Compression Proxy Server: Design and Implementation

Chi-Hung Chi, Jing Deng, Yan-Hong Lim
School of Computing
National University of Singapore
Singapore 119260
Email: chich@comp.nus.edu.sg

Abstract

Automatic data compression in the web proxy server is an important mechanism that can potentially reduce network bandwidth consumption and web access latency significantly. However, unlike traditional data compression, web protocols and data have unique characteristics that make compression challenging. These include data block streaming, wide range of data object sizes and types, and real-time response. In this paper, we focus on automatic web data compression in the HTTP proxy server. A new classification of web data compression based on system complexity and HTTP requirements is proposed: stream, block and file compression. Then, the concept of hybrid web data compression is introduced. To understand the potentials of web data compression better, an implementation of the proposed hybrid compression in the Squid proxy server is described. The result is very promising, as about 30% of the bandwidth can be saved easily. Furthermore, even with a low end Pentium 266 MHz PC as the proxy machine, the compression overhead is less than 1% of the transfer time.

1. Introduction

With the popularity of the Internet and the world wide web from academia to home and entertainment, network bandwidth has already become a scarce, valuable resource and an important obstacle to WWW surfing. Networks are getting more congested with an increasing amount of multimedia data, thus resulting in slower web response time and the comment of "World Wide Wait". To address this WWW bandwidth problem, one good approach is to compress web multimedia data on the network.

Compression of web multimedia data can be achieved in one of two ways: explicit and implicit. In the explicit case, it is the responsibility of the web data owner to store the compressed version of the data in the web server. Since HTTP 1.1 provides a compression data MIME type [12], any client browser that supports HTTP 1.1 can view the compressed data automatically. For network management, this is the simplest approach, because compression is completely transparent to it. However, this causes inconvenience to content management because any maintenance or update of web information must perform explicit data decompression. More importantly, unless the compression format is a standard type supported by HTTP, implicit collaboration between the web client and server for compression might have a certain degree of difficulty. The existence of a tremendous amount of uncompressed web data also raises the demand for automatic web data compression. Finally, non-standard and older versions of browsers might not support automatic decompression.

Under the implicit model, it is the web server and proxy server that will handle web data compression automatically. The price for this shifting, however, is a new technology and system architecture design that can overcome the inherited problems of web data compression such as data block streaming, the wide range of data object sizes and types, and real-time response. Just like the situation in the operating system and the hard-disk storage, automatic web data compression is possible, but is not as trivial as it appears.

In this paper, we will investigate the system architecture design and support for automatic web data compression in the HTTP proxy server. Due to the unique environment of the web, new requirements and constraints for web data compression will be identified first. In particular, the implications of data block streaming of web information will be investigated. Then, hybrid data compression will be proposed. This hybrid compression will be implemented in the Squid proxy server architecture for the evaluation of bandwidth reduction and the overhead incurred. The result is very promising, as about 30% of the bandwidth can be saved easily. Furthermore, even with a low end Pentium 266 MHz PC as the proxy machine, the compression overhead is less than 1% of the transfer time. This is important because compression is completely transparent to the web surfers and can work co-operatively with other bandwidth saving mechanisms.

2. Constraints & Implications of Network Compression

In the WWW environment, there are unique characteristics of web information and constraints of the proxy and web servers that make automatic networked multimedia data compression challenging. Some of the major ones are listed below and their implications on web data compression are discussed.

Data Object Sizes

The size distribution of web objects ranges from a few hundred bytes to tens/hundreds of thousand bytes. More importantly, due to the practice of web page design, there is a significant portion of the web objects whose sizes are quite small. This makes web data compression challenging. It is because traditional data compression algorithms are much less effective towards small sized files. It is also found that the occurrence frequency of repeated character strings in a web object (such as an HTML file) might not be very high (because of the limited size). One the other hand, this frequency turns out to be very high across multiple web objects. A good example is HTML script marks. The HTML language syntax has a very limited set of character strings and they appear in almost every web page. Hence, it will be beneficial to have a static table that will map the frequently used HTML strings into variable size tokens.

Data Object Types

Another unique characteristic of web objects is the wide range of data types. There are simple HTML text files, application/octet-streams, gif/jpeg images, avi/asf/mpeg videos, JAVA applets, executables, etc.. With different data encoding formats and information entropy, the compressibility of these object types varies significantly. Furthermore, some object types might allow lossy compression (e.g. images) while the others insist on lossless compression (e.g. HTML files). This feature makes a single universal data compressor difficult to find (if at all possible). As a result, hybrid data compression, with both lossy and lossless compressors, is likely be the direction to handle such a wide variety of object types.

Data Block Streaming

This is one of the toughest constraints that the web imposes on networked multimedia data compression. According to the current HTTP protocols, data is streamed to the client in blocks. This is to improve the client's perception of the web object retrieval time and to reduce system resource consumption. Data block streaming implies that the compression/decompression process can only work on those blocks that are in the buffer space of the HTTP proxy (not necessarily the entire object). As a result, for those data compressors that require multiple passes on the object content, extra effort will be required to save the data being streamed in the server before the (de-)compression process can take place. Although single-pass data compressors are available, their compression effectiveness is not as good as those of multi-pass ones.

3. Classification of Web Data Compression

In this section, we propose a new classification for web data compression that is defined in terms of the system resource requirements and data block streaming in the current HTTP protocol. There are basically three classes of compression approaches for the web:

[1] *Whole File Compression*
Under this class, the compression needs to work on the entire file of a web object. There is no compression/decompression on the partial object data.

