Engineering a Differencing and Compression Data Format

David G. Korn and Kiem-Phong Vo
AT&T Laboratories – Research
180 Park Avenue, Florham Park, NJ 07932, U.S.A.
dgk,kpv@research.att.com

Abstract

Compression and differencing techniques can greatly improve storage and transmission of files and file versions. Since files are often transported across machines with distinct architectures and performance characteristics, compressed data should be encoded in a form that is portable and efficient to decode. This paper describes the Vcdiff encoding format for differencing and compression data and presents an empirical study showing its effectiveness.

1 Introduction

Data differencing computes a compact transformation to take a source file to a target file based on their differences. *Data compression* compresses data in a single file. The UNIX utility *diff* is an example data differencing tool while *compress* and *gzip* are well-known data compressors. Differencing and compressed data are good for storage and transmission as they are often much smaller than the originals. Differencing and compression techniques are traditionally treated as distinct forms of data processing. Our work on the Vdelta compressor [3, 4] showed that compression and differencing can be treated uniformly by unifying the Lempel-Ziv'77 string parsing scheme [14] and Tichy's block-move technique [12]. This unification is called *delta compression*.

Compressed data need to be encoded in a portable and efficient format so that they can be transported across a network such as the Internet which consists of diverse hardware and software platforms. Many compressors are available, each with its own way to represent data. However, little has been published on the encoding formats used by these compressors. A notable exception is *gzip* whose encoding format is published in the IETF Standard *Deflate* [1]. Data differencing is much less developed than data compression so there are few tools available. The *diff* utility only works on text files and outputs editing commands to be processed by the UNIX line editor *ed*. The only published format for differencing of binary data is the W3C Standard *Gdiff* [13]. Outside of this work, there is no published encoding format for delta compression, i.e., a format suitable for both compression and differencing. Given the intended applications, we stipulate that such a data encoding format should have the following attributes:

- *Algorithm independence*: The encoding format must be independent from the algorithms used to compress data. This allows a receiver to decode compressed data without having to know how it was computed.
- *Data portability*: The encoding format must be free from hardware architecture issues such as byte order and word size. This allows a receiver to decode data without knowing the architecture of the encoding machine.
- *Output compactness*: The encoding format must compactly represent compressed and delta data. It should also be transparently extensible by encoders to maximize compression efficiency.
- *Decoding efficiency*: The encoding format must be decodable on machines with limited computational power and memory. This is important for web-based applications with small clients such as PCs or hand-held devices.

The mentioned Vdelta software was instrumental in the work to extend HTTP1.1 for Delta Encoding [9, 10]. However, the encoding format used by Vdelta was not sufficiently compact for compression data (i.e., when single files are compressed) and not easily extensible. Since then, we have designed a new format Vcdiff for delta compression. This format incorporates a number of innovations that enable compact data representation with extensibility to exploit application-specific knowledge in gaining further compression. This paper discusses the essential elements of the Vcdiff encoding format and presents performance data showing its effectiveness. A full description of the format is given in a current IETF Proposed Standard [6].

2 Algorithm independence

Techniques such as Vdelta, Lempel-Ziv and block-move are based on *string matching algorithms* [4, 7, 8] to find matches either across files or in the same file. Each such match can then be compactly encoded by its location and length. Different string matching algorithms will typically find different matches.

String matching algorithms often stress memory resources so, on current computers, they are not effective for processing large files in the order of hundreds of megabytes or gigabytes. To deal with this, a target file can be partitioned into sufficiently small contiguous segments of data called *target windows*, each of which is to be compressed separately. To improve compression, such a target window may be compared against some *source window*, a contiguous segment of data from either the source file or the target file itself. In the latter case, the source window is required to come from some part of the target file preceding the current target window so that, during decoding, the data for such a window is well-defined. Finding the right matching source window for a given target window is crucial for compressing data. Algorithms to do this are called *windowing algorithms*.

String matching and windowing algorithms clearly affect compression effectiveness. However, from the point of view of designing an encoding format, it is preferable to abstract away the details of such algorithms. In this way, simple and generic decoders can be constructed without knowing how the data was encoded. An additional benefit is that software vendors and/or researchers can continue improving the encoding algorithms without affecting the receivers of compressed data. We discuss how to design such an encoding format next.

2.1 Windowing data

Source and target windows may have different sizes but their sizes are chosen so that they can be processed entirely in memory. For data differencing, the traditional method simply aligns source and target windows by file offsets. For data compression, the popular rolling window method uses a small data segment immediately before the target window as the source window. These algorithms work well with small files since the window choices are limited (so they are mostly right by fiat) but they are suboptimal for large files as matching data may occur randomly and much further apart. In a work-in-progress, Vo explored a content-based method [11] to find source windows that would likely match well with given target windows (Section 5). Regardless of what window selection algorithm is used, a decoder does not need to know about it as long as the encoding format records the following data about source windows:

- An indication of whether the source window is from the source file or the target file,
- The starting position of the source window in the respective file, and
- The length of the source window.

