

ROOT I/O compression algorithms

Oksana Shadura, Brian Bockelman
University of Nebraska-Lincoln



Introduction

Compression Algorithms

Compression algorithms



Lossy

Reduces size by permanently eliminating certain redundant information

Audio, video = removing small details to reduce data size



Impossible to restore back

Lossless

Statistical models can be used to generate codes for specific characters based on their probability of occurring, and assigning the shortest codes to the most common data. Common statistical methods: Entropy encoding, run-length encoding, compression using a dictionary.

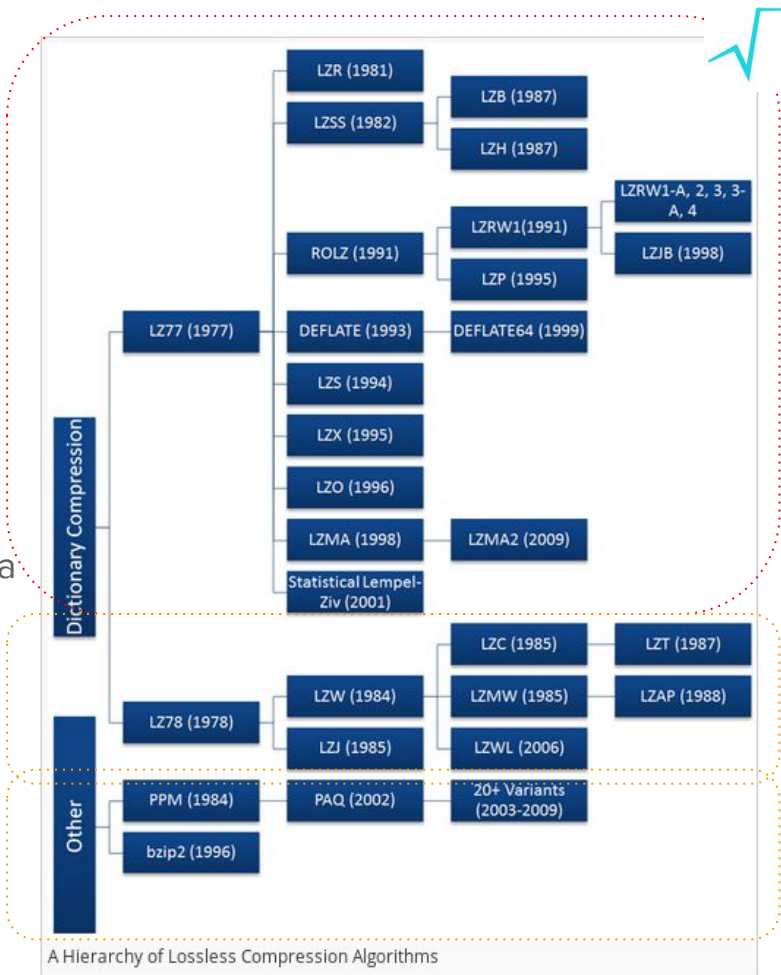
Txt, binaries, and etc. = only reduction of data size



Possible to restore back

History of lossless algorithms

- Morse code, invented in 1838, is the earliest instance of data compression:
 - Idea most common letters in the English language such as “e” and “t” are given shorter Morse codes.
- LZ77 algorithm (1977) - “LZ1”
 - First algorithm to use a dictionary to compress data
- LZ78 algorithm (1978) - “LZ2”





Lossless compression: LZ77 & LZ78

- Represent variable-length symbols with fixed-length codes.

