Acta Cybernetica **24** (2019) 121–129.

LZ based Compression Benchmark on PE Files

Zsombor Paróczi[a]

Abstract

The key element in runtime compression is the compression algorithm itself, that is used during processing. It has to be small in enough in decompression bytecode size to fit in the final executable, yet have to provide the best possible compression ratio. In our work we benchmark the top LZ based compression methods on Windows PE (both EXE and DLL) files, and present the results including the decompression overhead and the compression rates.

Keywords: lz based compression, compression benchmark, PE benchmark

1 Introduction

During runtime executable compression an already compiled executable is modified in ways, that it still retains the ability to execute, yet the transformation produces smaller file size. The transformations usually exists from multiple steps, changing the structure of the executable by removing unused bytes, adding a compression layer or modifying the code itself. During the code modifications the actual bytecode can change, or remain the same depending on the modification.

In the world of x86 (or even x86-64) PE compression there are only a few benchmarks, since the ever growing storage capacity makes this field less important. Yet in new fields, like IOT and wearable electronics every application uses some kind of compression, Android apk-s are always compressed by a simple gzip compression. There are two mayor benchmarks for PE compression available today, the Maximum Compression benchmark collection [1] includes two PE files, one DLL and one EXE, and the PE Compression Test [2] has four EXE files. We will use the 5 EXE files PE files during our benchmark, referred as *small corpus*. For more detailed results we have a self-collected corpus of 200 PE files, referred to as *large corpus*.

When approaching a new way to create executable compression, one should consider three main factors. The first is the actual compression rate of the algorithms, since it will have the biggest effect on larger files. The second is the overhead in terms of extra bytecode within the executable, since the decompression algorithm have to be included in the newly generated file, using large pre-generated dictionary is usually not an option. This is especially important for small (less than

[a]Budapest University of Technology and Economics, E-mail: paroczi@tmit.bme.hu

DOI: 10.14232/actacyb.24.1.2019.9

100kb) executables. The third factor has the lowest priority, but still important: the decompression speed. The decompression method should not require a lot of time to run, even on a resource limited machine. This eliminates whole families of compression methods, like neural network based (PAQ family) compressions.

```
83 e6 01          and   esi, 1
8d 3c 96          lea   edi, dword ptr [esi+edx*4]
39 44 b9 1c       cmp   dword ptr [ecx+edi*4+1ch], eax
75 77             jne   short L15176
Opcode+ModRM — Jump Offset — Displacement — SIB — Immediate
```

Figure 1: Annotated asm code

Split-stream methods are well-know in the executable compression world, these algorithms take advantage of the structural information of the bytecode itself, separating the opcode from all the modification flags. Each x86 instruction can be separated into multiple parts, prefix, opcode, mod r/m, etc., an annotated asm snippet can be seen on Figure 1. The idea behind split-stream is to annotate each byte by these parts, and collect them into one chunk. By doing this, each chunk can be compressed better due to local redundancies. During decompression the original bytecode is reconstructed using a small compiler. We used a reference implementation from the packer kkrunchy [6].

2 LZ based compression methods

LZ based compression methods (LZ77/LZSS/LZMA families) are well fitted for this compression task, since they usually have relatively small memory requirement (less than 64 Mb), they use Lempel-Ziv compression methods [3] and maybe some Huffman tables or hidden Markov model based approaches. These methods are simple algorithms, resulting in small size in terms of decompression bytecode. During the last few years there are a lot of new LZ based compression methods, the mayor ones are Zstandard (zstd) from Facebook and Zopfli from Google. The selected libraries can be seen on Table 1, these are the top LZ familly libraries for generic purpose compression regarding an extensive LZ benchmark [4,5].

The compression rates on generic dataset (non-code section of an executable) can be seen on Figure 2 and Table 2. All of these tests and results are in sync with the LZ benchmark mentioned previously, the only exception is Brotli which worked quite well on our dataset. Brotli, Lzlib and LZMA have the best compression ratio on average, followed by CSC, Zopfli and zstd. aPlib has the worst compression ratio, since it only implements a very simple LZ77 variant.

