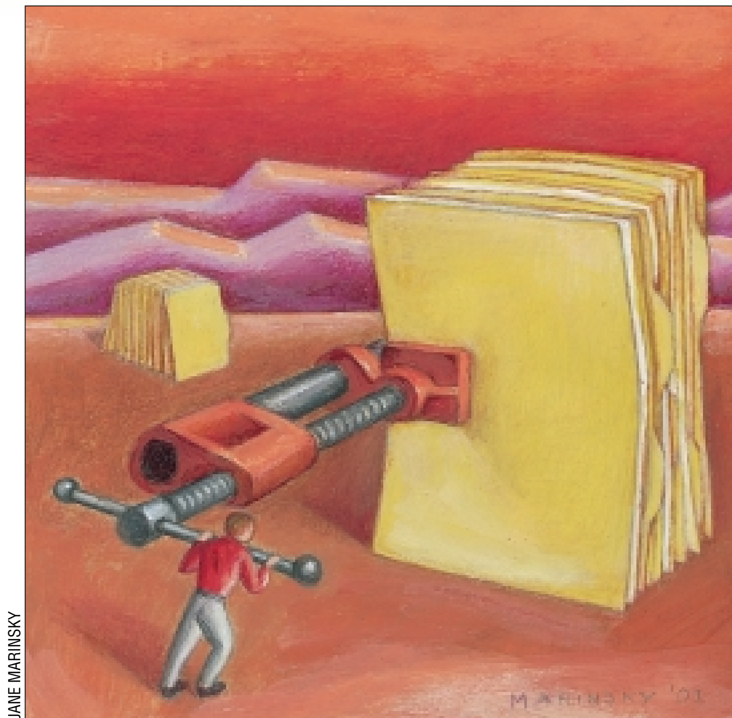
Work

by Jeffreys Copeland and Haemer



JANE MARINSKY

In small proportions
we just beauties see, /
And in short measures
life may perfect be.
– Ben Jonson, "To the
Immortal Memory of
Sir Lucius Cary and
Sir Henry Morison"

In the universe great acts
are made up of small deeds /
The sage does not attempt
anything very big, /
And thus achieves greatness.
– Lau Tsu, *Tao Te Ching*

Squishing Data

Leafing through a book on Linux system administration, we have learned that many compressed kernels are now `bzimage` instead of the older `zimage`, because they're now compressed with `bzip2`.

`bzip2`?

One wonderful thing about UNIX is that there's always some useful little corner of it that we haven't visited. We'd never heard of `bzip2`; discovering it gave us a few pleasant hours playing with compression.

Abstract Compressionism

A little voyage into abstract thinking about compression lands you squarely on an interesting conclusion: compression algorithms don't compress.

Let's do a gedanken experiment. Imagine that you have a compression filter that reads and writes a byte stream (I/O is done in bytes). You also have a decompression filter–the compression is reversible. Finally, someone has given you a collection of all possible files two bytes or shorter. This means 256 distinct, one-byte files, and 65,536 distinct, two-byte files.

OK, run each two-byte file through the compression filter. First, the 256 one-byte files can't be compressed. You can't get shorter than a byte. Second, only 256 of the two-byte files could compress. If more compressed into a single byte, two files would have the same one-byte output and the compression wouldn't be reversible. Finally, every two-byte file that compresses into a single byte removes a possible compression target for a one-byte file.

This same reasoning extends to larger files. No matter what filter you use, for every file it compresses, there is another file that the same filter will expand.

So why do compression algorithms seem to work? Only because they're designed to preferentially compress the files we want to compress.

