Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

zdelta: An Efficient Delta Compression Tool

Dimitre Trendafilov Nasir Memon Torsten Suel

Department of Computer and Information Science

Technical Report
TR-CIS-2002-02
12/26/2002

zdelta: An Efficient Delta Compression Tool *

Dimitre Trendafilov *Nasir Memon* *Torsten Suel*

CIS Department
Polytechnic University
Brooklyn, NY 11201

`dtrend01@utopia.poly.edu {memon,suel}@poly.edu`

Abstract

In this report we describe a tool for delta compression, i.e., the efficient encoding of a given data set in relation to another one. Its possible applications include archiving multiple versions of data, distribution of software updates, delta compression of backup files, or compression at the file system level. The compressor, called *zdelta*, could be viewed as a modification of the *zlib* compression library [4] with some additional ideas inspired by the *vdelta/vcdiff* tool of Vo [5]. We also present experimental results comparing *zdelta* to other delta compression tools.

Additional information about *zdelta*, including source code and updates, is available at `http://cis.poly.edu/zdelta/`.

*This project was supported by a grant from Intel Corporation. Torsten Suel was also supported by NSF CAREER Award NSF CCR-0093400.

1 Introduction

Compression in computer systems is used in order to reduce space requirements and/or I/O and network traffic. Most compression techniques are concerned with processing a single data set with a specific format. Delta compression on the other hand is concerned with compressing one data set, referred to as the *target data set*, in terms of another one, called the *reference data set*, by computing a *delta*. The delta can be viewed as an encoding of the difference between the target and the reference data set. Thus, the target data set can be later recovered from the delta and the reference data set. In this paper, the term *data set* denotes a sequence of bytes.

Delta compression found its initial application in revision control systems [8]. By storing deltas of different versions, instead of the actual data, these systems were able to reduce storage requirements significantly. Due to the ever increasing CPU speeds, many new applications can now benefit from delta compression. For example, delta compression can be used at the file system level; an example of a file system implemented with delta compression is the *Xdelta File System* (XDFS) of MacDonald [6]. Another application of delta compression is software distribution. This is relevant particularly for software distributed over the Internet. By distributing deltas, or essentially patches, one can significantly reduce network traffic. Delta compression could also be used to improve HTTP performance. By exploiting the similarity between different pages on a given website or between the different versions of a given web page, one can reduce the latency for web access [1, 2, 3]. The *zdelta* tool, presented in this paper, is fairly general and could be efficiently used in any of these applications.

The organization of this paper is as follows: Section 2 contains a brief survey of different delta techniques and of available delta compression tools. Section 3 provides a description of the *zdelta* architecture. Section 4 describes experiments and discusses their results. Section 5 discusses potential future improvements of *zdelta*.

2 Fundamentals

There are several techniques for computing a delta between two data sets. One of the earliest approaches was to look at the two data sets as two strings and to represent the targeted string as a sequence of insert, delete and update operations over the first string. There are obvious limitations of this method. For example, it will have poor performance if a single pattern in the reference data appears multiple times in the target string, or if certain patterns appear in the target data in a different order than in the reference data.

These limitations are resolved by the copy-based approach, which represents the targeted data set as combinations of copy operations from the reference data set. The well known *Lempel-Ziv* string compression algorithms [9] provide a natural extension to this approach. In particular, *LZ77* compresses a string by substituting its prefix with a reference to already compressed data. Thus, by treating the reference data as already compressed and performing *LZ77* on the target data, one can effectively build a delta of the two data sets.

Currently available delta compressors include *xdelta* and *vdelta* and its newer variant *vcdiff*. In this paper we present a new delta compression algorithm, which we call *zdelta*. All these compressors build the data difference using a copy-based approach. The three techniques, however, use different methods for finding the optimal set of copy instructions; they also have different ways of representing these copy instructions. The major difference in their performance, however,

comes from the different encoding schemes for the already found difference. *xdelta* builds simply a difference and does not encode it at all, *vcdiff* uses a byte-based encoding, and *zdelta* encodes the difference using Huffman codes.

3 The *zdelta* Algorithm and Implementation

As mentioned above, *zdelta* is built by modifying the *zlib* compression library. The main idea in *zdelta*, just as in *vdelta*/*vcdiff*, is to represent the target data as a combination of copies from the reference data and the already compressed target data. This representation is further Huffman encoded. Thus, the compression process could be divided into two parts: building a difference and encoding the difference. These two parts run concurrently; whenever the first process generates a certain amount of output it calls the second one. The output of the encoder is the final compressed data.

3.1 Building the Difference

In order to identify copies, *zdelta* maintains two hash tables - one for the reference data and one for the already compressed data. The two hash tables have exactly the same structure. The reference hash table is built in advance, at least for small reference files. (For larger files, a certain window of the reference file is in the hash table at any point in time.) The target hash table, just as in *zlib*, is built on the fly once the compression process starts. Hashing a substring is done on the basis of its first three characters. When building a hash table, an insertion is performed for each input character. In particular, each block of three consecutive characters (each 3-gram) is hashed and inserted into the corresponding hash table. As in *zlib*, hash table insertions are designed to be fast, and deletions are never done (except that the table is periodically flushed.)

