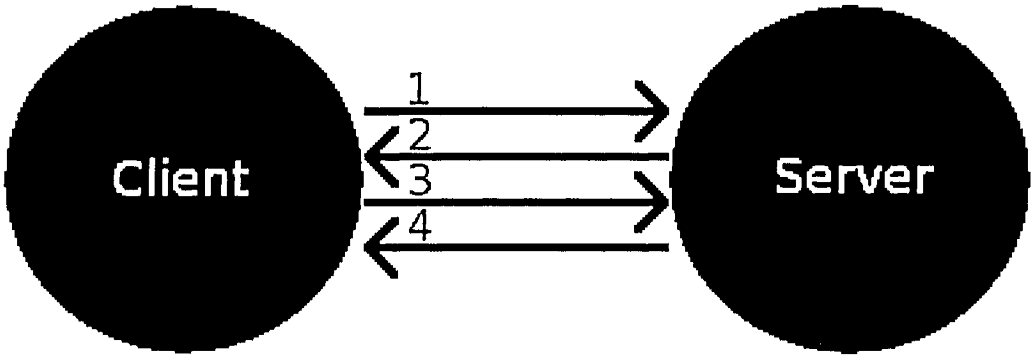

Figure 3.0: Normal Web Page Load Sequence

1. The client requests an HTML file from the server.
2. The HTTP Server scans through the HTML file and creates a list of all embedded objects that

can be forwarded to the client.

3. When sending the response to the client, the embedded objects are attached to the end of the response

4. The client receives the HTML file and all embedded objects in a single transmission.

A normal HTTP client, particularly a web browser, does not have the ability to receive multiple files in a single transmission. In order to accomplish this, the client must be modified. There are two possible approaches to client modification;

- Implementation of the modification within the web browser.
- Addition of a special light-weight proxy on the client machine to translate between the modified HTTP protocol used by the server and the standard HTTP protocol used by the browser.

Modification to the browser is impractical, since it would require modification of all browsers in order for this system to become effective. We focused on designing a special proxy translator since it will be compatible with all web browsers.

The proxy translator receives a proxy request from the client browser. It then modifies the request to an HTTP request and appends a header indicating its compatibility with the modified HTTP protocol.

Prefetching: true

When the server receives this request and reads the added header, it formulates a response containing all embedded objects and sends that to the proxy. If the server receives a request that does not have the added header, it will generate a normal HTTP response. If the added header is received by a server that does not understand it, the header will be ignored and a normal response will be sent.

In the modified HTTP response, the path of each embedded object is added to the response header for the HTML file. This is to indicate to the proxy immediately, which files will be sent. This is necessary because the HTML file is sent first in the response, and is forwarded immediately from the proxy to the client browser without waiting for the full transmission. This makes it possible for the client to generate a request for an embedded object before it has arrived. It is important for the client browser to be able to generate requests before receiving the full transmission in order for it to retrieve objects stored on other servers. Prefetching can only forward embedded objects located on the same server that is performing the prefetching. Unlike HTTP/1.0 and most applications of HTTP/1.1 protocol, this modified protocol is strictly dependent on the Content-Length header, this is because the client uses the content length to judge where in the transmission the headers for the next file begins. Files in the transmission are organized as following;

Header for HTML file with list of embedded object paths in order of transmission
{blank line}
HTML file
{blank line}
Header for first embedded object
{blank line}
First embedded object
{blank line}
Header for second embedded object
etc.

3.4. EXPERIMENTAL DESIGN

Our tests are designed to simulate a client's access to several web pages and to measure the change in the page load time in an environment similar to what real web browsers would use to connect to HTTP servers. Lists of popular web sites provided by the IRCache project and SeekingAlpha were used to generate mirrors of several popular websites. The websites chosen have a wide variation in their numbers of embedded objects. We need this varying number of embedded objects for testing the performance change resulting from employing our modified HTTP protocol and the relationship between the performance improvements and the number of embedded objects.

We did not use a real web browser to perform the tests, but instead used GNU Wget in much the same manner as it was used to originally mirror the web pages used for the tests. Wget is incapable of downloading multiple embedded objects concurrently as a web browser is. However, we will show that the performance benefit of using concurrent connections is negligible compared to the performance benefit of server to client prefetching. Additionally, due to the nature of server to client prefetching, establishing concurrent connections will not provide any performance improvement over using a single connection.

Our tests were performed under two environments. The two environments differ in the overall data transmission rate as well as the overall network latency. Our first environment situated the Apache HTTP Server with our custom prefetching module on a residential cable network capable of receiving at 6 Mbps and sending at 800 Kbps. The client was situated on a much faster network. The purpose of this first environment is to simulate a typical busy network as would be encountered in a real application of this modified HTTP protocol. The second environment tested consists of a client connected to a server via 100 Mbps LAN. We use the second test to determine if or how the performance of prefetching changes with the speed of the network. We expect that as the network latency decreases, so will the effects of prefetching.

In our tests, we ran the server containing mirrors of the web pages at http://www.amazon.com, http://www.aol.com, http://www.google.ca, http://www.mapquest.com, and http://www.wikipedia.com. On the client system, we ran the lightweight proxy server capable of interacting with a web server employing modified HTTP protocol. The proxy server used is modified from what would be used by an end user in that it responds to a signal to immediately clear its cache. When the server and the proxy processes are running, the client software, in our case GNU Wget, is run to download the given web page.

Our testing procedure consisted of running GNU Wget in a loop 100 times to download each website being tested. The time it takes for Wget to retrieve the entire web page is recorded in a log for every iteration of the loop. Between every run of Wget, the proxy cache is cleared. Immediately after running a test using the modified HTTP protocol, the same test is performed again, but without using the proxy. When the proxy is not used, the header indicating the client's capability of receiving responses using the modified HTTP protocol is not included, resulting in the server reverting to standard HTTP protocol. In this manner, we are able to test using precisely the same server and client in order to eliminate any chance of contaminating the results by using alternate software that may perform differently than ours.

Following testing, the logs were processed to determine the average page load times for each web page tested. Each web page tested using modified HTTP protocol was compared to the results from the same web page using standard HTTP protocol. The difference in performance for each web page was compared with the difference in performance for the other web pages. The comparisons were made to determine the relationship between the performance of modified HTTP protocol and the number of embedded objects.

3.5. RESULTS

Our modification to HTTP protocol allows the server to anticipate future requests from the client, in turn allowing the server to forward the responses to future requests to the client without explicit requests.

From the log files generated by our testing, we can show an improvement in web page delivery time as anticipated.

As previously explained, our testing client, GNU Wget, is not capable of concurrent connections. Due to the nature of our experiment, it was necessary to use a client that can be run repeatedly from a script. A normal web browser is capable of creating concurrent connections to web servers, which causes an improvement in performance over Wget, however, it is not possible to run a normal web browser and record page load times automatically, forcing us to use Wget. Table 3.1 and Figure 3.1 show the results of manually testing 5 web pages sequentially, concurrently, and through prefetching. These web pages were tested 10 times each connecting to Apache HTTP Server, which is designed to support concurrent connections, using Mozilla

Firefox 2.0.0.2 [36] with the Fasterfox 2.0.0 plugin [37]. The Fasterfox plugin is used for disabling concurrent connections and measuring page load times. Figure 3.2 shows the results from testing server to client prefetching as described using Wget.