To help see that, let's build an example. Here's a trivial run-length-compression filter, which compresses runs of characters into a NUL + a count plus the character.

```
/* $Id: rl.c,v 1.6 … jsh Exp $ */
#include <stdio.h>

#define NUL '\0'

main() {
   int c, runc;
   int n = 1;

  for (runc = getchar();
runc != EOF; runc=c) {
     c = getchar();
     if ((c == runc) &&
(n < 256))
       n++;
     else if (n==1)
       putchar(runc);
    else {
      putchar(NUL);
      putchar(n);
      putchar(runc);
      n = 1;
    }
  }
}
```

And here's the corresponding decompression filter:

```
/* $Id: url.c,v 1.1 2001/01/06 … jsh Exp $ */
#include <stdio.h>

#define NUL '\0'

main() {
   int n, c, runc;

   while((c = getchar()) != EOF) {
       if (c != NUL)
         putchar(c);
  else {
        n = getchar();
        runc = getchar();
            while(n-- > 0)
        putchar(runc);
     }
  }
}
```

Now let's put it to work:

```
$ echo -n 'aaaabbbb' | rl | od -bc
0000000 000 004 141 000 004 142
  \0 004   a  \0 004   b
0000006
```

Is this a useful compression scheme? Sometimes.

```
$ man unexpand | wc -c
2240
$ man unexpand | rl | wc -c
1991
$ echo 1991/2240 | bc -l
.88883928571428571428
```

(Here, and later, we'll use `wc -c` to count characters and `bc -l` to do floating-point calculations; the `-l` flag to `bc` tells it to use an arbitrary-precision math library. Without this flag, the division of two integers would do an integer divide and give us the answer "0." Both are good command line filters for playing around like this.)

Compressing the man page for unexpand gives us about an 11% space savings. But not everything compresses as nicely.

```
$ wc -c url.c
 303 url.c
$ rl < url.c | wc -c
 298
$ wc -c rl.c
 366 rl.c
$ rl < rl.c | wc -c
 366
$ rl < rl.c | diff - rl.c
Binary files - and rl.c differ
$ rl < url | wc -c
  11001
$ wc -c url
  11861 url
```

So the source for our decompression filter, `url`, barely compresses the source for our compression filter itself, `rl` doesn't change in size at all (even though it does change in content!), and the binary for the decompression filter actually grows by about 8%.

Much worse, the careful reader will have noticed that our simple-minded compression scheme will mishandle input files that contain embedded `NUL` (' ') characters. Compressing the three-character file '\0', 'X','Y' will leave it unchanged. The decompressor, thinking the NUL is a flag, will produce a file of 88 'Y' characters. ('X' in ASCII is decimal 88.) To fix this, we'd need to add code to escape our special character. So, our simple-minded compressor is OK for ASCII, but not great for binaries.

Squishy History

When someone mentions early compression schemes, many of you will think of `compress/uncompress` (or maybe even `pack/unpack`), which was used for compression on AT&T's System V UNIX releases.

The Open Group's Single UNIX Specification man page for `pack` (`http://www.opengroup.org/onlinepubs/007908799/xcu/pack.html`), notes that "Typically, text files are reduced to 60-75% of their original size. Object files, which use a larger character set and have a more uniform distribution of characters, show little compression, the packed versions typically being about 90% of the original size."

Real old-timers will, instead, think of `unexpand/expand`, which used the radical idea of saving space by turning runs of fixed numbers of blanks into a special character: the tab. This is not to be sneezed at.

```
$ wc -c rl.c
 366 rl.c
$ expand rl.c | wc -c
 443
$ echo 366/443 | bc -l
.82618510158013544018
```

Notice that using tabs instead of spaces saved us more space than our run-length encoding did. The savings are independent (we ran our `rl` filter on the version with tabs), but a surprising number of non-UNIX systems (and even now, a few badly run UNIX development organizations that we've passed through) require all leading whitespace be blanks.

Here, too, the filter is very specific: `unexpand` is tailored for ASCII sources with a lot of leading blanks. Still, being able to compress your source and text files by 20% or so can help a lot.

Binary

So how about compression schemes that aren't ASCII-specific? Let's look at four that are widely available: `compress`, `zip`, `gzip`, and `bzip2`.

