Delta Encoding in a Compressed Domain

Shmuel T. Klein and Moti Meir

Department of Computer Science
Bar Ilan University
Ramat Gan 52900, Israel
Tel: (972–3) 531 8865 Fax: (972–3) 736 0498
tomi@cs.biu.ac.il, moti.meir@gmail.com

Abstract. A delta compression algorithm is presented, working on an LZSS compressed reference file and an uncompressed version, and producing a delta file that can be used to reconstruct the version file directly in its compressed form. This has applications to accelerate data flow in network environments.

1 Introduction

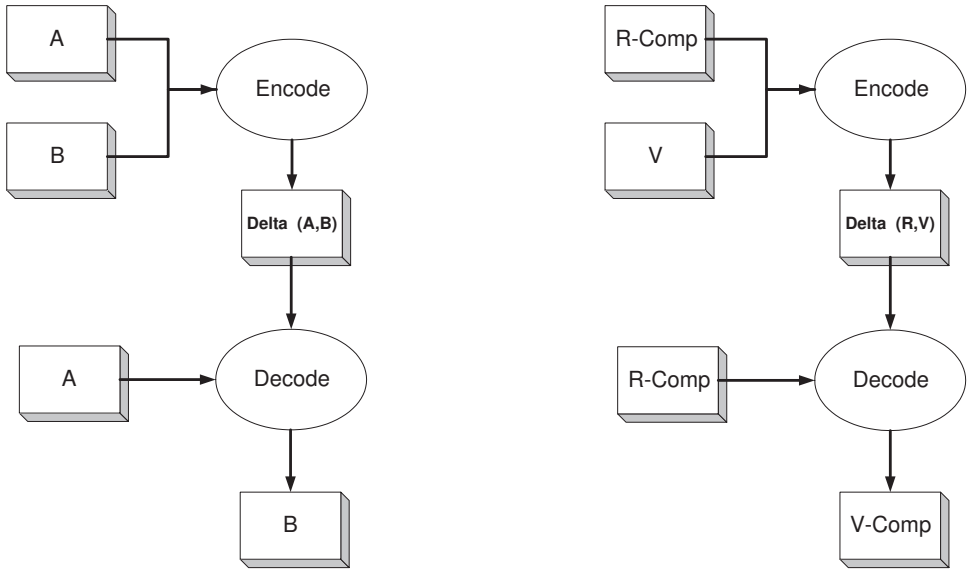
This paper presents an algorithmic approach to work with highly correlated files in the compressed domain. The idea is to allow small changes to be reflected upon a reference file each time a newer file version becomes available. This ability is highly required by caching, versioning and additive backup systems.

The standard delta compression scheme [3,1,4,9,2] takes two files and outputs the difference between those files. Such a scheme tries to output a small amount of data which represents the difference between the two files in their uncompressed state. As a result, using the standard delta file scheme, one can reconstruct a version file by using a reference file and the delta file (see Figure 1.A). Our scheme, however, encodes a delta file that reflects the differences between the files in their *compressed state*. That is, using the delta file and the compressed reference file, the decoder outputs the *compressed* version file (See Figure 1.B), contrarily to standard delta schemes that would output a version file in uncompressed form. Our scheme uses an uncompressed version file and a compressed reference file as the inputs to the encoder, a scenario defined as *semi-compressed delta encoding* in [7], in contrast with the *fully-compressed* alternative treated in [6].

We provide a conceptual solution considering the fact that textual data is produced in uncompressed form and also by examination of the entire network route between the encoding part (usually a server) and the decoding part (usually a client), which includes intermediate network elements. The scheme considers the fact that most of the network's intermediate elements are indifferent to the data content and only wish to store and forward the most recent copy of the data. For such elements, having the ability to alter their cached copy without decompressing it first as needed when a regular delta encoding is used, presents a great advantage. Standard schemes producing uncompressed version files would require an encoding phase, that is, compressing the file in order to save storage space and network bandwidth. This imposes a penalty in terms of both CPU utilization and temporary storage space, which can be saved by directly dealing with a compressed output.