3 Decompression code

For each compression method the related library also supplies the decompression method as well. In most cases it's tightly coupled with the compression code, so

Table 1: Libraries used in the benchmark

Compression method	Version	Source
aPlib	1.1.1	http://ibsensoftware.com/products_aPLib.html
Lzlib	1.10	https://www.nongnu.org/lzip/lzlib.html
LZMA	9.35	https://www.7-zip.org/sdk.html
Zopfli	2017-07-07	https://github.com/google/zopfli
Zstandard	1.3.3	https://facebook.github.io/zstd/
CSC	2016-10-13	https://github.com/fusiyuan2010/CSC
Brotli	1.0.3	https://github.com/google/brotli

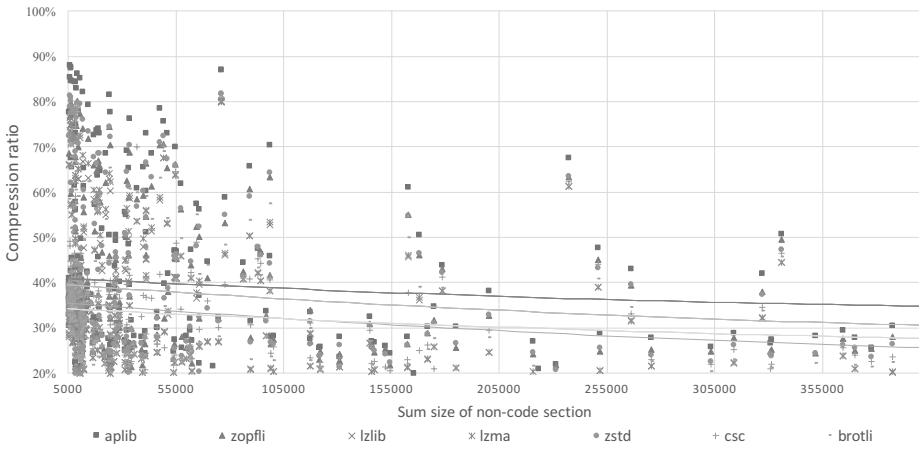

Figure 2: Compression rates on non-code section by input size

Table 2: Compression rates on non-code section

Method	aPlib	Zopfli	Lzlib	LZMA	zstd	CSC	Brotli
Rate	40.2%	37.2%	34.7%	34.2%	38.3%	34.6%	33.4%

in the first step we separated the compression method from the decompression one and created small executables which included only the decompression method and a sample from the compressed data, so we can verify that the decompression still works. All these LZ based compression libraries are written in C / C++, during this step we used GCC for ease of debugging. The aPlib library includes an ASM written decompression method which is already small enough, so we didn't do any modification on it. Lzlib, LZMA, zlib, zstd, CSC and Brotli are at some point use

dynamic memory allocation, Brotli and CSC has some other external dependencies. We opted to remove (or inline) all dependencies, since loading external DLLs or extra functions takes up more space than an inlined function. We managed to fully remove memory allocation from LZMA and Lzlib by simply creating a large chunk of zerofilled memory at the end of the executable, and absolutely referencing those with some pointers. In the other libraries we could inline some trivial functions (zerofill, memcpy) but due to the nature of those algorithms there are a lot of dynamic allocations that require external libraries. We also remove all error reporting functionality from the code, these are designed to detect if the compressed data is damaged in any way. All of the modifications were tested with multiple samples to retain the ability of decompressing compressed data.