[2] *Data Block Compression*
This class of compression processes the information that is stored in the HTTP proxy buffer. Since a web object is made up of one or more data blocks, compression on a web object implies individual compression and decompression processes on data blocks that appear in the proxy buffer.

[3] *Data Stream Compression*
As its name implies, this class of compression treats data as a continuing stream. Whenever a proxy server receives some data, it will compress/decompress the data immediately and the result will then be passed to the next level of the network. There is no need to buffer data for future compression/decompression.

Based on the different stages of triggering of the compression process, each of these three approaches have different implications on the complexity of the proxy system and the compressor. This is given in Table 1. From this table, we see that data block compression is more suitable for web multimedia data in the HTTP proxy than either the stream or whole file compression approaches. Under this approach, additional data buffering is minimal, as the proxy also needs to keep the data of the currently transferred blocks in the buffer anyway. The maximum data size stored in the buffering space implies that data does not need to be stored on disk; thus no additional disk operation is expected. The extra access delay time, as perceived by the web surfer, is also not important because this delay time is defined by the fixed buffer size.

	Stream	Block	Whole File
Additional Memory Buffering Req.	No	No	Resource to hold entire web object
Additional Disk Operations	No	Not expected	Yes
Partial Web Data Transfer	Same	In size of proxy buffer	No
Implementation Complexity	Slightly increased	Most complicated	Increased
Compression Efficiency	Lowest	Close of that of whole file compression	Highest
Choice of Compression Algorithms	Only single pass	All kinds	All kinds

Table 1: Comparison of Three Web Compression Types (With Respect to the Current Proxy Requirements)

As far as compression algorithms are concerned, data block streaming can incorporate all kinds of compressors, including multi-pass ones. The tradeoff, however, is the complexity of the proxy server architecture. Among these three classes, data block compression interferes with the proxy data flow the most, resulting in the most complicated system structure. There will also be limitations to the applicability of multi-pass compressors for web data: a multi-pass algorithm can only work on the object that can fit in the proxy buffer. If the object size is greater than the buffer size, either one of these two situations will happen. For web data objects whose blocks can be compressed independently (e.g. `text` files), performance will be lost slightly. It is because a large file is now divided into smaller segments, each of which is compressed independently. However, for those objects whose blocks cannot be compressed independently (e.g. `jpeg`), data block compression will not work.

4. Algorithm Selection

In Section 2, we mentioned that it is important to have hybrid compressors, each of which is optimized for one predefined type of data object. In this section, we would like to propose a sample selection of data compressors for the three compression approaches mentioned above. This will help in the testing and evaluation of the compression proxy potentials. *Note that this is just a reasonable set of combinations and may not be optimal. There is no intention in this paper to define any optimal algorithms.*

Let us look at each object type and see what kind of compression can be performed under the three approaches of web data compression:

<u>`gif` *Objects*</u>

It is important to compress `gif` objects because about one third of the web bandwidth consumption is due to `gif`. To compress these objects, we propose to use the GIF-to-JPEG transformer with 25% lossy factor. Lossy compression is used here because most `gif` objects are for decoration purposes and can tolerate some degree of loss. Furthermore, the loss in the image quality due to compression is very small to be noticed by web surfers. Due to the multi-pass nature of the `gif-to-jpeg` transformer, it cannot be used in data stream compression. Furthermore, for data block compression, there are two additional requirements:

- If the `gif` object size is greater than the proxy buffer size, no `gif-to-jpeg` compression will be performed. This is because the transformer cannot work on partial images.

- If the `gif` object size is less than 4 KBytes, no `gif-to-jpeg` compression will be performed. Table 2 shows the compression ratio of the `gif-to-jpeg` transformer with 25% lossy factor. From this table, we see that the `gif-to-jpeg` transformer actually expands a `gif` file if the file size is between 0 and 4 KBytes.

`gif` Object Size Range	**Compression Ratio**
0 – 1 K	0.3322
1 – 4 K	0.7891
4 – 8 K	1.3136
8 – 16 K	1.9825
16 – 32 K	2.9885
32 – 64 K	5.6460
64 – 128 K	9.9781
> 128 k	18.6050

Table 2: Compression Ratio of `gif-to-jpeg`

<u>`text/octet-stream` *Objects*</u>

The choice of compressors for these objects is relatively simple because many text compressors can work very well on them. The only complication is the complexity of the compression algorithm: whether multi-pass algorithms should be allowed. For data block compression, we use the ZLIB library [5]; for whole file compression, we suggest using GZIP. They are some of the most common compressors for text. For stream compression, LZW is used instead because it is a single pass compressor. Furthermore, for HTML objects, mapping of the HTML script marks to variable size tokens is performed with the help of a predefined table.

`jpeg` *Objects*

Unlike `gif` objects, no transcoding compression is done on `jpeg` objects for all the three types of compression. There are a few reasons for this. Firstly, the `jpeg` objects are already in compressed format. It will be relatively difficult to compress the objects further without losing image quality substantially. Secondly, it is observed that on the web, `jpeg` objects are used much less often for decoration than `gif` objects are; hence losing image quality for further `jpeg` transcoding might result in user dissatisfaction. Thirdly, the algorithm to change from normal `jpeg` to progressive `jpeg` is a multi-pass algorithm. Since the size of a `jpeg` object is usually larger than the size of the HTTP proxy buffer, it will not have a significant effect on data block compression. Note that just like the `gif-to-jpeg` compression, `jpeg` transcoding is possible as long as the user is willing to trade off the image quality for bandwidth.