The proposed encoding algorithm has two inputs: the compressed reference file R_c, and the uncompressed version file V. The output of the encoder is a delta file Δ, which, together with R_c, is the input to the decoding algorithm that outputs the compressed version file V_c.

Shmuel T. Klein, Moti Meir: Delta Encoding in a Compressed Domain, pp. 55–64.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2



1.A 1.B

Figure 1. Schematic representation of delta encoding

The efficiency of our scheme is based on several assumptions regarding both version and reference files:

1. The files are highly correlated
2. The changes are local and sparse
3. The changes are very small compared to the size of V.

The algorithm is an extension of LZSS [5] encoding and uses an ordered hash table adapted from [10], to store previous occurrences of substrings. The parameters which control the encoding are:

– Window size — the length of the sliding window. This attribute defines the maximum valid size of the jumpbacks and thereby the maximum "memory" size of the hash table.
– Minimum Match — determines the minimum number of characters required to match, in order to create a back pointer to the previous text.
– Synch Chunk — the maximum number of characters that are affected by a single change. That is, the length of all changes are smaller than this number.

The output of the encoder is a set of `COPY`, `ADD`, `UPDATE` and `SPLIT` commands and a set of characters stored in Δ, acting as a set of control commands to a decoder for changing R_c. These commands are described in more detail in Section 4 below. The decoder uses Δ and R_c to build an updated compressed file equivalent to V_c. This is done without decoding R_c but rather by changing it while it is still compressed (see Figure 1.B)

Let $V_{cr}[i, j]$ be the substring of V_c with index in the real uncompressed form, that is, the indices i and j refer to the indices of V in their uncompressed form. For example, if the 100 first bytes of V are compressed to the first 20 elements of V_c, then we have $V_c[1, 20] = Compress(V[1, 100]) = V_{cr}[1, 100]$. This provides a reverse mapping between the compressed domain and the uncompressed domain locations. We shall use the same notation when referring to just one character of V or R, that is $V_{cr}[50] = i$ is the location in V_c that corresponds to character $V[50]$.

2 The Encoding Algorithm

2.1 Overview

The algorithm encodes V and iteratively compares the results to R_c. During the processing, a sequence of characters and back pointers is generated, and checked for matches with R_c. If a mismatch is detected — denote the location of this *local mismatch point* by LMP — the original LZSS algorithm would output a file that will greatly differ from R_c. The result would be two encoded files, which were highly correlated when uncompressed, and when compared in the compressed domain loose their high resemblance. The current encoding fixes this problem by synchronizing both files. As a consequence there will be a *local synchronization point* (LSP) after which the output will match exactly the reference file, up to the next LMP. We assume here that the distance from LSP to LMP is at least Synch Chunk, otherwise the two changes are considered as one.

Several types of mismatches need to be addressed:

- one element is a back pointer while the other is a character;
- both elements are characters, but different ones;
- both elements are back pointers, but their copy length differs;
- both elements are back pointers, but their jump back length differs.

Maintaining a *Local Reconstructed Buffer* (LRB) in combination with the assumption of a limited change results in the ability to track the change. At each step, the algorithm checks whether a substitution, insertion or deletion of characters can explain the change, and continues according to the results by locating the LSPs in the uncompressed domain. When these points are found, the hash table is updated accordingly.

For example, consider the case of inserting a line of text into the version file. The LRB created by the reference instructions and the version data will match at a point in the version text which is beyond the inserted change. Since the change is assumed to be relatively small, the number of characters inserted is found by running a loop up to *synch chunk size*, trying to find this substring in the version file.

In order to compare the new version with the reference, we must be able to reconstruct the original substrings which might be represented by pointers in the reference file. This has led to the need of decoding the reference file in order to reconstruct the original data in the *change area* of V [7], which is the string starting one element prior to the actual change and ending at most Synch Chunk characters after the end of the change. The idea of running from right to left in R_c from elements prior to LMP, collecting the needed reference data characters, does not work well in most cases due to the fact that a back pointer in the encoded file can point to another back pointer, and so on, creating a chain. In addition to being expensive, this right to left decoding is not local. Since we want to maintain a local approach which greatly decreases memory needs, the semi compressed domain is exploited by using the reference characters and pointers to reconstruct the original data in the change area.