Given this basic data, a decoder can obtain the appropriate source data to be used with the compressed data to decode a target window. Next we discuss what comprises compressed data.

2.2 String matching and delta instructions

When a target window T with size t is compressed given a source window S of size s, we shall think of S and T as substrings of a superstring U formed by concatenating them like this:

$$S_0S_1...S_{s-1}T_0T_1...T_{t-1}$$

The *address* of a byte in S or T is referred to by its location in U. Thus, for any $k \leq t$, the address of T_k is $s + k$. The compressed data consists of a sequence of instructions called *delta instructions*. There are three types:

- ADD: This instruction has two arguments, a size λ and a sequence of λ bytes to be copied.
- COPY: This instruction has two arguments, a size λ and an address α in the string U. These arguments specify the substring of U that must be copied into the target window being constructed. For programming convenience, we assert that such a substring must be entirely contained in either S or T.
- RUN: This instruction has two arguments, a size λ and a byte that will be copied λ times.

Let $\lambda(i)$ be the size of any delta instruction i and $\alpha(i)$ the associated address if i is a COPY. Let $I = i_1i_2...i_n$ be a sequence of delta instructions. Then each instruction i_k encodes a data segment $\sigma(i_k)$ of size $\lambda(i_k)$. Let $p = \sum_{1\leq m\leq k-1} \lambda(i_m)$. We say that I is a *faithful* representation of T if:

- For all k, $\sigma(i_k)$ is equal to the substring of T starting at p with size $\lambda(i_k)$;
- If i_k is a COPY, then $\alpha(i_k) < p + s$; and
- $\sum_{1\leq m\leq n} \lambda(i_m)$ is equal to the size of T.

1. Set $p = 0$ and $k = 0$.
2. If i_k is a RUN instruction, copy the associated data byte $\lambda(i_k)$ times to T starting at p.
3. If i_k is an ADD instruction, copy the associated data to T starting at p.
4. If i_k is a COPY instruction,
   (a) If $\alpha(i_k) < s$, copy $\lambda(i_k)$ bytes from S starting at $\alpha(i_k)$ to T starting at p;
   (b) Else, copy $\lambda(i_k)$ bytes from T starting at $\alpha(i_k) - s$ to T starting at p.
5. Set $p = p + \lambda(i_k)$ and $k = k + 1$.
6. If $k \leq n$, go to 2.

Figure 1: Decoding delta instructions

Let S be a source string of size s, T a target window of size t and I a faithful representation of T. Figure 1 shows the algorithm to reconstruct T from I and S. A string copy operation is assumed to be carried out from left to right so that Step 4.b is well-defined. Since the total running time of the algorithm is proportional to the number of bytes copied, the below result immediately follows from the definition of a faithful representation:

Theorem 1 *A target window encoded with a faithful sequence of delta instructions can be decoded in $O(t)$ time and space where t is the size of the window.*

```
S: abcdefghijklmnop
T: abcdwxyzefghefghefghefghzzzz

COPY  4, 0
ADD   4, wxyz
COPY  4, 4
COPY 12, 24
RUN   4, z
```

Figure 2: Delta instructions transforming S into T

Figure 2 shows example source and target windows and a sequence of delta instructions encoding the target data. It is easy to verify that this sequence is a faithful representation of T. The first COPY instruction copies 4 bytes from address 0, i.e., the string `abcd` in the source window S. Next is an ADD instruction that adds the 4 specified bytes `wxyz`. Note that the fourth instruction copies data from T itself since address 24 is position 8 in T. This instruction also shows that the data to be copied can overlap with the data being copied from as long as the latter starts earlier. This enables efficient encoding of periodic sequences, i.e., sequences with regularly repeated subsequences. The final RUN instruction compactly encodes the last four bytes of T.

Given a pair of target and source windows, there are usually many different faithful representations of the target data. For example, the target data in Figure 2 can also be faithfully represented with a single ADD instruction that includes all the data. From a compression point of view, it is desirable to find the representation that requires the least number of bytes to encode. Unfortunately, this problem is NP-hard even when sizes and addresses are encoded with some fixed number of bits (SR22 and SR23 in Garey and Johnson [2]). This situation improves when relaxed to just finding the minimum number of delta instructions without worrying about whether or not their encoding minimizes the compressed output. In this case, greedy approaches such as Lempel-Ziv parsing or Tichy block-move [12] do compute the minimal number of delta instructions in linear time and space given appropriate string matching algorithms [7, 8]. The Vdelta algorithm [4] relaxes this minimality to trade for faster string matching and less working memory. In any case, the point with delta instructions is that, no matter how they are computed, Theorem 1 guarantees that a generic decoder can be written that always runs in linear time and space.