A match is unambiguously represented by its length, its offset from one of a few *pointers*, the *pointer*, and the direction of the offset. The current *zdelta* implementation supports matches with length up to 1026 and offsets from 0 to 32766. If there is no match of length at least 3, then *zdelta* emits the first character as literal and attempts to find a match starting at the position of the second character. The *pointers* are simply positions within the target or reference data relative to which offsets are taken. If *zdelta* has a choice, it always uses the pointer that results in the smallest offset. Experiments revealed that *zdelta* has best performance using three reference pointers - one into the target data and two into the reference data. The pointer in the target data, similar to *zlib*, is implicit; it always points to the start of the string currently being compressed. The two pointers in the reference data, however, can point to any position within the reference buffer. In general *zdelta* attempts to predict the location of the next reference data match and to position a pointer close to it. The emitted offset can be viewed as an error correction of this prediction. The pointer movement is motivated by the following likely cases:

- If the data sets are similar, there is a high probability for the next match to be close to the location of the current one. In particular, the start of the next match is often close to the end of the current one. In this case we benefit if we move one of the pointers to the end of the previous match.
- Sometimes we get isolated matches, meaning there will be no other copies from the vicinity of the current match. In this case, if we have two pointers, we can keep one of them in the

vicinity of the previous match, while moving the other one to the end of the current isolated match.

Our pointer movement strategy based on this observation is as follows: If the current match is close to the previous one, say its offset is less than 256, then move the pointer used to specify the current match; otherwise, move the other pointer. Clearly, this strategy fails in the case of two or more consecutive isolated matches. Experiments showed, however, that increasing the number of pointers results in only minor savings for expressing the offset, but incurs more extra bits for specifying which pointer is used.

Due to the movement of the reference pointers we could get positive or negative offsets. *zdelta* solves this problem by storing an extra flag for the match direction. The matches within the target data always have a negative direction, of course.

When searching for matches, a greedy strategy is used. All possible matches in the reference and target data are examined and the best one is selected. Only matches with length between 3 and 1026 and offset smaller than 32766 are considered. A match m_1 is considered better than match m_2 if m_1 is longer than m_2 or if m_1 has the same length as m_2 but smaller offset. In addition, if the offset is fairly large, say larger than 4096, the match is penalized, by decreasing the match length used in the comparison by some constant (1 in the current *zdelta* implementation). The motivation for this is that a shorter but closer match will give better compression than a slightly longer but much farther match. When a match is found, it is not emitted immediately, but compared against the best match found at the next position; this idea is taken from *zlib*. The complete difference-building algorithm is as follows:

I. Preprocessing the Reference File:

For $i = 0$ to $len(f_{ref}) - 3$:

(a) Compute $h_i = h(f_{ref}[i, i + 2])$, the hash value of the first three characters starting from position i in f_{ref}.

(b) Insert a pointer to position i into hash bucket h_i of T_{ref}.

II. Computing the difference:

Initialize reference data pointers p_1, p_2 to zero
Set $j = 0$
Set $l_{prev} = 0$
While $j \leq len(f_{target})$:

(a) Compute $h_j = h(f_{target}[j, j + 2])$, the hash value of the first three characters starting from position j in f_{target}.

(b) Search hash bucket h_j in both T_{ref} and T_{target} to find "the best match".

(c) Insert a pointer to position j into hash bucket h_j of T_{target}.

(d) If the found match has length less or equal to l_{prev}, emit the previous match. Hash all substrings of $f_{target}[j+1, j+l_{prev}]$ and insert them in T_{target}. Increase j by l_{prev}; set $l_{prev} = 0$.

If the found match has length greater than l_{prev}, emit the literal $f_{target}[j-1]$. Set $j = j+1$; set l_{prev} to the current match length.

If there was no previous match, set l_{prev} to the length of the current match, set $j = j+1$; wait for the next iteration.

This implementation, although good in theory, is not always practical. One important issue arises when the two data sets are large. Hashing them completely would increase the memory requirements beyond acceptable levels. Large hash tables also require more time for searching. In order to overcome this problem, *zdelta* limits the data inserted into the hash tables to some fixed size, 64 KB in the current implementation. Thus, there are two fixed size windows, one in the reference data and another one in the target data, and *zdelta* is used to build a delta between these two windows. The target window is moved in exactly the same way as the trailing window in *zlib*. The movement of the reference window is determined by a simple heuristic that uses a weighted average of the positions of the most recently used matches to determine when to slide the window. The window is always moved forward by half its size. When the window is moved, the reference data hash table is flushed completely and then rebuilt again.

Another problem that could affect execution time is the occurrence of extremely large hash buckets. This could occur even when the size of the hashed data is limited. There are two reasons for this to happen: a bad hash function, or too many repetitions of a given pattern. We assume that the *zlib* hash function, which is reused in *zdelta*, distributes the hash values fairly well. To handle the second case, *zdelta* simply limits the maximum number of elements to be searched in a given hash bucket - this limit is set to 1024 in the current implementation.

There are alternative solutions to this problem based on the following observation: if a match of length N exists, then there could be up to $N-2$ different hash buckets containing pointers to some part of this match. (If the match has no repeating substrings, and there are no hash collisions between the substrings, then there will be exactly $N-2$ hash chains like this.) Thus, we could find the match by searching any one of those hash chains. In the example on Figure 3.1, the word "current" is to be compressed and a match exists in the reference data. This match can be located by traversing the hash chain given by $hash(cur)$. We can see that the other four hash chains, $hash(ren)$, $hash(urr)$, $hash(rre)$, $hash(ent)$, also contain pointers to the match. Therefore, we could locate the match by traversing the shortest of these chains.

Of course, in reality, we do not know the length of the match that will be found. However, we can adapt this approach as follows. We start searching the first hash chain, $hash(cur)$. If a match longer than this prefix is found, we try to switch to a shorter hash chain that contains a pointer to the extended prefix. We can repeat the process until we reach the end of a hash chain. Although this method could possibly decrease the number of traversed elements, the drawback is that we have to maintain the length of each hash chain and check for shorter hash chains. Experiments showed that this idea does not improve performance on the real data sets that we used, though it does improve slightly over our solution in worst-case scenarios.