Table 3.1. Prefetching compared with sequential and concurrent downloading (s)

Website	Sequential	Concurrent	Prefetching
www.amazon.com	15.017	14.796	4.216
www.aol.com	8.956	8.567	3.163
www.google.ca	0.257	0.249	0.263
www.mapquest.com	4.915	4.371	2.327
www.wikipedia.org	4.830	4.320	2.012

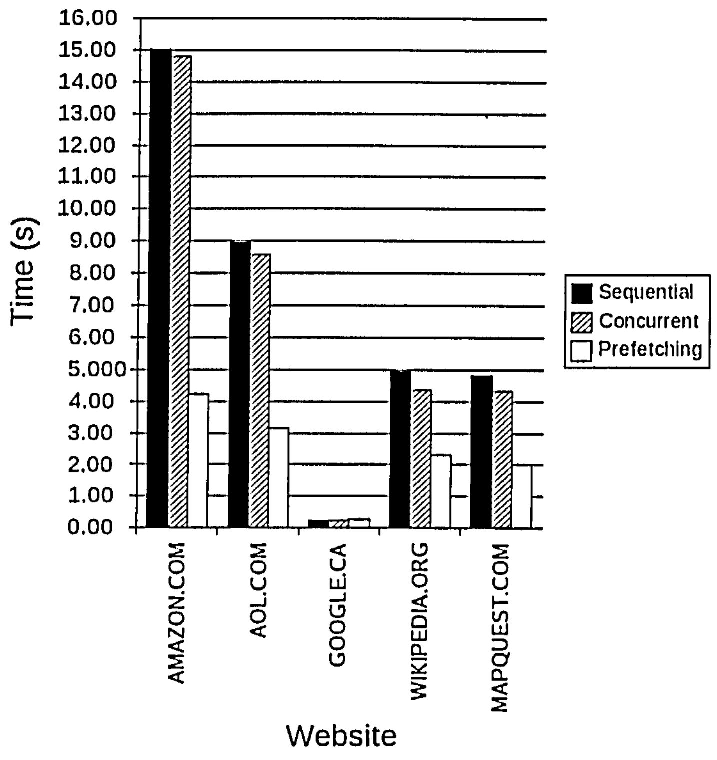

Figure 3.1: Prefetching compared with sequential and concurrent downloading.

The results of Table 3.1 and Figure 3.1 are not accurate over long term, since each value is the average over just 10 trials. However, Table 3.1 does show a general tendency that compared to the effects of prefetching, the effect of downloading web pages using concurrent connections is minimal. We use the results of this test with Firefox to validate the results of further testing where prefetching is compared only to sequential downloading using GNU Wget.

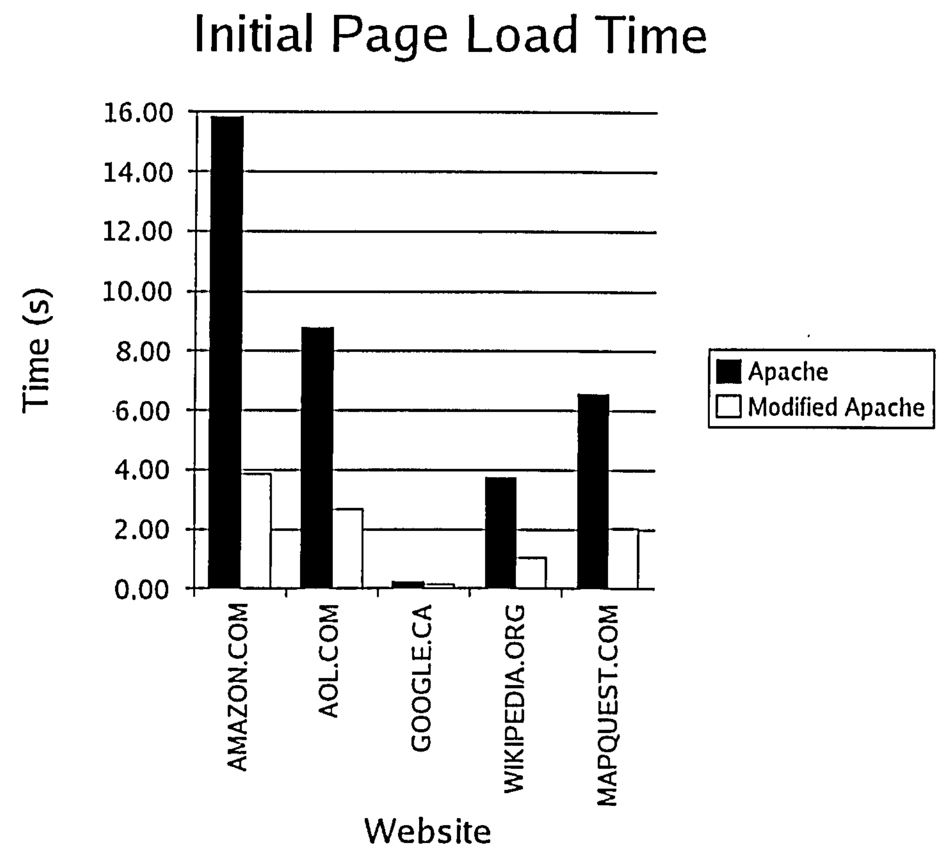

Figure 3.2: Page load time improvement for modified Apache.

Clearly, the improvements resulting in prefetching embedded objects from server to client far outweigh the benefits of concurrent downloading shown in Table 3.1. Table 3.2 shows the overall results obtained from testing. Figure 3.2 and Table 3.2 show the results of testing using GNU Wget. Each web page is tested 100 times.

Table 3.2. Page load time improvement for modified HTTP protocol.

Website	Objects	Time (standard) (s)	Time (prefetching) (s)	% of standard
www.amazon.com	60	15.85	3.88	24.48%
www.aol.com	34	8.78	2.66	30.30%
www.mapquest.com	28	6.55	2.04	31.15%
www.wikipedia.org	15	3.72	1.03	27.61%
www.google.ca	1	0.23	0.16	69.57%

As anticipated, there is a correlation between the number of objects embedded in a web page and the benefit achieved through prefetching from server to client. As tested, the website having the greatest number of embedded objects tend to have the greatest improvement due to prefetching, and the website having the fewest embedded objects had the lowest improvement.

Our second set of tests involved measuring the performance change when testing over a high speed local area network. The anticipation in these tests is that there will be less benefit to prefetching as prefetching is a means of overcoming network latency and in a high speed local area network, network latency is very low.

As with the previous tests, each web page is tested 100 times both for prefetching enabled and for prefetching disabled. Unlike the previous tests, our testing for prefetching disabled include using the prefetching proxy, but with prefetching disabled. We did this because with the higher network speed, the overhead from running the proxy will have a far greater impact on the overall times recorded.

Figure 3.3 shows the relationship between the page load times of standard HTTP/1.0 and modified prefetching HTTP/1.0. Table 3.3 summarizes Figure 3.3. Table 3.4 compares the benefit of prefetching on a slow network with the benefit of prefetching on a fast network.

