The Affection of HTTP Compression in the Performance of Core web Components

Khushali Tirghoda

Abstract— HTTP compression addresses some of the performance problems of the Web by attempting to reduce the size of resources transferred between a server and client thereby conserving bandwidth and reducing user perceived latency.Currently, most modern browsers and web servers support some form of content compression. Additionally, a number of browsers are able to perform streaming decompression of gzipped content. Despite this existing support for HTTP compression, it remains an underutilized feature of the Web today. This can perhaps be explained, in part, by the fact that there currently exists little proxy support for the Vary header, which is necessary for a proxy cache to correctly store and handle compressed content.To demonstrate some of the quantitative benefits of compression, I conducted a test to determine the potential byte savings for a number of popular web sites.

Index Terms HTTP Compression Comparison, Compression Ratio Measurements, Apache Web Server Performance

1 Introduction

User perceived latency is one of the main performance problems plaguing the World Wide Web today. At one point or another every Internet user has experienced just how painfully slow the "World Wide Wait" can be. As a result, there has been a great deal of research and development focused on improving Web performance.

Currently there exist a number of techniques designed to bring content closer to the end user in the hopes of conserving bandwidth and reducing user perceived latency, among other things. Such techniques include prefetching, caching and content delivery networks. However, one area that seems to have drawn only a modest amount of attention involves HTTP compression.

Many Web resources, such as HTML, JavaScript, CSS and XML documents, are simply ASCII text files. Given the fact that such files often contain many repeated sequences of identical information they are ideal candidates for compression. Other resources, such as JPEG and GIF images and streaming audio and video files, are precompressed and hence would not benefit from further compression. As such, when dealing with HTTP compression, focus is typically limited to text resources, which stand to gain the most byte savings from compression.

Encoding schemes for such text resources must provide lossless data compression. As the name implies, a lossless data compression algorithm is one that can recreate the original data, bit-for-bit, from a compressed file. One can easily imagine how the loss or alteration of a single bit in an HTML file could affect its meaning.

The goal of HTTP compression is to reduce the size of certain resources that are transferred between a server and client. By reducing the size of web resources, compression can make more efficient use of network bandwidth. Compressed content can also provide monetary savings for those individuals who pay a fee based on the amount of bandwidth they consume. More importantly, though, since fewer bytes are transmitted, clients would typically receive the resource in less time than if it had been sent uncompressed. This is especially true for narrowband clients. Modems typically present what is referred to as the weakest link or longest mile in a data transfer; hence methods to reduce download times are especially pertinent to these users.

Furthermore, compression can potentially alleviate some of the burden imposed by the TCP slow start phase. The TCP slow start phase is a means of controlling the amount of congestion on a network. It works by forcing a small initial congestion window on each new TCP connection thereby limiting the number of maximum-size packets that can initially be transmitted by the sender [2]. Upon the reception of an ACK packet, the sender's congestion window is increased. This continues until a packet is lost, at which point the size of the congestion window is decreased [2]. This process of increasing and decreasing the congestion window continues throughout the connection in order to constantly maintain an appropriate transmission rate [2]. In this way, a new TCP connection avoids overburdening a network with large bursts of data. Due to this slow start phase, the first few packets that are transferred on a connection are relatively more expensive than subsequent ones. Also, one can imagine that for the transfer of small files, a connection may not reach its maximum transfer rate because the transfer may reach completion before it has the chance to get out of the TCP slow start phase. So, by compressing a resource, more data effectively fits into each packet. This in turns results in fewer packets being transferred thereby lessening the effects of slow start (reducing the number of server stalls) [4, 6, 14, 3].

