Network Working Group   D. Korn
Request for Comments: 3284   AT&T Labs
Category: Standards Track   J. MacDonald
                                                         UC Berkeley
                                                            J. Mogul
                                            Hewlett-Packard Company
                                                               K. Vo
                                                           AT&T Labs
                                                           June 2002

       The VCDIFF Generic Differencing and Compression Data Format

Status of this Memo

Copyright Notice

Abstract

   This memo describes VCDIFF, a general, efficient and portable data
   format suitable for encoding compressed and/or differencing data so
   that they can be easily transported among computers.

Table of Contents

1. Executive Summary

   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, such data should be encoded in a form
   that is portable and can be decoded with little or no knowledge of
   the encoders. This document describes Vcdiff, a compact portable
   encoding format designed for these purposes.

   Data differencing is the process of computing a compact and
   invertible encoding of a "target file" given a "source file". Data
   compression is similar, but without the use of source data. The UNIX
   utilities diff, compress, and gzip are well-known examples of data
   differencing and compression tools. For data differencing, the
   computed encoding is called a "delta file", and for data compression,
   it is called a "compressed file". Delta and compressed files are
   good for storage and transmission as they are often smaller than the
   originals.

   Data differencing and data compression are traditionally treated as
   distinct types of data processing. However, as shown in the Vdelta
   technique by Korn and Vo [1], compression can be thought of as a
   special case of differencing in which the source data is empty. The
   basic idea is to unify the string parsing scheme used in the Lempel-
   Ziv'77 (LZ'77) style compressors [2] and the block-move technique of
   Tichy [3]. Loosely speaking, this works as follows:

        a. Concatenate source and target data.
        b. Parse the data from left to right as in LZ'77 but make sure
           that a parsed segment starts the target data.
        c. Start to output when reaching target data.

   Parsing is based on string matching algorithms, such as suffix trees
   [4] or hashing with different time and space performance
   characteristics. Vdelta uses a fast string matching algorithm that
   requires less memory than other techniques [5,6]. However, even with
   this algorithm, the memory requirement can still be prohibitive for
   large files. A common way to deal with memory limitation is to
   partition an input file into chunks called "windows" and process them
   separately. Here, except for unpublished work by Vo, little has been
   done on designing effective windowing schemes. Current techniques,
   including Vdelta, simply use source and target windows with
   corresponding addresses across source and target files.

   String matching and windowing algorithms have great influence on the
   compression rate of delta and compressed files. However, it is
   desirable to have a portable encoding format that is independent of
   such algorithms. This enables the construction of client-server
   applications in which a server may serve clients with unknown
   computing characteristics. Unfortunately, all current differencing
   and compressing tools, including Vdelta, fall short in this respect.
   Their storage formats are closely intertwined with the implemented
   string matching and/or windowing algorithms.

   The encoding format Vcdiff proposed here addresses the above issues.
   Vcdiff achieves the characteristics below:

      Output compactness:
         The basic encoding format compactly represents compressed or
         delta files. Applications can further extend the basic
         encoding format with "secondary encoders" to achieve more
         compression.

      Data portability:
         The basic encoding format is free from machine byte order and
         word size issues. This allows data to be encoded on one
         machine and decoded on a different machine with different
         architecture.

      Algorithm genericity:
         The decoding algorithm is independent from string matching and
         windowing algorithms. This allows competition among
         implementations of the encoder while keeping the same decoder.

Decoding efficiency:
    Except for secondary encoder issues, the decoding algorithm
    runs in time proportionate to the size of the target file and
    uses space proportionate to the maximal window size. Vcdiff
    differs from more conventional compressors in that it uses only
    byte-aligned data, thus avoiding bit-level operations, which
    improves decoding speed at the slight cost of compression
    efficiency.

The combined differencing and compression method is called "delta
compression" [14]. As this way of data processing treats compression
as a special case of differencing, we shall use the term "delta file"
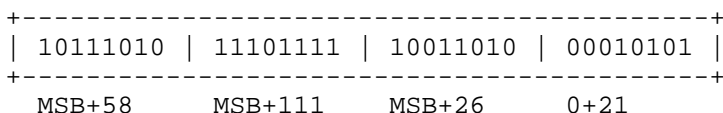to indicate the compressed output for both cases.

2. Conventions

The basic data unit is a byte. For portability, Vcdiff shall limit a
byte to its lower eight bits even on machines with larger bytes. 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.

For purposes of exposition in this document, we adopt the convention
that the LSB is numbered 0, and the MSB is numbered 7. Bit numbers
never appear in the encoded format itself.

Vcdiff encodes unsigned integer values using a portable, variable-
sized format (originally introduced in the Sfio library [7]). This
encoding treats an 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:

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

For example, consider the value 123456789, which can be represented
with four 7-bit digits whose values are 58, 111, 26, 21 in order from
most to least significant. Below is the 8-bit byte encoding of these
digits. Note that the MSBs of 58, 111 and 26 are on.

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

Henceforth, the terms "byte" and "integer" will refer to a byte and
an unsigned integer as described.

Algorithms in the C language are occasionally exhibited to clarify
the descriptions. Such C code is meant for clarification only, and
is not part of the actual specification of the Vcdiff format.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in BCP 14, RFC 2119 [12].

3. Delta Instructions

A large target file is partitioned into non-overlapping sections
called "target windows". These target windows are processed
separately and sequentially based on their order in the target file.