There is, however, one exception to the above. Since we use elements from R_c to reconstruct the reference data using the version data, care has to be taken in the case of self-pointers, i.e., when the copy *length* is larger than the *offset*. Indeed, the possibility of self-pointers is one of the major features of LZ77 schemes like LZSS, enabling the compression of variable-length repetitive strings; for example, a string

of 50 a's can be compressed as a[offset = 1, length = 49]. The solution is to use the already reconstructed buffer as a reference for self pointers.

2.2 Substitution

This is the simplest case, e.g., a date field has been updated in the version file. The following steps are executed, referring to Figure 2:

1. find the actual change size by comparing characters from the LRB and V, bounded by the Synch Chunk parameter;
2. insert a new quarantine zone $[K, K+i]$ to the mismatch list;
3. output the relevant commands (split and update pointers) to Δ;
4. advance the R_c index to point to the location of the LSP, and also advance the version index to the same LSP.

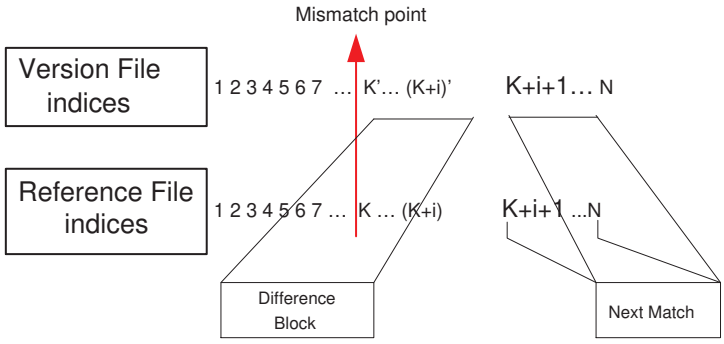


Figure 2. Schematic representation of substitution

The algorithm works because skipping over the change in both files and backwards updating V to the pre-change state, brings us to a point (LSP) where we just need to synchronize the hash tables. After this synchronization, the encoding process is the same, hence the output of the encoder are `COPY` and `UPDATE` commands up to the next mismatch. As we can see in Figure 2, jumping in both files as close as possible to index $K+i+1$ will bring us to the LSP. The reason that one does not always get exactly to $K+i+1$ is that the exact pointers from R_c are used and they are not split. However, breaking the back pointers to get to the exact spot in V is feasible and could in certain cases result in better compression.

2.3 Insertion

An inserted string may dramatically change the original LZSS output, as shown in Figure 3. V is scanned for an occurrence of the content of LRB inside the defined limited area, defined to start from $K-j$ up to $K+i$, where i is the Synch Chunk size and j is the maximum change size. The simplest way to deal with the insertion is to adjoin the inserted block to Δ with a single `ADD` and one `ADJUST` command. One could also consider using the hash table to add the newly inserted text as a sequence of pointers, which might improve compression.

In any case, a set of adjustment commands needs to be added to correct the back pointers of R_c. Each back pointer which points to the index in V prior to LMP has to be increased by the length of the insertion. If the increased length exceeds the maximum defined, the back pointer is split and the exceeding part is inserted as individual characters.

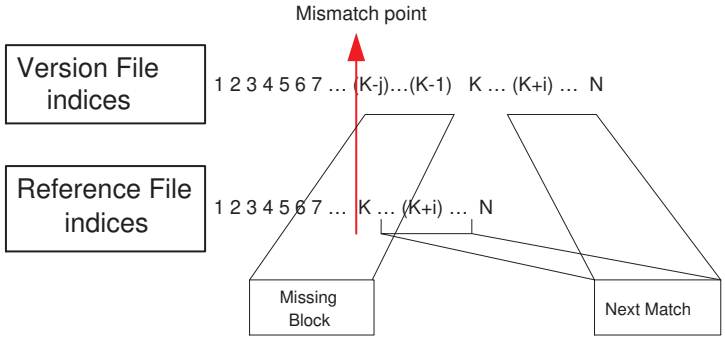


Figure 3. Schematic representation of insertion

2.4 Deletion

Deletion is similar to insertion, but here the LRB is scanned for an occurrence of a substring of V. The steps are, referring to Figure 4:

1. LRB $\longleftarrow$ $decode(R_c[K, K+i], V)$
2. search for $V[K+j, K+i]$ in LRB;
3. output to Δ a `COPY` command up to LMP; if LMP lies within a string compressed by a pointer, this pointer has to be split or some prefix of this string needs to be inserted by an `ADD` command;
4. output to Δ an `ADJUST` command for $[K, K+i]$ of R_c to reduce the relevant offsets by the length of the deleted string.

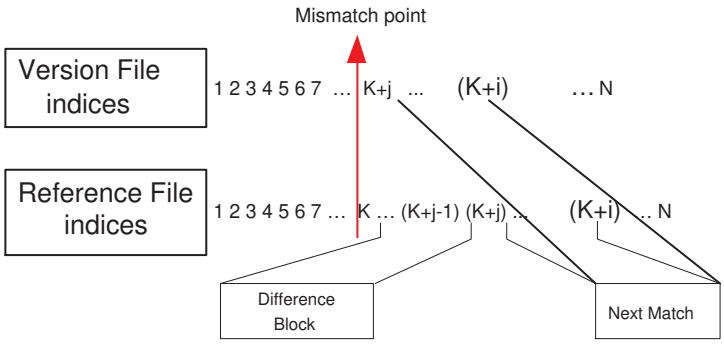


Figure 4. Schematic representation of deletion

3 The Decoding Algorithm

The descriptive nature of Δ is used to apply change commands to R_c, thereby transforming it into a file which is equivalent to V_c. The decoding process is linear in time and storage but is slightly more complex than the original LZSS decoding. This is due to the fact that pointers need to be adjusted along the way as data is inserted to or deleted from V_c. Moreover, the chain breaking mechanism might result in pointer splitting, as discussed above.

The algorithm is as follows, with the SPLITTO3 command explained below:

```
while Δ commands exist do:
    if COPY command
        copy substring of R_c to V_c
    if ADD command
        insert string into V_c
    if ADJUST pointers [i, j]
        do, up to window size from j + 1
            add the change size to the jump value of the pointers that
              point to a location preceding i;
            if the pointer is invalidated, issue a SPLITTO3 command
        end do
    end if
end while
```

As can be seen, the decoder runs in linear time and does not require additional memory. The algorithm performs a few more passes on parts of the data, but assuming the input is large relative to the window size, these passes are negligible. Further, if we consider a worst case scenario of needing to act upon many pointers pointing to the changed area, the pointers are bounded by the *offset* bits allocated for the pointer which bounds the entire procedure to part of the data. We assume that this is a small part compared to the input size. Again, if the changes are frequent in the input, the result will be a larger delta file and will require much work in the decoder. In such a case, it might be better to send the entire compressed version file and start fresh.

4 The Delta File

The delta file encapsulates commands which instruct the decoder how to convert its old compressed reference into a new compressed version. By applying the commands of the delta file, the decoding algorithm outputs the compressed version file with very low computational needs. The delta file is constructed such that the decoding complexity will be as small as possible and most of the computational effort is done by the encoder. This policy was chosen in order to be consistent with the *hop by hop* scheme to which this new compression suits. Reducing the complexity of the decoder in both time and storage allows small or computationally weak caching devices to be one of the hops along the way. Also, since the changes are the result of changes in the server side, the server can do most of the work for all the hops along the path to the clients. This will allow better utilization of the network nodes along the path between the server and the clients including the clients themselves.

Most of the previous work on delta files refers to an uncompressed domain, for example VCDIFF [8]. In our case, one needs in addition to the ADD, COPY and RUN commands of VCDIFF also means to split a pointer in order to insert changes. The new SPLITTO3 command breaks a pointer into three parts: the (possibly empty) prefix represents the pointer to the data portion which did not change; the middle part represents the change to be inserted and the (again possibly empty) suffix points to the representation of the remaining data. This methodology allows us to break the pointers of the compressed file and perform chain breaking with very little effort by the decoder.