Others

For objects other than those mentioned above, no compression will be performed. It is because the percentages of their bandwidth consumption are not high and their optimal compressors are unknown.

5. Proxy-To-Proxy Compression in Squid

To get a better understanding of the design feasibility of automatic web data compression in the proxy server, we are going to describe an implementation of the block compression mechanism between two Squid proxy servers in this section. In this implementation exercise, the Squid proxy version 2.1 [23] is chosen to be the basic platform for experimentation. In Squid, data is transferred block by block. Whenever a Squid proxy receives data from its upper web server or proxy level, it will send the data to the client/proxy at the next network level as soon as possible. To make our discussion easier, the following terms are used. The "*compression proxy*" is the proxy server that does the compression of web data and then sends it to the decompression proxy server. The "*decompression proxy*" is the proxy server that receives the compressed data from the compression proxy and decompresses it before it sends the data to its client.

5.1. Implementation Considerations

During the implementation of data block compression in the Squid proxy server, there are at least three design issues that need to be handled properly.

5.1.1. Encoding of Compression Messages

In the implementation of proxy data compression, it is extremely important to handle the handshaking mechanism between two proxy servers properly. On one end, the decompression proxy needs to insert a message into the request header to notify the compression proxy that it has the decompression capability. The compression proxy will also need to write a message in the reply header to indicate that the entity body is compressed. Furthermore, both proxies need to delete these additional messages in the request and reply headers before they pass the request to the next level of the network. According to the HTTP/1.1 protocol [12], we propose the following solution:

- The decompression proxy adds a "`Transfer-Encoding: Block-Decoding`" field in the request header to notify the compression proxy what compression method(s) it supports.

- The compression proxy sends a "`Transfer-Encoding: Block-Encoding`" field in the reply header to notify the decompression proxy what compression method it uses to compress the data.

One important set of parameters that needs to be communicated between the two proxy servers is the compressed block size and the uncompressed block size. In the block compression on Squid, data will be compressed in blocks; thus the decompression proxy server will not receive the same number of bytes as in the original uncompressed block. Even worse, it is possible that the decompression proxy buffer might receive more than one block of data at a time, or a whole compressed data set is received in multiple blocks. To solve this problem, we propose to add a four bytes "block header" to each data set of compressed blocks in the proxy buffer. The first two bytes of the header record the size of the compressed set while the last two bytes record the original (uncompressed) data set size.

With this information, the decompression proxy will use the first two bytes to separate each compressed data set from the whole trunk of the received data. It will also use the last two bytes of the header for decompression. Furthermore, if the last two bytes record "0", it will mean that the data block is not compressed. This happens either when the compression proxy cannot perform compression on that data type or it is not beneficial to perform compression on the given data (as we mentioned previously in Section 4). With the two bytes mechanism to record the data sizes, the maximum working set size is bounded by 64 KBytes. As will be shown later, the recommended buffer size for data block compression is 32 Kbytes. Hence, the two bytes mechanism should be sufficient in most practical implementations. In the case where a really large block size is used, more bytes can be allocated to hold this information.

Another problem in the hand-shaking mechanism is the length of the transfer file. The web server records the length of the transfer file in the reply header field "Content-Length". Once the proxy performs compression or decompression, the actual length of the file will definitely be affected. To handle this situation, our decompression proxy will not check the incoming file length with the "Content-Length" field. That is, even though the compression proxy server transfers an object with file length less than the value of the "Content-Length" field, no error will be reported.

5.1.2. Memory Allocation

To call the "compress()" and "uncompress()" functions in the proxy server, some working memory will be required. Furthermore, the decompression proxy also needs memory to keep the compressed data block. All these can be handled by the group of memory management functions provided by Squid; we use the xmalloc() and xfree() functions to allocate and free memory in the compression process.

5.1.3. Data Structure

To complete the support for automatic web proxy compression, the compression status information of a web request needs to be reflected in its data structure inside the proxy server. In Squid, the request data and the reply data of a request are linked together through the data structure "_HttpStateData". It records all the necessary information about the request and the reply. This structure is generated when the proxy processes a request and it will be used during the entire request and reply processes. In the implementation of our compression proxy, we added a field "need_compress" in the "_HttpStateData" structure to specify that this reply can be compressed. We also added another field "can_compress" in the _StoreEntry structure to indicate that if reply data type is actually in the compressed format. On the decompression proxy side, the "_StoreEntry" structure records the information of the reply. It is linked to the structure "_HttpStateData". Another variable, "can_decompress" is also added as the compression flag in this structure. If the decompression proxy finds that the incoming data is in compressed format, the flag will be set.

5.2. Workflow of Squid Compression Proxy

In this section, we would like to put together the actual workflow path of a web request in the Squid compression proxy environment. This not only helps us to understand the complete design of the Squid compression proxy server, but it also allows us to appreciate the design decisions and to see how various techniques fit together in a single system. The modified workflow for Squid compression proxy is summarized below.

Workflow in the Squid Decompression Proxy

Step A1:
When the Squid decompression proxy receives a request from a client, it will add a "Transfer-Encoding: Block-Decoding" field in the request header.
(Afterwards, it will wait for data to come back from the compression proxy).