A target window T, of length t, may be compared against some source
data segment S, of length s. By construction, this source data
segment S comes either from the source file, if one is used, or from
a part of the target file earlier than T. In this way, during
decoding, S is completely known when T is being decoded.

The choices of T, t, S and s are made by some window selection
algorithm, which can greatly affect the size of the encoding.
However, as seen later, these choices are encoded so that no
knowledge of the window selection algorithm is needed during
decoding.

Assume that S[j] represents the jth byte in S, and T[k] represents
the kth byte in T. Then, for the delta instructions, we treat the
data windows S and T as substrings of a superstring U, formed by
concatenating them like this:

        S[0]S[1]...S[s-1]T[0]T[1]...T[t-1]

The "address" of a byte in S or T is referred to by its location in
U. For example, the address of T[k] is s+k.

The instructions to encode and direct the reconstruction of a target
window are called delta instructions. There are three types:

   ADD: This instruction has two arguments, a size x and a sequence
         of x bytes to be copied.
   COPY: This instruction has two arguments, a size x and an address
         p in the string U. The arguments specify the substring of U
         that must be copied. We shall assert that such a substring
         must be entirely contained in either S or T.

RUN: This instruction has two arguments, a size x and a byte b,
      that will be repeated x times.

Below are example source and target windows and the delta
instructions that encode the target window in terms of the source
window.

    a b c d e f g h i j k l m n o p
    a b c d w x y z e f g h e f g h e f g h e f g h z z z z

    COPY 4, 0
    ADD   4, w x y z
    COPY 4, 4
    COPY 12, 24
    RUN   4, z

Thus, the first letter 'a' in the target window is at location 16 in
the superstring. Note that the fourth instruction, "COPY 12, 24",
copies data from T itself since address 24 is position 8 in T. This
instruction also shows that it is fine to overlap the data to be
copied 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 RUN
instruction is a compact way to encode a sequence repeating the same
byte even though such a sequence can be thought of as a periodic
sequence with period 1.

To reconstruct the target window, one simply processes one delta
instruction at a time and copies the data, either from the source
window or the target window being reconstructed, based on the type of
the instruction and the associated address, if any.

4. Delta File Organization

A Vcdiff delta file starts with a Header section followed by a
sequence of Window sections. The Header section includes magic bytes
to identify the file type, and information concerning data processing
beyond the basic encoding format. The Window sections encode the
target windows.

Below is the overall organization of a delta file. The indented
items refine the ones immediately above them. An item in square
brackets may or may not be present in the file depending on the
information encoded in the Indicator byte above it.

```
Header
    Header1                                    - byte
    Header2                                    - byte
    Header3                                    - byte
    Header4                                    - byte
    Hdr_Indicator                              - byte
    [Secondary compressor ID]                  - byte
    [Length of code table data]                - integer
    [Code table data]
        Size of near cache                     - byte
        Size of same cache                     - byte
        Compressed code table data
Window1
    Win_Indicator                              - byte
    [Source segment size]                      - integer
    [Source segment position]                  - integer
    The delta encoding of the target window
        Length of the delta encoding           - integer
        The delta encoding
            Size of the target window          - integer
            Delta_Indicator                    - byte
            Length of data for ADDs and RUNs   - integer
            Length of instructions and sizes   - integer
            Length of addresses for COPYs      - integer
            Data section for ADDs and RUNs     - array of bytes
            Instructions and sizes section     - array of bytes
            Addresses section for COPYs        - array of bytes
Window2
...
```
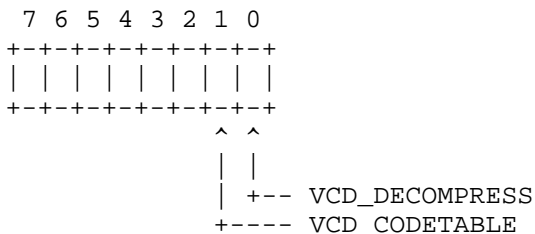
4.1 The Header Section

   Each delta file starts with a header section organized as below.
   Note the convention that square-brackets enclose optional items.

```
    Header1                                    - byte = 0xD6
    Header2                                    - byte = 0xC3
    Header3                                    - byte = 0xC4
    Header4                                    - byte
    Hdr_Indicator                              - byte
    [Secondary compressor ID]                  - byte
    [Length of code table data]                - integer
    [Code table data]
```

The first three Header bytes are the ASCII characters 'V', 'C' and
'D' with their most significant bits turned on (in hexadecimal, the
values are 0xD6, 0xC3, and 0xC4). The fourth Header byte is
currently set to zero. In the future, it might be used to indicate
the version of Vcdiff.

The Hdr_Indicator byte shows if there is any initialization data
required to aid in the reconstruction of data in the Window sections.
This byte MAY have non-zero values for either, both, or neither of
the two bits VCD_DECOMPRESS and VCD_CODETABLE below:

```
   7 6 5 4 3 2 1 0
  +-+-+-+-+-+-+-+-+
  | | | | | | | | |
  +-+-+-+-+-+-+-+-+
                ^ ^
                | |
                | +-- VCD_DECOMPRESS
                +---- VCD_CODETABLE
```

If bit 0 (VCD_DECOMPRESS) is non-zero, this indicates that a
secondary compressor may have been used to further compress certain
parts of the delta encoding data as described in Sections 4.3 and 6.
In that case, the ID of the secondary compressor is given next. If
this bit is zero, the compressor ID byte is not included.