Table 3: Decompression bytecode size

Compression method	Bytecode size	Compressed with aPlib
aPlib	150	-
Lzlib	7.168	3.943
LZMA	8.602	3.155
Zopfli	14.351	8.173
Zstandard	106.525	26.632
CSC	23.714	10.671
Brotli	215.665	92.736

GCC has several flags for optimizing for space, speed and even some internal optimization options are available, but after several failed attempts to make the pure decompression bytecode smaller, we started to experiment with other compilers. Clang and Microsoft Visual C++ compiler produced almost the same bytecode size, even with extra optimization options, but Watcom Compiler (Open Watcom 1.9) managed to create 10%-15% smaller bytecode than any of the other compilers (second best was gcc with size optimization flags). This is due to the fact that generic registers (registers storing and passing variables between functions) can be fine-tuned in Watcom, using esi, edi, ebp registers in the produced binaries. After several iterations of modifying the code, testing and compiling we managed to create really small sized decompression code for each library. We also noticed that compressing the various decompressing bytecodes with aPlib and decompressing them during runtime is a great way to create smaller sized binaries. The aPlib decompression bytecode is 150 bytes after all. Table 3 contains the bytecode size on both the decompression code bytesize as is, and the compressed decompression bytecode size. Also worth noting that Brotli can be compiled without the built-in dictionary, which results in 66.930 bytes (and 23.425 bytes compressed with aPlib), but the dictionary has huge benefits during compression / decompression. Any data compressed with Brotli with dictionary can only be decompressed, if the decompressor code also has the dictionary.

4 Benchmark

During the benchmark we constructed a system, which is capable of extracting different sections from the executables, apply split-stream and a compression method on it to create a well detailed benchmark result. During the benchmark we run each compression method on each section, then run each compression method with split-stream on executable sections. The benchmark system was created using C++ and Node.js, the Node.js part was responsible for the instrumentation of the compressions, the C++ part was responsible for extracting the section and verifying the modified decompression method we created. If there is any side effect from the decompression code modification explained in the third section, we are not seeing it.

5 Results: compression ratio

The detailed results for each test case on the *small corpus* can be seen on Figure 3. As you can see applying split-stream before the compression is useful in most of the cases (except for the smallest executable, which suffered from the overhead of this method - splitting 1 byte instructions into base instruction + mod flags). The rates for each compression varies between test cases, but Lzlib, LZMA, Brotli

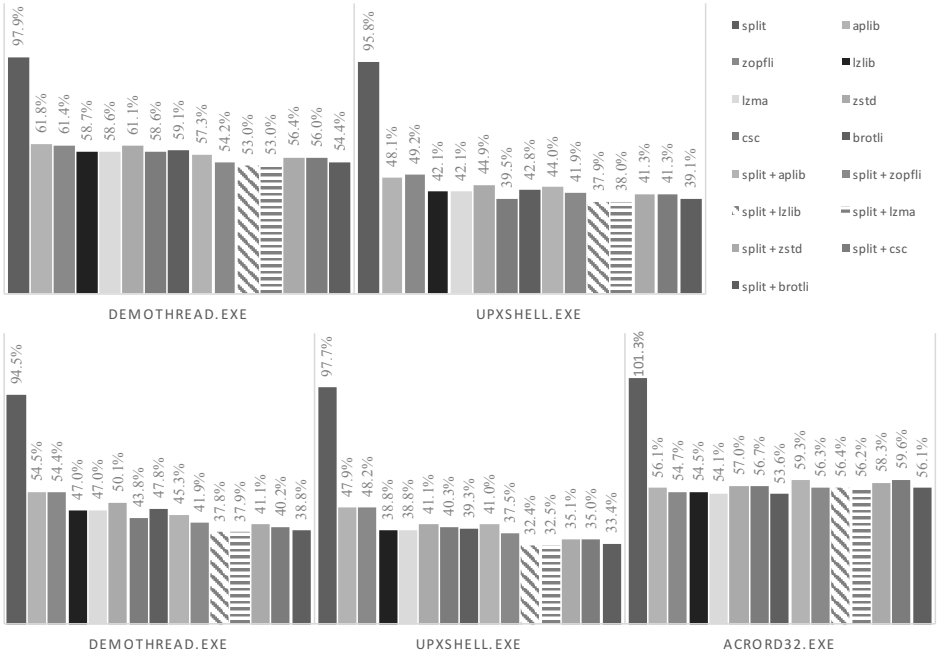

Figure 3: Resulting section size compered to the original on the *small corpus* files