Step A2:
When the decompression proxy receives the reply data from the compression proxy, it will analyze the reply

header. If it finds that a "`Transfer-Encoding: Block-Encoding`" field is set and the "`Content-Type`" matches its decompressor, it will set the variable "`StoreEntry->can_decompress`".

Step A3:
When the "`StoreEntry->can_decompress`" field is set, the decompression proxy will read the compressed and the uncompressed set sizes from the "`block header`". If the incoming data set size is greater than or equal to the compressed block size, the decompression proxy will extract one compressed block, then call the function "`uncompress()`", and afterwards go back to step *A3*; otherwise the data will be saved in `StoreEntry->keep_buf`.

Step A4:
If the `can_decompress` field is set, the decompression proxy will erase the "`Transfer-Encoding: Block-Encoding`" field from the reply header.

Workflow in the Squid Compression Proxy

Step B1:
If the request header has a "`Transfer-Encoding: Block-Decoding`" field, the compression proxy will set the variable "`_HttpStateData->need_compress`". Then, it erases the "`Transfer-Encoding: Block-Decoding`" field from the request header.

Step B2:
The compression proxy checks the reply header field "`Content-Type`". If this field matches with the proxy's supported compression data type(s), both the "`HttpStateData->need_compress`" and "`StoreEntry->can_compress`" variables will be set.

Step B3:
When the "`StoreEntry->can_compress`" field is set, the compression proxy will call the function "`compress()`". It also writes the set size of the compressed and the uncompressed blocks in the header of the data block.

Step B4:
It adds the "`Transfer-Encoding: Block-Encoding`" field in the reply header and will then send the compressed data block to the decompression proxy.

6. Experimental Results

To measure the effectiveness of web multimedia data compression in the HTTP proxy, we repeated the web surfing pattern of the proxy trace that we collected in a junior college in Singapore. The proxy server used was the Squid proxy on a low-end DEC Alpha workstation. The proxy trace log was collected for about one year, and the standard proxy information was recorded. The workload of the proxy was about 5,000 to 20,000 requests per day and the school had a leased line of 128Kbps

Two sets of experiment were conducted with the same sequence of web object references:

- *client browser ⇔ original SQUID proxy ⇔ original SQUID proxy ⇔Web server*
- *client browser ⇔decompression SQUID proxy ⇔ compression SQUID proxy ⇔Web server*

Each of the four proxy servers (either with or without compression) was Squid version 2.1 with 30 Mbytes of allocated memory and 200 Mbytes of cache space. The proxy server was run on a dual Pentium II 266 MHz PC with 64 MBytes memory and 128 MBytes swap space and the operating system was LINUX. For each set of experiments, both proxy servers were run on the same machine. We did this because the overhead of compression and decompression could be estimated better by avoiding any data transfer between the two testing proxy servers in the public network. In our experiments, we used one day's web access sequence, which consists of about 10,000 requests. The access latency of the web object and the bandwidth consumption were measured. The overhead of compression/decompression in the proxy can be estimated by comparing the time consumed by the two sets of experiments.

Before we discuss the performance of the three different compression approaches, it will be helpful to have statistical data on the web objects that pass through the proxy server.

6.1. Distribution of Web Object Types

Web objects have a wide variety of data types. A data type distribution of WWW objects provides hints on what kind of compression algorithms should be supported and how the proxy compression system should be structured and optimized. We collected this statistic by extracting the "`file type`" and "`file` size" fields of entries in the proxy trace. The result is shown in Figure 1. The percentage is defined in terms of the bytes transferred instead of the number of objects requested.

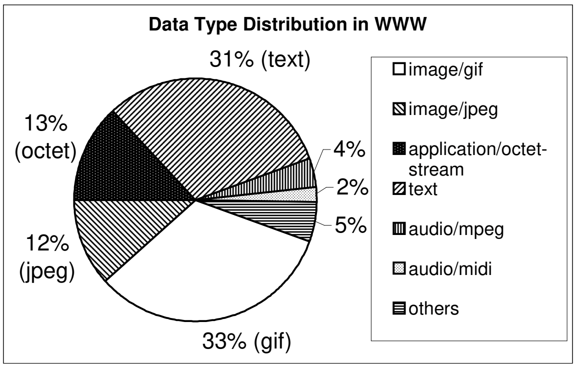

Figure 1: Distribution of Web Object Types (in terms of the Bytes Transferred)

From the figure, we see that the main data types of web objects are: `gif` data (33%), all kinds of `text` data (31%), `octet-stream` data (13%), and `jpeg` image data (12%).

6.2. Distribution of Web Object Sizes

Understanding the distribution of web object sizes is important to estimate the overall benefits of web data compression. Given a web object type, the size distribution can indicate how effective the data compression will be. Usually, small objects are much more difficult to compress than large ones. Due to data block streaming in the web environment, the statistic also shows how often a web object can fit into the HTTP proxy buffer.