Another approach is taken by *vcdiff*. Instead of jumping to shorter hash chains, *vcdiff* keeps its hash table as small as possible. When hashing a given string, *vcdiff* first searches for it. If it is

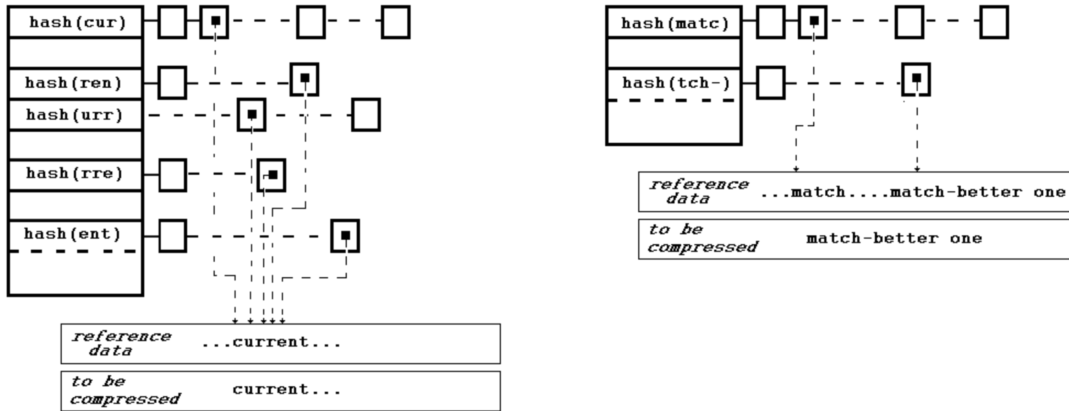

Figure 3.1: Maintainance and traversal of hash tables (left: *zlib*, right: *vcdiff*)

not found in the already hashed data, then it is inserted. Otherwise *vcdiff* inserts $h(Data[j + L - 3, j + L])$, where L is the length of the longest available match and j is the start of the string that we are processing (note that *vcdiff* uses a 4-character hash function). When searching for a match, *vcdiff* first finds an occurrence of a string with the same 4-byte prefix. Once a match of length L is found, if there is an even longer match, *vcdiff* can find it by traversing the $h(Data[j+L-3, j+L])$ hash chain.

An example is shown in the right half of Figure 3.1: when preprocessing the reference data, *vcdiff* inserts into its hash table the first occurrence of "matc", "atch", etc. Afterwards, when preprocessing the string "match-better one", *vcdiff* will try to find the longest possible match in the already hashed data, which in this case is "match". Thus, the compressor will insert no new hashes for "matc", and "atch", and will only insert "tch-" and the subsequent 4-grams. Suppose that subsequently *vcdiff* has to compress the string "match-better one" in the target data, and needs to find the occurrence of this string in the reference data. It will start by traversing the $hash(matc)$ chain and it will locate only the first occurence of "match". Then *vcdiff* will try find a longer match by traversing the $hash(tch-)$ chain and will locate the "match-better one" string in the reference data.

There were several reasons for not choosing this method in *zdelta*. Firstly, it complicated the code significantly, and eliminated most of the *zlib* match finding optimizations. The major reason, however, was that this approach guarantees only finding the first occurrence of the longest possible match. Therefore, it cannot be used for selecting the match with the smallest offset; this resulted in somewhat degraded compression in experiments.

3.2 Encoding the Differences

After the difference is built, it is encoded. Our *zdelta* relies on the Huffman coding facilities provided by *zlib*. The output of the differencing phase contains matches and literals. Matches are represented by length, pointer, offset and direction. As mentioned above, the match length could be up to 1026 characters. The *zlib* Huffman encoder, however, supports only codes for lengths from 0 to 255 characters. In order to encode greater lengths, *zdelta* expresses the length as $L = (l + 3) + 256 * c$, where l is then given to the Huffman encoder, and c is encoded elsewhere. Similar to *zlib*, *zdelta* encodes the offsets in one Huffman space, and the literals and the lengths in

c (length)	$-ptr_{target}$	$+ptr_{ref(1)}$	$-ptr_{ref(1)}$	$+ptr_{ref(2)}$	$-ptr_{ref(2)}$
0 (3-258)	code 1	code 2	code 3	code 4	code 5
1 (259-514)	code 6	code 7	code 8	code 8	code 10
2 (515-770)	code 11	code 12	code 13	code 14	code 15
3 (771-1026)	code 16	code 17	code 18	code 19	code 20

Table 3.1: *zdelta* flags

another one. In addition, we have a third code space for *zdelta flags*. These flags encode match pointers, offset direction, and the length coefficient c. There are 4 possible coefficients, 3 possible pointers and 2 possible directions. The target data pointer, of course, has always negative direction. Thus, there are a total 20 *zdelta flags* to be encoded, see Table 3.1.

3.3 *zdelta* and the *zlib* Compression Libraries

This subsection discusses the relationship between *zdelta* and the *zlib* compression libraries. It is for people familiar with the implementation of *zlib*.

As mentioned above, the LZ77-based techniques compress a given string through references to the already compressed data. The copy-based delta compressors are very similar, but use references to the already compressed data and to the reference data. Thus, we decided to implement *zdelta* by modifying the LZ77-based *zlib* library, in particular by adding code for supporting a reference data set, identifying matches in it, and maintaining a window into the reference data. This way we take advantage of the good high-level design and the efficient and well tested code of *zlib*.