3 Data portability

The Vcdiff encoding format is byte-oriented. Each byte is limited to its lower eight bits for portability. The bits in a byte are ordered from right to left so that the least significant bit (LSB) has value 1, and the most significant bit (MSB), has value 128.

Sizes and file offsets are unsigned integers encoded via a portable variable-sized format (originally introduced in the Sfio library [5]). This encoding treats an unsigned integer as a number in base 128. Then, each digit in this representation is encoded in the lower seven bits of a byte. Except for the least significant byte, other bytes have their most significant bit turned on to indicate that there are still more digits in the encoding. The two key properties of this integer encoding that are beneficial to a data compression format are:

- The encoding is portable among systems using 8-bit bytes, and
- Small values are encoded compactly.

Below is the encoding of the integer 123456789 in four 7-bit digits whose values are 58, 111, 26, 21 in order from most to least significant. In the 8-bit representation of these digits, the MSBs of 58, 111 and 26 are on.

`10111010`	`11101111`	`10011010`	`00010101`
MSB+58	MSB+111	MSB+26	0+21

4 Encoding delta instructions

The delta instructions represent string matching results. In data differencing applications of text files, changes between source and target data are often small, resulting in long common substrings. When that is the case, any straightforward representation of the delta instructions would be adequate. However, for differencing of binary files or general compression, matched substrings are often short so that the delta instructions must be encoded well to achieve good compression rates. The key to compact encoding revolves around the questions of how to encode addresses of COPY instructions efficiently and how to deal with instructions having small sizes or limited number of sizes. This leads to the ideas of *address encoding modes* and *instruction code tables* which are discussed next.

4.1 Address caches and encoding modes

Data in local regions are often replicated with minor changes. This is especially true in data differencing where target files are created from small changes in source files. Thus, the addresses of successive COPY instructions often occur close by or even exactly equal to one another. To take advantage of this phenomenon, Vcdiff maintains two types of address caches:

- A *near* cache is an array with s_near slots of previously matched addresses. An address p can be encoded against a cached address q as $p - q$ if $p \geq q$.
- A *same* cache is an array with $s\_same * 256$ slots of previously matched addresses. If an address p is equal to $same[p\%(s\_same * 256)]$, then p can be encoded with the single byte value $p\%256$.

It is clear that an encoder and a decoder must be in synch with respect to maintaining the address caches. The protocol to enforce this is as follows:

1. Before processing (i.e., encoding or decoding) a target window, all cache slots are initialized to zero.
2. After processing each COPY instruction, its address p is used to update the caches as follows:
   (a) The slots in the $near$ cache are managed as a circular buffer with a current $index$. The address p is first added to $near[index]$, then $index$ is incremented modulo s_near.
   (b) The $same$ cache is a hash table of size $s\_same * 256$. The address p is added to $same[p\%(s\_same * 256)]$.

In the above cache usage, the *address encoding mode*, i.e., the manner in which the address p of a COPY instruction is encoded must be recorded in the encoding data. Let $here$ be the current position in the target data (i.e., the start of the data about to be encoded or decoded). Below are the address modes:

- VCD_SELF: This mode has value 0 and indicates that p was encoded as itself.
- VCD_HERE: This mode has value 1 and indicates that p was encoded as $here - p$.
- *Near*: There are s_near modes in the range $[2, s\_near + 1]$. If m is the mode of the address encoding then p was encoded as $p - near[m - 2]$.
- *Same*: There are s_same modes in the range $[s\_near+2, s\_near+s\_same+1]$. If m is the mode of the encoding then p was encoded as a single byte b such that $same[(m - (s\_near + 2)) * 256 + b]$ is equal to p.

By default, Vcdiff uses 4 for s_near and 3 for s_same resulting in a total of 9 different addressing modes.

4.2 Instruction code tables

Successive delta instructions often represent short matches separated by small amounts of unmatched data. So the sizes of the COPY and ADD instructions are often small. This is particularly true of binary data such as executable files or semi-structured data such as HTML or XML. In such cases, it is beneficial to combine sizes, instruction types and even successive pairs of instructions. The effectiveness of such combinations depend on many factors including the data being processed and the string matching algorithm in use. For example, in a case where many COPY instructions with the same data sizes are generated, it may be worth encoding these instructions more compactly than others.

To maintain independence from the choices made in encoding algorithms, we introduce the notion of *instruction code tables*, each of which consists of 256 entries. These entries describe combinations of sizes, instruction types and pairs of instructions. The encoder and decoder(s) of a compressed dataset must share the same table. Then, the encoding only records the indices of the table entries, each of which fits in a single byte.

As depicted below, an entry in an instruction code table conceptually consists of two triples, each of the form `(inst,size,mode)`:

inst1	*size1*	*mode1*	*inst2*	*size2*	*mode2*

- *inst*: This field can be one of: NOOP, ADD, RUN or COPY to indicate the instruction types. NOOP means that no instruction is specified.