File Size in Bytes	Gif	text	octet
0 – 1 K	3.77%	1.32%	0.04%
1 – 4 K	15.25%	7.75%	0.66%
4 – 8 K	20.92%	16.96%	0.69%
8 – 16 K	28.89%	30.15%	1.14%
16 – 32 K	14.12%	28.75%	0.89%
32 – 64 K	8.44%	9.30%	1.53%
64 – 128 K	5.00%	2.79%	2.40%
> 128 K	3.60%	2.99%	92.66%

Table 3: File Size Distribution of `gif`, `text`, and `octet-stream` (in terms of Bytes Transferred)

Table 3 shows the distribution of the amount of data retrieved under an object size range for a given object type (again, not the number of files). It shows that for `text` objects, most of the data are clustered in the range of 4 KBytes to 32 KBytes. This is a good range for most text compressors to have reasonable performance. The size distribution of the `gif` objects is quite similar to that of `text` objects except that there is a higher percentage of data with file sizes under 4 KBytes. As was discussed in the previous section, no aggressive transformation or compression will be done on this group of objects - the objects will be expanded upon compression. Data distribution of the `octet-stream` objects is quite different from the rest. Most of the objects are larger than 128 KBytes.

6.3. Compression Effectiveness

The results of compression effectiveness of the proposed approaches are given in Figures 2-4. The x-axis is the compression types: stream, block with various buffer sizes, and file. The results are extremely encouraging and illustrate the potentials of web data compression.

Figures 2(a)-(d) show the bandwidth saved by the three web data compression approaches. Overall, the saving from whole file compression is still the highest, with an average of about 37%. This is expected because the whole file is available for compression. With a reasonable HTTP proxy buffer size, the performance of data block compression is actually very impressive. For a block size of 32 KBytes, the bandwidth reduction is 32%; and with a 64 KBytes block size, the reduction is 34.39%. These are only a few percent lower than the upper limit. Moreover, the incremental performance gain in the bandwidth reduction starts to level off when the block size is greater than 32 KBytes. For data stream compression, about 14.66% of bandwidth reduction can be expected. This is quite good; however, it is only about half of what can be obtained from data block compression with 32 KBytes block size. This justifies our argument that data block compression is worth the increase in system design complexity.

Comparing Figure 2(a), 2(b) and 2(c), we see that `text` objects are the most compressible ones. Usually, 50% to 70% of the bandwidth consumption due to `text` objects can be saved. Also, it is interesting to notice that there is practically no difference in bandwidth saving for buffer sizes larger than 4KBytes. This further supports our choice for data block compression. For `gif` objects, the situation is completely different. With buffer size less than 4 KBytes, there is no bandwidth saving. This is expected because no compression takes place; compression only increases the object size. With buffer size greater than 4 KBytes, the bandwidth reduction increases quite rapidly with the buffer size and then levels off after 64 KBytes. Note that although the compression ratio of `gif` to `jpeg` is very high, the bandwidth reduction is still bounded by about 43%. This is because there exists a set of `gif` objects (with size less than 4 KBytes) that will not benefit by the `gif-to-jpeg` transformation. The contribution of the `octet-stream` compression to the bandwidth reduction is the least. This is probably due to the lesser compressibility of the `octet-stream` objects.

Figure 3 shows the average time required to perform compression and decompression in the Squid proxy server. It shows that the compression overhead is actually quite small, with an average of about 5 milli-seconds per KBytes. Note that there is a drop in compression/decompression time for octet objects from data stream compression to 1 Kbytes data block compression. Careful investigation shows that this is due to the change of compressors from LZW to ZLIB. Furthermore, by the nature of the ZLIB algorithm, its decompression time is smaller than its compression time. Similar situation happens to text objects from 64 KBytes data block compression to whole file compression. Despite the similar cascading concept used by ZLIB and GZIP, the implementation of GZIP is more efficient than that of ZLIB.

Figure 4 shows the ratio of the compression time overhead to the original access latency (before compression). The overhead is indeed very small; the average is only about 0.6% of the access latency. With the consideration of a 30% reduction in the network bandwidth, the overall web access latency will definitely improve. Furthermore, the proxy machine used in the experiment is an outdated one. With a reasonable machine configuration for the proxy server, the compression overhead will be reduced further.

Finally, we estimated the improvement of web access latency due to data compression. The testing was done on a network segment that covered two local internet service providers and their internet exchange gateway. The result is shown in Figure 5. Stream data compression improves the web access latency by about 10%, which is quite good. With block data compression, the improvement goes up to about 25%-30%. This is expected because more data can be compressed and more effective two-pass compressors can now be used. However, for whole file compression, the improvement drops back to about 18%. This is mainly due to the buffering of the whole object for compression before it is sent to the decompression proxy server.

7. Related Work

Data compression [14,17,21] is a fundamental area of research in information theory. Traditional system design (including the operating system) and disk controllers provide a good foundation for supporting automatic data compression [3,6,7].

Despite the long history of data compression, the application of data compression to the web environment is relatively new. Nielsen et al. [18] investigated deflate data compression [8] from the viewpoint of a web server. Both Mogul et al. [16] and Velasco et al. [24] realized the potential benefits of compression in the HTTP proxy by compressing the web objects with text compressors such as GZIP. Santos et al. [22] looked at mechanisms to suppress the transfer of replicated data in the web environment. While their studies gave a good foundation for web data compression, there was no discussion on the constraints of the web environment, the design considerations of the compression proxy, and the implementation issues.