In the first part of computing the delta (that is, building the difference between the two data sets), *zdelta* reuses the hash table implementation of *zlib*, and simply maintains an additional hash table for the reference data set. This hash table is initialized once in the beginning of the *deflate* process. It is later updated only if the window into the reference data is moved. The decision to move this window is based on the average, weighted by copy length, of the positions of the most recently performed copies. When this average moves too far into the second half of the current window, we move the window by half its width. Thus, whenever a match is emitted, *zdelta* updates the corresponding statistical data.

The code for identifying matches in the already compressed target data is identical to that in *zlib*. Since *zdelta* uses two pointers into the reference data set, the code for identifying matches there is slightly different as we may need to check the offset of a match with respect to both pointers. In the second part of the delta compression process, (that is, encoding the data difference) *zdelta* completely reuses the Huffman encoding facilities of *zlib*. The current *zdelta* implementation even reuses the default Huffman trees of *zlib* for the offset and copy length Huffman trees. The only difference is that *zdelta* needs an extra Huffman tree to encode the match pointer, direction, and length coefficient c.

Another important difference is that *zdelta* compression must be done in a single step. The library user provides both the target and reference file at once, and *zdelta* produces the output at once. This might be a problem on systems with very limited memory. For the target file, this is due

to our implementation and could be fixed with a little work; for example, *vcdiff* allows the user to feed the target file in over a period of time. However, the reference file needs to be supplied in advance in order to get good compression (though only the current window really needs to be in memory at the same time).

4 Results

This section presents a few experimental results comparing *zdelta* against *vcdiff*, *xdelta*, and *gzip*. The experiments were conducted on two different sets of files[1]:

1. The *gcc* and *emacs* data sets used in the performance study in [5], consisting of versions 2.7.0 and 2.7.1 of *gcc*, and 19.28 and 19.29 of *emacs*. The newer versions of *gcc* and *emacs* consist of 1002 and 1291 files, respectively.

2. A set of artificially created files that model the degree of similarity between two files. In particular, we created two completely random files f_0 and f_1 of fixed length, and then performed delta compression between f_0 and another file f_m created by a "blending" procedure that copies text from either f_0 and f_1 according to a simple Markov process. By varying the parameters of the process, we can create a sequence of files f_m with similarity ranging from 0 ($f_m = f_1$) to 1 ($f_m = f_0$) on a nonlinear scale[2].

Three different classes of experiments were performed. First, we observed the compression performance, achieved compression ratio, and compression/decompression speed on real data. The data for this experiment were the *gcc* and the *emacs* data sets. In the second experiment, we observed how the compression parameters are affected by the input data similarity. For this experiment, we used the artificial *morph* data sets; the input file size was fixed to 1 MB and the file similarity was varied. The last experiment was concerned with the relation between the input size and the compression performance. For this experiment, we again used the *morph* data set; this time the file similarity was fixed, and the file size was varied.

The experiments were conducted on three different platforms:

1. Machine I - E450 Sun Enterprise, with two UltraSparc II_e CPUs at 400MHz and 4 GB of RAM, using 5 SCSI disks with RAID-5 configuration.

2. Machine II - Dell PowerEdge 2400, with two Pentium-III CPUs at 800MHz and 1 GB of RAM, using 5 SCSI disks with RAID-0 configuration.

3. Machine III - Sun ULTRA 10, with one UltraSparc II_e CPU at 400MHz and 384 MB of RAM, using two IDE Western Digital WD600BB disk drives.

[1] All used benchmark data are available at `cis.poly.edu/zdelta/`.

[2] More precisely, our process has two states, s_0, where we copy a character from f_0, and s_1, where we copy a character from f_1, and two parameters, p, the probability of staying in s_0, and q, the probability of staying in s_1. In the experiments, we set $q = 0.5$ and vary p from 0 to 1. Clearly, a complete evaluation would have to look at several settings of q to capture different granularities of file changes.

gcc	size	Mach. I compress	Mach. I decomp	Mach. II compress	Mach. II decomp	Mach. III compress	Mach. III decomp
uncompressed	27289	-	-	-	-	-	-
gzip	7563	31	18	13	9.0	48	29
xdelta	462	21	15	6.8	5.0	36	29
vcdiff	290	33	16	20	8.9	55	26
zdelta SIO	251	34	17	15	8.8	56	28
zdelta direct	251	27	9.6	8.6	2.7	42	17
emacs	size	Mach. I compress	Mach. I decomp	Mach. II compress	Mach. II decomp	Mach. III compress	Mach. III decomp
uncompressed	27327	-	-	-	-	-	-
gzip	8577	36	22	16	12	56	37
xdelta	2132	30	21	10	7.0	51	38
vcdiff	1822	35	21	20	11	58	34
zdelta SIO	1466	44	22	19	11	69	36
zdelta direct	1466	35	12	11	3.4	55	22

Table 4.1: Compressed sizes and running times for the *gcc* and *emacs* data sets (sizes in KB and times in seconds)

Note that Machine I and Machine II have an extremely fast I/O sub-system, and this minimized the importance of I/O while measuring the execution speed of the compressors. In order to minimize further the impact of I/O on our results, compression/decompression speeds were measured as follows: the process was run several times on the same input data; the recorded value was the average of all runs not including the first one. Due to the differences in processing the input files, the impact of I/O varies for different compressors. For the conducted experiments *gzip* and *vcdiff* used Standard I/O, *xdelta* used direct file access, and *zdelta* used both. Note also that for Machine I and Machine II only one CPU was effectively used since all processes were run sequentially; for Machine III, only one disk was used.