In the case where an HTML document is sent in a compressed format, it is probable that the first few packets of data will contain more HTML code and hence a greater number of inline image references than if the same document had been sent uncompressed. As a result, the client can subsequently issue requests for these embedded resources quicker hence easing some of the slow start burden. Also, inline objects are likely to

- *Khushali is currently working as Assistant Professor in MCA Department at IITE,Ahmedabad,India*
- *E-mail: Tirghoda_khushali@yahoo.co.in*

be on the same server as this HTML document. Therefore an HTTP/1.1 compliant browser may be able to pipeline these requests onto the same TCP connection [6]. Thus, not only does the client receive the HTML file in less time he/she is also able to expedite the process of requesting embedded resources [6, 14].

Currently, most modern browsers and web servers support some form of content compression. Additionally, a number of browsers are able to perform streaming decompression of gzipped content. This means that, for instance, such a browser could decompress and parse a gzipped HTML file as each successive packet of data arrives rather than having to wait for the entire file to be retrieved before decompressing. Despite all of the aforementioned benefits and the existing support for HTTP compression, it remains an underutilized feature of the Web today.

2 Popular Compression Schemes

Although, there exists many different lossless compression algorithms today, most are variations of two popular schemes: Huffman encoding and the Lempel-Ziv algorithm.

Huffman encoding works by assigning a binary code to each of the symbols (characters) in an input stream (file). This is accomplished by first building a binary tree of symbols based on their frequency of occurrence in a file. The assignment of binary codes to symbols is done in such a way that the most frequently occurring symbols are assigned the shortest binary codes and the least frequently occurring symbols assigned the longest codes. This in turn creates a smaller compressed file [7].

The Lempel–Ziv algorithm, also known as LZ-77, exploits the redundant nature of data to provide compression. The algorithm utilizes what is referred to as a sliding window to keep track of the last n bytes of data seen. Each time a phrase is encountered that exists in the sliding window buffer, it is replaced with a pointer to the starting position of the previously occurring phrase in the sliding window along with the length of the phrase [7].

The main metric for data compression algorithms is the compression ratio, which refers to the ratio of the size of the original data to the size of the compressed data [13]. For example, if we had a 100 kilobyte file and were able to compress it down to only 20 kilobytes we would say the compression ratio is 5-to-1, or 80%. The contents of a file, particularly the redundancy and orderliness of the data, can strongly affect the compression ratio.

3 Proxy Support for Compression

Currently one of the main problems with HTTP compression is the lack of proxy cache support. Many proxies cannot handle the Content-Encoding header and hence simply forward the response to the client without caching the resource. As was mentioned above, IIS attempts to ensure compressed documents are not served stale by setting the Expires time in the past. Caching was handled in HTTP/1.0 by storing and retrieving resources based on the URI [2]. This, of course, proves inadequate when multiple versions of the same resource exist - in this case, a compressed and uncompressed representation. This problem was addressed in HTTP/1.1 with the inclusion of the Vary response header. A cache could then store both a compressed and uncompressed version of the same object and use the Vary header to distinguish between the two. The Vary header is used to indicate which response headers should be analyzed in order to determine the appropriate variant of the cached resource to return to the client [2].

4 Related Work

In [1], Mogul et al. quantified the potential benefits of delta encoding and data compression for HTTP by analyzing lives traces from Digital Equipment Corporation (DEC) and an AT&T Research Lab. The traces were filtered in an attempt to remove requests for precompressed content; for example, references to GIF, JPEG and MPEG files. The authors then estimated the time and byte savings that could have been achieved had the HTTP responses to the clients been delta encoded and/or compressed. The authors determined that in the case of the DEC trace, of the 2465 MB of data analyzed, 965 MB, or approximately 39%, could have been saved had the content been gzip compressed. For the AT&T trace, 1054MB, or approximately 17%, of the total 6216 MB of data could have been saved. Furthermore, retrieval times could have been reduced 22% and 14% in the DEC and AT&T traces, respectively. The authors remarked that they felt their results demonstrated a significant potential improvement in response size and response delay as a result of delta encoding and compression.