In the wireless mobile environment, there are research efforts to reduce the network bandwidth requirements for mobile devices and PDAs [25]. One representative project is the GloMop from Berkeley [9,10]. It achieved better end-to-end performance and higher quality display output for low-end clients through dynamic distillation. However, the data streaming nature of the WWW data cannot be handled by their techniques. Another approach to address the WWW bandwidth problem is to use delta encoding for the web response. When a cache object is outdated in the local proxy, only the delta of the change in content will be transferred from the server to the proxy. This idea was proposed in [2,11,13,26] and [16] later verified its potentials with realistic traces. The idea is good, but it only works on objects that are outdated in cache. This limitation is important to network bandwidth reduction because over half of web objects are referenced only once [15]. In network caching, researchers address the network bandwidth problem by keeping those web objects that are expected to be reused in the local memory or cache [1,4]. However, the network proxy cache cannot reduce the network bandwidth consumption for first time object accesses. To address this issue, the idea of prefetching [19,20] is proposed. The main difficulty in web prefetching is to have a very high prefetch accuracy; the spatial property of web objects is not well defined and many web objects are referenced only once in cache [15].

8. Conclusion

In this paper, we investigated web data compression as the mechanism to solve the Internet bandwidth problem. Based on the unique features of the web protocol and data, we proposed a new classification for web data compression: stream, block, and file. We argued that data block compression fits into the web environment best because it handles the streaming of web data properly and allows multi-pass compressors to work on web data without explicit file storage. Then we proposed the hybrid web compression mechanism and implemented it on the Squid proxy server to test out its feasibility and to evaluate its performance. The result is very impressive; about 30% of the network bandwidth can be saved and the compression and decompression overhead is less than 1% of the web access latency (even on an outdated PC proxy server). The result is important because the compression process is completely transparent to the web surfers and it can work co-operatively with other bandwidth saving mechanisms.

Acknowledgements

The authors would like to thank P. Krishnan and Jessica Kornblum who helped to improve the final version of this paper. We would also like to thank the anonymous referees for their valuable comments on earlier drafts of this paper.

Reference

[1] Abrams, M., Standridge, C.R., Abdulla, G., Williams, S., "Caching Proxies: Limitations and Potentials," *Proceedings of the Fourth International World Wide Web Conference*, Boston, U.S.A., December 1995.

[2] Banga, G., Douglis, F., Rabinovich, M., "Optimistic Deltas for WWW Latency Reduction," *Proceedings of the 1997 USENIX Annual Technical Conference*, Anaheim, California, USA , January 1997.

[3] Burrows, M., Jerian, C., Lampson, B., Mann, T., "On-line Data Compression in a Log-Structured File System," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 2-9.

[4] Chankhunthod, A., Danzig, P., Neerdaels, C., Schwartz, M.F., Worrell, K.J., "A Hierarchical Internet Object Cache," *Proceedings of the 1996 USENIX Annual Technical Conference*, January 1996.

[5] Deutsch, P., Gailly, J., "ZLIB Compressed Data Format Specification Version 3.3", RFC 1950, May 1996.

[6] Douglis, F., "On the Role of Compression in Distributed Systems," *Proceedings of the Fifth ACM SIGOPS European Workshop*, Mont St.-Michel, France, September 1992.

[7] Douglis, F., "The Compression Cache: Using On-line Compression to Extend Physical Memory," *Proceedings of 1993 Winter USENIX Conference*, San Diego, CA, January 1993, pp. 519-529.

[8] Deutsch, P., "DEFLATE Compression Data Format Specification version 1.3," RFC 1951, Aladdin Enterprises, May 1996.

[9] Fox, A., Brewer, E.A., "Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation," *Proceedings of the Fifth International WWW Conference*, May 1996.

[10] Fox, A., Gribble, S., Brewer, E.A, "Adapting to Network and Client Variation via On-Demand Dynamic Transcoding," *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996, pp. 160-170.

[11] Housel, B.C., Lindquist, D.B., "WebExpress: A System for Optimizing Web Browsing in a Wireless Environment," *Proceedings of the ACM Second Annual International Conference on Mobile Computing and Networking*, Rye, NY., November 1996, pp. 108-116.

[12] HTTP 1.1 Protocol. Request for Comments: 2616, June 1999.

[13] Hunt, J., Vo, K.P., Tichy, W., "An Empirical Study of Delta Algorithms," *Proceedings of the IEEE Software Configurations and Maintenance Workshop*, Berlin, March 1996.

[14] Lelewer, D., Jacobson, V., "Data Compression," *ACM Computing Surveys*, 19(3), 1987, pp. 261-296.

[15] Lim, S.N., "Live Range Modelling of WWW References," *M.Sc. Thesis, School of Computing, National University of Singapore*, 1999.

[16] Mogul, J.C., Douglis, F., Feldmann, A., Krishnamurthy, B., "Potential benefits of delta-encoding and data compression for HTTP," *Proceedings of the ACM SIGCOMM '97 Conference,* September 1997.

[17] Nelson, M., Gailly, J.L., *The Data Compression Book*, M&T Books, 1996.

[18] Nielsen, H.F., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H.W., Lilley, C., "Network Performance Effects of HTTP/1.1, CSS1, and PNG," 1997. http://www.w3.org/Protocols/HTTP/Performance/pipeline.html.

[19] Padmanabhan, V.N., Mogul, J.C., "Improving HTTP Latency," *Computer Networks and ISDN Systems*, 28(1-2), December 1995, pp. 25-35.

[20] Padmanabhan, V.N., Mogul, J.C., "Using Predictive Prefetching to Improving World Wide Web Latency," *Computer Communication Reviews*, 26(3), 1996, pp. 22-36.

[21] Salomon, D., *Data Compression: the Complete Reference*, Springer Press, 1997.