In order to maintain this idea of lowering the computational demands of the decoder, we write all needed commands to Δ such that the decoder algorithm will not need to trace the restriction zones. This might lead to a larger delta file, but the size penalty is reasonable when the complexity needed by the hops is lower by doing most of the chain breaking at the encoder side.

Summarizing, the complete delta file command set consists of:

- `COPY` — similar to the copy of `VCDIFF`, it tells the decoder to copy a substring from the reference file to the decoded file.

- `ADDP` — Adds a pointer to V_c. This command is similar to `VCDIFF`'s `ADD` command, but adds a pointer, not characters.

- `ADDS` — Adds a string. Instructs the decoder to embed the command's parameter string into V_c in its current index. This command is similar to other delta file encoding `ADD` commands.

- `SPLITTO3` — the basic pointer breaking technique when a change which is in the middle of an area covered by a pointer is encountered. The decoder has to replace the changed pointer so that the result will be a reflection of the change in the compressed output. We split the pointer into three parts, the prefix part up to the change, the inserted part, which can be some string or a pointer to an early appearance, and a suffix part covering the remainder. The `SPLITTO3` command is also used when the encoder gets to a backpointer which points to a restricted zone. Since restricted zones need to be ignored as they represent invalid data, the command is used to break these pointers. The encoding algorithm stores each restricted zone and detects the first pointers that point to a restricted zone, then splits them. This way, we apply chain breaking, since, if we have P_1 pointing to a point in a restricted zone, and P_2 pointing to P_1, then by fixing P_1 we also take care of P_2 and the rest of the pointers that are connected to them. Also, this way we remove the decoder's need to trace pointers to restricted zones, hence simplifying the decoding algorithm.

- `ADJUSTJP` — Instructs the decoder to adjust all the offset sizes of the pointers, starting from a given start index up to a window size in R_c.

The size of the delta file is a major factor in the proposed scheme. Imagine a case where the compressed version is similar in size to the delta file. This overabundance of data is not needed since we could have sent V_c instead of the delta file. Sending V_c results is consuming less resources since there is no need for decoding along the path in each hop. Therefore, the rule is that if $|\Delta| \ll |V_c|$, send Δ. Otherwise, it is better to send V_c.

The following is a small sample of Δ in its textual form. In practice we used a binary code in order to encode the commands. Further, when dealing with very large files, in which the delta file is also large enough, Δ itself can be compressed in order to further reduce its size.

```
COPY [from = 0, to = 77]
SPLITTO3 [offset = 114, length = 113]
    PrefixPointer = [offset = 114, length = 103]
    Change = ['2'] // substitute one character
    SuffixPointer = [offset = 9, length = 9]
COPY [from = 81, to = 180]
```

The sample above represents a Δ of changing a single character in a file V consisting of 2864 1-byte numbers. In this example, a single number 1 has been changed into the number 2. The original file was compressed using LZSS to become R_c of size 180 bytes. The decoder algorithm manipulates R_c using the above Δ to output V_c.

5 Experiments

Table 1 summarizes the benefits of using the proposed algorithm. Real life HTML files have been used, e.g., www.cnn.com, with uncompressed size of 107 KByte, text files such as RFC's and a set of synthetic files, including many repetitions. The change type has been restricted in the tests to substitutions (S) and insertions (I), as deletions behave symmetrically to insertions. The results are compared to the binary uncompressed delta encoding scheme XDelta, which uses zlib to compress its delta file that requires a decompression phase before it can be used to reconstruct the uncompressed version. Our scheme takes into consideration that network elements are indifferent to the content of the data and thus prefer to maintain cached files in their compressed form and not to decompress them for running the delta encoding algorithm or recompress them after reconstruction. The results are compared to re-encoding the new version data, since this scenario represents a reference implementation of compression based transactions between a client and a server, for both types of regular recompression and the proposed delta encoding.