In [8], the authors attempted to determine the performance benefits of HTTP compression by simulating a realistic workload environment. This was done by setting up a web server and replicating the CNN site on this machine. The authors then accessed the replicated CNN main page and ten subsections within this page (i.e. World News, Weather, etc), emptying the cache before each test. Analysis of the total time to load all of these pages showed that when accessing the site on a 28.8 kbps modem, gzip content coding resulted in 30% faster page loads. They also experienced 35% faster page loads when using a 14.4 kbps modem.

Finally, in [3] the authors attempted to determine the performance effects of HTTP/1.1. Their tests included an analysis of the benefits of HTTP compression via the deflate content-coding. The authors created a test web site that combined data from the Netscape and Microsoft home pages into a page called “MicroScape”. The HTML for this new page totaled 42 KB with 42 inline GIF images totaling 125 KB. Three different network environments were used to perform the test: a Local Area Network (high bandwidth, low latency), a Wide Area Network (high bandwidth, high latency) and a 28.8 kbps modem (low bandwidth, high latency). The test involved measuring the time required for the client to retrieve the Microscape web page from the server, parse, and, if necessary, decompress the HTML file on-the-fly and retrieve the 42 inline images. The results showed significant improvements for those clients on low bandwidth and/or high latency connections. In fact, looking at the results from the entire test environments, compression reduced the total number of packets transferred by 16% and the download time for the first time retrieval of the page by 12%.

5 Experiments

We will now analyze the results from a number of tests that were performed in order to determine the potential benefits and drawbacks of HTTP compression.

5.1 Compression Ratio Measurements

The first test that was conducted was designed to provide a basic idea of the compression ratio that could be achieved by compressing some of the more popular sites on the Web. The objective was to determine how many fewer bytes would need to be transferred across the Internet if web pages were sent to the client in a compressed form. To determine this I first found a web page, [15], that ranks the Top 99 sites on the Web based on the number of unique visitors. Although the rankings had not been updated since March 2001 most of the indicated sites are still fairly popular. Besides, the intent was not to find a definitive list of the most popular sites but rather to get a general idea of some of the more highly visited ones. A freely available program called wget [16] was used to retrieve pages from the Web and Perl scripts were written to parse these files and extract relevant information.
The steps involved in carrying out this test consisted of first fetching the web page containing the list of Top 99 web sites. This HTML file was then parsed in order to extract all of the URLs for the Top 99 sites. A pre-existing CGI program [17] on the Web that allows a user to submit a URL for analysis was then utilized. The program determines whether or not the indicated site utilizes gzip compression and, if not, how many bytes could have been saved were the site to implement compression. These byte savings are calculated for all 10 levels of gzip encoding. Level 0 corresponds to no gzip encoding. Level 1 encoding uses the least aggressive form of phrase matching but is also the fastest, as it uses the least amount of CPU time when compared to the other levels, excluding level 0. Alternatively, level 9 encoding performs the most aggressive form of pattern matching but also takes the longest, utilizing the most CPU resources [13].
A Perl script was employed to parse the HTML file returned by this CGI program, with all of the relevant information being dumped to a file that could be easily imported into a spreadsheet. Unfortunately, the CGI program can only determine the byte savings for the HTML file. While this information is useful it does not give the user an idea of the compression ratio for the entire page - including the images and other embedded resources. Therefore, I set my web browser to go through a proxy cache and subsequently retrieved each of the top 99 web pages. We then used the trace log from the proxy to determine the total size of all of the web pages. After filtering out the web sites that could not be handled by the CGI program, wget or Perl scripts I was left with 77 URLs. One of the problems encountered by the CGI program and wget involved the handling of server redirection replies. Also, a number of the URLs referenced sites that either no longer existed or were inaccessible at the time the tests were run.
The results of this experiment were encouraging. First, if we consider the savings for the HTML document alone, the average compression ratio for level 1 gzip encoding turns out to be 74% and for level 9 this figure is 78%. This clearly shows that HTML files are prime candidates for compression. Next, we factor into the equation the size of all of the embedded resources for each web page. We will refer to this as the total compression ratio and define it as the ratio of the size of the original page, which includes the embedded resources and the uncompressed HTML, to the size of the encoded page, which includes the embedded resources and the compressed HTML.