[22] Santos, J., Wetherall, D., "Increasing Effective Link Bandwidth by Suppressing Replicated Data," *Proceedings of the USENIX 1998 Annual Technical Conference*, June 1998.

[23] Squid Web Site.
http://Squid.nlanr.net

[24] Velasco, J.R., Velasco, L.A., "Benefits of Compression in HTTP Applied to Caching Architectures," *Proceedings of the Third International WWW Caching Workshop*, 1998. http://wwwcache.ja.net/events/workshop/32/manchester.html.

[25] Wachsberg, S., Kunz, T.,. Wong, J., "Fast World Wide Web Browsing over Low Bandwidth Links," June 1996. Available as http://ccnga.uwaterloo.ca/~sbwachsb/paper.html.

[26] William, S., Abrams, M., Standridge, C.R., Abdulla, G., Fox, E.A., "Removal Policies in Network Caches for World-Wide-Web Documents," *Proceedings of the ACM SIGCOMM '96 Conference*, August 1996.

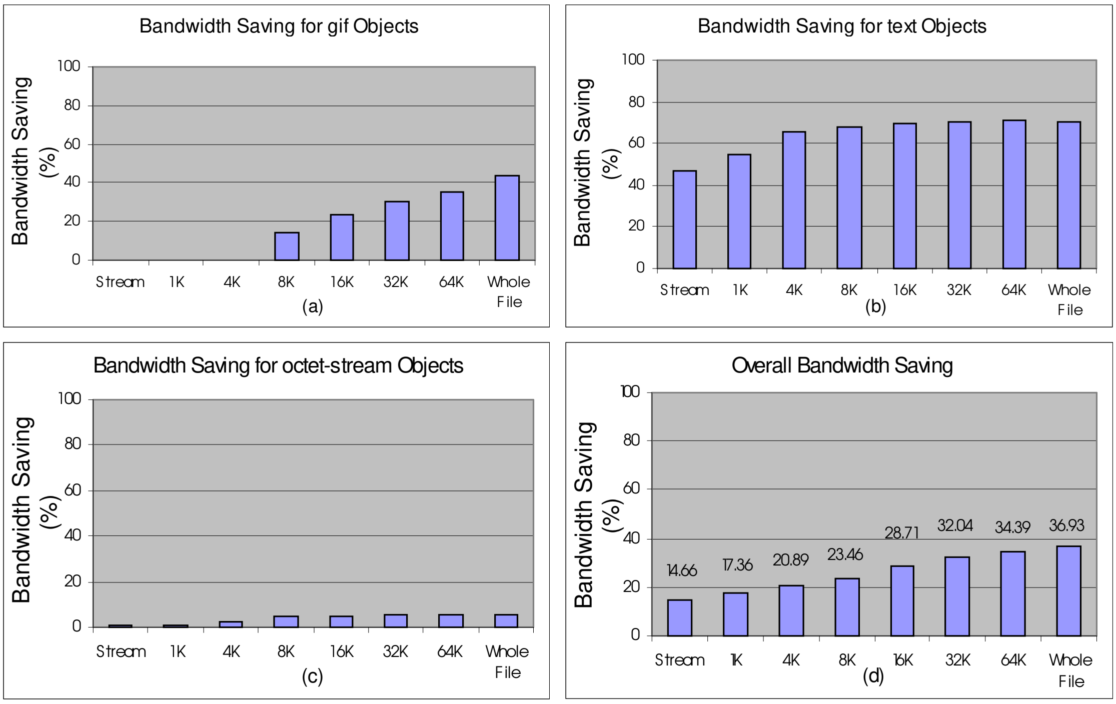

Figure 2: Bandwidth Saving By Web Compression
(x-axis: stream, block with various buffer sizes, file)