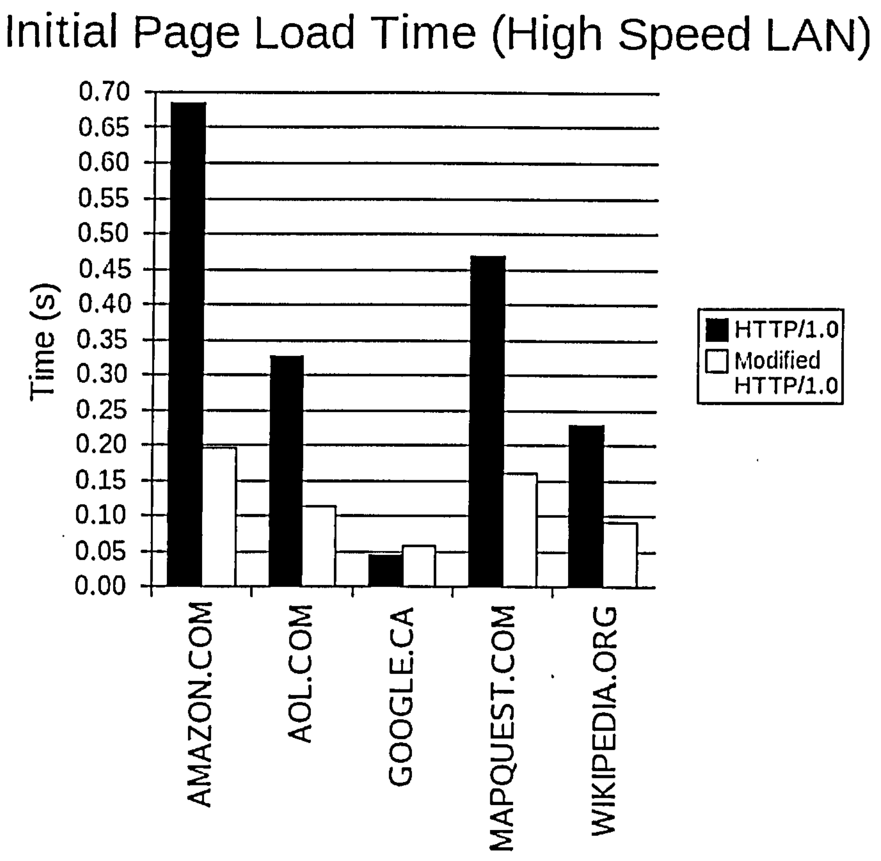

Figure 3.3: Standard and Modified HTTP/1.0 on a high speed LAN

Table 3.3. Summary of Standard and Modified HTTP/1.0 on a high speed LAN

Website	Objects	Time (standard) (s)	Time (prefetching) (s)	% of standard
www.amazon.com	60	0.68	0.20	28.55%
www.aol.com	34	0.33	0.11	34.54%
www.mapquest.com	28	0.47	0.16	34.06%
www.wikipedia.org	15	0.23	0.09	40.31%
www.google.ca	1	0.05	0.06	128.11%

The number of objects associated with a web page is determined by the configuration of the web pages being tested. These numbers represent the actual web pages available at the specified URL's at the time when the testing was performed.

The results from testing modified HTTP/1.0 on a high speed LAN show that there is still a benefit in prefetching. The overall difference in time found by applying HTTP prefetching on a high speed network is lower than what was found when prefetching with a low speed network, but the overall transfer time is also lower because of the higher transfer speeds. Because of this, prefetching over a high speed network yields similar percentage improvements to prefetching over a low speed network. As anticipated, the greater the network latency, the greater the reduction in overall time achieved by prefetching. Table 3.4 summarizes the results of prefetching over a high speed network compared to prefetching over a low speed network.

Table 3.4. Benefit of prefetching on low speed network vs high speed network

Website	**% of standard – low speed**	**% of standard – high speed**
www.amazon.com	24.48%	28.55%
www.aol.com	30.30%	34.54%
www.mapquest.com	31.15%	34.06%
www.wikipedia.org	27.61%	40.31%
www.google.ca	69.57%	128.11%

3.6. CONCLUSION

In testing the modified Apache for the popular websites www.amazon.com, www.aol.com, www.google.ca, www.mapquest.com, and www.wikipedia.org, we have determined that prefetching from server to client offers significant benefit over standard HTTP protocol. Our testing provided a reduction in page load times for the mentioned web sites equal to 24.48%, 30.30%, 69.57%, 31.15%, and 27.61% of standard time respectively when using a low speed WAN, and 28.55%, 34.54%, 128.11%, 34.06%, and 40.31% respectively when using a high speed LAN. The results of testing over show that even for a high speed network, prefetching causes a significant reduction in overall web page delivery time. It should be noted that since the overall download times are much smaller in the high speed network than in the low speed network, the overall improvement times are also.

Our testing of concurrent vs. sequential web page retrieval validates the above results which were obtained by comparing prefetching web page retrieval against sequential web page retrieval. It was determined that the benefit of concurrent downloading is minor when compared to the benefit of prefetching.

We can conclude, based on our results, that prefetching has a greater effect on web page delivery time where there are a greater number of embedded objects, and on networks where there is a longer latency.

CHAPTER 4

AN EFFICIENT PREFETCHING HTTP SERVER

4.1. INTRODUCTION

As a continuation of our work on prefetching Apache HTTP server, we have developed a custom HTTP/1.0 server to implement the same advanced server to client prefetching strategy as we previously applied to Apache.

Apache HTTP Server is a large and complex server and is adaptable to nearly all uses. It is far more complex than what is required by most web servers, for example, it supports authentication and encryption, features which are necessary for many websites, but not all. Apache pays for this complexity in having a greater resource overhead. The source code for Apache HTTP Server version 2.2.4 is 28 MB when uncompressed. The size of our custom HTTP Server is under 100 KB.

We developed server to client prefetching for the purpose of improving the delivery performance of web pages. We now extend our solution to improve the efficiency of the server which is responsible for prefetching. Our custom HTTP server addresses efficiency in terms of required CPU time, memory, and disk space.

4.2. DATA COLLECTION AND METHODOLOGY

We perform multiple experiments to determine the performance of our custom HTTP server. Our initial tests are the same as the ones performed for testing the server to client prefetching module for Apache HTTP Server. We also compare the CPU time and memory footprint of our HTTP server to that of Apache, as well as comparing the difference in the server's capacity.

4.3. PROPOSED STRATEGY

As mentioned before, for reducing the system requirements of the HTTP server, we wrote a new HTTP server that is much simpler than Apache, while maintaining all the features of HTTP/1.0 required by the majority of web sites. In addition to the overall simplicity of our HTTP server, modifications to it for application to specific purposes will be much easier due to the much smaller code base.

As with our custom module for server to client prefetching in Apache HTTP Server, we implement the same features in our HTTP server. We implement these features such that selection between Apache and our custom server requires no changes to the client.

4.4. EXPERIMENTAL DESIGN

We performed three tests to measure the characteristics of our custom HTTP server. The first test is the same as the tests performed on the custom module for Apache HTTP Server as discussed in the previous chapter. This test is to measure the effects of prefetching with a simpler and more predictable server.

Our second test, for measuring the load handling capacity of our custom server, involved using the tool ApacheBench 2.0.40 [38] (Unix command "ab") for measuring and comparing the load handling capacities for Apache and our custom server. This utility is run from the same machine as is running the HTTP server in order to measure the peak server performance. This test measures the number of requests that can be fulfilled per second with varying file sizes and number of concurrent downloads.

The third test is a measurement of memory usage for both Apache and for our custom HTTP server.

The design of our tests are organized as follows; testing server to client prefetching performance is described in section 4.4.1, testing server workload capacities with ApacheBench are explained in section 4.4.2, testing memory usage is presented in section 4.4.3.

4.4.1. SERVER TO CLIENT PREFETCHING IN CUSTOM HTTP SERVER

We perform the same testing for our custom HTTP server as is outlined in section 3.4 using the same popular websites identified by IRCache and SeekingAlpha. These sites are http://www.amazon.com, http://www.aol.com, http://www.google.ca, http://www.mapquest.com, and http://www.wikipedia.org.

The server running our custom HTTP server was located on the same residential network as in the previous tests, bearing peak downstream bandwidth of 6.0 Mbps and peak upstream bandwidth of 800 Kbps. The client was the same machine running on the same network as in the previous tests, and the client software and testing scripts are unaltered from previous tests. The client software Wget was used to download web pages, including all embedded objects, from the server through our custom client-side proxy interface for server to client prefetching.

4.4.2. LOAD HANDLING CAPACITY

ApacheBench provides two measurements for HTTP server performance; requests per second, and transfer rate. It is capable of performing any number of requests either sequentially, or with a configurable level of concurrency.

ApacheBench is capable of making requests through a proxy, however, it is not suitable for testing prefetching performance since it can only request one file - it can not follow a request for an HTML file with requests for embedded objects. For this reason, we do not use ApacheBench to measure server to client prefetching performance, but rather we use it to compare the non-prefetching performance of both servers.

ApacheBench tests server work capacity for specific files hosted on the web server. We have created files of multiple sizes to compare the performance of our custom HTTP server with Apache HTTP Server. We have created files of size 500 B, 90 KB, 200 KB, 275 KB, 920 KB, 17.5 MB, and 100 MB for testing the servers. It is shown in [39] that the average size of a web file is about 20 KB. Small files are ones that are smaller than average, and similarly, large files are ones that are larger than average. We also tested with varying levels of concurrency, including 1, 5, 10, 25, 50, and 100 simultaneous connections. The level of concurrency is the number of parallel downloads that the tool ApacheBench will perform simultaneously.