If bit 1 (VCD_CODETABLE) is non-zero, this indicates that an
application-defined code table is to be used for decoding the delta
instructions. This table itself is compressed. The length of the
data comprising this compressed code table and the data follow next.
Section 7 discusses application-defined code tables. If this bit is
zero, the code table data length and the code table data are not
included.

If both bits are set, then the compressor ID byte is included before
the code table data length and the code table data.
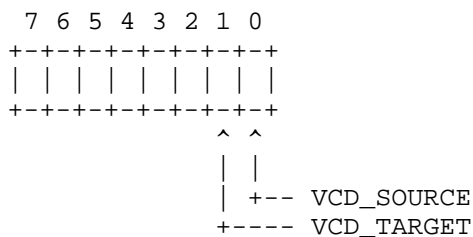
4.2 The Format of a Window Section

Each Window section is organized as follows:

```
    Win_Indicator                               - byte
    [Source segment length]                     - integer
    [Source segment position]                   - integer
    The delta encoding of the target window
```

Below are the details of the various items:

Win_Indicator:
   This byte is a set of bits, as shown:

```
 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+
| | | | | | | | |
+-+-+-+-+-+-+-+-+
             ^ ^
             | |
             | +-- VCD_SOURCE
             +---- VCD_TARGET
```

If bit 0 (VCD_SOURCE) is non-zero, this indicates that a
segment of data from the "source" file was used as the
corresponding source window of data to encode the target
window. The decoder will use this same source data segment to
decode the target window.

If bit 1 (VCD_TARGET) is non-zero, this indicates that a
segment of data from the "target" file was used as the
corresponding source window of data to encode the target
window. As above, this same source data segment is used to
decode the target window.

The Win_Indicator byte MUST NOT have more than one of the bits
set (non-zero). It MAY have none of these bits set.

If one of these bits is set, the byte is followed by two
integers to indicate respectively, the length and position of
the source data segment in the relevant file. If the indicator
byte is zero, the target window was compressed by itself
without comparing against another data segment, and these two
integers are not included.

The delta encoding of the target window:

This contains the delta encoding of the target window, either
in terms of the source data segment (i.e., VCD_SOURCE or
VCD_TARGET was set) or by itself if no source window is
specified. This data format is discussed next.

4.3 The Delta Encoding of a Target Window

   The delta encoding of a target window is organized as follows:

        Length of the delta encoding   - integer
        The delta encoding
            Length of the target window   - integer
            Delta_Indicator   - byte
            Length of data for ADDs and RUNs   - integer
            Length of instructions section   - integer
            Length of addresses for COPYs   - integer
            Data section for ADDs and RUNs   - array of bytes
            Instructions and sizes section   - array of bytes
            Addresses section for COPYs   - array of bytes

        Length of the delta encoding:
           This integer gives the total number of remaining bytes that
           comprise the data of the delta encoding for this target
           window.

        The delta encoding:
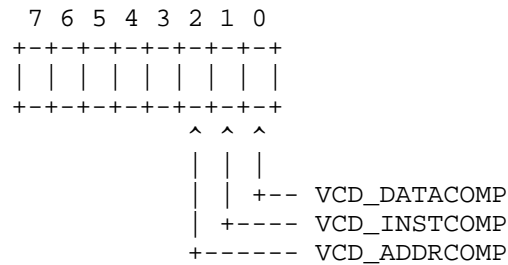   This contains the data representing the delta encoding which
           is described next.

        Length of the target window:
           This integer indicates the actual size of the target window
           after decompression. A decoder can use this value to
           allocate memory to store the uncompressed data.

        Delta_Indicator:
           This byte is a set of bits, as shown:

         7 6 5 4 3 2 1 0
        +-+-+-+-+-+-+-+-+
        | | | | | | | | |
        +-+-+-+-+-+-+-+-+
                  ^ ^ ^
                  | | |
                  | | +-- VCD_DATACOMP
                  | +---- VCD_INSTCOMP
                  +------ VCD_ADDRCOMP

            VCD_DATACOMP: bit value 1.
            VCD_INSTCOMP: bit value 2.
            VCD_ADDRCOMP: bit value 4.

As discussed, the delta encoding consists of COPY, ADD and RUN
instructions. The ADD and RUN instructions have accompanying
unmatched data (that is, data that does not specifically match
any data in the source window or in some earlier part of the
target window) and the COPY instructions have addresses of
where the matches occur. OPTIONALLY, these types of data MAY
be further compressed using a secondary compressor. Thus,
Vcdiff separates the encoding of the delta instructions into
three parts:

   a. The unmatched data in the ADD and RUN instructions,
   b. The delta instructions and accompanying sizes, and
   c. The addresses of the COPY instructions.

If the bit VCD_DECOMPRESS (Section 4.1) was on, each of these
sections may have been compressed using the specified secondary
compressor. The bit positions 0 (VCD_DATACOMP), 1
(VCD_INSTCOMP), and 2 (VCD_ADDRCOMP) respectively indicate, if
non-zero, that the corresponding parts are compressed. Then,
these parts MUST be decompressed before decoding the delta
instructions.