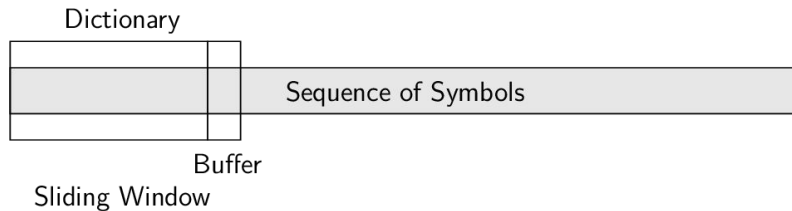


Figure 1. Sliding window concept

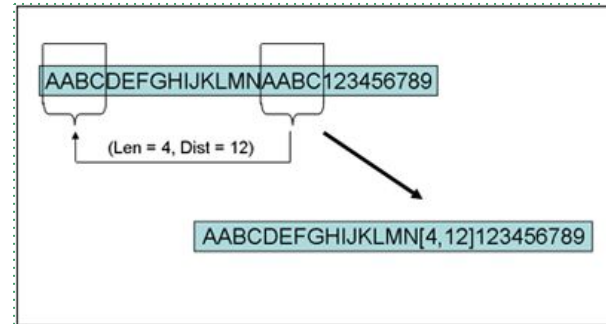


Figure 2. Deflate algorithm

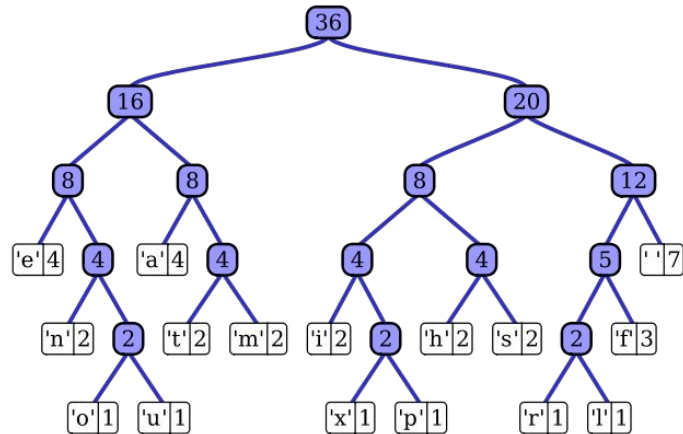
- LZ77 and LZ78 are theoretically **dictionary coders**
- LZ77 maintains a **sliding window during compression** [it is shown to be equivalent to the *explicit dictionary* constructed by LZ78 and they are only equivalent when the entire data is intended to be decompressed]. Since LZ77 encodes and decodes from a sliding window over previously seen characters, **decompression must always start at the beginning of the input.**
- **LZ78 decompression could allow random access** to the input only if the entire dictionary is known in advance.



Lossless compression: Huffman algorithm

- Represent fixed-length symbols with variable-length codes.

Huffman tree generated from text "this is an example of a huffman tree".



Output from Huffman's algorithm:
variable-length code table for encoding a
source symbol.

36x8bit = 180 bit
VS
147 using Huffman coding

https://en.wikipedia.org/wiki/Huffman_coding

Char ↕	Freq ↕	Code ↕
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010



ROOT compression algorithms

- **ZLIB** - LZ77 preprocessor *with Huffman coding* **{old ROOT default}**
- Old ROOT compression algorithm (backward compatibility)
- **LZMA** - LZMA uses a dictionary compression algorithm (a variant of LZ77 with huge dictionary sizes and special support for repeatedly used match distances), whose output is then encoded with a range encoder, using a complex model to make a probability prediction of each bit.
- **LZ4** - LZ77-type compressor with a fixed, byte-oriented encoding and *no Huffman coding pass* **{new ROOT default}**
- **ZSTD** - dictionary-type algorithm (LZ77) with large search window and fast implementations of entropy coding stage, using either very fast Finite State Entropy (tANS) or Huffman coding.

ROOT I/O concepts

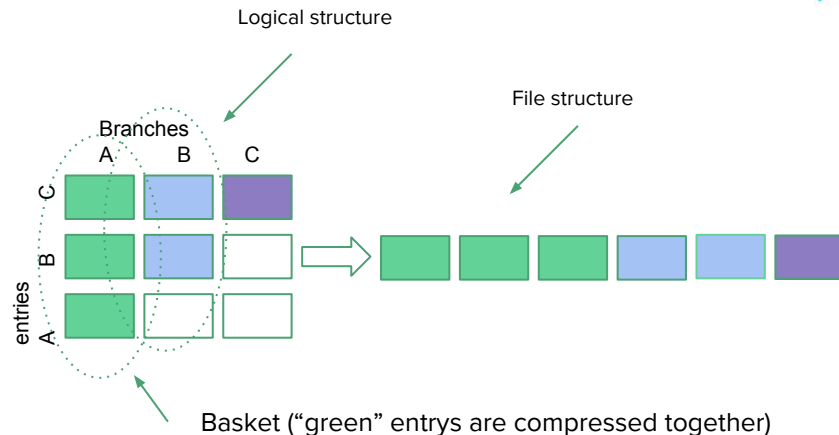
Entry: The entry is the atomic unit of work - **rows in a table**. Each entry is composed of multiple objects.