- *size*: This field is either zero or positive. Zero means that the size associated with the instruction is encoded separately in the encoding data. A positive value defines the actual data size so the encoding data will omit it.

- *mode*: This field is significant only when the instruction type is COPY. It defines the encoding mode used to encode the associated address.

Thus, each entry in the instruction code table can encode a single instruction (one of the triples is a NOOP) or two successive instructions. Vcdiff itself defines a *default instruction code table* for the case when the $near$ cache has 4 slots and the $same$ cache has $3 * 256$ slots. Thus, there are 9 address modes for COPY instructions. The first two are VCD_SELF(0) and VCD_HERE(1). Modes 2, 3, 4 and 5 are for addresses coded against the *near* cache. And, modes 6, 7 and 8 are for addresses coded against the *same* cache. This default table is assumed to be available with each encoder and decoder. The Vcdiff encoding format also allows an encoder to define its own custom code table but then it has to encode this table in the data itself [6].

Table 1 depicts the default instruction code table. Each numbered line represents one or more entries (recall that an entry in the instruction code table may represent up to two combined delta instructions). The last column ("Index") shows which index value or range of index values of the entries covered by that line. The first 6 columns of a line in the depiction describe the pairs of instructions used for the corresponding index value(s). For example, line 1 shows the single RUN instruction with index 0. As the size field is 0, this RUN instruction always has its actual size encoded separately in the encoding data. Line 2 shows the 18 single ADD instructions. The ADD instruction with size field 0 (i.e., the actual size is coded separately) has index 1. ADD instructions with sizes from 1 to 17 use code indices 2 to 18 and their sizes are as given (so they will not be separately encoded). Lines 12 to 21 show the pairs of instructions that are combined together. For example, line 12 depicts the 12 entries in which an ADD instruction is combined with an immediately following COPY instruction. The entries with indices 163, 164, 165 represent the pairs in which the ADD instructions all have size 1 while the COPY instructions have mode VCD_SELF(0) and sizes 4, 5 and 6 respectively.

Table 2 shows two different encodings of the delta instructions from Figure 2: Plain and Optimized. In the Plain encoding, each instruction was simply encoded. For example, the first COPY instruction used code index 19 so its size and address were separately encoded entailing a total cost of three bytes. Similarly, the ADD instruction used index 1 with separately encoded size so the cost was 6 bytes. Altogether, the Plain coding used 18 bytes which substantially improved over the original data size of 28 but was not optimal.

In the Optimized encoding, the first COPY instruction used code index 20 with implicit size 4. Thus, its encoding took only two bytes. The second and third instructions, ADD and COPY, were combined via code index 172 with both sizes implicitly defined. Thus, both instructions were encoded in 6 bytes instead of the original 9 bytes in the Plain encoding. Altogether, the size of the Optimized encoding improved to 13 bytes.

The Optimized encoding shows that judicious use of instructions with implicitly defined sizes and combined instructions can substantially improve the compression rate. We discuss next how to compute the optimal encoding given a fixed code table.

4.3 Optimizing instruction encoding

Section 4.2 showed that an encoder has a wide latitude in choosing when and how to combine and encode delta instructions. In fact, for any fixed instruction code table, one can optimize the encoding of a sequence of delta instructions using dynamic programming. Toward this end, let $I = i_1 i_2 ... i_n$ be a sequence of delta instructions. We shall use I_k to denote the subsequence of I starting from the k^{th} instruction and extending to the end of I. For example, $I = I_1$. We define I_k to be the empty sequence whenever $k > n$.

The code entries in an instruction code table assumes that the addresses of COPY instructions and the data of ADD and RUN instructions are always coded separately. Thus, to optimize the encoding, we only need to consider the sizes of the instructions and their types, i.e., ADD, RUN, COPY and any addressing modes. Now, for each instruction i, let the cost of i, $c(i)$, be the number of bytes required to encode i and its size using the best choice from the instruction code table. We assume that the instruction code table has been defined so that there is at least one way for doing this. Likewise, for any two consecutive instructions i and j, let the cost $c(i, j)$ be the number of bytes required using the best table entry that

