On Exploiting gzip's Content-Dependent Compression

Matteo Franzil, Luis Augusto Dias Knob

1. Introduction

Despite the development of more recent compression techniques like `bzip2` and `xz`, `gzip` is still a well-liked UNIX compression tool. Gzip is employed as the default compression technique in the majority of Linux and UNIX distributions. Although it has a somewhat poor compression ratio when compared to other utilities, this can be justified by its simplicity and quick compression and decompression times [1], [3]. This is also true for some tools, like `containerd`, which employs gzip as its standard compression technique for its image layers [4], [5].

In this report, we explore the possibility of exploiting gzip's algorithm for artificially increasing the decompression time of a file. By creating files filled with semi-random data generated with various methods, we show that compression and decompression times may vary significantly depending on the content of the file and the compression level used. Indeed, we show that the decompression time of a file can be increased by a factor of 3 in the worst case, when compared to the decompression time of a file containing English text.

2. Results

2.1. System setup

We run our experiments on a machine with a 4-core Intel Xeon Silver 4112 CPU @ 2.60GHz, 64GB of RAM, running Ubuntu Server 20.04.2 LTS. We used `gzip` version 1.12 on both machines. Tests were run in isolation and with CPU pinning, in order to reduce the impact of other processes on the results, and were run 5 times to reduce the impact of noise.

Our tests comprised the following steps:

1) Generate a file of 100MB in size, with a specific content
2) Compress the file with gzip
3) Decompress the file with gzip

We measured the time taken by each step, the size of the compressed and uncompressed files, the CPU usage, and the compression ratios. We repeated this test for each of `gzip`'s compression levels (1-9).

2.2. Popular tools

We first started by analyzing the compression and decompression times of files generated with some popular random data generators. We generated files with the following tools [2]:

- `od --format=x /dev/urandom | head -c 1G`
- `base64 /dev/urandom | head -c 1G`
- `cat /dev/urandom | tr -dc 'a-zA-Z0-9' | head -c 1G`
- `openssl rand -out myfile "$( echo 1G | numfmt --from=iec )"`
- `lorem -c 1000000000`

We then compressed and decompressed these files with gzip, using the methods described above. The decompression times are shown in Figure 1.

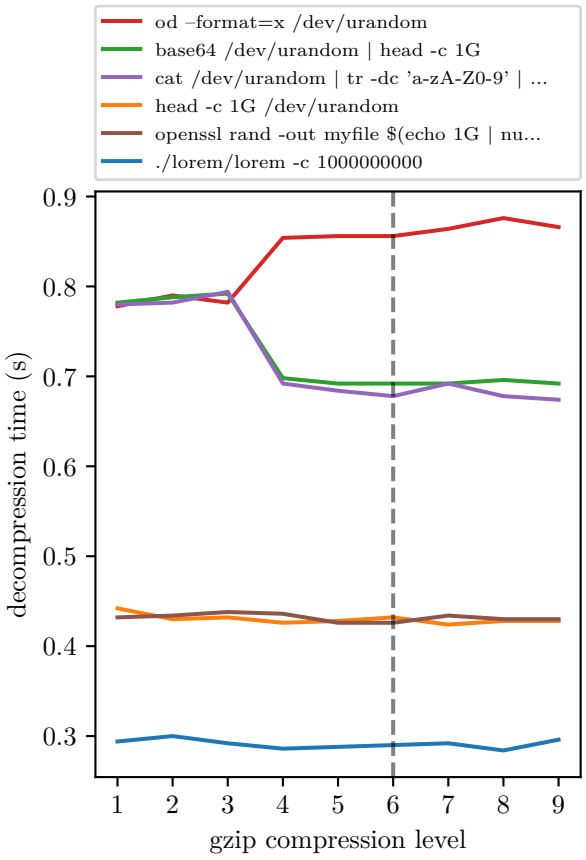

Figure 1: Decompression times for files generated with popular tools.