4.4.3. MEMORY USAGE

For measuring memory usage, we use the utility smem.py [40], which reads a process' smaps - /proc/$pid/smaps, and translates it into a human readable form, showing virtual memory size, and both shared and private RSS. RSS measures the pages that are mapped to memory. We measure the HTTP server memory usage when not serving files because there is no way to measure memory usage real-time while the processes are serving data.

4.5. RESULTS

Results are organized as follows; results of performance testing of prefetching in custom HTTP Server are in section 4.5.1, results of benchmarking with ApacheBench are in section 4.5.2, memory usage statistics are in section 4.5.3

4.5.1. RESULTS OF SERVER TO CLIENT PREFETCHING IN CUSTOM SERVER

As with our server to client prefetching module for Apache HTTP Server, our custom server, when prefetching is enabled, demonstrated a significant performance improvement over a standard HTTP server.

The results of testing our custom HTTP server are similar to the results obtained from modified Apache in chapter 3. The performance measurements we obtained for Apache HTTP Server are slightly better than our custom server for prefetching. This could in part be due to variations in network traffic when testing. The two servers were not tested concurrently, as doing so would cause the two tests to interfere with each other.

Initial page load times for our custom HTTP server with prefetching are shown in Figure 4.1(a). Figure 4.1(b) shows the performance of server to client prefetching in modified Apache HTTP Server. Figure 4.1(b) was not generated from testing in this section, but is provided for the purpose of comparing Apache with our custom HTTP server.

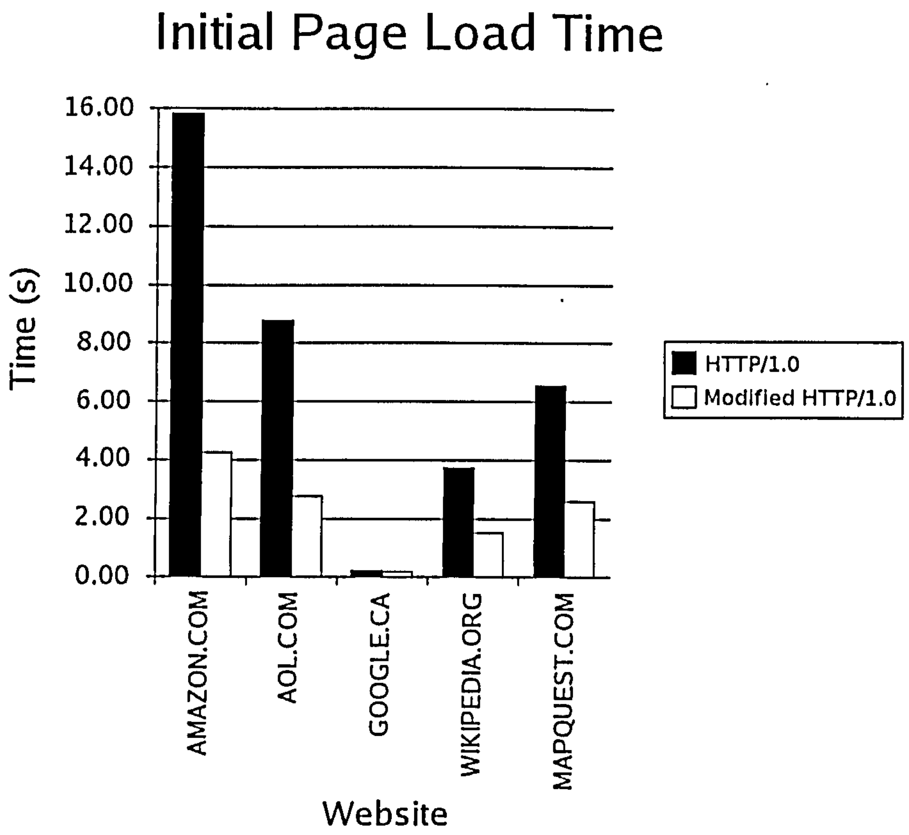

Figure 4.1(a): Page load time improvement for modified HTTP protocol, custom server.

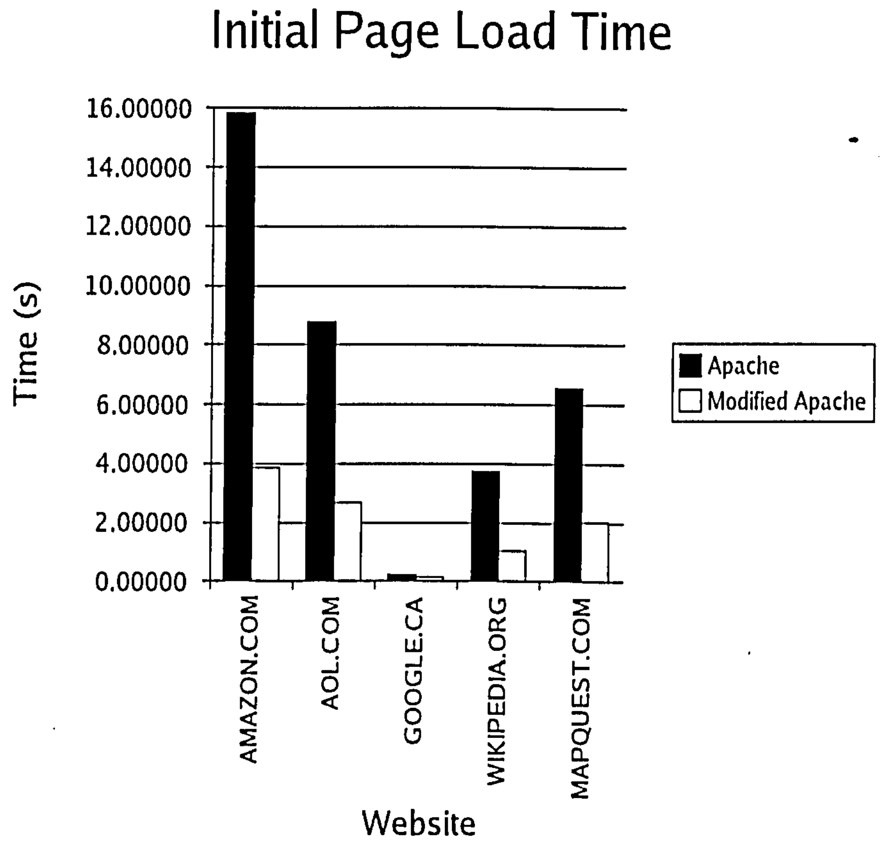

Figure 4.1(b): Page load time improvement for Apache with server to client prefetching.

Table 4.1 summarizes Figures 4.1(a) and 4.1(b).

Table 4.1. Page load times for Apache, modified Apache, and custom HTTP server (s).

Website	Apache	Modified Apache	Custom Server
www.amazon.com	15.85	3.88	4.25
www.aol.com	8.78	2.66	2.73
www.mapquest.com	6.55	2.04	2.56
www.wikipedia.org	3.72	1.03	1.44
www.google.ca	0.23	0.16	0.24

4.5.2. RESULTS FOR LOAD HANDLING TEST

From ApacheBench, we can see from Table 4.4, that in terms of a high volume server, Apache has an edge over our custom server. However, our custom server shows significant strength in serving smaller files. For serving small files, our server shows performance significantly greater than Apache. In the 500 KB file size with 5 concurrent downloads, our server shows an 82% performance improvement over Apache. For all number of concurrent connections with 500 KB file size, our server shows an advantage over Apache. For files up to 200 KB with 1 connection, our server performs better than Apache.

Results for our custom HTTP server are shown in Table 4.2, results for Apache are shown in Table 4.3, percentage improvement for our server are shown in Table 4.4.

Table 4.2(a). Number of requests per second - Custom HTTP server.