The results show that the average total compression ratio comes to about 27% for level 1 encoding and 29% for level 9 encoding. This still represents a significant amount of savings, especially in the case where the content is being served to a modem user.

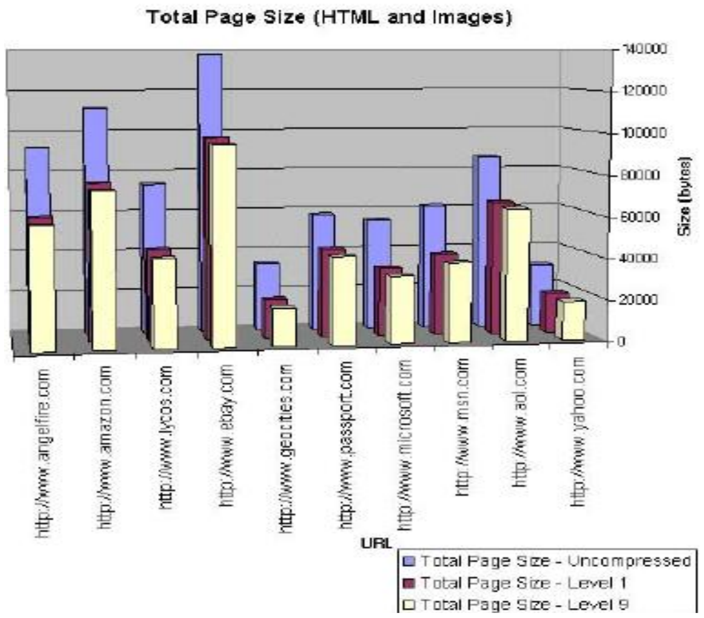

Fig.1 – The total page size (including HTML and embedded resources) for the top ten web sites.

Fig. 1 shows the difference in the total number of bytes transferred in an uncompressed web page versus those transferred with level 1 and level 9 gzip compression. Note the small difference in compression ratios between level 1 and level 9 encoding.
Table 1 shows a comparison of the total compression ratios for the top ten web sites. You can see that there is only a slight difference in the total compression ratios for levels 1 and 9 of gzip encoding. Thus, if a site administrator were to decide to

enable gzip compression on a web server but wanted to devote the least amount of CPU cycles as possible to the compression process, he/she could set the encoding to level 1 and still maintain favorable byte savings.

Table 1.
Comparison of the total compression ratios of level 1 and level 9 gzip encoding for the indicated URLs

URL	Level 1	Level 9
www.yahoo.com	36.353	38.222
www.aol.com	26.436	25.697
www.msn.com	35.465	37.624
www.microsoft.com	38.850	40.189
www.passport.com	25.193	26.544
www.geocities.com	43.316	45.129
www.ebay.com	29.030	30.446
www.lycos.com	40.170	42.058
www.amazon.com	31.334	32.755
www.angelfire.com	34.537	36.427

Ultimately, what these results show is that, on average, a compression-enabled server could send approximately 27% less bytes yet still transmit the exact same web page to supporting clients. Despite this potential savings, out of all of the URLs examined, www.excite.com was the only site that supported gzip content coding. This is indicative of HTTP compression's current popularity, or lack thereof.

5.2 Web Server Performance Test

Number footnotes separately in superscripts (Insert | Footnote)1. Place the actual footnote at the bottom of the column in which it is cited; do not put footnotes in the reference list (endnotes). Use letters for table footnotes (see Table 1). Please do not include footnotes in the abstract and avoid using a footnote in the first column of the article. This will cause it to appear of the affiliation box, making the layout look confusing.