Table 1: The default instruction code table

	Type	Size	Mode	Type	Size	Mode	Index
1	RUN	0	0	NOOP	0	0	0
2	ADD	0, [1,17]	0	NOOP	0	0	[1,18]
3	COPY	0, [4,18]	0	NOOP	0	0	[19,34]
4	COPY	0, [4,18]	1	NOOP	0	0	[35,50]
5	COPY	0, [4,18]	2	NOOP	0	0	[51,66]
6	COPY	0, [4,18]	3	NOOP	0	0	[67,82]
7	COPY	0, [4,18]	4	NOOP	0	0	[83,98]
8	COPY	0, [4,18]	5	NOOP	0	0	[99,114]
9	COPY	0, [4,18]	6	NOOP	0	0	[115,130]
10	COPY	0, [4,18]	7	NOOP	0	0	[131,146]
11	COPY	0, [4,18]	8	NOOP	0	0	[147,162]
12	ADD	[1,4]	0	COPY	[4,6]	0	[163,174]
13	ADD	[1,4]	0	COPY	[4,6]	1	[175,186]
14	ADD	[1,4]	0	COPY	[4,6]	2	[187,198]
15	ADD	[1,4]	0	COPY	[4,6]	3	[199,210]
16	ADD	[1,4]	0	COPY	[4,6]	4	[211,222]
17	ADD	[1,4]	0	COPY	[4,6]	5	[223,234]
18	ADD	[1,4]	0	COPY	4	6	[235,238]
19	ADD	[1,4]	0	COPY	4	7	[239,242]
20	ADD	[1,4]	0	COPY	4	8	[243,246]
21	COPY	4	[0,8]	ADD	1	0	[247,255]

Table 2: Encoding the delta instructions in Figure 2

Delta Inst.	Plain	Optimized
COPY 4, 0	19 4 0	20 0
ADD 4, wxyz	1 4 wxyz	172 wxyz 4
COPY 4, 4	19 4 4	
COPY 12, 24	19 12 24	28 24
RUN 4, z	0 4 z	0 4 z

combines both instructions. If there is no way to combine i and j, we let $c(i,j)$ be infinite. Finally, let $C(I)$ be the cost of encoding the sequence I. Then, $C(I)$ can be obtained by solving the following dynamic program:

$$C(I) = \begin{cases} 0 & \text{if } I = \phi; \\ c(i_1) & \text{if } |I| = 1; \text{otherwise,} \\ min\{c(i_1) + C(I_2), c(i_1, i_2) + C(I_3)\}. \end{cases}$$

The first case states that the cost of encoding an empty sequence is 0. The second case states the cost of encoding a sequence with a single instruction. The last case computes the optimal cost by minimizing between two alternatives: encoding the first instruction by itself or combining the first two instructions. In each alternative, recursion is used to deal with the rest of the sequence.

Since each $C(I_k)$ is uniquely determined by the sequence I_k, we can keep track of all processed subsequences in $O(|I|)$ space so that the recursion can be pruned whenever it arrives at a processed subsequence. We have shown:

Theorem 2 *Given a fixed instruction code table, any sequence of delta instructions I can be encoded optimally in $O(|I|)$ time and space.*

In addition to optimizing delta instruction encoding given a fixed code table, it is also possible for an application to define its own code tables inside the encoding data [6]. This enables an application to gain further compression by specially treating certain instructions or pairs of instructions that are much more popular than others. We are investigating the question of how to compute such an optimal instruction code table. The mentioned IETF Proposed Standard [6] also discusses the use of secondary compressors to further compress the instruction encoding.

5 Performance

We show the effectiveness of the Vcdiff encoding format in two ways. The first set of experiments was based on three different source code archives of the Gnu C compiler, gcc-2.95.1.tar, gcc-2.95.2.tar and gcc-2.95.3.tar. We used the Vcodex/Vcdiff software (Section 8) to compare various options of Vcdiff against the *gzip* and *compress* tools. These files were very large so that some windowing scheme must be used. In the second set of experiments, we collected the home page of www.cnn.com every hour for 10 days and computed the deltas using various methods.

5.1 Comparing with *compress* and *gzip*

We compared *Vcdiff* against *compress* and *gzip* using the mentioned three source code archives of the GNU C compiler. The experiments were done on an SGI-MIPS3, 400MHZ. Timing results were obtained by running each program three times and taking the average of the total cpu+system times. Below are the different Vcdiff runs:

- *Vcdiff-c*: *Vcdiff* was used for compression only. That is, no source file was used. This directly compared *Vcdiff*, *gzip* and *compress* as compressors.
- *Vcdiff-d*: *Vcdiff* was used for differencing only. That is, matching was allowed only between source and target data. Windows were simply matched by positions across target and source files.
- *Vcdiff-dc*: This is similar to *Vcdiff-d* but matching within target data, was allowed, i.e., delta compression was used.
- *Vcdiff-dcw*: This is similar to *Vcdiff-dc* but a content-based windowing algorithm [11] was used to select source windows more likely to match with given target windows. Thus, file offsets of source and target windows would seldom align.

Table 3 shows the experimental results. Note that compression times were typically dominated by the string matching and encoding algorithms. For example, the large time variation in the *Vcdiff* rows was strictly due to the windowing and string matching algorithms used in the Vcodex/Vcdiff software. Such measurements were somewhat irrelevant from the point of view of evaluating an algorithm-independent encoding format. However, the decompression times were indicative of how the different formats would perform in practice.