Size of File	1 Conn.	5 Conn.	10 Conn.	25 Conn.	50 Conn.	100 Conn.
500 B	3215.12	5197.78	3541.08	3520.01	3618.73	3440.80
90 KB	1956.91	2080.34	1720.58	1607.92	1607.67	1642.39
200 KB	1274.91	1330.85	1282.87	1268.20	1220.41	1156.76
275 KB	948.59	1034.25	967.77	777.90	745.42	902.36
920 KB	304.12	295.34	300.35	364.86	306.63	302.92
17.5 MB	20.85	20.44	20.89	19.14	16.48	16.04
100 MB	3.77	3.28	3.48	3.06	2.95	2.78

Table 4.2(b). Number of requests per second - Apache HTTP Server.

Size	1 Conn.	5 Conn.	10 Conn.	25 Conn.	50 Conn.	100 Conn.
500 B	2613.35	2875.38	3202.36	2843.17	2866.97	2789.24
90 KB	1778.44	2166.80	1987.56	1965.14	1941.63	1883.77
200 KB	1252.65	1555.35	1537.23	1483.31	1460.90	1400.99
275 KB	1055.82	1243.64	1178.24	1153.34	1110.31	1185.77
920 KB	425.16	456.89	510.60	484.92	462.20	460.62
17.5 MB	28.03	25.97	27.93	29.30	27.16	29.37
100 MB	4.16	4.69	5.09	4.97	4.64	4.26

Tables 4.2(a) and 4.2(b) show the number of requests per second that each server is capable of fulfilling with varying file size and number of concurrent downloads. The numbers of concurrent downloads tested are 1, 5, 10, 25, 50, and 100.

Table 4.3. Percent benefit of Custom server over Apache HTTP Server, requests per second.

Size	1 Conn.	5 Conn.	10 Conn.	25 Conn.	50 Conn.	100 Conn.
500 B	23.03	80.77	10.58	23.81	26.22	23.36
90 KB	10.04	-3.99	-13.43	-18.18	-17.20	-12.81
200 KB	1.78	-14.43	-16.55	-14.50	-16.46	-17.43
275 KB	-10.16	-16.84	-17.86	-32.55	-32.86	-23.90
920 KB	-28.47	-35.36	-41.18	-24.76	-33.66	-34.24
17.5 MB	-25.62	-21.29	-25.21	-34.68	-39.32	-45.39
100 MB	-9.38	-30.06	-31.63	-38.43	-36.42	-34.74

Table 4.3 was generated by taking the difference between the results from tables 4.2(a) and 4.2(b), and calculating the percent improvement with respect to Apache.

In Table 4.3, positive numbers indicate the custom HTTP server outperforming Apache HTTP Server. Negative numbers indicate Apache outperforming the custom HTTP server.

4.5.3. MEMORY USAGE STATISTICS

Our custom server and Apache function entirely differently. Apache uses a main listening process to receive connections, which it passes on to worker processes. The main process controls the number of worker processes by the number of connections that are currently made to it by client machines, and within the constraints established by the configuration file.

The main Apache process uses 10312 KB shared memory, and 3112 KB private memory with a virtual address space of 166884 KB. Worker processes use 7252 KB shared memory, 216 KB private memory, and have a virtual address space of 7468 KB. A default configuration of Apache establishes a minimum 8 worker processes, a minimum of 5 spare worker processes to handle new requests quickly, and a maximum of 256 worker processes.

The private memory footprint of a server is determined by adding the total private memory used by all processes associated with that server.

The minimum private memory footprint of Apache HTTP Server is;

3112 KB + 8 * 216 KB = 4840 KB of private memory.

The maximum private memory footprint of Apache HTTP Server, not considering the additional memory used by worker processes while they are operating, is;

3112 KB + 256 * 216 KB = 58408 KB of private memory.

Amount of shared memory is not considered since it is not necessarily used by Apache.

Our custom HTTP server, on the other hand, uses 856 KB shared memory and 172 KB private memory with a virtual address space of 16572 KB. This is only 21.24% of the minimum memory usage by Apache. Additionally, our custom HTTP server does not create new processes, but simply threads, which share address space.

4.6. CONCLUSION

Our custom HTTP server's performance is very similar to the performance of Apache HTTP Server when used for client to server prefetching. For serving small to average files, our server has a definite advantage over Apache, delivering files as much as 82% faster than Apache. With files much larger than average, Apache still performs very strongly. Additionally, our server has a much smaller memory footprint than Apache.

With the characteristics of our custom HTTP server, it is ideally suited to a low to moderate workload with content of average sizes. Since prefetch eliminates the need for a client to establish concurrent connections with the server, use of prefetching reduces the overall need for concurrency. Since prefetching reduces the need for concurrency, the custom HTTP server can perform better than Apache over a larger range of sites than what are shown by the ApacheBench test results. For example, if there is one client creating 5 or 10 concurrent connections to download the same number of under 200 KB files, then prefetching will reduce that to 1 connection, allowing our custom server to outperform Apache.

The simplicity of our server also allows it to be much more easily modified over Apache. This means that if it is necessary to alter a web server to perform a task that no other web server already can perform, then the amount of labor involved in modifying our custom HTTP server is significantly lower than what would be involved in creating similar modifications to Apache HTTP Server.

On the other hand, for a server needed to handle a high volume of requests or large file sizes, Apache is better suited.

CHAPTER 5

PREFETCHING EMBEDDED MULTIMEDIA

5.1. INTRODUCTION

This chapter is regarding a special case of use for HTTP protocol; multimedia services. Though the focus of this project is not specifically the serving of multimedia content, it is, nevertheless, a necessary application of HTTP protocol that must be considered in building a complete general purpose web server.

The previous chapters discussed improvements to the HTTP protocol to reduce the time that it takes for web pages with general text and image type content to be delivered from the server to the client. Some web pages go beyond text and images and may include other content such as videos.

One of the problems with prefetching embedded multimedia files is their size. In order to achieve a balance between smooth playback and quality, embedded multimedia files, such as videos, may take nearly as long to download as they take to play. For a five minute video, the download time can be as high as five minutes. In the event that a multimedia file is embedded in a web page where it is not positioned such that it will be prefetched after all other embedded content, the embedded objects following it, which could be very small images, will have to wait for the media file to download.

In general, videos can be played in two ways; either from a static file, or by streaming. A streaming video is one that plays while it is still being received. A static media file is one that is stored in its entirety on the server before ever being requested by a client.

HTTP uses Transport Control Protocol (TCP) connections to transfer data between server and client. TCP guarantees delivery of data at the expense of some throughput. As a result of the lowered performance in TCP, other custom media protocols use User Datagram Protocol (UDP). For improving throughput, UDP sacrifices reliability [24]. When multiple packets are sent from a server to a client using UDP, packets can be lost by, for example, overflowing a buffer on some device between the server and the client. When a packet is lost, there is no way of resending it. In video data, this can mean loosing frames or parts of frames that are not received. Since UDP is faster, if it is not critical that all of the data be received perfectly, it can be a better solution for streaming video. Regardless, in some cases, it can be easier or preferable to use a protocol using TCP connections, such as HTTP. In particular, it is easier both for the content provider and the client to use existing HTTP servers rather than adding specific software on both the server and on the client in order to use less common UDP protocols. Media files served by an HTTP server can be received by any client with a web browser and can be distributed by any HTTP server. In cases where media delivery must be guaranteed, such as security videos, HTTP over TCP is superior to any protocol using UDP connections.

The HTTP protocol cannot deliver true streaming media. It is capable only of serving static files, however, it does deliver data in order. Most static audio and video files are read by a media player from start to end as they are played, this means that the media player will use data from the file in the same order that an HTTP server will transmit it. What this means is that there may be enough data from a static video available at the client to begin playback before the entire file is transmitted, allowing static media to be streamed over HTTP. Many popular media players, such as the open sourced mplayer, are capable of playing media files while they are being received.

The amount of data required to begin playback of a file depends on the media player. In order to compensate for fluctuations in network performance, media players will usually buffer a few seconds of video before beginning playback, however, the buffer is optional and its size is controllable. We use our prefetching technique to reduce the wait time between issuing a request for a web page, and beginning playback of an embedded multimedia object. For the purpose of testing, we assume that the media player can begin playback as soon as the first byte of the media has been received. The results of testing server to client prefetching with multimedia data were published in [41].