Branch: There are similar objects in each entry – these are organized as branches - like **columns in a table**.

Basket: When serialized, ROOT writes (and compresses) the objects in the same branch – and from many contiguous entries – into a basket.

Cluster: All the data from a group of entries is written contiguously as part of an entry cluster.

B.Bockelman and Z.Zhang [ACAT2017]



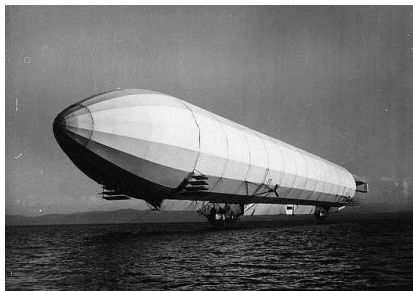
How works ROOT TTree compression?

- **Each basket is compressed and written to the ROOT file;**

ROOT TTree compression is not a trivial task!



LZ4

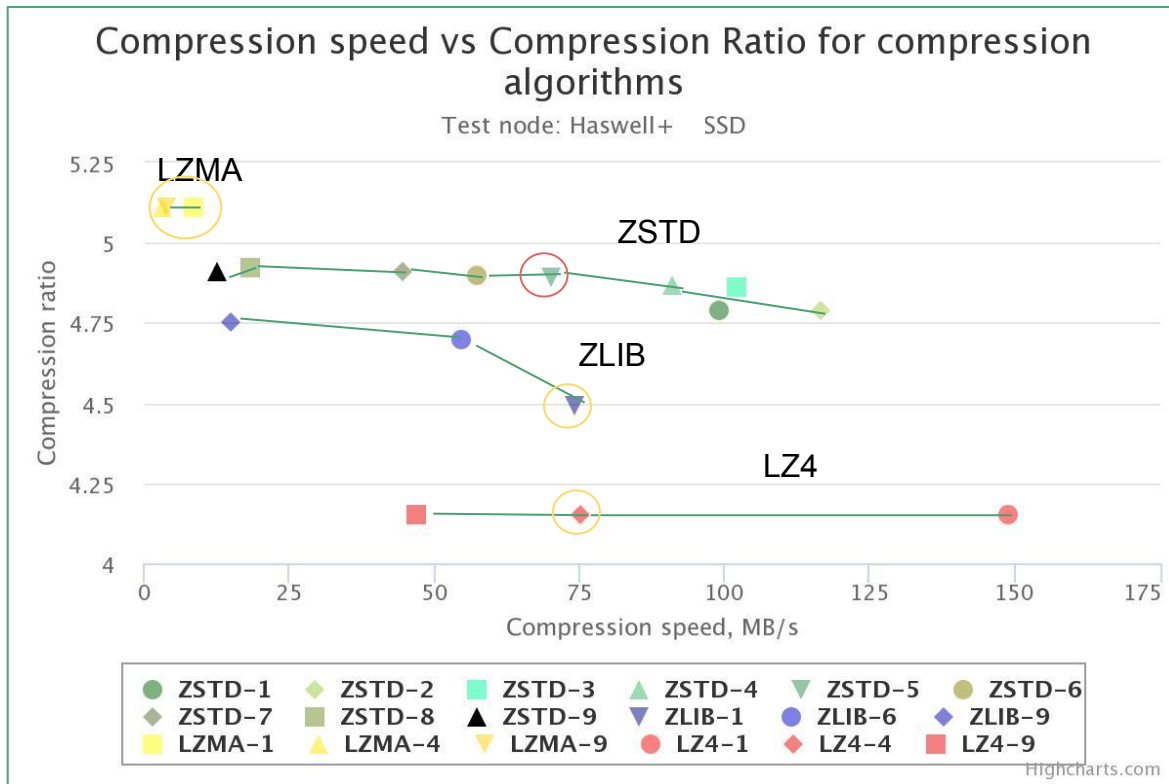


...sadly search gives no logo...but only pictures of Zeppelin LZ4 aircraft :-)