The tests were designed to determine the maximum throughput of the two servers by issuing a series of requests for compressed and uncompressed documents. Using Autobench I was able to start by issuing a low rate of requests per second to the server and then increase this rate by a specified step until a high rate of requests per second were attempted to be issued. An example of the command line options used to run some of the tests is as follows:

```
./autobench_gzip_on --low_rate 10 --
 high_rate 150 --rate_step 10 --
 single_host --host1 192.168.0.106 --
 num_conn 1000 --num_call 1 --output_fmt
 csv --quiet --timeout 10 --uri1
/google.html --file google_compr.csv
```

These command line options indicate that initially requests for the google.html file will be issued at a rate of 10 requests per second. Requests will continue at this rate until 1000 connections have been made. For these tests each connection makes only one call. In other words no persistent connections were used. The rate of requests is then increased by the rate step, which is 10. So, now 20 requests will be attempted per second until 1000 connections have been made. This will continue until a rate of 150 requests per second is attempted. when looking at the results that the client may not be capable of issuing 150 requests per second to the server. Thus a distinction is made between the desired and actual number of requests per second.

[1]It is recommended that footnotes be avoided (except for the unnumbered footnote with the receipt date on the first page). Instead, try to integrate the footnote information into the text.

6 Apache Performance Benchmark

We will first take a look at the results of the tests when run against the Apache server. Fig. 2 & Fig. 3 represent graphs of some of the results from respective test cases.

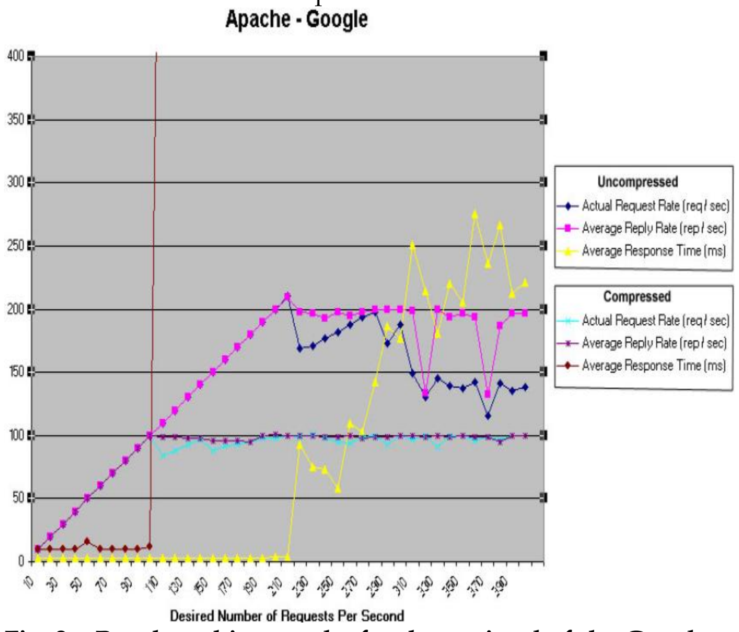

Fig. 2 – Benchmarking results for the retrieval of the Google HTML file from the Apache Server.

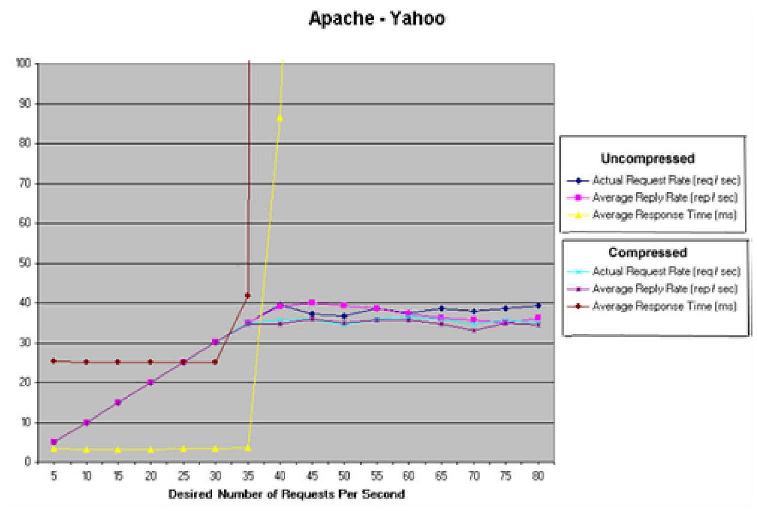

Fig. 3 – Benchmarking results for the retrieval of the Yahoo

HTML file from the Apache Server.