The pure compressor *Vcdiff-c* gave worse compression rate than *gzip* but better than *compress*. However, it always decompressed fastest. Version gcc.2.95.2 was similar to version gcc.2.95.1. Thus, compressing gcc.2.95.2 given gcc.2.95.1 gained up to a factor of 500 in size reduction as shown in the last three rows. On the other hand, the files in the archive gcc.2.95.3 were were sufficiently changed and rearranged from gcc.2.95.2 so that simply matching source and target windows by file positions were ineffective. As a result, *Vcdiff-d* and *Vcdiff-dc* did not perform well even though delta compression did help *Vcdiff-dc* to beat *Vcdiff-c* and come close to *gzip*. *Vcdiff-dcw* still worked well due to the content-based windowing algorithm. There was a clear time cost for using such an algorithm during encoding but decoding time was not affected.

Finally, the *cat* row of the table shows the times required to just copy the files gcc.2.95.2.tar and gcc.2.95.3.tar, respectively 1.08 and 1.05 seconds. Thus, in the best case of *Vcdiff-dcw*, decompression times were only about 70-90% worst than plain copying of the data. The dramatic size reduction meant that, with an appropriate encoder, the Vcdiff encoding format presented a good mechanism for transporting data without taxing client machines on decoding.

5.2 Compressing a set of web pages

We collected the home pages of www.cnn.com every hour starting at 12:00AM on 10/23/2001 and ending at 11:00PM 11/01/2001. The below methods for delta compression were used:

- *diff+gzip*: This method runs the *diff -e* program to compute the differences, then pipes the result to *gzip* for further compression.
- *gdiff*: We instrumented the Vcodex/Vcdiff software to run the Vcdiff string matching algorithm but output results in the *Gdiff* format.
- *gdiff+gzip*: This is like the above but the result is piped to *gzip* for further compression.
- *vcdiff*: This uses the Vcdiff encoding format.

We ran two different experiments. In the *First* experiment, each file is compressed against the first file collected while, in the *Successive* experiment, each file is compressed against the one in the previous hour. Table 4 summarizes the compression results. The *raw* row shows

Table 3: Comparing *Vcdiff* to *gzip* and *compress* using the gcc-2.95.[123] archives

	2.95.2 (55,797,760)			*2.95.3 (55,787,520)*		
Compressor	Size	Comp.(s)	Decomp.(s)	Size	Comp.(s)	Decomp.(s)
cat	55,797,760	1.08	1.08	55,787,520	1.05	1.05
compress	19,939,390	13.85	7.09	19,939,453	13.54	7.00
gzip	12,973,443	42.99	5.35	12,998,097	42.63	5.62
Vcdiff-c	15,358,786	20.04	4.65	15,371,737	20.09	4.74
	2.95.2 given 2.95.1 (55,746,560)			*2.95.3 given 2.95.2 (55,797,760)*		
Vcdiff-d	100,971	10.93	1.92	26,383,849	71.41	6.41
Vcdiff-dc	97,246	20.03	1.84	14,461,203	42.48	4.82
Vcdiff-dcw	256,445	44.81	1.84	1,248,543	61.18	1.99

Table 4: Delta compression of www.cnn.com

	First			Successive		
Method	Min.	Max.	Avg.	Min.	Max.	Avg.
raw	44,602	50,033	46,036	44,602	50,033	46,036
diff+gzip	20	6,257	4,955	20	3,146	1,017
gdiff	11	5,597	4,277	11	1,767	458
gdiff+gzip	31	4,335	3,336	31	1,496	434
vcdiff	23	4,249	3,274	23	1,423	385

the Minimum, Maximum and Average sizes of the collected files. Later rows show the same statistics for the compressed data. Delta compression was effective in reducing the data sizes overall. The best method, *vcdiff* reduced data by a factor of about 15 in the *First* experiment and about 120 in the *Successive* experiment.

Figure 3 shows in detail the sizes of the compressed data in the *First* experiment. The order of the methods from worst to best was *diff+gzip*, *gdiff*, *gdiff+gzip* and *vcdiff*. Since *gdiff* and *vcdiff* were based on the same underlying algorithms to compute delta instructions, the results compared directly the effectiveness of the different encoding formats. Even with the additional compression step using *gzip* (i.e., the *Deflate* format) the *gdiff+gzip* results were still slightly worse than *vcdiff*. The fact that delta compression was still effective after a fairly long duration of 10 days suggested that these pages were generated from some large template that seldom changed.

Figure 4 shows results from the *Successive* experiment. The *diff+gzip* data fluctuated wildly because *diff* was line-oriented and could not handle small changes made on many lines. Due to this large fluctuation, the format used in Figure 3 did not show the data well. Therefore, we plotted all data points relative to the ones from *vcdiff* by simply dividing each data point by the corresponding value from *vcdiff*. Thus, the flat horizontal line at 1 represented *vcdiff* data. The encoding formats of *gzip*, *gdiff* and *vcdiff* required fixed overhead sets of bytes (shown in the *Min.* columns in Table 4 as the compared files were identical in that case). We subtracted such overheads from the data points before dividing to remove the large distortion in the ratios when files changed little. *Vcdiff* was again the best encoding format for delta compression in this experiment. In fact, Table 4 showed that it typically reduced data by more than 2 orders of magnitude since files changed very little in successive hours.