Write Tests - Write Speed and Compression Ratio



Larger is better ↑



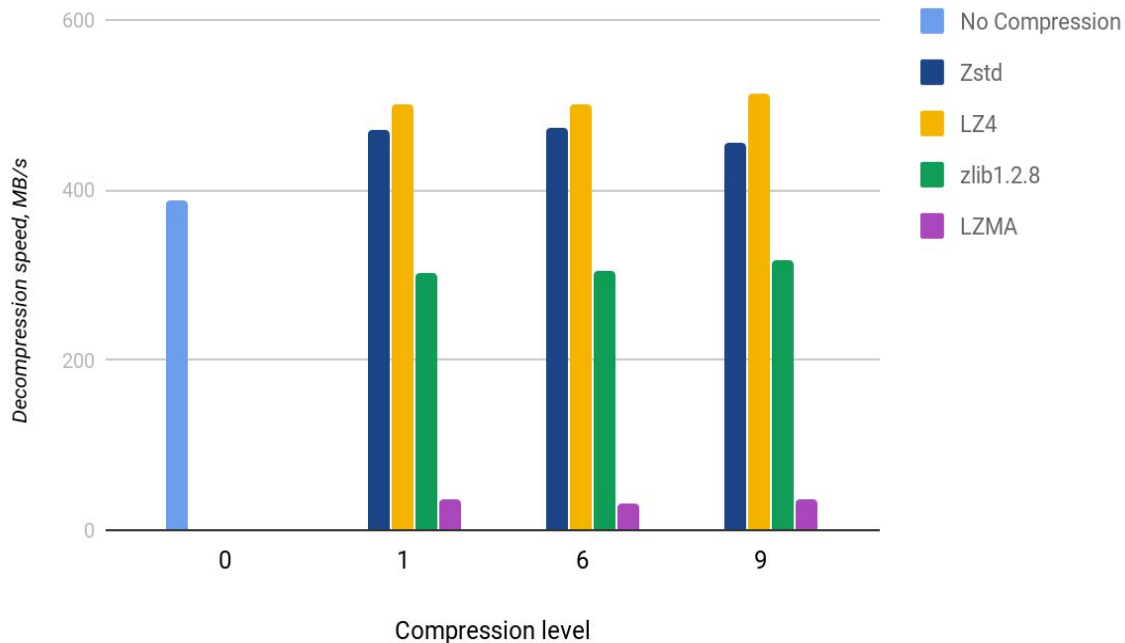
Larger is better

Test used: roottest-io-compression-make with 2000 entries




Read Speed - Compare across algorithms

Decompression speed, 2000 event TTree, MB/s





LZ4 is default compression algorithm

- **It is a good trade off between compression ratio and compression / decompression speed!**
- Was enabled as default in ROOT 6.14.01 (temporary disabled in 6.14.04 for the further investigation)
- We got reported some corner cases:
 - Tree generated with variable-sized branches embed an “entry offset” array in their on-disk representation.
 - Genomic data processed by GeneROOT 