are clearly the best for the *small corpus*, followed by zstd, CSC, Zopfli and aPlib. There is a constant improvement when using split-stream. Only for really small executable aPlib is the best, due to the simplicity of the algorithm itself. All of these results were verified during our *large corpus* benchmark.

The actual compression rates on the *large corpus* can be seen on Table 2 and 4 (split-stream is annotated as s). As you can see the ratio between each compression rate on average is really small, for code sections split-stream really helps. For code section LZMA, Lzlib and Brotli are the best, followed by Zopfli and CSC. For non-code section we had a larger variety of results, since the non-code sections can contain any datatype. The non-code section has a more loose structure and less density, the compression rates are higher. It is interesting, that Brotli is the winner in these tests, but as it turned out Brotli has a large dictionary prebuilt into the

Table 4: Average compression rates on code section

Compression method	compression rate
aPlib	47.0%
LZMA	42.1%
s + aPlib	44.3%
s + Zopfli	41.3%
s + Lzlib	39.6%
s + LZMA	39.5%
s + zstd	42.4%
s + CSC	42.0%
s + Brotli	40.0%

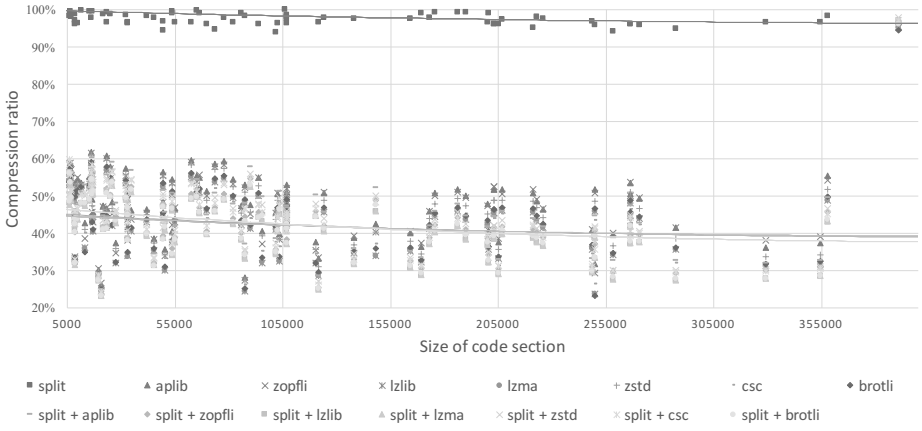

Figure 4: Compression rates vs file size the code section

algorithm, that helps with compressing text. LZMA, Lzlib, CSC produced just 1-2% lower rates, followed by zstd and Zopfli. Obviously aPlib was the worst in both tests, since it contains the most simple algorithm for compression. PE sections tend to be less then 3 Mb, the larger the section the more compression rate we can achieve.

6 Results: final file size with decompression bytecode

Since the decompression code has to be included in the final executable, we also benchmarked how the decompression overhead code effects the final file size. As you can see on Figure 4 for smaller executables the overhead is what really defines the final result. All of the decompression methods were packed with aPlib, since aPlib has a decompression code size of 150 bytes, and above 1.000 bytes it is better to compress the decompression code with aPlib. Some of the more complex methods (namely zstd, Brotli, CSC) has relatively large data tables in the decompression code. Same goes for the split-stream code, which is above 1kByte uncompressed, and 540 byte compressed with aPlib.

Our final results suggest, that there is no "golden" LZ based compression with split-stream method for all the executables.