Length of data for ADDs and RUNs:
   This is the length (in bytes) of the section of data storing
   the unmatched data accompanying the ADD and RUN instructions.

Length of instructions section:
   This is the length (in bytes) of the delta instructions and
   accompanying sizes.

Length of addresses for COPYs:
   This is the length (in bytes) of the section storing the
   addresses of the COPY instructions.

Data section for ADDs and RUNs:
   This sequence of bytes encodes the unmatched data for the ADD
   and RUN instructions.

Instructions and sizes section:
   This sequence of bytes encodes the instructions and their
   sizes.

Addresses section for COPYs:
   This sequence of bytes encodes the addresses of the COPY
   instructions.

5. Delta Instruction Encoding

   The delta instructions described in Section 3 represent the results
   of string matching. For many data differencing applications in which
   the changes between source and target data are small, any
   straightforward representation of these instructions would be
   adequate. However, for applications including differencing of binary
   files or data compression, it is important to encode these
   instructions well to achieve good compression rates. The keys to
   this achievement is to efficiently encode the addresses of COPY
   instructions and the sizes of all delta instructions.

5.1 Address Encoding Modes of COPY Instructions

   Addresses of COPY instructions are locations of matches and often
   occur close by or even exactly equal to one another. This is because
   data in local regions are often replicated with minor changes. In
   turn, this means that coding a newly matched address against some
   recently matched addresses can be beneficial. To take advantage of
   this phenomenon and encode addresses of COPY instructions more
   efficiently, the Vcdiff data format supports the use of two different
   types of address caches. Both the encoder and decoder maintain these
   caches, so that decoder's caches remain synchronized with the
   encoder's caches.

   a. A "near" cache is an array with "s_near" slots, each containing an
      address used for encoding addresses nearby to previously encoded
      addresses (in the positive direction only). The near cache also
      maintains a "next_slot" index to the near cache. New entries to
      the near cache are always inserted in the next_slot index, which
      maintains a circular buffer of the s_near most recent addresses.

   b. A "same" cache is an array with "s_same", with a multiple of 256
      slots, each containing an address. The same cache maintains a
      hash table of recent addresses used for repeated encoding of the
      exact same address.

   By default, the parameters s_near and s_same are respectively set to
   4 and 3. An encoder MAY modify these values, but then it MUST encode
   the new values in the encoding itself, as discussed in Section 7, so
   that the decoder can properly set up its own caches.

   At the start of processing a target window, an implementation
   (encoder or decoder) initializes all of the slots in both caches to
   zero. The next_slot pointer of the near cache is set to point to
   slot zero.

Each time a COPY instruction is processed by the encoder or decoder,
the implementation's caches are updated as follows, where "addr" is
the address in the COPY instruction.

a. The slot in the near cache referenced by the next_slot index is
   set to addr. The next_slot index is then incremented modulo
   s_near.

b. The slot in the same cache whose index is addr%(s_same*256) is set
   to addr. [We use the C notations of % for modulo and * for
   multiplication.]

5.2 Example code for maintaining caches

To make clear the above description, below are examples of cache data
structures and algorithms to initialize and update them:

```
typedef struct _cache_s
{
    int*  near;      /* array of size s_near         */
    int   s_near;
    int   next_slot; /* the circular index for near */
    int*  same;      /* array of size s_same*256     */
    int   s_same;
} Cache_t;

cache_init(Cache_t* ka)
{
    int   i;

    ka->next_slot = 0;
    for(i = 0; i < ka->s_near; ++i)
        ka->near[i] = 0;

    for(i = 0; i < ka->s_same*256; ++i)
        ka->same[i] = 0;
}

cache_update(Cache_t* ka, int addr)
{
    if(ka->s_near > 0)
    {   ka->near[ka->next_slot] = addr;
        ka->next_slot = (ka->next_slot + 1) % ka->s_near;
    }

    if(ka->s_same > 0)
        ka->same[addr % (ka->s_same*256)] = addr;
}
```

5.3 Encoding of COPY instruction addresses

   The address of a COPY instruction is encoded using different modes,
   depending on the type of cached address used, if any.

   Let "addr" be the address of a COPY instruction to be decoded and
   "here" be the current location in the target data (i.e., the start of
   the data about to be encoded or decoded). Let near[j] be the jth
   element in the near cache, and same[k] be the kth element in the same
   cache. Below are the possible address modes:

      VCD_SELF: This mode has value 0. The address was encoded by
         itself as an integer.

      VCD_HERE: This mode has value 1. The address was encoded as the
         integer value "here - addr".

      Near modes: The "near modes" are in the range [2,s_near+1]. Let m
         be the mode of the address encoding. The address was encoded
         as the integer value "addr - near[m-2]".

      Same modes: The "same modes" are in the range
         [s_near+2,s_near+s_same+1]. Let m be the mode of the encoding.
         The address was encoded as a single byte b such that "addr ==
         same[(m - (s_near+2))*256 + b]".

5.4 Example code for encoding and decoding of COPY instruction addresses

   We show example algorithms below to demonstrate the use of address
   modes more clearly. The encoder has the freedom to choose address
   modes, the sample addr_encode() algorithm merely shows one way of
   picking the address mode. The decoding algorithm addr_decode() will
   uniquely decode addresses, regardless of the encoder's algorithm
   choice.