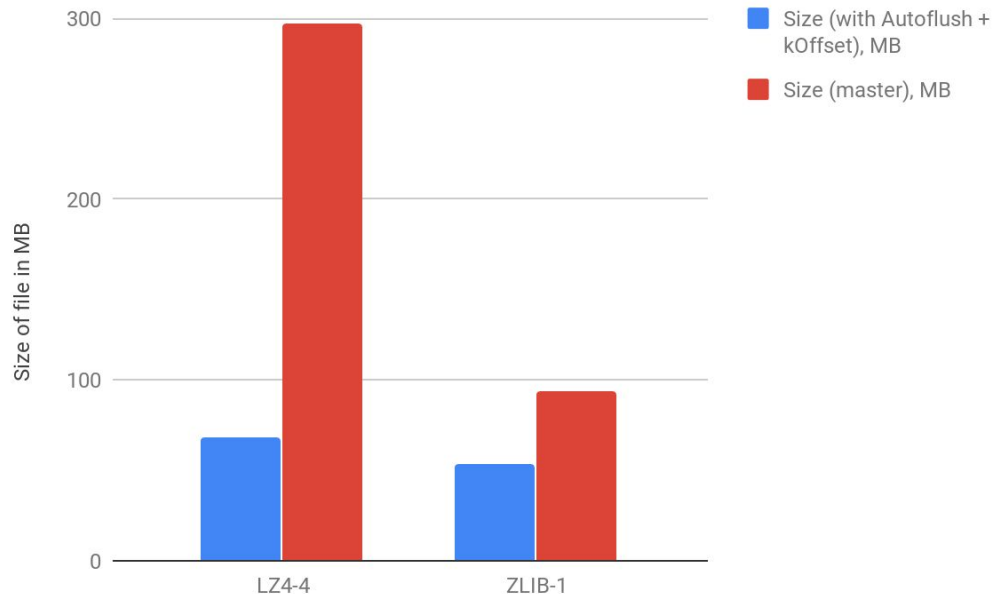
*Both cases are involving compression of big arrays of integers!
We are working on the fix!*

Recommendation to the users

- **Ratio between compression ratio and compression/decompression speed:** the best choice is **LZ4**
- **Size of the file:** the best choice is **LZMA**
- **Recovering data from partial file (in case of crash): tune AutoSave!**
 - Default frequency is to save the metadata every 10 clusters or so.
- **Memory use or physical I/O performance: tune AutoFlush!**
 - Default is number of entries needed to reach 32 Mb of compressed data
 - Can be expressed in number of entries or compressed data size
 - Memory requirement:
 - i. read and write: $\text{AutoFlush frequency in number of entries} * \text{average size of an entry.}$
 - ii. Plus for reading: $\text{AutoFlush frequency in number of entries} * \text{average compressed size of an entry (TTreeCache size)}$
 - Size of a cluster in compressed size will be the 'unit of reading' when accessing the file

[Thanks for recommendations to Philippe Canal]

Optimization of TTree with Int_V branches: AutoFlush(1000000) & kGenerateOffsetMap



```
t->SetAutoFlush(1000000);  
ROOT::TIOFeatures features;  
features.Set(ROOT::Experimental::EIOFeatures::  
kGenerateOffsetMap);  
t->SetIOFeatures(features);
```

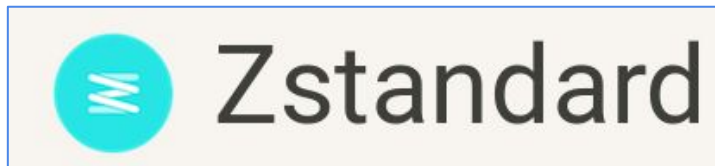
Challenge: not forward compatible.