You can see the best performing algorithm on Figure 6 for the *large corpus*. For smaller files a more detailed result of this can be seen on Figure 5. There is a clear

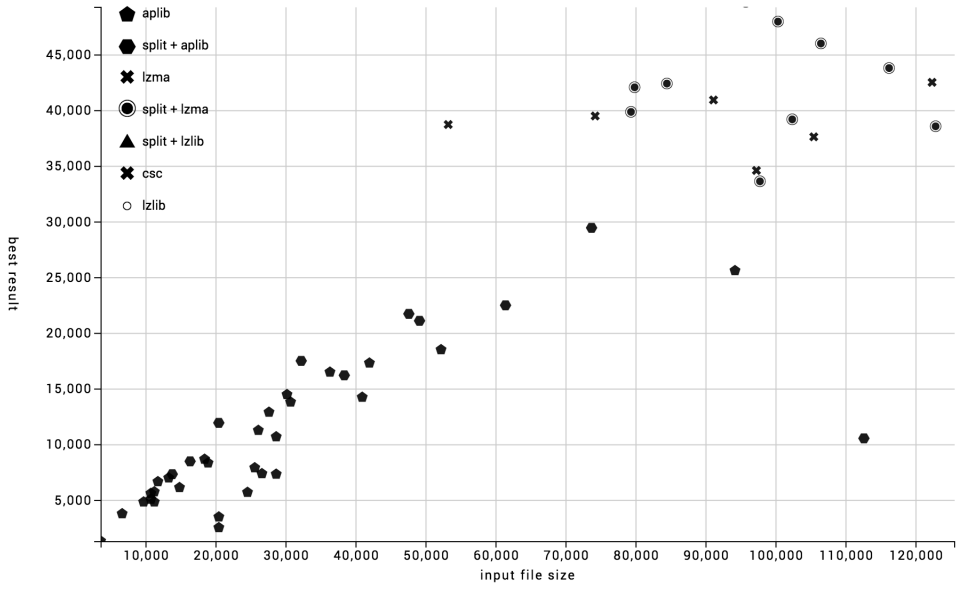

Figure 5: Raw and compressed file size using the best method on smaller files

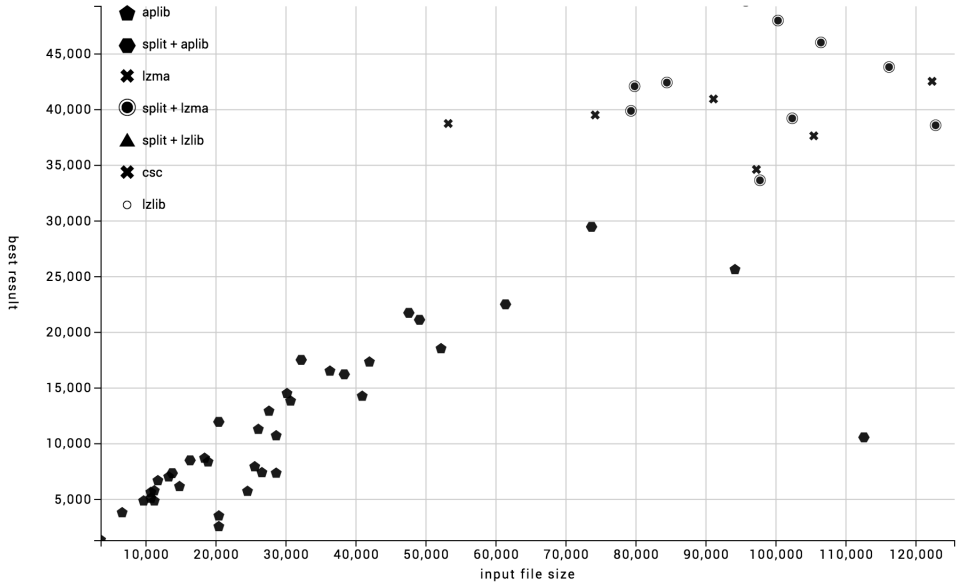

Figure 6: Raw and compressed file size using the best method on larger files

tendency, that some algorithms perform better on smaller files, partly due to the fact that the decompression code is small, and others perform well on larger files. Since smaller files tend to be more code section heavy, and larger files are more like a generic datafile (much more strings, xml, images within the sections), there is an interesting trend how each compression method behaves on different sized binaries.