The oldest, `compress`, implements Lempel-Ziv-Welch (LZW) compression. This is a particular flavor of the LZ family of compression algorithms also used by `zip` and `gzip`. Two things make it special: (1) It was the first widely distributed, general compression algorithm in the UNIX world; (2) It was patented by Unisys.

The first made it successful. The second killed it off. In the

early '80s, everyone used `compress` to squish files and ship them around. The popular image format, GIF, used LZW. The program `tar` even added a `-Z` flag to compress and uncompress its output automatically: `tar -Zcvf foo.tar.Z`

By the early '80s, the IEEE 1003.2 Committee considered adding it to the POSIX shell-and-tools standard. Then Unisys started talking about "liberal licensing terms." A lot of people, including corporations like IBM and Sun, had trouble with that. To see whether you would, too, take a look at `http://www.unisys.com/unisys/lzw/`.

Unisys hasn't helped the situation by changing their internal policy on this patent several times. They only have a limited amount of time to extract money from the technology, though: the original Lempel-Ziv patent expires on Aug. 7 this year, and the LZW patent expires next December.

The FSF (Free Software Foundation) was also uncomfortable; however, it actually did something: it produced and began giving away a better LZ-compression program named `gzip`.

```
$ man compress | wc -c
8435
$ man compress | compress | wc -c
4082
$ man compress | gzip | wc -c
3177
```

`tar` added a `-z` flag, and the world was quickly filled with `.tar.z` (or `.tgz`) files–gzipped archives.

In the Microsoft world, a similar program, `PKZIP`, developed by the late Phil Katz, spread around. It combines the functions of `tar` and `gzip`, though it doesn't provide either separately. On UNIX systems, the `zip/unzip` pair can produce and interpret this format. It's not quite as efficient as `gzip`. If you think for a moment, you'll realize that `zip/unzip` can't be as efficient as separate `tar/gzip` because in the former the compression is done to each file in the archive, not to the archive as a whole. This means that you don't get the economies of collapsing common sequences in the larger file because there is extra header information to provide the `tar`-like functionality.

```
$ man compress | zip | wc -c
   adding: - (deflated 63%)
3275
```

And how about `bzip2`? Built around an algorithm from a different family, Burroughs-Wheeler compression, `bzip2` is even more efficient.

```
$ man compress | bzip2 | wc -c
3050
```

The man page says that `bzip2` is faster and produces files typically 10-15% smaller than `gzip`.

As with `compress` and `gzip`, there's even a `tar` flag, `-I`, to produce and recognize `bzip2` files on-the-fly.

But does it always compress? We know the answer is no. But how can we look at this conveniently? Let's just feed it random bit patterns and look at the size change.

First, we need a random file of a known size. We can get that by using `dd` to copy from `/dev/urandom` to a scratch file.

```
$ dd if=/dev/urandom ibs=1024k/
count=1 > /tmp/random_bits
1+0 records in
2048+0 records out
$ wc -c /tmp/random_bits
1048576 /tmp/random_bits
```

The file `/dev/urandom`, contains random bit patterns generated by the kernel's random-number generator. A cousin of `/dev/null`, it will return as many random bits as you ask for. See *random*(4) man page for more details. We use the command `dd` to grab the number of bits we want–one 1,024k block.

We suggest you not let the output go to your screen, because arbitrary bit patterns can do very odd things to terminal emulators. If you're curious about what random noise looks like, you can peruse `/tmp/random_bits` with either a text editor or `od`, both of which convert the bytes to safer representations.

We tried running `/tmp/random bits` through `strings`, (which extracts all printable strings), and then discovered ours contained the complete works of Shakespeare, but only the first time. Every other time it looked like it was either random garbage, or the complete works of Shakespeare–in Klingon.