5.2. DATA COLLECTION AND METHODOLOGY

To test the improvement in the media playback start time, we test the difference in the time required to deliver an HTML file with one embedded object of length 1 byte from server to client. We use an embedded object of 1 byte because the delivery of the first byte of the object signifies the start of playback in the media player.

5.3. PROPOSED STRATEGY

Our strategy to improve the delivery of multimedia objects from server to client involves using server to client prefetching method that we introduced in Chapters 3 and 4 for Apache HTTP Server and for our custom HTTP server.

For serving large multimedia files, our strategy requires some modification to the Apache prefetching module, our custom HTTP server, and to the client side proxy we use to enable server to client prefetching. Due to the problem in prefetching large embedded media files as outlined in 5.1, our strategy must be modified to allow the smaller objects following the embedded media file to be prefetched before the entire media file is sent to the client. To accomplish this, we only prefetch part of the embedded multimedia object, and have the proxy request the rest of the media object by establishing a second connection with the server. Since we have already prefetched part of the embedded multimedia object, the media handler will have enough data to begin playback while the second connection is established for the rest of the file. Because the media file will be delivered slightly faster than the media handler will require data, the time lost in establishing the second connection will not impact the playback of the file.

Our modification to the prefetching strategy is to improve fairness to other embedded objects while allowing multimedia object start time to benefit from prefetching.

5.4. EXPERIMENTAL DESIGN

We perform two experiments to measure the improvement in the start time for embedded multimedia objects. The first experiment uses Apache HTTP Server with prefetching, the second uses our custom prefetching HTTP server. The experiments are identical.

Our experiments are very straightforward. We have created several HTML documents with a number of embedded objects. One of our embedded objects is a multimedia object of length 1 byte. These generated web pages also contain a number of average sized images, specifically, 0, 5, 10, and 25 images. We serve these web pages from server to client 50 times each with prefetching disabled and 50 times with prefetching enabled, recording the total elapsed time each time the page is served, and calculate the averages for prefetching and for standard HTTP. The web browser invokes the media handling plugin the moment that it detects that there is an embedded video within the web page, but until the first byte of the media file is received, there is nothing for the media handler. The media handler will begin working on the media file being received the moment the first byte is received, so the difference in the elapsed time for standard HTTP and for prefetching is the improvement in the time it takes for an embedded media object to begin playback on the client.

5.5. RESULTS

The results for this section are divided into two sections; results for modified Apache HTTP Server in section 5.5.1, and results for our custom prefetching HTTP server in section 5.5.2.

5.5.1. RESULTS FOR MODIFIED APACHE HTTP SERVER

For loading our test web page using standard Apache HTTP Server, we have measured an average page load time of 0.074809 seconds. With prefetching enabled, the page load time is reduced by 0.004843 seconds to 0.069965 seconds.

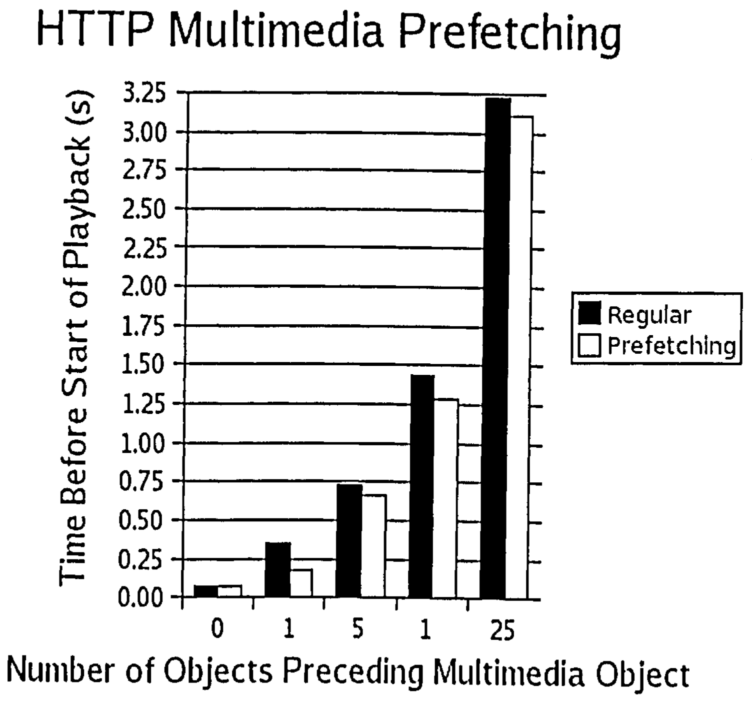

Figure 5.1: Prefetching improvement on embedded media playback start time - Apache

Table 5.1. Multimedia object playback start time - Apache (s)

Number of objects	Standard Apache	Prefetching Apache	Improvement
0	0.074809	0.069965	0.004843
1	0.348832	0.170879	0.177953
5	0.735006	0.659930	0.075077
10	1.435844	1.282452	0.153392
25	3.222278	3.111592	0.110686

The improvement obtained is amount of time, in seconds, that prefetching saves from the multimedia object playback start time.

5.5.2. RESULTS FOR CUSTOM HTTP SERVER

For loading our test web page using our custom HTTP Server, we have measured an average page load time of 0.060630 seconds. With prefetching enabled, the page load time is reduced by 0.013122 seconds to 0.047508 seconds.

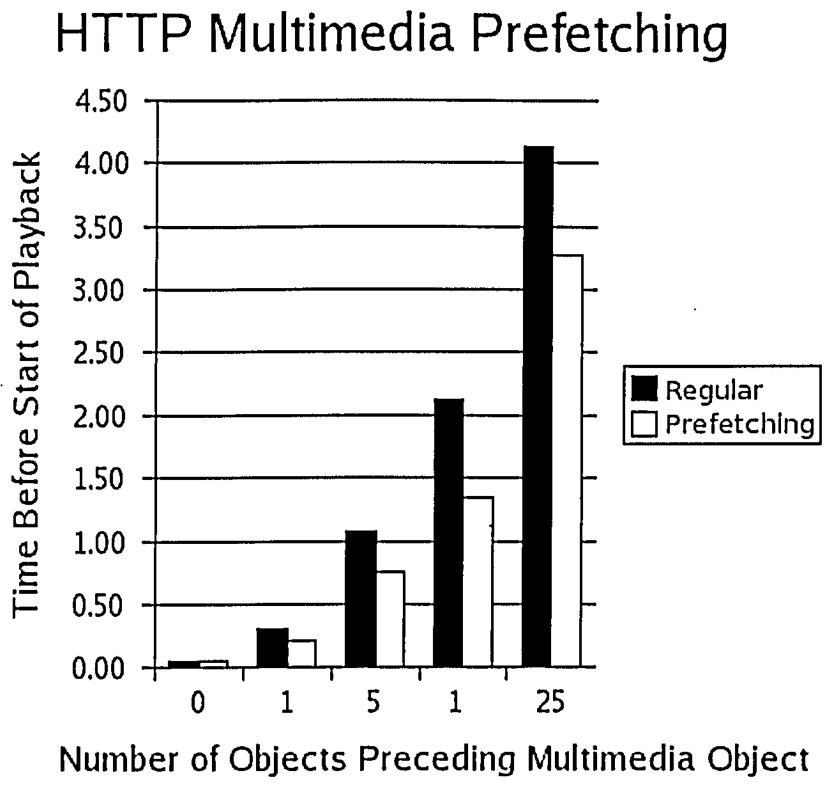

Figure 5.2: Prefetching improvement on embedded media playback start time - Custom

Table 5.2. Multimedia object playback start time - Custom

Number of objects	Standard HTTP	Prefetching HTTP	Improvement
0	0.060630	0.047508	0.013122
1	0.305684	0.219592	0.086092
5	1.067207	0.751699	0.315508
10	2.129906	1.337465	0.792441
25	4.132104	3.273403	0.858701

As in section 5.5.1, the improvement is again the difference in seconds between the playback start time for normal retrieval and for prefetching.