   Note that the address caches are updated immediately after an address
   is encoded or decoded. In this way, the decoder is always
   synchronized with the encoder.

```
int addr_encode(Cache_t* ka, int addr, int here, int* mode)
{
    int  i, d, bestd, bestm;

    /* Attempt to find the address mode that yields the
     * smallest integer value for "d", the encoded address
     * value, thereby minimizing the encoded size of the
     * address. */

    bestd = addr; bestm = VCD_SELF;      /* VCD_SELF == 0 */

    if((d = here-addr) < bestd)
        { bestd = d; bestm = VCD_HERE; } /* VCD_HERE == 1 */

    for(i = 0; i < ka->s_near; ++i)
        if((d = addr - ka->near[i]) >= 0 && d < bestd)
            { bestd = d; bestm = i+2; }

    if(ka->s_same > 0 && ka->same[d = addr%(ka->s_same*256)] == addr)
        { bestd = d%256; bestm = ka->s_near + 2 + d/256; }

    cache_update(ka,addr);

    *mode = bestm; /* this returns the address encoding mode */
    return  bestd; /* this returns the encoded address       */
}
```

Note that the addr_encode() algorithm chooses the best address mode
using a local optimization, but that may not lead to the best
encoding efficiency because different modes lead to different
instruction encodings, as described below.

The functions addrint() and addrbyte() used in addr_decode(), obtain
from the "Addresses section for COPYs" (Section 4.3), an integer or a
byte, respectively. These utilities will not be described here. We
simply recall that an integer is represented as a compact variable-
sized string of bytes, as described in Section 2 (i.e., base 128).

```
    int addr_decode(Cache_t* ka, int here, int mode)
    {   int   addr, m;

        if(mode == VCD_SELF)
            addr = addrint();
        else if(mode == VCD_HERE)
            addr = here - addrint();
        else if((m = mode - 2) >= 0 && m < ka->s_near) /* near cache */
            addr = ka->near[m] + addrint();
        else /* same cache */
        {   m = mode - (2 + ka->s_near);
            addr = ka->same[m*256 + addrbyte()];
        }

        cache_update(ka, addr);

        return addr;
    }
```

5.4 Instruction Codes

   Matches are often short in lengths and separated by small amounts of
   unmatched data. That is, the lengths of COPY and ADD instructions
   are often small. This is particularly true of binary data such as
   executable files or structured data, such as HTML or XML. In such
   cases, compression can be improved by combining the encoding of the
   sizes and the instruction types, as well as combining the encoding of
   adjacent delta instructions with sufficiently small data sizes.
   Effective choices of when to perform such combinations depend on many
   factors including the data being processed and the string matching
   algorithm in use. For example, if many COPY instructions have the
   same data sizes, it may be worthwhile to encode these instructions
   more compactly than others.

   The Vcdiff data format is designed so that a decoder does not need to
   be aware of the choices made in encoding algorithms. This is
   achieved with the notion of an "instruction code table", containing
   256 entries. Each entry defines, either a single delta instruction
   or a pair of instructions that have been combined. Note that the
   code table itself only exists in main memory, not in the delta file
   (unless using an application-defined code table, described in Section
   7). The encoded data simply includes the index of each instruction
   and, since there are only 256 indices, each index can be represented
   as a single byte.

Each instruction code entry contains six fields, each of which is a
single byte with an unsigned value:

```
+------------------------------------------------+
| inst1 | size1 | mode1 | inst2 | size2 | mode2 |
+------------------------------------------------+
```

Each triple (inst,size,mode) defines a delta instruction. The
meanings of these fields are as follows:

   inst: An "inst" field can have one of the four values: NOOP (0),
         ADD (1), RUN (2) or COPY (3) to indicate the instruction
         types. NOOP means that no instruction is specified. In
         this case, both the corresponding size and mode fields will
         be zero.

   size: A "size" field is zero or positive. A value zero means that
         the size associated with the instruction is encoded
         separately as an integer in the "Instructions and sizes
         section" (Section 6). A positive value for "size" defines
         the actual data size. Note that since the size is
         restricted to a byte, the maximum value for any instruction
         with size implicitly defined in the code table is 255.

   mode: A "mode" field is significant only when the associated delta
         instruction is a COPY. It defines the mode used to encode
         the associated addresses. For other instructions, this is
         always zero.

5.6 The Code Table

   Following the discussions on address modes and instruction code
   tables, we define a "Code Table" to have the data below:

         s_near: the size of the near cache,
         s_same: the size of the same cache,
         i_code: the 256-entry instruction code table.

   Vcdiff itself defines a "default code table" in which s_near is 4 and
   s_same is 3. Thus, there are 9 address modes for a COPY instruction.
   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.

	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]

The default instruction code table is depicted above, in a compact
representation that we use only for descriptive purposes. See
section 7 for the specification of how an instruction code table is
represented in the Vcdiff encoding format. In the depiction, a zero
value for size indicates that the size is separately coded. The mode
of non-COPY instructions is represented as 0, even though they are
not used.

In the depiction, each numbered line represents one or more entries
in the actual instruction code table (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 are covered by that line. (The
notation [i,j] means values from i through j, inclusively.) The
first 6 columns of a line in the depiction, describe the pairs of
instructions used for the corresponding index value(s).

If a line in the depiction includes a column entry using the [i,j]
notation, this means that the line is instantiated for each value in
the range from i to j, inclusively. The notation "0, [i,j]" means
that the line is instantiated for the value 0 and for each value in
the range from i to j, inclusively.