We can see that the decompression times vary significantly depending on the tool used to generate the file. For example, the file generated by `lorem`, containing English text, is decompressed in 3 seconds. `openssl` and direct `/dev/urandom` output are decompressed in 4.5 seconds. Finally, the files generated by `base64` and `tr` require between 7 and 8 seconds

to be decompressed, depending on the compression level used. Finally, `od`'s output is decompressed in 9 seconds.

2.3. `od`'s output

Wishing to understand why `od`'s output was decompressed slower than the others, we decided to fully leverage `od`'s various output formats. We thus generated files with the following output formats:

- `x` (hexadecimal)
- `a`, `c` (ASCII both named and unnamed)
- `d1`, `d2`, `d4`, `d8` (decimal)
- `f` (floating point)
- `o` (octal)
- `u1`, `u2`, `u4`, `u8` (unsigned decimal)

We decided to use the decimal and unsigned decimal format variants with 1, 2, 4, and 8 bytes in order to see if the size of the numbers (and the minus sign in decimal formats) had any impact on the decompression time. It must be remembered we voluntarily un-padded the results of `od`, and thus, the files are comprised of a single line of content with no spaces or newlines.

We repeated the same tests as before, and the results are shown in Figure 2, although with 100MB files instead of 1GB files, due to the long time required to generate the files.

We can see that the decompression times vary significantly depending on the output format chosen. For our nefarious purposes, the best output formats (at level 6, the default one) are `a` and `c` (ASCII named and unnamed), although `x` almost matches them.

2.4. Finding the optimal file

Finally, the graph in Figure 3 shows the decompression times for `od`'s best output formats along with the other tools.

2.5. Other results

To verify that our results were not specific to our machine, we also ran the same tests on a MacBook Pro (late 2020) with an M1 CPU, 16GB of RAM, and running macOS Ventura 13.3.1 (a). We used the same version of `gzip` as on Ubuntu (1.12). Results were similar, although the MacBook Pro was on average faster than the Ubuntu machine. We thus do not include the results in this report.

Furthermore, we tried various file sizes, between 100MB and 1GB, in order to verify there was also no correlation between the file size and the results. Again, we found that the results were comparable, and thus we only reported the results once for the 100MB files.

3. Conclusions

We have shown that the decompression time of a file can vary significantly depending on the content

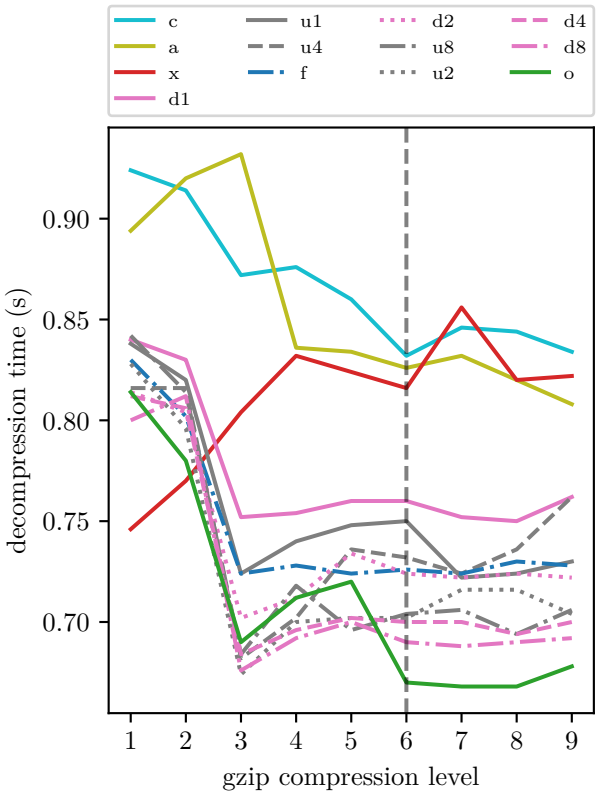

Figure 2: Decompression times for files generated with od 's various output formats.

of the file and the compression level used. Indeed, we have shown that the decompression time of a file can be increased by a factor of 3 when files are filled with datafrom `od`'s `a` and `c` output formats, when compared to the decompression time of a file containing English text.