File	Number of Changes	Change type	Change size (bits)	GZip (bits)	CDDelta (bits)	Xdelta (bits)
CNN	0	No change	0	168472	67	1160
CNN	1	S	8	169944	145	1688
CNN	1	I	128	170024	265	1808
CNN	2	S	48	168496	255	1784
CNN	2	I	1608	168744	1912	2408
CNN	3	I	1872	168744	2254	2592
CNN	4	I	2144	168912	2652	2864
CNN	5	I	2408	169088	3042	3032
CNN	6	I	2672	169120	3384	3144
synthetic	1	S	8	1960	201	1712
synthetic	1	I	24	2136	217	1800
synthetic	1	S	72	2016	257	1728

Table 1: *Experimental results*

The first row of Table 1 describes the case of no change in the version file. Even though this case looks odd at first, it is a real life case. When considering the network elements that cache the data from a server, combined with the HTTP cache control commands, we see that many data objects, including textual HTML files, have an

expiration time. This expiration time causes the caching element to ask the server if the data was modified (the `if-modified-since` HTTP attribute) and in some cases, depending on both the network element and the server type, it causes the network element to ask for a new version. In this typical case, our scheme will always output a 67 bit command which is to copy the entire reference file.

The encoder performance has been tested on an Intel's core 2 due processor 2.4 GHz, and achieved a throughput of 36 MByte/sec on one CPU. This performance is similar to our LZSS encoder implementation. A 3 bit field was used to represent the command, and 32 bits for an index in the file, such as the from field of the `COPY` command. When referring to offsets, as in the `ADJUSTP` command, 16 bits were used, since the window size was up to 64 Kbytes.

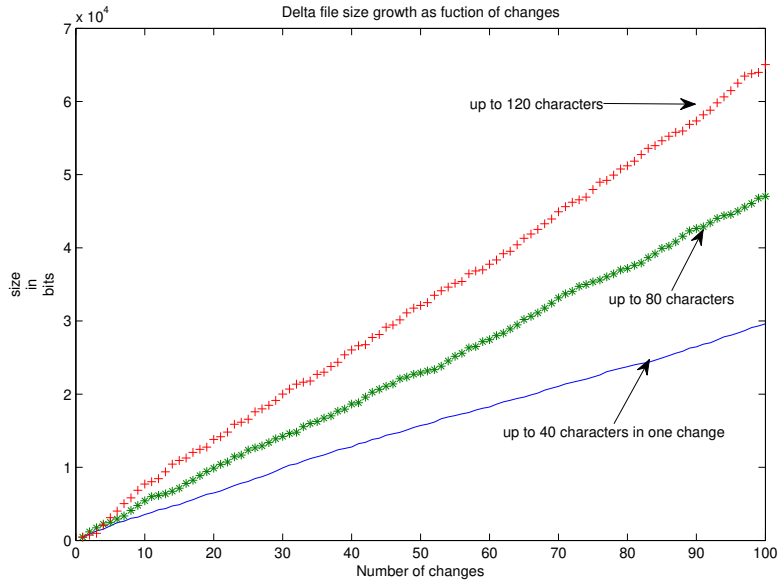


Figure 5. Size of delta file as function of the number of changes

From Table 1 we can see that on this data, a factor of about 46 is gained when compared to the compressed size of the file. The Delta file size increases when there are more changes. However, changes that were tested reflect a change of 7.4 lines with an average size of 45 characters per line. It must be noted that these results are without compression of the delta file and with no compression of the changed data relative to itself or relative to the entire file as a reference.

Figure 5 shows the size increase in bits of the delta file as a function of the number of changes, where each such change affects a randomly chosen set of 40, 80 and 120 characters. We can see that the size of the delta file increases linearly with the number of changes.

Figure 6 visualizes the data included in Table 1 concerning the size relation between a gzip compressed file (CNN) and the Delta file for our insertion tests. The x-axis gives the number of insertions, and the y -axis the size of the file on a logarithmic scale. We can see that the Delta file size increases with the number of inserted characters, while the insertion has only a negligible effect on the size of the new version gzip compressed file. The Delta file consists mostly of the inserted characters themselves.