   If a line in the depiction includes more than one entry using the
   [i,j] notation, implying a "nested loop" to convert the line to a
   range of table entries, the first such [i,j] range specifies the
   outer loop, and the second specifies the inner loop.

   The below examples should make clear the above description:

   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.

   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.)

   Following the single ADD instructions are the single COPY
   instructions ordered by their address encoding modes. For example,
   line 11 shows the COPY instructions with mode 8, i.e., the last of
   the same cache. In this case, the COPY instruction with size field 0
   has index 147. Again, the actual size of this instruction will be
   coded separately.

   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 0 (VCD_SELF) and sizes 4, 5 and 6
   respectively.

   The last line, line 21, shows the eight instruction pairs, where the
   first instruction is a COPY and the second is an ADD. In this case,
   all COPY instructions have size 4 with mode ranging from 0 to 8 and
   all the ADD instructions have size 1. Thus, the entry with the
   largest index 255 combines a COPY instruction of size 4 and mode 8
   with an ADD instruction of size 1.

   The choice of the minimum size 4 for COPY instructions in the default
   code table was made from experiments that showed that excluding small
   matches (less then 4 bytes long) improved the compression rates.

6. Decoding a Target Window

   Section 4.3 discusses that the delta instructions and associated data
   are encoded in three arrays of bytes:

        Data section for ADDs and RUNs,
        Instructions and sizes section, and
        Addresses section for COPYs.

   Further, these data sections may have been further compressed by some
   secondary compressor. Assuming that any such compressed data has
   been decompressed so that we now have three arrays:

        inst: bytes coding the instructions and sizes.
        data: unmatched data associated with ADDs and RUNs.
        addr: bytes coding the addresses of COPYs.

   These arrays are organized as follows:

      inst: a sequence of (index, [size1], [size2]) tuples, where
            "index" is an index into the instruction code table, and
            size1 and size2 are integers that MAY or MAY NOT be included
            in the tuple as follows. The entry with the given "index"
            in the instruction code table potentially defines two delta
            instructions. If the first delta instruction is not a
            VCD_NOOP and its size is zero, then size1 MUST be present.
            Otherwise, size1 MUST be omitted and the size of the
            instruction (if it is not VCD_NOOP) is as defined in the
            table. The presence or absence of size2 is defined
            similarly with respect to the second delta instruction.

      data: a sequence of data values, encoded as bytes.

      addr: a sequence of address values. Addresses are normally
            encoded as integers as described in Section 2 (i.e., base
            128). However, since the same cache emits addresses in the
            range [0,255], same cache addresses are always encoded as a
            single byte.

   To summarize, each tuple in the "inst" array includes an index to
   some entry in the instruction code table that determines:

   a. Whether one or two instructions were encoded and their types.

   b. If the instructions have their sizes encoded separately, these
      sizes will follow, in order, in the tuple.

   c. If the instructions have accompanying data, i.e., ADDs or RUNs,
      their data will be in the array "data".

   d. Similarly, if the instructions are COPYs, the coded addresses are
      found in the array "addr".

   The decoding procedure simply processes the arrays by reading one
   code index at a time, looking up the corresponding instruction code
   entry, then consuming the respective sizes, data and addresses
   following the directions in this entry. In other words, the decoder
   maintains an implicit next-element pointer for each array;
   "consuming" an instruction tuple, data, or address value implies
   incrementing the associated pointer.

   For example, if during the processing of the target window, the next
   unconsumed tuple in the inst array has an index value of 19, then the
   first instruction is a COPY, whose size is found as the immediately
   following integer in the inst array. Since the mode of this COPY
   instruction is VCD_SELF, the corresponding address is found by
   consuming the next integer in the addr array. The data array is left
   intact. As the second instruction for code index 19 is a NOOP, this
   tuple is finished.

7. APPLICATION-DEFINED CODE TABLES

   Although the default code table used in Vcdiff is good for general
   purpose encoders, there are times when other code tables may perform
   better. For example, to code a file with many identical segments of
   data, it may be advantageous to have a COPY instruction with the
   specific size of these data segments, so that the instruction can be
   encoded in a single byte. Such a special code table MUST then be
   encoded in the delta file so that the decoder can reconstruct it
   before decoding the data.

   Vcdiff allows an application-defined code table to be specified in a
   delta file with the following data:

         Size of near cache   - byte
         Size of same cache   - byte
         Compressed code table data

   The "compressed code table data" encodes the delta between the
   default code table (source) and the new code table (target) in the
   same manner as described in Section 4.3 for encoding a target window
   in terms of a source window. This delta is computed using the
   following steps:

   a. Convert the new instruction code table into a string, "code", of
      1536 bytes using the below steps in order:

        i. Add in order the 256 bytes representing the types of the first
           instructions in the instruction pairs.
       ii. Add in order the 256 bytes representing the types of the
           second instructions in the instruction pairs.
      iii. Add in order the 256 bytes representing the sizes of the first
           instructions in the instruction pairs.
       iv. Add in order the 256 bytes representing the sizes of the
           second instructions in the instruction pairs.
        v. Add in order the 256 bytes representing the modes of the first
           instructions in the instruction pairs.
       vi. Add in order the 256 bytes representing the modes of the
           second instructions in the instruction pairs.

   b. Similarly, convert the default code table into a string "dflt".

   c. Treat the string "code" as a target window and "dflt" as the
      corresponding source data and apply an encoding algorithm to
      compute the delta encoding of "code" in terms of "dflt". This
      computation MUST use the default code table for encoding the delta
      instructions.