5.6. CONCLUSION

In prefetching Apache HTTP Server, we have improved the playback start time for a multimedia object by 0.004843 seconds for web pages with no additional embedded objects, to as much as 0.110686 seconds for web pages with 25 additional embedded objects. Our improvement for the playback start time for our custom HTTP server is 0.013122 seconds for web pages with no additional embedded objects, to as high as 0.858701 seconds for web pages with 25 additional embedded objects.

Similarly to previous tests, the number of embedded objects is related to the improvement in file access time. The more embedded objects contained within a web page, the greater the improvement due to prefetching.

CHAPTER 6

CONCLUSIONS

In this chapter, we begin by restating the problems in distributing-web data over the HTTP protocol in Section 6.1. We then evaluate each of our four solutions in Section 6.2.

6.1. DISTRIBUTING WEB DATA OVER HTTP

HTTP is one of the primary protocols used for transferring data on the Internet. The client makes requests to HTTP servers for specific hosted files and the server responds by sending the requested files. Web pages that are written in the HTML document format may require additional files before they can be properly displayed by the client web browser.

The fact that HTML contains embedded objects in the form of distinct files means that the web browser must receive the HTML file before it knows what additional objects to request. This delay before requesting embedded objects is directly perceivable as client latency. Also perceivable as client latency, is server delay. Server delay is the time it takes for a server to generate a response to a request it receives.

The underlying transport protocol, TCP, used by HTTP is very inefficient for making small transfers due to *slow start*, which is used to maximize bandwidth utilization on long transfers.

6.2. CONTRIBUTIONS

The contributions of this thesis include an enhancement made internally to the Apache HTTP Server, and externally to the HTTP protocol. We discuss Apache disk to memory prefetching in section 6.2.1, Apache server to client prefetching in section 6.2.2, our efficient custom server to client prefetching HTTP server in section 6.2.3, and the impact we make on the distribution of embedded streaming media content in section 6.2.4.

6.2.1. DISK TO MEMORY PREFETCHING IN APACHE HTTP SERVER

In Chapter 2, we introduced a form of disk to memory prefetching implemented in Apache HTTP Server. In this modification, we were able to reduce the effects of disk latency on the service time for Apache HTTP Server. We tested this modification by measuring the memory hits when re-enacting the activity recorded in access traces and calculating the file hit ratio and byte hit ratio. We were able to obtain file hit ratio improvements in the range of 23.52% through 27.83%. Byte hit ratio improvements were in the range of 0.67% through 3.00%. Byte hit ratio improvements were much lower than hit ratio improvements due to the presence of very large non-embedded objects in the traces.

We also tested the change in user perceived latency by measuring the change in page load times caused by disk to memory prefetching. We found an improvement in the page load times of up to 31.3% for one site tested. Of the remaining 4 sites tested, one showed reduced performance of 3.9%, and the last three showed marginal improvements in the range of 0.6% to 2.1%.

6.2.2. SERVER TO CLIENT PREFETCHING IN APACHE HTTP SERVER

In Chapter 3, we introduced, in the form of a module for Apache HTTP Server, a mechanism for prefetching web data from server to client. In this approach, the HTTP server scans through requested HTML documents to determine which additional files will be required by the client. It then appends those files onto the response to the client. The client web browser is able to receive this modified response through the use of a custom built translator that the browser accesses through its proxy interface.

Since user perceived latency is more the effect of network latency than server latency, the result of server to client prefetching showed a much greater improvement in client latency than disk to memory prefetching. We were able to reduce user perceived latency to as little as 24.48% of what would be experienced without prefetching. The page load times for the 5 web pages tested were in the range of 69.57% through 24.48% of what their load times were before prefetching.

6.2.3. EFFICIENT SERVER TO CLIENT PREFETCHING CUSTOM HTTP SERVER

Most commercially available and open source web servers, such as Apache HTTP Server, are designed to be very robust and capable of handling huge traffic volumes and file sizes. Many applications of HTTP servers do not require the robustness and many features available in these servers. In Chapter 4, we describe a custom HTTP server that we built, implementing our server to client prefetching scheme, and being much smaller and more efficient than Apache HTTP Server.

Our custom HTTP server performed very similarly to Apache HTTP Server under the same tests that were performed in Chapter 3. In Chapter 4, we performed additional tests on both servers, showing a significant reduction in the code size of our custom HTTP server, a reduction in the memory usage, and an improved performance for light traffic loads when compared to Apache. The source code for our custom server is only 100 KB where Apache 2.2.3 is 28 MB. When running, our server uses only 21.24% of the minimum memory consumed by Apache. For small files and low concurrency, our server may be as much as 80.77% faster than Apache.

6.2.4. PREFETCHING OF EMBEDDED STREAMING MULTIMEDIA

One of the downsides to our original implementation of server to client prefetching was the lack of fairness among embedded objects. In a page with many small embedded objects and one or more very large embedded object, such as a streaming multimedia object, using prefetching would cause the small objects located after the large object to have to wait until after the large object was sent before being transmitted to the client. This problem was corrected by prefetching only the first part of large embedded objects. The amount prefetched would be enough to satisfy the media player long enough to create another connection to the server and begin receiving the rest of the object.

In Chapter 5, we tested the change in the playback start time for embedded multimedia objects. Since in our test we used generated websites instead of actual popular websites, we do not calculate improvement as a proportion, but only as an absolute time. We find that the absolute time saved from the playback start time of an embedded multimedia object is a direct function of the number of embedded object preceding it, and the network latency. Using our test configuration, we were able to obtain time reductions in the range of 10 milliseconds for having no embedded objects preceding the multimedia object, through to 860 milliseconds for having 25 objects preceding the multimedia object. As the number of objects preceding the multimedia object increases, so do the time savings, so for a web page containing 250 objects preceding a multimedia object, the time savings are expected to be approximately 9 seconds.

APPENDIX A - List of Acronyms

CD - Compact Disk. An optical storage medium capable of storing 650 MB of data.
CPU - Central Processing Unit.
CRLF - Carriage Return, Line Feed. The two byte identification for the ends of lines in HTTP.
CUPS - Common Unix Printing System.
DNS - Domain Name System. An association between IP addresses and human readable names used for referencing servers on networks.
DVD - Digital Versatile Disk. An optical storage medium capable of storing 4.7 GB of data on a single layer, or 8.5 GB on two layers.
GHB - Global History Buffer. In data prefetching, this is used to store the historic cache miss pattern.
GNU - A recursive acronym for "GNU's Not Unix". This is an operating system composed entirely of free software.
GPL - GNU General Public License. This is a free software license that causes derived work to also be subject to the same license. This license includes the free availability of program source code.
GWS - The web server software used by Google. It is believed to be an acronym for "Google Web Server".
HTML - Hypertext Markup Language. A language that defines the structure of web pages.
HTTP - Hypertext Transfer Protocol. This is a high level communications protocol used to transfer data on the Internet.
IP - Internet Protocol.
LAN - Local Area Network. This is a network that covers a small area, such as a home or office.
LRU - Least Recently Used. This is a cache replacement policy used to remove cached data.
RAM - Random Access Memory. Temporary data storage in the form of integrated circuits.
RSS - Resident Set Size. This is a measure of physical RAM usage.
TCP - Transmission Control Protocol. A protocol used for transferring data on networks. This protocol sacrifices some performance in exchange for guaranteeing successful delivery of data.
UDP - User Datagram Protocol. A protocol used for transferring data on networks. This protocol transfers data faster than TCP by not waiting for verification that data has transmitted successfully.
URL - Uniform Resource Locator. This is a name used to reference data on the Internet.
UTF-8 - 8-bit Unicode Transformation Format. This is a character encoding for Unicode.
WAN - Wide Area Network. This is a network that covers a wide area, such as a city or country.