Now let's use that random file to see how good the compression schemes are. Here's a shell script to test them.

```
1  #!/bin/sh
2  # $Id: compression ratios,v 1.3 … jsh
   Exp jsh  $

3  # find the size of the compressed file
4  csize() {
5   dd if=$source ibs=${kb}k/
          count=1 2>/dev/null |
6   $filter |
7   wc -c
8  }

9  # calculate how much we compressed
10  cratio() {
11    echo "
12    scale=4
13      $(csize)/(${kb}*1024)
14    " |
15    bc -l
16  }

17  # calculate compression ratios
18  # for $kb Kb file,
19  # $samples times
20  # for each filter
21  source=${1:-/dev/urandom}
22  kb=${2:-1}
23  nsamples=${3:-5}
```

```
24  for filter in compress 'zip -q' gzip bzip2
25  do
26    echo $filter:
27    for j in $(seq 1 $nsamples)
28    do
29      echo -ne '\t'
30      cratio $filter
31    done
32  done
```

This may require a little explanation.

Lines 1 and 2 are our usual: line 1 tells the kernel that this is a POSIX shell script; line 2 is a bookkeeping line for RCS, because we keep everything under source-code control. We think these two lines are a must for professional shell programming.

Lines 4 to 8 are a shell function, `csize`, that generates a random file of a fixed size, compresses it, and returns the number of characters in the compressed file. Line 5 generates the file (in the same way we did earlier, though we've parameterized the file size), line 6 compresses it, and line 7 gives us the character count.

Lines 9 to 16 are a second function, `cratio`, that calculates the ratio of compressed file size to original file size. Here, we're using `echo` to feed a full program to `bc -l`. Line 12 truncates output to four decimal places, and line 13 says to divide the size of the compressed file by the original file size. We put the program in double quotes to protect `bc`'s syntactic elements, such as ')', from the shell.

Lines 21, 22, and 23 set parameters used elsewhere in the program: the source of the input bits, the number of kilobytes to use from the source, and the number of times to try each filter. Each can be set from the command line, but has a default value. For example, the source is the first command-line argument, but the first argument is missing (or empty), the default source to compress is `/dev/urandom`.

The remainder of the program is a pair of nested loops. The inner loop, lines 27 to 31, prints the compression ratio for `$nsamples` (default = 5) samples from `$source`. Line 29 indents this output by a tab stop to make it easier to read. The outer loop does this for each compression filter. (The `zip` filter chatters needlessly to standard error, and we shut it up with its `-q` switch.) So, running this, what can we see? First, with the defaults.

```
$ compression_ratios
compress:
  1.2441
  1.2470
  1.2402
  1.2441
  1.2412
zip -q:
  1.1181
  1.1181
  1.1181
  1.1181
  1.1181
gzip:
  1.0224
  1.0224
  1.0224
  1.0224
  1.0224
bzip2:
  1.2958
  1.2832
  1.2919
  1.2832
  1.2880
```

That's correct. All four compression schemes are actually expanding the random, 1 Kb bit patterns that they get from `/dev/urandom`. The `gzip` algorithm seems to do the best, and the `bzip2` algorithm the worst. Increasing the size of the input to a megabyte dramatically improves the performance of everything but `compress`.

```
$ compression_ratios /dev/urandom 1024
compress:
  1.2377
  1.2382
  1.2383
  1.2389
  1.2398
zip -q:
  1.0002
  1.0002
  1.0002
  1.0002
  1.0002
gzip:
  1.0001
  1.0001
  1.0001
  1.0001
  1.0001
bzip2:
  1.0045
  1.0046
  1.0045
  1.0046
  1.0046
```

A quick look with `od` makes the reason clear. Using `gzip` as our example gives us the results found in Listing 1 (Page 36).

These filters are smart enough not to try to compress random data; they simply wrap the original file in a small header and trailer. The larger the input file, the lower the overhead.

In contrast, `bzip2` is actually doing a little work–the ratios vary a little from run to run–but not too much. The overhead on a 1 MB file of random data is typically around 5,000 bytes.