Timing results were not shown in the above experiments but *diff+gzip* and *gdiff+gzip* were much slower than *vcdiff* and *gdiff* because of the use of multiple processes and, in the case of *diff+gzip*, slow text alignment algorithms. For *vcdiff* and *gdiff*, it was also hard to obtain meaningful measurements since the files were small and the encoding algorithms were sufficiently fast so that process start-up time became the dominating factor.

6 Summary

We described Vcdiff, a general and portable encoding format for delta compression, i.e., combined compression and differencing. This is the first fully described encoding format for this type of data processing. Vcdiff introduced the novel idea of an instruction code table to allow combining delta instructions to optimize compression rate. We showed how to compute minimal encodings given a fixed code table using dynamic programming. More importantly, the nature of the encoding format enables construction of decoders free from any knowledge of encoders and guaranteed to run in linear time and space. Thus, Vcdiff is suitable for web-based client-server applications in which a big server can send data to much smaller clients with different hardware architectures. We presented performance results showing that Vcdiff compares favorably to other formats for data differencing including the W3C Gdiff Standard and the use of *diff* and *gzip*. Vcdiff is the subject of a current IETF Proposed Standard [6].

7 Acknowledgements

Thanks are due to Balachander Krishnamurthy, Jeff Mogul and Arthur Van Hoff who provided much encouragement to publicize Vcdiff.

8 Code availability

The Vcdiff data format described here is free from any patent claims. An implementation of Vcdiff is available as a part of the Vcodex package written by Phong Vo. The code can be obtained from http://www.research.att.com/sw/tools.

References

[1] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. In *http://www.ietf.org*. IETF RFC1951, 1996.

[2] M.R. Garey and D.S. Johnson. *Computers and Intractibility*. W.H. Freeman and Company, NY, 1979.

[3] J.J. Hunt, K.-P. Vo, and W.F. Tichy. An Empirical Study of Delta Algorithms. In *IEEE Software Configuration and Maintenance Workshop*, 1996.

[4] J.J. Hunt, K.-P. Vo, and W.F. Tichy. Delta Algorithms: An Empirical Analysis. *ACM Transactions on Software Engineering and Methodology*, 7:192–214, 1998.

[5] D.G. Korn and K.-P. Vo. SFIO: Safe/Fast String/File IO. In *Proc. of the Summer '91 Usenix Conference*, pages 235–256. USENIX, 1991.

[6] D.G. Korn and K.-P. Vo. The VCDIFF Generic Differencing and Compression Data Format. www.research.att.com/sw/tools/vcodex, www.ietf.org, 2000.

[7] U. Manber and G. Meyer. Suffix Array: A New Method for On-Line String Searches. *SIAM J. Computing*, 22:935–948, 1993.

[8] E. M. McCreight. A space-economical suffix tree construction algorithm. *JACM*, 23:262–272, 1976.

[9] J.C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *SIGCOMM '97, Cannes, France*, 1997.

[10] J.C. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, and A. Van Hoff. Delta Encoding in HTTP. In *draft-mogul-http-delta-10*. IETF, 2000.

[11] S. Suri and K.-P. Vo. Effective Windowing for Delta Compression. Work-in-progress, 2001.

[12] Walter F. Tichy. The String-to-String Correction Problem with Block Moves. *ACM Transactions on Computer Systems*, 2(4):309–321, November 1984.

[13] A. van Hoff and J. Payne. Generic Diff Format Specification. In *http://www.w3.org/TR/NOTE-gdiff-19970825.html*, 1997.