We consider 3 categories based on the executable size: for small files (less than 50kB) size aPlib is the clear winner with 150 byte decompression code, maybe with split-stream if the executable section is large. For medium size (less than 500 kB) split-stream with aPlib or split-stream with LZMA (aPlib compressed) should be used. For larger files split-stream with LZMA (aPlib compressed) or split-stream with Lzlib (aPlib compressed) should be used.

For some special cases any combination can be the winner in the final compression size. CSC (without split-stream), Lzlib (without split-stream) and LZMA (without split-stream) can outperform the others in some cases.

7 Summary

By providing a good ruleset for choosing the right compression method or methods based on file size, we hope that future executable compression authors can improve the compression rate of their tools. Besides that we see a clear trend, that even LZ based compression libraries are getting more complex (dynamic memory allocations, large dictionary size, etc.), making small size, compact decompression bytecode

creation a lot harder. We provided our insights of how to make small decompression bytecode by simply modifying the decompression method in different ways, using different compilers and compressing the bytecode itself.

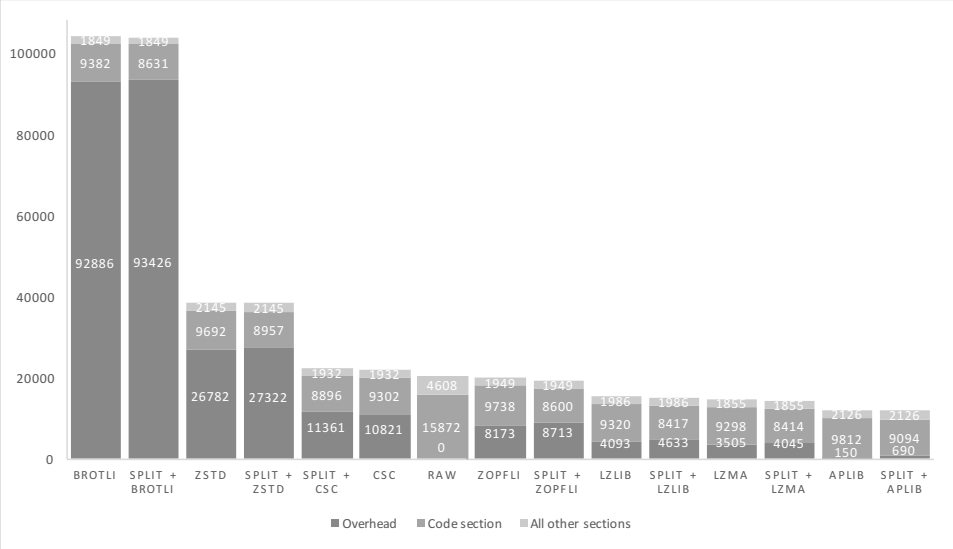

Figure 7: Final executable size example on DemoThread.exe (20kByte)

References

[1] Lossless data compression software benchmarks/comparisons. `https://www.maximumcompression.com/` (Visited 2018-03-04).

[2] PE Compression test by Ernani Weber. `http://pect.atspace.com/` (Visited 2018-03-04).

[3] Ziv, J. and Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. on Inf. Th. IT-23*, 337-343, 1977.

[4] LZbench. `https://github.com/inikep/lzbench/` (Visited 2018-03-04).

[5] Kunkel, Julian. SFS: A Tool for Large Scale Analysis of Compression Characteristics *Research Papers (4)*, Research Group: Scientific Computing, University of Hamburg

[6] Giesen, Fabian. Working with compression. *Breakpoint conference*, 2006.