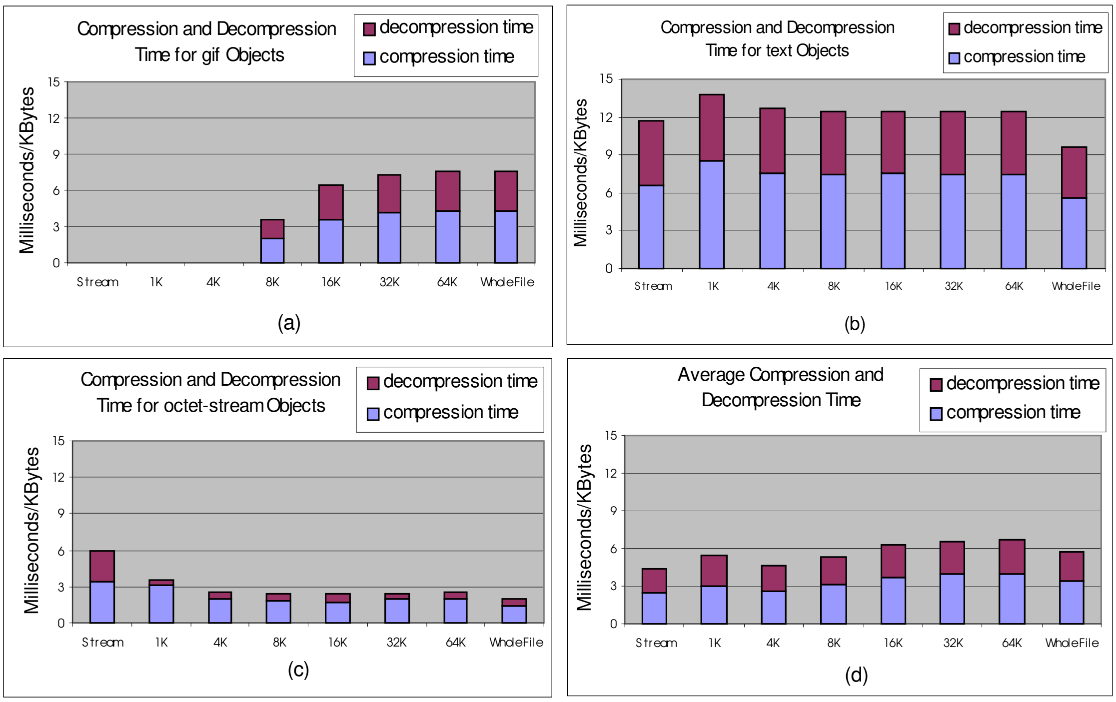

Figure 3: Compression and Decompression Overhead
(x-axis: stream, block with various buffer sizes, file)

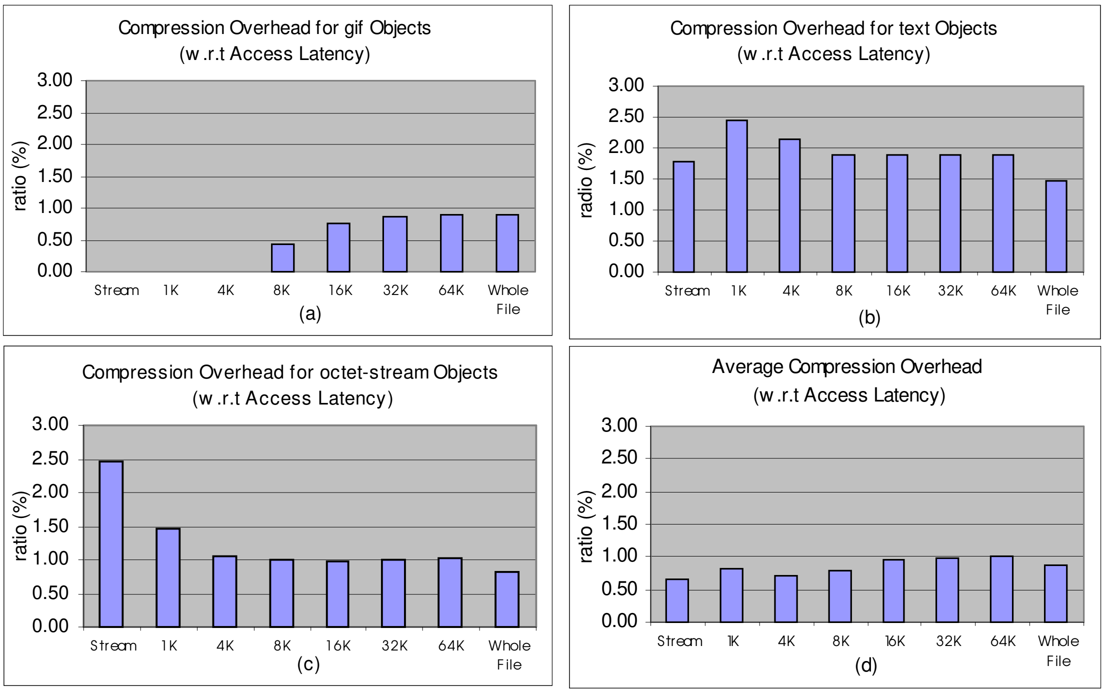

Figure 4: Normalized Web Compression Overhead
(with respect to the Access Latency)
(x-axis: stream, block with various buffer sizes, file)

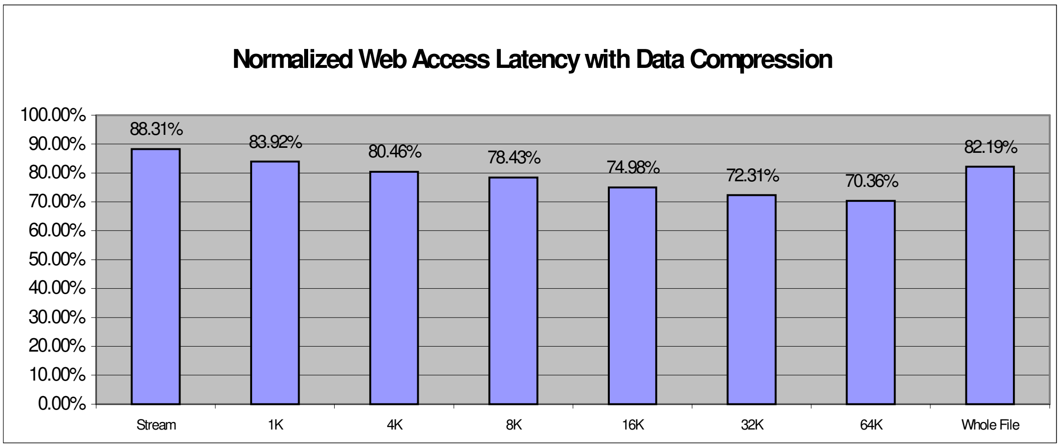

Figure 5: Normalized Web Access Latency with Web Data Compression
(with respect to the Access Latency without Compression)
(x-axis: stream, block with various buffer sizes, file)