We believe that this is a serious issue, as it can be used to artificially increase the decompression time of a file, which can be used to slow down systems that rely on gzip for decompression. For example, this could be used to slow down the unpacking of Docker images, which use gzip for compression of the various layers.

4. Bibliography

References

[1] L. P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," Internet Engineering Task Force, Request for Comments RFC 1951, May 1996. DOI: 10.17487/RFC1951. [Online]. Available: https://datatracker.ietf.org/doc/rfc1951 (visited on 05/11/2023).

[2] P. E. Strandberg, *Lorem*, Dec. 2022. [Online]. Available: https://github.com/per9000/lorem (visited on 05/11/2023).

[3] *GNU Gzip*. [Online]. Available: https://www.gnu.org/software/gzip/manual/gzip.html#index-options-4 (visited on 05/11/2023).

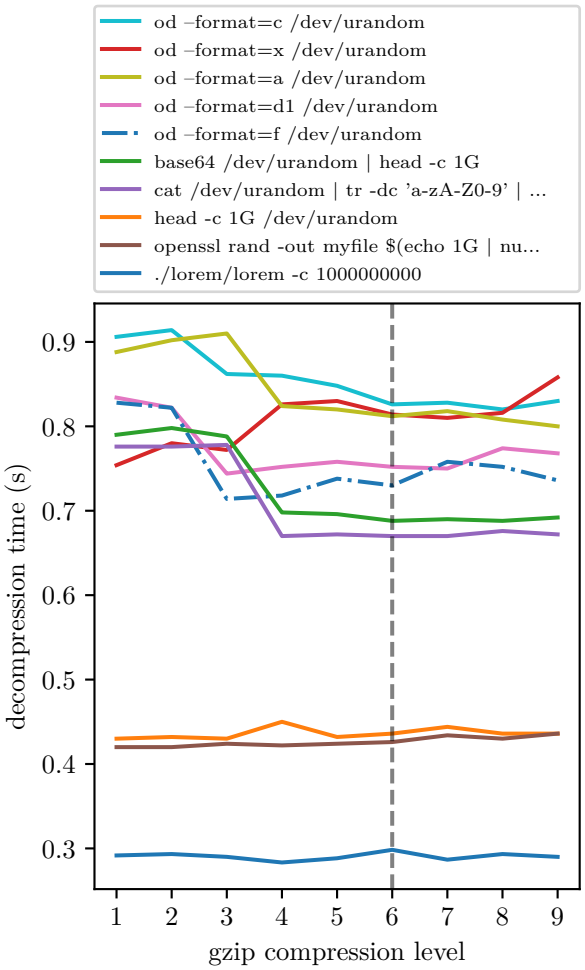

Figure 3: Decompression times for files generated with od 's best output formats and the other tools.

[4] *Make image (layer) downloads faster by using pigz by sargun · Pull Request #35697 · moby/moby.* [Online]. Available: https://github.com/moby/moby/pull/35697 (visited on 05/11/2023).

[5] *Support parallel decompression (pigz) by mxpv · Pull Request #2640 · containerd/containerd.* [Online]. Available: https://github.com/containerd/containerd/pull/2640 (visited on 05/11/2023).

5. Appendix

The following are the full results of our tests for the last graph in the previous section.

TABLE 1: **od –format=c ...**

Level	Compr. ratio	Compr. size	Compr. time	Decompr. time
1	0.540	58M	1.720	0.906
2	0.531	57M	1.950	0.914
3	0.522	56M	3.120	0.862
4	0.513	55M	2.860	0.860
5	0.513	55M	5.270	0.848
6	0.504	54M	9.230	0.826
7	0.504	54M	9.390	0.828
8	0.504	54M	9.420	0.820
9	0.504	54M	9.440	0.830

TABLE 2: **od –format=x ...**

Level	Compr. ratio	Compr. size	Compr. time	Decompr. time
1	0.590	56M	1.170	0.754
2	0.590	56M	1.550	0.780
3	0.580	55M	1.870	0.772
4	0.580	55M	2.500	0.826
5	0.580	55M	3.040	0.830
6	0.580	55M	3.050	0.814
7	0.580	55M	3.020	0.810
8	0.580	55M	3.040	0.816
9	0.580	55M	3.100	0.858