So what are these filters good for? Ordered data. Instead of taking input from `/dev/urandom`, let's take it from `/dev/zero`, which emits a stream of all zeroes.

```
$ compression_ratios /dev/zero 1024
compress:
   .0017
   .0017
   .0017
   .0017
   .0017
zip -q:
   .0010
   .0010
   .0010
   .0010
   .0010
gzip:
   .0010
   .0010
   .0010
   .0010
   .0010
bzip2:
   0
   0
   0
   0
   0
```

Wait … the ratio for `bzip` is *zero*? Remember, we only asked for four significant digits.

```
dd if=/dev/zero ibs=1024k count=1 2>/dev/null |
bzip2 | wc -c
45
```

In other words, `bzip2` turns a megabyte of zeroes into a 45 byte file.

Further

The field of data compression is rich. On the other hand, we didn't really write this column to teach you about data compression: we wrote it to show you how easily we could amuse ourselves using UNIX tools to expand our knowledge. Until a week ago, we'd never heard of `bzip2`. The first few folks we talked to about this had never even heard of either `/dev/urandom` or `/dev/zero`. There may be other things in here that are new to you, too. Meanwhile, here are some sites you can go to for further reading about the specific algorithms:

- gzip: `http://www.gzip.org`; `http://www.cdrom.com/pub/infozip/zlib/`.
- zip: `http://ftp.freesoftware.com/pub/infozip/ index.html/`.
- bzip2: `ftp://sourceware.cygnus.com/pub/bzip2/docs/manual_toc.html/`.

For both `gzip` and `bzip2` there are libraries you can link into your program to call the associated functions directly.

Until next time, happy trails.

Jeffrey Copeland *(`copeland@alumni.caltech.edu`) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer *(`jsh@usenix.org`) works at Minolta-QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work` or alternately at `ftp://ftp.cpg.com/pub/Work`.

Listing 1. Results of *gzip*

```
$ od -c /tmp/random_bits | head
0000000 331   q   %   _   H 264   b   m   , 373 315 370 221 354 330 252
0000020   ~   b 245  \v 275   \ 341  \n 362 304 205 212  \v 261   B 020
0000040 200   ( 345   9 336   2 244   a   e 327   @ 370 333 377 373   0
0000060   \ 221 352 006  \r 177 326 301 227 337   C 205   d 020  \b 001
0000100 367 272  \0 024 376 352 200   A 345 226  \r 037   |   x 312   \
0000120 202 025 326 245 226 355 232   s 373   B 355 333   2   ^ 312   M
0000140 020 031   | 232  \b 341   V   * 016   g   t   3   I 210 020 325
0000160 210   { 331 335 333   t 342   J 306   q 207 343 305 243   E   s
0000200 237   Y 304 370   b 003 332 322 363   1  \0 276 253 251   ?   k
0000220   g   u 204       W 234   ` 222 230 263 017   h 346   4 364   S
$ gzip < /tmp/random_bits | od -c | head
0000000 037 213  \b  \0 020 001   Y   :  \0 003 001  \0 004 377 373 331
0000020   q   %   _   H 264   b   m   , 373 315 370 221 354 330 252   ~
0000040   b 245  \v 275   \ 341  \n 362 304 205 212  \v 261   B 020 200
0000060   ( 345   9 336   2 244   a   e 327   @ 370 333 377 373   0   \
0000100 221 352 006  \r 177 326 301 227 337   C 205   d 020  \b 001 367
0000120 272  \0 024 376 352 200   A 345 226  \r 037   |   x 312   \ 202
0000140 025 326 245 226 355 232   s 373   B 355 333   2   ^ 312   M 020
0000160 031   | 232  \b 341   V   * 016   g   t   3   I 210 020 325 210
0000200   { 331 335 333   t 342   J 306   q 207 343 305 243   E   s 237
0000220   Y 304 370   b 003 332 322 363   1  \0 276 253 251   ?   k   g
```