APPENDIX B - Performance Analysis Tools

In this appendix, we describe the tools used to analyze performance.

ab: the Apache HTTP server benchmarking tool [38]. This tool is for benchmarking Apache HTTP Server. It generates requests to specified HTTP URLs and measures the number of requests that are fulfilled and the amount of time taken for each request. It is capable of performing tests using a user selectable level of concurrency. The output of the program breaks down the request time into time to connect, processing time, and waiting time. Most importantly, it outputs the number of requests per second that the server is capable of, as well as the overall transfer rate during the tests.

Firefox with Fasterfox: The Mozilla Firefox Web Browser [36], using the Fasterfox plugin [37], is able to display complete web page load times for any website tested. In addition, Fasterfox allows the level of concurrency to be modified in a field labeled as *"Max Connections"*. For testing the effect of download concurrency, we use concurrency settings of 0 and the default for Firefox, which is 24 connections.

smem.pl: This perl script is used to read Linux smaps. Linux smaps are files generated for every running process that keep track of memory usage for that process, including addresses and quantities, for executable and object files in use by the process. The smem.pl [40] program reads the smaps file for particular processes and translates them into human readable form. We use the output of this program for comparing the memory usage of Apache HTTP Server with the memory usage of our custom HTTP server.

prefetch testing script: We wrote a custom bash script for testing server to client prefetching. The script contains a simple loop that runs a predetermined number of times. Inside the loop, the system time is recorded, Wget is executed for the particular URL being tested, the system time is again recorded, and the difference between the finishing time and the starting time is output to a log file.

REFERENCES

[1] Y. C. Chehadeh, A. Z. Hatahet, A. E. Agamy, M. A. Bamakhrama, S. A. Banawan, "Investigating Distribution of Data of HTTP Traffic: An Empirical Study." *IEEE Innovations in Information Technology (IIT'06)*, pp. 1-5, Dubai, Nov. 2006

[2] B. A. Forouzan, *Data Communications and Networking*, Third Edition, pp. 603-624, 731-838, McGraw-Hill, 2004

[3] RFC 1945 - Hypertext Transfer Protocol -- HTTP/1.0, May 1996. Available at: http://www.faqs.org/rfcs/rfc1945.html

[4] RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1, section 14, subsection 35, July 1999. Available at http://www.faqs.org/rfcs/rfc2616.html

[5] Apache HTTP Server (The Apache Software Foundation). Available at: http://httpd.apache.org

[6] Netcraft Web Server Survey, available at http://news.netcraft.com

[7] mod_disk_cache module (The Apache Software Foundation). Available at: http://httpd.apache.org/docs/2.2/mod/mod_disk_cache.html

[8] mod_mem_cache module (The Apache Software Foundation). Available at: http://httpd.apache.org/docs/2.2/mod/mod_mem_cache.html

[9] mod_cache module (The Apache Software Foundation). Available at: http://httpd.apache.org/docs/2.2/mod/mod_cache.html

[10] htcacheclean (The Apache Software Foundation). Available at: http://httpd.apache.org/docs/2.2/programs/htcacheclean.htm

[11] T. Fagani, R. Rerego, F. Silvestri, "Boosting the Performance of Web Search Engines: Caching and Prefetching Query Results by Exploiting Historical Usage Data", *ACM Transactions on Information Systems*, v. 24, pp. 51-78, ACM Press, New York, NY, USA, 2006

[12] K. Nesbit, J. Smith, "Data Cache Prefetching Using a Global History Buffer." *IEEE Micro 25*, pp. 90-97, Dec. 2005

[13] S. Sair, T. Sherwood, B. Calder. "A Decoupled Predictor-Directed Stream Prefetching Architecture." *IEEE Transactions on Computers*, v. 52, n. 3, pp. 260-276, Oct. 2003

[14] D. Joseph, D. Grunwald, "Prefetching Using Markov Predictors", *IEEE Transactions on Computers*, v. 48, n. 2, pp. 121-133, June 1999

[15] J. Lee, S. Jeong, S. Kim, C. C. Weems, "An Intelligent Cache System with Hardware Prefetching for High Performance." *IEEE Transactions on Computers*, v. 52, n. 5, Oct. 2003

[16]A. Gendler, A. Mendelson, Y. Birk, "A PAB-Based Multi-Prefetcher Mechanism." *International Journal of Parallel Programming*, v. 34, n. 2, Aug. 2006

[17] N. A. Lewycky, B. M. Benhan, A. Abhari, "Improving the Performance of the Squid Proxy Cache", *In Proceeding of 9th Communications and Networking Simulation Symposium (CNS06)*, Huntsville, AL., USA, April 2006

[18] A. Pons, "Object Prefetching Using Semantic Links." *Database for Advances in Information Systems*, pp. 97-110, 2006

[19] V. Safronov, M. Parashar, "Optimizing Web servers using Page rank prefetching for clustered accesses." *Information Sciences*, v. 150, pp. 165-176, 2003

[20] X. Chen, X. Zhang, "A Popularity-Based Prediction Model for Web Prefetching.", *IEEE*

Computer Society, v. 36, pp. 63-70, March 2003

[21] E. Cohen, H. Kaplan, "Prefetching the means for document transfer: a new approach for reducing Web latency." *IEEE Computer and Communications Joint Conference (IEEE INFOCOM 2000)*, pp. 854-863, 2000

[22] D. Carr, "How Google Works", Baseline Magazine, July 2006

[23] Common Unix Printing System, available at http://www.cups.org

[24] B. A. Forouzan, *Data Communications and Networking, Third Edition*, pp. 601-603, McGraw-Hill, 2004

[25] S. Jun, A. Bestavros, A. Iyengar, "Accelerating Internet Streaming Media Delivery using Network-Aware Partial Caching." *Proceedings of the 22nd International Conference on Distributed Computing Systems*, July 2002

[26] W. Liao, P. Shih, "Architecture of Proxy Partial Caching Using HTTP for Supporting Interactive Video and Cache Consistency." *Proceedings of 11th International Conference on Computer Communications and Networks*, Oct. 2002.

[27] J. Z. Wang, P. S. Yu, "Fragmental Proxy Caching for Streaming Multimedia Objects." *IEEE Transactions on Multimedia*, v. 9, n. 1, Jan. 2007

[28] W. Qu, K. Li, M. Kitsuregawa, T. Nanya, "An optimal solution for caching multimedia objects in transcoding proxies." *Computer Communications*, v. 30, n. 8, June. 2007

[29] A. Abhari, A. Serbinski, M. Gusic, "Improving the Performance of Apache Web Server", *10th Communications and Networking Simulation Symposium (CNS'07)*, Mar. 2007

[30] IRCache Trace Files, The IRCache Project, Available at http://www.ircache.net/Traces/

[31] GNU Wget, GNU Project, The Free Software Foundation, available at http://www.gnu.org/software/wget

[32] RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1. Available at http://www.faqs.org/rfcs/rfc2616.html

[33] B. A. Forouzan, *Data Communications and Networking, Third Edition*, pp. 640-641, McGraw-Hill, 2004

[34] IRCache Top 50 Origins, available at http://www.ircache.net/Statistica/Top-Fifty-Servers

[35] Seeking Alpha, "The 20 Most Popular Websites", available at http://internet.seekingalpha.com/article/25309

[36] Firefox Web Browser, The Mozilla Project. Available at http://www.mozilla.com

[37] Fasterfox Plugin for Firefox Web Browser, Available at http://fasterfox.mozdev.org

[38] Apache HTTP server benchmarking tool, Unix command "ab", Manual page available at http://httpd.apache.org/docs/1.3/programs/ab.html

[39] R. Levering, M. Cutler. "The Portrait of a Common HTML Web Page", *ACM Symposium on Document Engineering (DocEng'06)*, Oct. 2006.

[40] B. Maurer, Program: smem.pl for reading Linux SMAPS, available at http://www.contrib.andrew.cmu.edu/~bmaurer/smem.pl

[41] A. Serbinski, A. Abhari. "Improving the Delivery of Multimedia Embedded in HTML over HTTP on Wireless Networks", *Third International Mobile Multimedia Communications Conference (Mobimedia 2007)*, to be held on Aug. 2007