TABLE 3: **od –format=a ...**

Level	Compr. ratio	Compr. size	Compr. time	Decompr. time
1	0.717	77M	2.240	0.888
2	0.717	77M	2.430	0.902
3	0.717	77M	2.840	0.910
4	0.717	77M	3.050	0.824
5	0.717	77M	4.650	0.820
6	0.717	77M	5.500	0.812
7	0.717	77M	6.430	0.818
8	0.717	77M	6.340	0.808
9	0.717	77M	6.330	0.800

TABLE 4: **od –format=d1 ...**

Level	Compr. ratio	Compr. size	Compr. time	Decompr. time
1	0.442	48M	1.240	0.834
2	0.442	48M	1.480	0.822
3	0.425	46M	2.800	0.744
4	0.425	46M	2.230	0.752
5	0.407	44M	5.000	0.758
6	0.398	43M	8.480	0.752
7	0.407	44M	9.590	0.750
8	0.398	43M	11.000	0.774
9	0.407	44M	11.120	0.768

TABLE 5: **od –format=f ...**

Level	Compr. ratio	Compr. size	Compr. time	Decompr. time
1	0.469	51M	1.240	0.828
2	0.460	50M	1.480	0.822
3	0.442	48M	2.490	0.714
4	0.442	48M	2.200	0.718
5	0.434	47M	4.350	0.738
6	0.434	47M	5.530	0.730
7	0.434	47M	6.340	0.758
8	0.434	47M	6.280	0.752
9	0.434	47M	6.120	0.736

TABLE 6: **base64 /dev/ur ...**

Level	Compr. ratio	Compr. size	Compr. time	Decompr. time
1	0.810	77M	3.220	0.790
2	0.810	77M	3.230	0.798
3	0.810	77M	3.200	0.788
4	0.810	77M	3.670	0.698
5	0.810	77M	3.670	0.696
6	0.810	77M	3.670	0.688
7	0.810	77M	3.660	0.690
8	0.810	77M	3.630	0.688
9	0.810	77M	3.730	0.692

TABLE 7: **cat /dev/urand ...**

Level	Compr. ratio	Compr. size	Compr. time	Decompr. time
1	0.717	77M	3.040	0.776
2	0.717	77M	3.150	0.776
3	0.717	77M	3.140	0.778
4	0.717	77M	3.620	0.670
5	0.717	77M	3.640	0.672
6	0.717	77M	3.640	0.670
7	0.717	77M	3.640	0.670
8	0.717	77M	3.630	0.676
9	0.717	77M	3.630	0.672

TABLE 8: **head -c 1G /de ...**

Level	Compr. ratio	Compr. size	Compr. time	Decompr. time
1	1.130	108M	2.100	0.430
2	1.130	108M	2.110	0.432
3	1.130	108M	2.130	0.430
4	1.130	108M	2.300	0.450
5	1.130	108M	2.300	0.432
6	1.130	108M	2.260	0.436
7	1.130	108M	2.320	0.444
8	1.130	108M	2.270	0.436
9	1.130	108M	2.250	0.436

TABLE 9: **openssl rand ...**

Level	Compr. ratio	Compr. size	Compr. time	Decompr. time
1	1.130	108M	2.110	0.420
2	1.130	108M	2.100	0.420
3	1.130	108M	2.110	0.424
4	1.130	108M	2.220	0.422
5	1.130	108M	2.200	0.424
6	1.130	108M	2.210	0.426
7	1.130	108M	2.210	0.434
8	1.130	108M	2.220	0.430
9	1.130	108M	2.650	0.436

TABLE 10: **./lorem ...**

Level	Compr. ratio	Compr. size	Compr. time	Decompr. time
1	0.014	1M	0.330	0.292
2	0.014	1M	0.320	0.293
3	0.009	801k	0.300	0.290
4	0.005	465k	0.390	0.283
5	0.005	465k	0.390	0.288
6	0.005	465k	0.380	0.298
7	0.005	465k	0.380	0.287
8	0.005	465k	0.380	0.293
9	0.005	465k	0.380	0.290