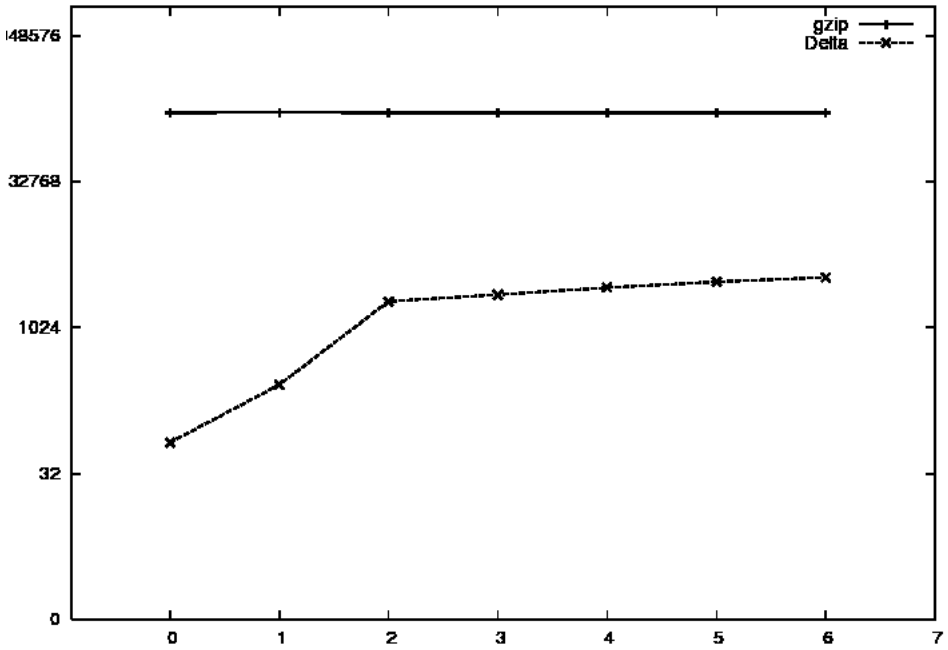


Figure 6. Comparing Gzip and Delta sizes

6 Conclusion

We focused on the semi-compressed delta encoding problem for LZSS encoded files, in an application in which the reconstructed version file is directly given in compressed form. This can greatly reduce network traffic and the CPU and storage requirements of the various network elements. The algorithm is based on a partial local reconstruction of a previous occurrence of the data, using the compressed reference and the new version.

References

1. M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer: *Compactly encoding unstructured input with differential compression.* Journal of the ACM, 49(3) 2002, pp. 318–367.
2. B. B. C. Xiao and G. Chang: *Delta compression for fast wireless internet download.* IEEE GLOBECOM Proceedings, 2005, pp. 474–478.
3. M. Chan and T. Woo: *Cache-based compaction: A new technique for optimizing web transfer.* Proceedings of the IEEE Infocom '99 Conference, 1999, pp. 117–125.
4. J. Hunt, K. Vo, and W. Tichy: *Delta algorithms: An empirical analysis.* ACM Trans. on Software Eng. and Methodology, 7(2) 1998, pp. 192–214.
5. J. Storer and T. Syzmanski: *Data compression via textual substitution.* Journal of the ACM, 29 1982, pp. 928–951.
6. S. Klein, T. Serebro, and D. Shapira: *Modeling delta encoding of compressed files.* International Journal of the Foundations of Computer Science, 19 2008, pp. 137–146.
7. S. Klein and D. Shapira: *Compressed delta encoding for LZSS encoded files.* Proceedings of Data Compression Conference, DCC–07, 2007, pp. 113–122.
8. D. Korn, J. MacDonald, J. Mogul, and K. Vo: *The VCDIFF Generic Differencing and Compression Data Format.* RFC 3284 (Proposed Standard), June 2002.
9. G. Motta, J. Gustafson, and S. Chen: *Differencial compression of executable code.* Proceedings of Data Compression Conference, DCC–07, 2007, pp. 103–112.
10. R. Williams: *An extremly fast Ziv-Lempel data compression algorithm.* Proceedings of Data Compression Conference, DCC–91, 1991, pp. 362–371.