Referring to the graphs, we can see that for each test case a saturation point was reached. This saturation point reflects the maximum numbers of requests the server could handle for the given resource. Looking at the graphs, the saturation point can be recognized by the point at which the server's average response time increases significantly, often times jumping from a few milliseconds up to hundreds or thousands of milliseconds. The response time corresponds to the time between when the client sends the first byte of the request and receives the first byte of the reply.

So, if we were to look at Yahoo (Figure 3), for instance, we would notice that the server reaches its saturation point at about the time when the client issues 36 requests per second for uncompressed content. This figure falls slightly, to about 33 requests per second, when compressed content is requested.

Table 2
Estimated saturation points for the Apache web server based on repeated client requests for the indicated document

WebSite	UnCompressed	Compressed
Google	215	105
Yahoo	36	33
AOL	27	25
EBay	16	15

Refer to Table 2 for a comparison of the estimated saturation points for each test case. These estimates were obtained by calculating the average number of connections per second handled by the server using data available from the benchmarking results. One interesting thing to note from the graphs is that, aside from the Google page, the server maintained almost the same average reply rate for a page up until the saturation point, regardless of whether the content was being served compressed or uncompressed.

After the saturation point the numbers diverge slightly, as is noticeable in the graphs. What this means is that the server was able to serve almost the same number of requests per second for both compressed and uncompressed documents.

The Google test case shows it is beneficial to impose a limit on the minimum file size necessary to compress a document. Both mod_gzip and IIS allow the site administrator to set a lower and upper bound on the size of compressible resources. Thus if the size of a resource falls outside of these bounds it will be sent uncompressed.

For these tests all such bounds were disabled, which caused all resources to be compressed regardless of their size. When calculating results we will only look at those cases where the demanded number of requests per second is less than or equal to the saturation point. Not surprisingly, compression greatly reduced the network bandwidth required for server replies. The factor by which network bandwidth was reduced roughly corresponds to the compression ratio of the document. Next I have discussed the performance effects that on-the-fly compression imposed on the server. To do so we will compare the server's average response time in serving the compressed and uncompressed document. The findings are summarized in Table 3.

Table 3
Average response time (in milliseconds) for the Apache server to respond to requests for compressed and uncompressed static

WebSite	UnCompressed	Compressed
Google	3.2	10.2
Yahoo	3.3	27.5
AOL	3.4	34.7
EBay	3.4	51.4

The results are not particularly surprising. We can see that the size of a static document does not affect response time when it is requested in an uncompressed form. In the case of compression, however, we can see that as the file size of the resource increases so too does the average response time. We would certainly expect to see such results because it takes a slightly longer time to compress larger documents. Keep in mind that the time to compress a document will likely be far smaller for a faster, more powerful computer. The machine running as the web server for these tests has a modest amount of computing power, especially when compared to the speed of today's average web server.

7 Summary / Suggestions

If the server generates a large amount of dynamic content one must consider whether the server can handle the additional processing costs of on-the-fly compression while still maintaining acceptable performance. Thus it must be determined whether the price of a few extra CPU cycles per request is an acceptable trade-off for reduced network bandwidth. Also, compression currently comes at the price of cacheability.