4.1 Experiments on Real Data

This subsection presents experimental results on real data - the *gcc* and *emacs* data sets. Note that the *uncompressed* and *gzip* numbers are those for the newer *gcc* and *emacs* releases. We see from the results that delta compression achieves significant improvements over *gzip* on these files, especially for the very similar *gcc* files. Among the delta compressors, *zdelta* gets the best compression ratio, mainly due to the use of Huffman coding instead of byte-based coding as in *vcdiff*. The *xdelta* compressor performs worst in these experiments. As described in [6], *xdelta* aims to separate differencing and compression, and thus a standard compressor such as *gzip* can be applied to the output of *xdelta*. However, in our experiments, subsequent application of *gzip* did not result in any significant improvement on these data sets.

Concerning running times, at first glance *xdelta* and *zdelta* with direct file access appear to be

fastest. However, we note that these are the two methods that use direct file access; as shown by the different numbers for *zdelta* with standard I/O and *zdelta* with direct file access, this makes a significant difference in speed. In general, it seems difficult to see a clear winner in terms of speed from these results and the results in the next subsections. We note the significant advantage of the Pentium-based Machine II over the UltraSparc-based Machine I, and some difference between the two UltraSparc-based systems (Machine I and Machine III) due to the more powerful I/O and bus subsystem of Machine I. We note that both data sets consist of a fairly large number of small files and thus the results do not measure throughput for large files.

4.2 Fixed File Size, Varying File Similarity

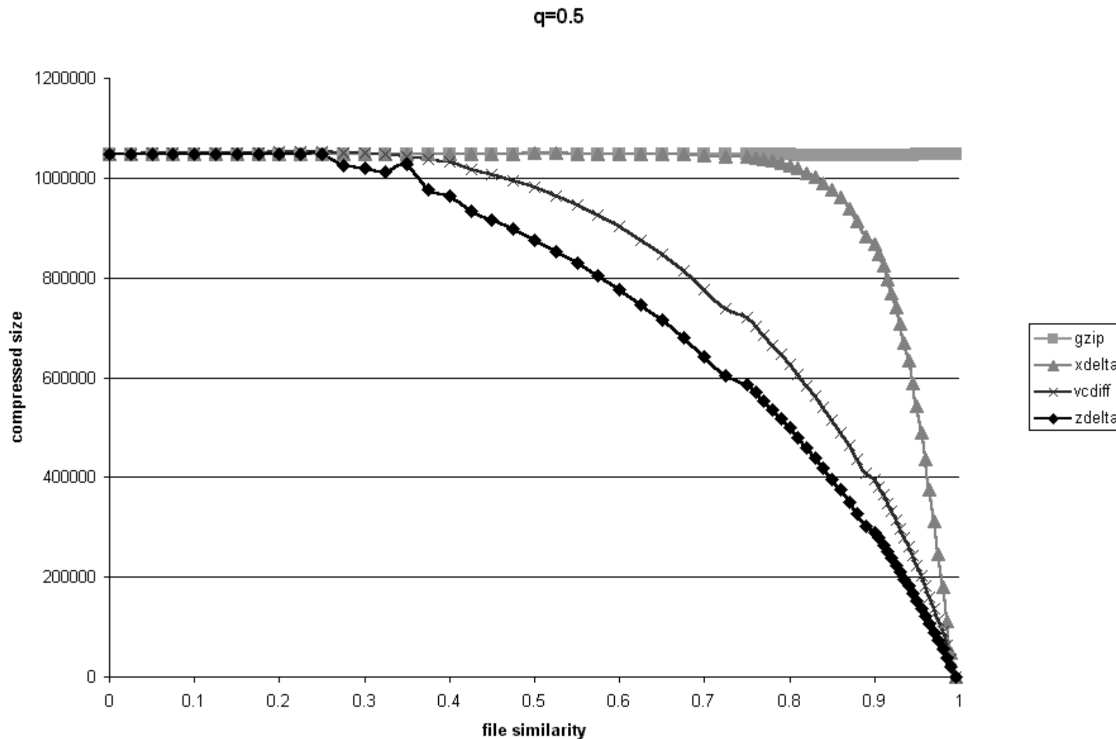

Figure 4.1: Compressed size (in bytes) versus file similarity

For this experiment, *zdelta* was run with direct file access, and all runs were done on Machine I. Looking at the compressed sizes for different file similarities in Figure 4.1, we see the same ordering as in the previous benchmark. Not surprisingly, when files are very different delta compression does not help at all, while for almost identical files all methods do quite well. However, we see that *vcdiff* and *zdelta* give benefits even for only slightly similar files for which *xdelta* does not improve over *gzip* at all. (Note that *gzip* itself does not provide any benefits here due to the incompressibility of the files.)

We see in Figure 4.2 that the running times for the delta compressors decrease as file similarity increases; this is due to the increasing lengths of the matches found in the reference files, which decrease the number of searches in the hash tables. This effect largely explains why the delta