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

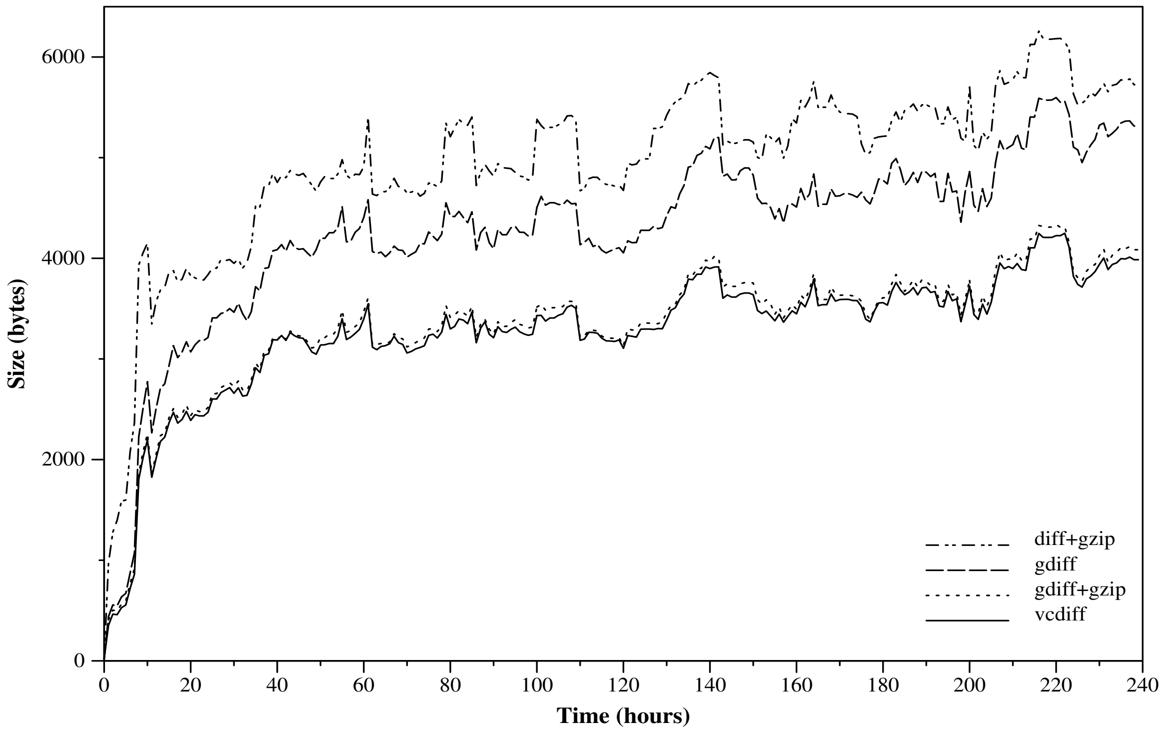

Figure 3: Delta compression of www.cnn.com against a fixed first page

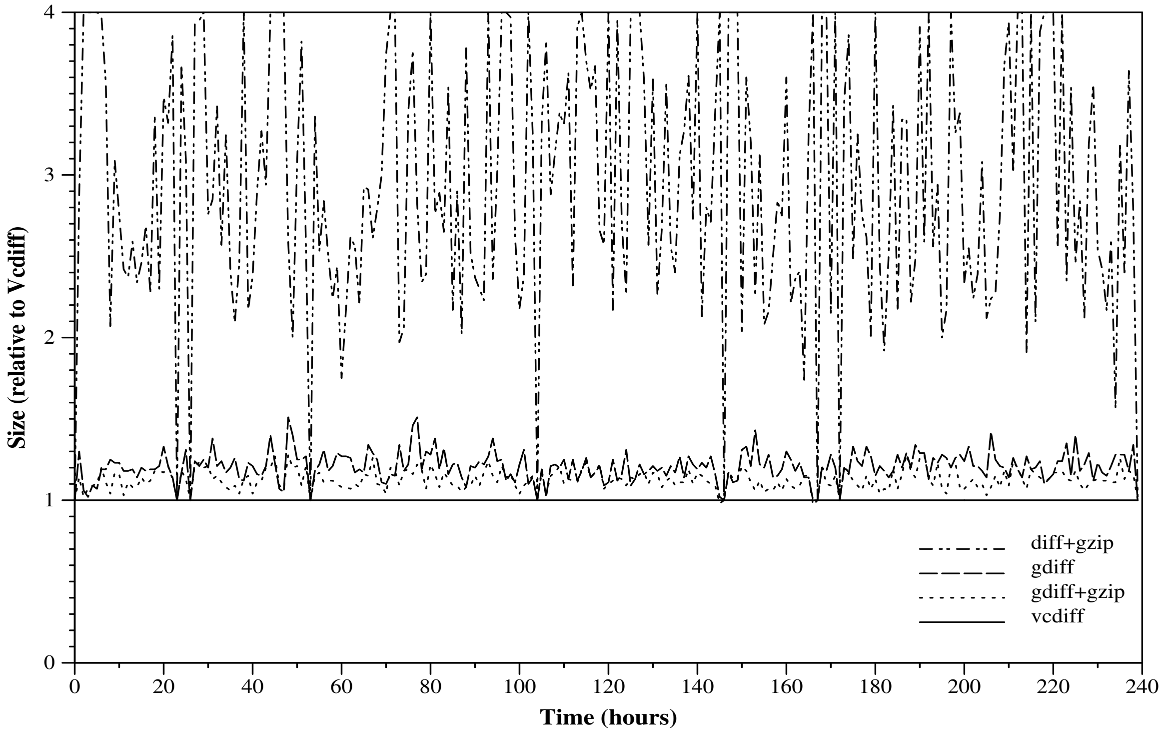

Figure 4: Delta compression of www.cnn.com in successive hours