Much Internet content is already compressed, such as GIF and JPEG images and streaming audio and video. However, a large portion of the Internet is text based and is currently being transferred uncompressed. As we have seen, HTTP compression is an underutilized feature on the web today. This despite the fact that support for compression is built into most modern web browsers and servers. Furthermore, the fact that most browsers running in the Windows environment perform streaming decompression of gzipped content is beneficial because a client receiving a compressed HTML file can decompress the file as new packets of data arrive rather than having to wait for the entire object to be retrieved. Our tests indicated that 27% byte reductions are possible for the average web site, proving the practicality of HTTP compression. However, in order for HTTP compression to gain popularity a few things need to occur.

First, the design of a new patent free algorithm that is tailored specifically towards compressing web documents, such as HTML and CSS, could be helpful. After all, gzip and deflate are simply general purpose compression schemes and do not take into account the content type of the input stream. Therefore, an algorithm that, for instance, has a predefined library

of common HTML tags could provide a much higher compression ratio than gzip or deflate.
Secondly, expanded support for compressed transfer coding is essential. Currently, support for this feature is scarce in most browsers, proxies and servers. As far as proxies are concerned, Squid appears only to support compressed content coding, but not transfer coding, in its current version. According to the Squid development project web site [18] a beta version of a patch was developed to extend Squid to handle transfer coding. However, the patch has not been updated recently and the status of this particular project is listed as idle and in need of developers. Also, in our research we found no evidence of support for compressed transfer coding in Apache.
The most important thing in regards to HTTP compression, in my opinion, is the need for expanded proxy support. As of now compression comes at the price of uncacheability in most instances. As we saw, outside of the latest version of Squid, little proxy support exists for the Vary header. So, even though a given resource may be compressible by a large factor, this effectiveness is negated if the server has to constantly retransmit this compressed document to clients who should have otherwise been served by a proxy cache.

References

[1] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP.In Proceedings of the ACM SIGCOMM '97 Symposium, Cannes, France, Sept 1997.

[2] B. Krishnamurthy and J. Rexford. Web Protocols and Practice. Addison-Wesley, May 2001.

[3] H.F. Nielsen, J. Gettys, A. Baird-Smith, E.Prud'hommeaux, H. W. Lie, and C. Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In Proceedings of ACM SIGCOMM'97 Conference, September1997.http://www.w3.org/Protocols/HTTP/Performance/Pipeline.html

[4] UnxSoft Ltd. WebSwift: Real Time HTML Compression.http://www.unxsoft.com/webswift/. 2001.

[5] D. Mosberger and T. Jin. httperf: a Tootl for Measuring Web Server Performance.http://www.hpl.hp.com/personal/David_Mosberger/httperf.html. Jun 29, 1998.

[6] H.F. Nielsen. The Effect of HTML Compression on a LAN. http://www.w3.org/Protocols/HTTP/Performance/Compression/LAN.html. Dec 6,2000.

[7] G. Goebel. Introduction / Lossless Data Compression.http://www.vectorsite.net/ttdcmp1.html.May 01, 2001.

[8] Google, Inc. The Google Search Engine.http://www.google.com/

[9] Yahoo! The Yahoo! Home Page.http://www.yahoo.com/

[10] America Online. The America OnlineHome Page. http://www.aol.com/.

[11] eBay. The eBay Home Page.http://www.ebay.com/. 2001.

[12] T.J. Midgley. Autobench.http://www.xenoclast.org/autobench/. Jun 27, 2001.

[13] Gzip manual page.http://www.linuxprinting.org/man/gzip.1.html. 2001.

[14] Packeteer,Inc.InternetApplication Acceleration using Packeteer'sAppCeleraICX. http://www.packeteer.com/PDF_files/ appcelera/icx/icx55_paper.pdf. 2001.

[15] Top9.com.Top99WebSites.http://www.top9.com/top99s/top99_web_sites.html.Mar2001

[16] GNU Project. The wget home page. http://www.gnu.org/software/wget/wget.html. Dec 11, 1999.

[17] Leknor.com. Code – gziped?http://leknor.com/code/gziped.php. 2001.

[18] S. R. van den Berg. Squid Development Projects. http://devel.squidcache.org/projects.html. Jan 6, 2002.