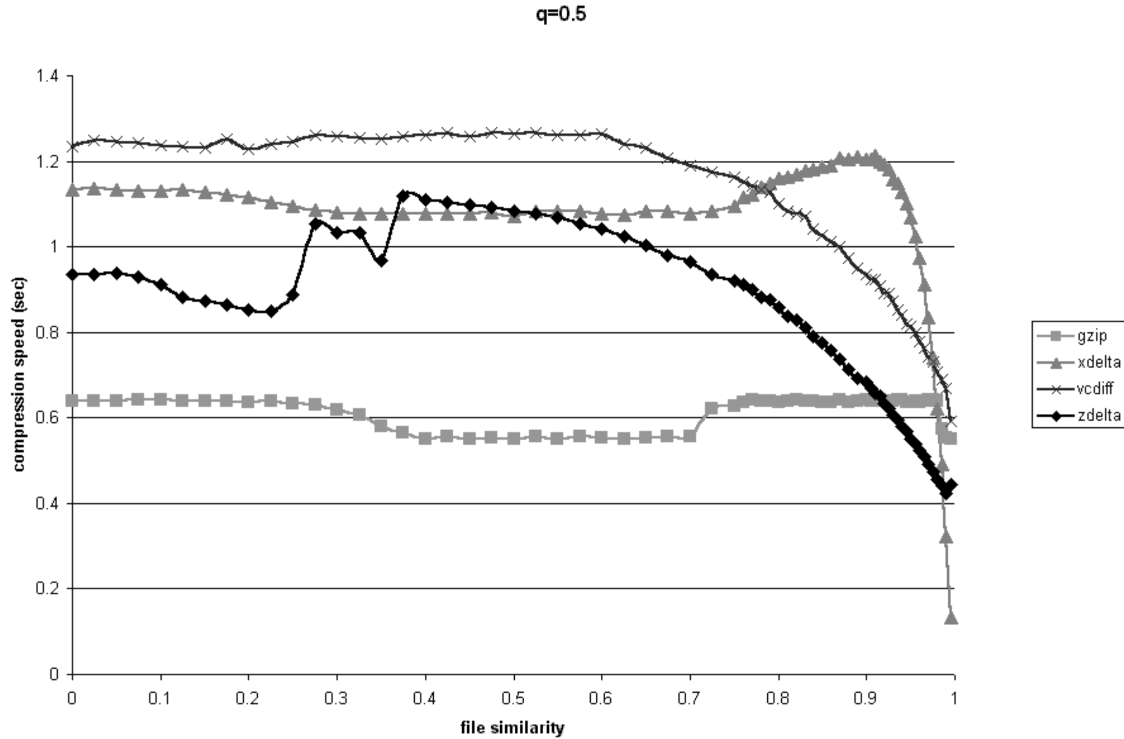

Figure 4.2: Compression time (in seconds) versus file similarity

compressors are about as fast as *gzip* on collections with large similarity such as *gcc* and *emacs*; for files with low degrees of similarity, the three delta compressors take between 60% and 200% longer than *gzip*. We also note the somewhat odd behavior for the decompression time in *xdelta*, which suddenly jumps as the files become more similar, and then goes down again. (We observed this in several different experiments on the artificial data, but not the real data, and do not currently have a good explanation.)

4.3 Fixed File Similarity, Varying File Size

For these experiments, *zdelta* was run again only with direct file access. To make the numbers symmetric, we plot both compression and decompression speed with respect to the sizes of the uncompressed target files. We set the parameter q in the creation process for the files to 0.5, as before, and fix p to three values representing target files that are very similar, moderately similar, and not similar at all to the reference files.

The results are presented in Figures 4.4 and 4.5 for compression and decompression, respectively. We can see that the dependence between the compression speed for all the compressors is roughly linear. A slight exception is *vcdiff*, for which execution time increases faster in the range from 100 KB to 200 KB. If we increase the file similarity, we can see that the graphs start to change in somewhat different ways. In particular, *xdelta* slightly decreases in speed, while *vcdiff* does not change much. On the other hand, *zdelta* increases in speed.

The graphs for decompression are more complicated. We can observe the slow decompression

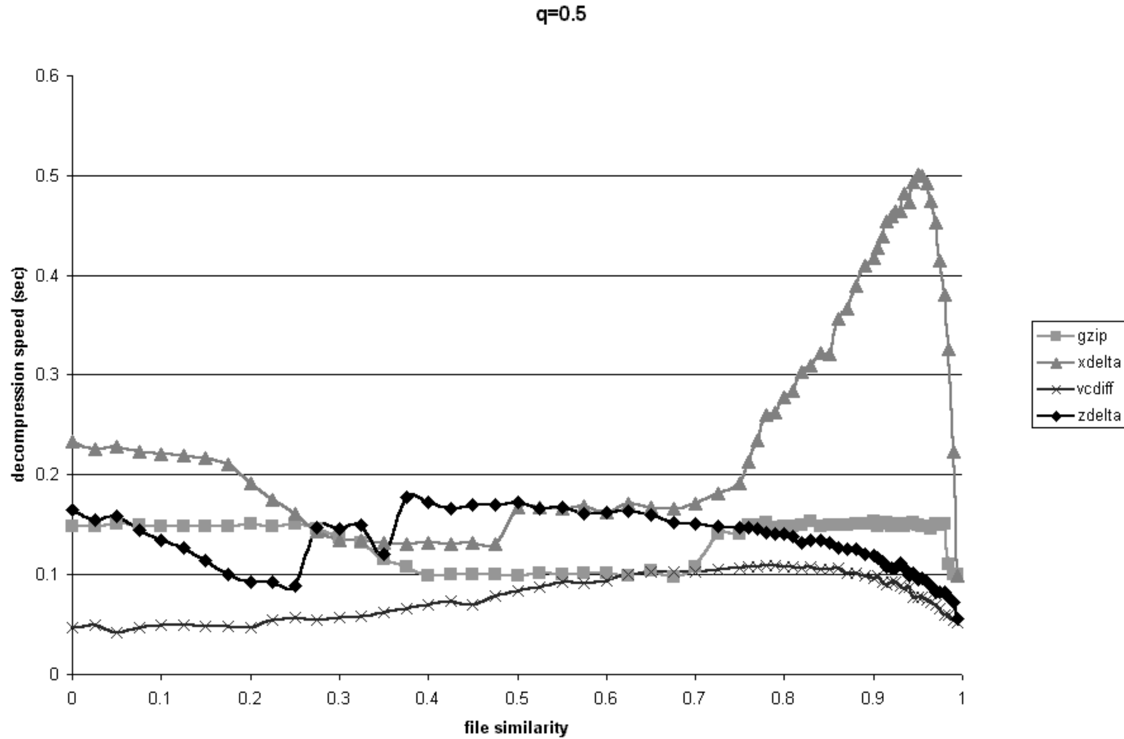

Figure 4.3: Decompression time (in seconds) versus file similarity

speed of *xdelta*, and see some irregularities in the running time of *vcdiff* as file size increases. We note that there is a difference in the relative performances on the machines. While on Machine I (UltraSparc) *vcdiff* almost always outperforms the other compressors in terms of execution speed, on Machine II (Pentium 3) it is only superior to *xdelta*. Also, as observed before, while *vcdiff* and *zdelta* improve their performance when the target and the reference files are similar, *xdelta* actually performs poorer.