   The decoder can then reverse the above steps to decode the compressed
   table data using the method of Section 6, employing the default code
   table, to generate the new code table. Note that the decoder does
   not need to know about the details of the encoding algorithm used in
   step (c). It is able to decode the new code table because the Vcdiff
   format is independent from the choice of encoding algorithm, and
   because the encoder in step (c) uses the known, default code table.

8. Performance

   The encoding format is compact. For compression only, using the LZ-
   77 string parsing strategy and without any secondary compressors, the
   typical compression rate is better than Unix compress and close to
   gzip. For differencing, the data format is better than all known
   methods in terms of its stated goal, which is primarily decoding
   speed and encoding efficiency.

   We compare the performance of compress, gzip and Vcdiff using the
   archives of three versions of the Gnu C compiler, gcc-2.95.1.tar,
   gcc-2.95.2.tar and gcc-2.95.3.tar. Gzip was used at its default
   compression level. The Vcdiff data were obtained using the
   Vcodex/Vcdiff software (Section 13).

Below are the different Vcdiff runs:

Vcdiff: vcdiff is used as a compressor only.

Vcdiff-d: vcdiff is used as a differencer only. That is, it only
    compares target data against source data. Since the files
    involved are large, they are broken into windows. In this
    case, each target window, starting at some file offset in the
    target file, is compared against a source window with the same
    file offset (in the source file). The source window is also
    slightly larger than the target window to increase matching
    opportunities.

Vcdiff-dc: This is similar to Vcdiff-d, but vcdiff can also
    compare target data against target data as applicable. Thus,
    vcdiff both computes differences and compresses data. The
    windowing algorithm is the same as above. However, the above
    hint is recinded in this case.

Vcdiff-dcw: This is similar to Vcdiff-dc but the windowing
    algorithm uses a content-based heuristic to select a source
    window that is more likely to match with a given target window.
    Thus, the source data segment selected for a target window
    often will not be aligned with the file offsets of this target
    window.

	gcc-2.95.1	gcc-2.95.2	gcc-2.95.3
1. raw size	55,746,560	55,797,760	55,787,520
2. compress	-	19,939,390	19,939,453
3. gzip	-	12,973,443	12,998,097
4. Vcdiff	-	15,358,786	15,371,737
5. Vcdiff-d	-	100,971	26,383,849
6. Vcdiff-dc	-	97,246	14,461,203
7. Vcdiff-dcw	-	256,445	1,248,543

The above table shows the raw sizes of the tar files and the sizes of
the compressed results. The differencing results in the gcc-2.95.2
column were obtained by compressing gcc-2.95.2, given gcc-2.95.1.
The same results for the column gcc-2.95.3 were obtained by
compressing gcc-2.95.3, given gcc-2.95.2.

Rows 2, 3 and 4 show that, for compression only, the compression rate
from Vcdiff is worse than gzip and better than compress.

The last three rows in the column gcc-2.95.2 show that when two file
versions are very similar, differencing can give dramatically good
compression rates. Vcdiff-d and Vcdiff-dc use the same simple window
selection method of aligning by file offsets, but Vcdiff-dc also does
compression so its output is slightly smaller. Vcdiff-dcw uses a
content-based algorithm to search for source data that likely will
match a given target window. Although it does a good job, the
algorithm does not always find the best matches, which in this case,
are given by the simple algorithm of Vcdiff-d. As a result, the
output size for Vcdiff-dcw is slightly larger.

The situation is reversed in the gcc-2.95.3 column. Here, the files
and their contents were sufficiently rearranged or changed between
the making of the gcc-2.95.3.tar archive and the gcc-2.95.2 archive
so that the simple method of aligning windows by file offsets no
longer works. As a result, Vcdiff-d and Vcdiff-dc do not perform
well. By allowing compression, along with differencing, Vcdiff-dc
manages to beat Vcdiff-c, which does compression only. The content-
based window matching algorithm in Vcdiff-dcw is effective in
matching the right source and target windows so that Vcdiff-dcw is
the overall winner.

9. Further Issues

This document does not address a few issues:

Secondary compressors:
   As discussed in Section 4.3, certain sections in the delta
   encoding of a window may be further compressed by a secondary
   compressor. In our experience, the basic Vcdiff format is
   adequate for most purposes so that secondary compressors are
   seldom needed. In particular, for normal use of data
   differencing, where the files to be compared have long stretches
   of matches, much of the gain in compression rate is already
   achieved by normal string matching. Thus, the use of secondary
   compressors is seldom needed in this case. However, for
   applications beyond differencing of such nearly identical files,
   secondary compressors may be needed to achieve maximal compressed
   results.

   Therefore, we recommend leaving the Vcdiff data format defined as
   in this document so that the use of secondary compressors can be
   implemented when they become needed in the future. The formats of
   the compressed data via such compressors or any compressors that
   may be defined in the future are left open to their
   implementations. These could include Huffman encoding, arithmetic
   encoding, and splay tree encoding [8,9].