5 Conclusions and Future Work

We have presented an efficient tool for delta compression, called *zdelta*, that is based on the *zlib* compression library. The compressor also uses some additional concepts from *vcdiff*, and a few original techniques for pointer representation and reference window movement. The applications for this tool include, but are not limited to, software distribution, version control systems, backup, and http traffic optimizations. We have compared *zdelta* to two other delta compressors, *vcdiff* and *xdelta*, and discussed the results of the experiments.

Although *zdelta* demonstrated good performance, there are some obvious areas with potential for improvement. Some additional tweaking could be done concerning the use of the Huffman codes, such as a better selection of default codes, or separate offset and length Huffman spaces for the different pointers and directions. The pointer movement policy and encoding could also be slightly improved, or we could add a cache for recently used copy locations, as done in *vcdiff*. Finally, we could try to use a more accurate model to decide which of two possible matches will

result in fewer bits used, beyond the simple approach of penalizing the copy length of a match with very long offset by one. (Some initial attempts did not result in any significant improvements, though, and care has to be taken not to slow down searches in the hash buckets too much.)

The current window-sliding scheme for the reference window does not guarantee any similarity between the target and the reference windows in a situation where two large files have similar pieces appearing in widely different order. A more sophisticated scheme would try to find good settings for the reference window based on some global view of the reference file. For example, *vcdiff* selects the window in the reference file based on precalculated "fingerprints" of the various parts of the file. A similar idea could be added to *zdelta* to obtain better compression in certain cases.

Another natural extension would be to allow compression of one data set in terms of two or more reference data sets. For the case of web pages from a common server, work in [2] shows that there can be significant benefit in using 2 to 6 reference data sets. Finally, the problem of identifying suitable reference files in a collection of files is an interesting problem; see, e.g., the discussion in [7].

References

[1] G. Banga, F. Douglis, and M. Rabinovich. Optimistic deltas for WWW latency reduction. In *1997 USENIX Annual Technical Conference, Anaheim, CA*, pages 289–303, January 1997.

[2] M. Chan and T. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proc. of INFOCOM'99*, March 1999.

[3] M. Delco and M. Ionescu. xProxy: A transparent caching and delta transfer system for web objects. May 2000. unpublished manuscript.

[4] J. Gailly. zlib compression library. Available at `http://www.gzip.org/zlib/`.

[5] J. Hunt, K. P. Vo, and W. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7, 1998.

[6] J. MacDonald. File system support for delta compression. MS Thesis, UC Berkeley, May 2000.

[7] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In Khalid Sayood, editor, *Lossless Compression Handbook*. Academic Press, 2002. to appear.

[8] W. Tichy. RCS: A system for version control. *Software - Practice and Experience*, 15, July 1985.

[9] J. Ziv and A. Lempel. A universal algorithm for data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

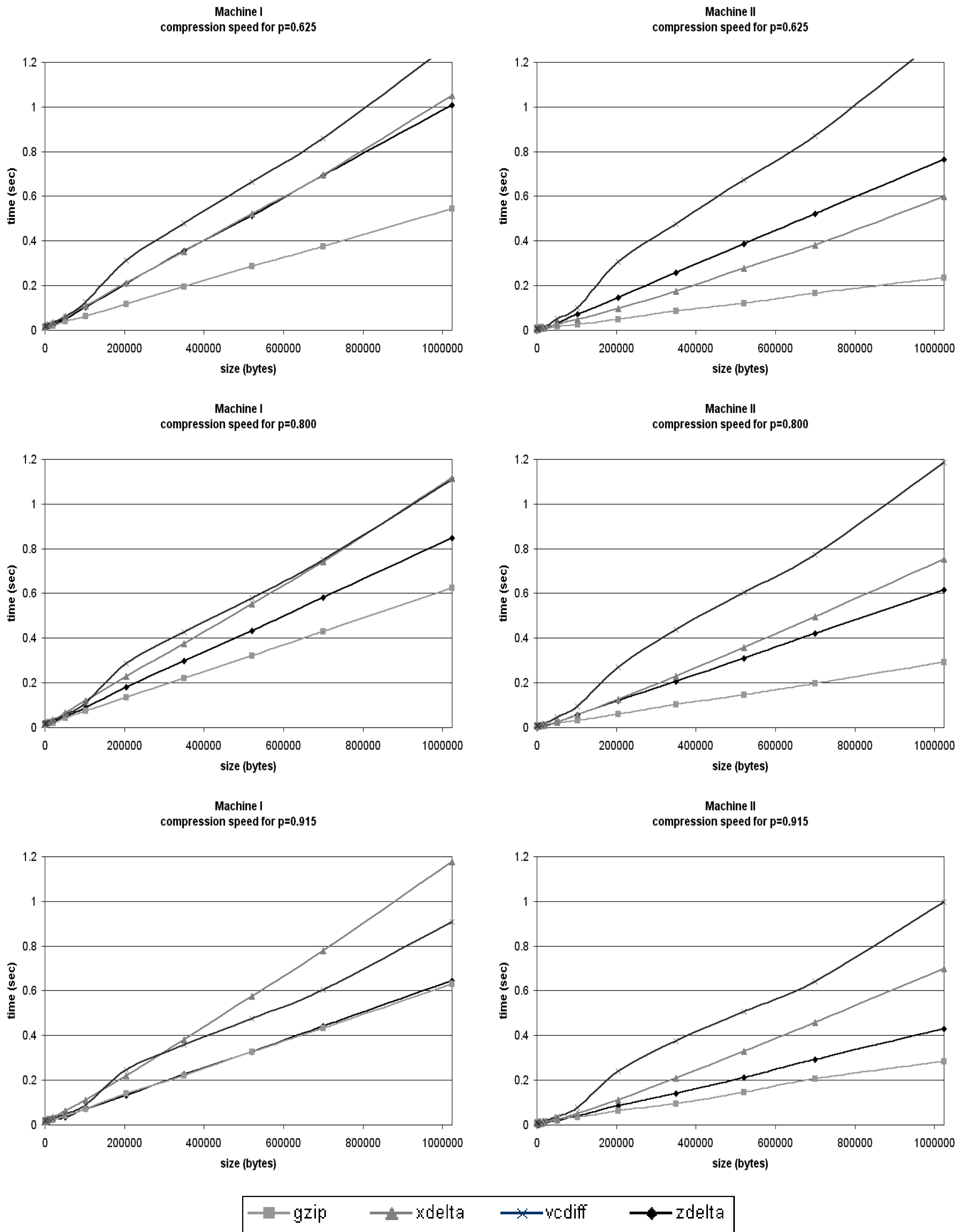

Figure 4.4: Compression time (in seconds) versus file size

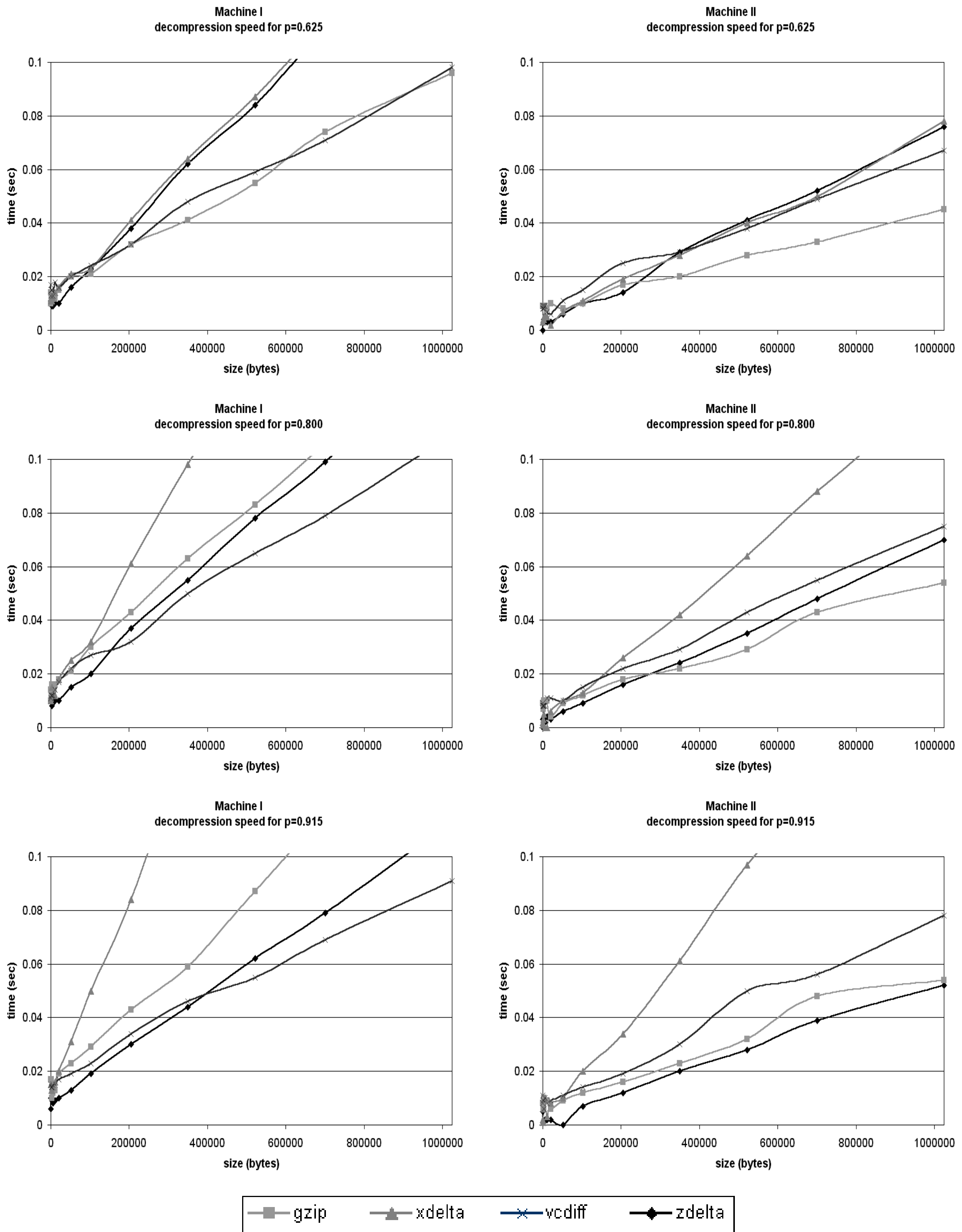

Figure 4.5: Decompression time (in seconds) versus file size