Large file system vs. small file system:
    As discussed in Section 4, a target window in a large file may be
    compared against some source window in another file or in the same
    file (from some earlier part). In that case, the file offset of
    the source window is specified as a variable-sized integer in the
    delta encoding. There is a possibility that the encoding was
    computed on a system supporting much larger files than in a system
    where the data may be decoded (e.g., 64-bit file systems vs. 32-
    bit file systems). In that case, some target data may not be
    recoverable. This problem could afflict any compression format,
    and ought to be resolved with a generic negotiation mechanism in
    the appropriate protocol(s).

10. Summary

   We have described Vcdiff, a general and portable encoding format for
   compression and differencing. The format is good in that it allows
   implementing a decoder without knowledge of the encoders. Further,
   ignoring the use of secondary compressors not defined within the
   format, the decoding algorithms run in linear time and requires
   working space proportional to window size.

11. Acknowledgements

   Thanks are due to Balachander Krishnamurthy, Jeff Mogul and Arthur
   Van Hoff who provided much encouragement to publicize Vcdiff. In
   particular, Jeff helped in clarifying the description of the data
   format presented here.

12. Security Considerations

   Vcdiff only provides a format to encode compressed and differenced
   data. It does not address any issues concerning how such data are,
   in fact, stored in a given file system or the run-time memory of a
   computer system. Therefore, we do not anticipate any security issues
   with respect to Vcdiff.

13. Source Code Availability

   Vcdiff is implemented as a data transforming method in Phong Vo's
   Vcodex library. AT&T Corp. has made the source code for Vcodex
   available for anyone to use to transmit data via HTTP/1.1 Delta
   Encoding [10,11]. The source code and according license is
   accessible at the below URL:

      http://www.research.att.com/sw/tools

14. Intellectual Property Rights

   The IETF has been notified of intellectual property rights claimed in
   regard to some or all of the specification contained in this
   document. For more information consult the online list of claimed
   rights, at <http://www.ietf.org/ipr.html>.

   The IETF takes no position regarding the validity or scope of any
   intellectual property or other rights that might be claimed to
   pertain to the implementation or use of the technology described in
   this document or the extent to which any license under such rights
   might or might not be available; neither does it represent that it
   has made any effort to identify any such rights. Information on the
   IETF's procedures with respect to rights in standards-track and
   standards-related documentation can be found in BCP 11. Copies of
   claims of rights made available for publication and any assurances of
   licenses to be made available, or the result of an attempt made to
   obtain a general license or permission for the use of such
   proprietary rights by implementors or users of this specification can
   be obtained from the IETF Secretariat.

15. IANA Considerations

   The Internet Assigned Numbers Authority (IANA) administers the number
   space for Secondary Compressor ID values. Values and their meaning
   must be documented in an RFC or other peer-reviewed, permanent, and
   readily available reference, in sufficient detail so that
   interoperability between independent implementations is possible.
   Subject to these constraints, name assignments are First Come, First
   Served - see RFC 2434 [13]. Legal ID values are in the range 1..255.

   This document does not define any values in this number space.

16. References

   [1] D.G. Korn and K.P. Vo, Vdelta: Differencing and Compression,
        Practical Reusable Unix Software, Editor B. Krishnamurthy, John
        Wiley & Sons, Inc., 1995.

   [2] J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data
        Compression, IEEE Trans. on Information Theory, 23(3):337-343,
        1977.

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

[4] E.M. McCreight, A Space-Economical Suffix Tree Construction
     Algorithm, Journal of the ACM, 23:262-272, 1976.

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

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

[7] D.G. Korn, K.P. Vo, Sfio: A buffered I/O Library, Proc. of the
     Summer '91 Usenix Conference, 1991.

[8] D. W. Jones, Application of Splay Trees to Data Compression,
     CACM, 31(8):996:1007.

[9] M. Nelson, J. Gailly, The Data Compression Book, ISBN 1-55851-
     434-1, M&T Books, New York, NY, 1995.

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

[11] Mogul, J., Krishnamurthy, B., Douglis, F., Feldmann, A., Goland,
     Y. and A. Van Hoff, "Delta Encoding in HTTP", RFC 3229, January
     2002.

[12] Bradner, S., "Key words for use in RFCs to Indicate Requirement
     Levels", BCP 14, RFC 2119, March 1997.

[13] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA
     Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[14] D.G. Korn and K.P. Vo, Engineering a Differencing and
     Compression Data Format, Submitted to Usenix'2002, 2001.

17. Authors' Addresses

   Kiem-Phong Vo (main contact)
   AT&T Labs, Room D223
   180 Park Avenue
   Florham Park, NJ 07932

   Phone: 1 973 360 8630
   EMail: kpv@research.att.com

   David G. Korn
   AT&T Labs, Room D237
   180 Park Avenue
   Florham Park, NJ 07932

   Phone: 1 973 360 8602
   EMail: dgk@research.att.com

   Jeffrey C. Mogul
   Western Research Laboratory
   Hewlett-Packard Company
   1501 Page Mill Road, MS 1251
   Palo Alto, California, 94304, U.S.A.

   Phone: 1 650 857 2206 (email preferred)
   EMail: JeffMogul@acm.org

   Joshua P. MacDonald
   Computer Science Division
   University of California, Berkeley
   345 Soda Hall
   Berkeley, CA 94720

   EMail: jmacd@cs.berkeley.edu

18. Full Copyright Statement

Acknowledgement