Future work on compression algorithms



ZLIB-CF & ZSTD



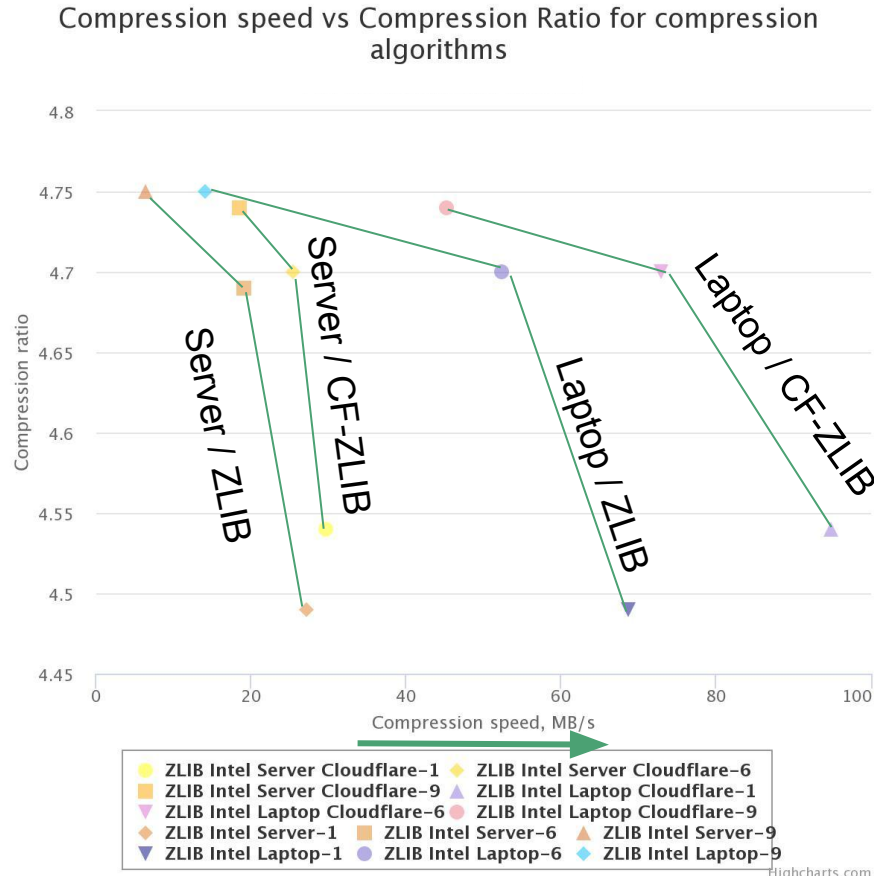
Future work: Cloudflare ZLIB vs ZLIB - Intel Laptop/Intel Server



Note: small dynamic range for y-axis.

The CF-ZLIB compression ratios *do* change because CF-ZLIB uses a different, faster hash function.

Larger is better ↑

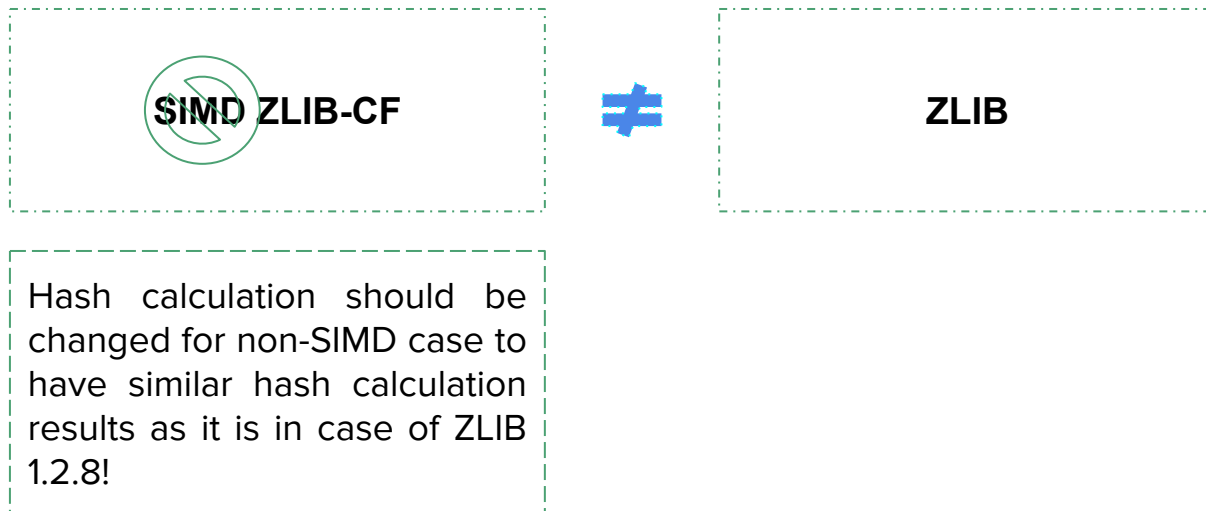




Future work: ZLIB-CF vs. ZLIB

ZLIB Cloudflare(CF) - <https://github.com/cloudflare/zlib/>

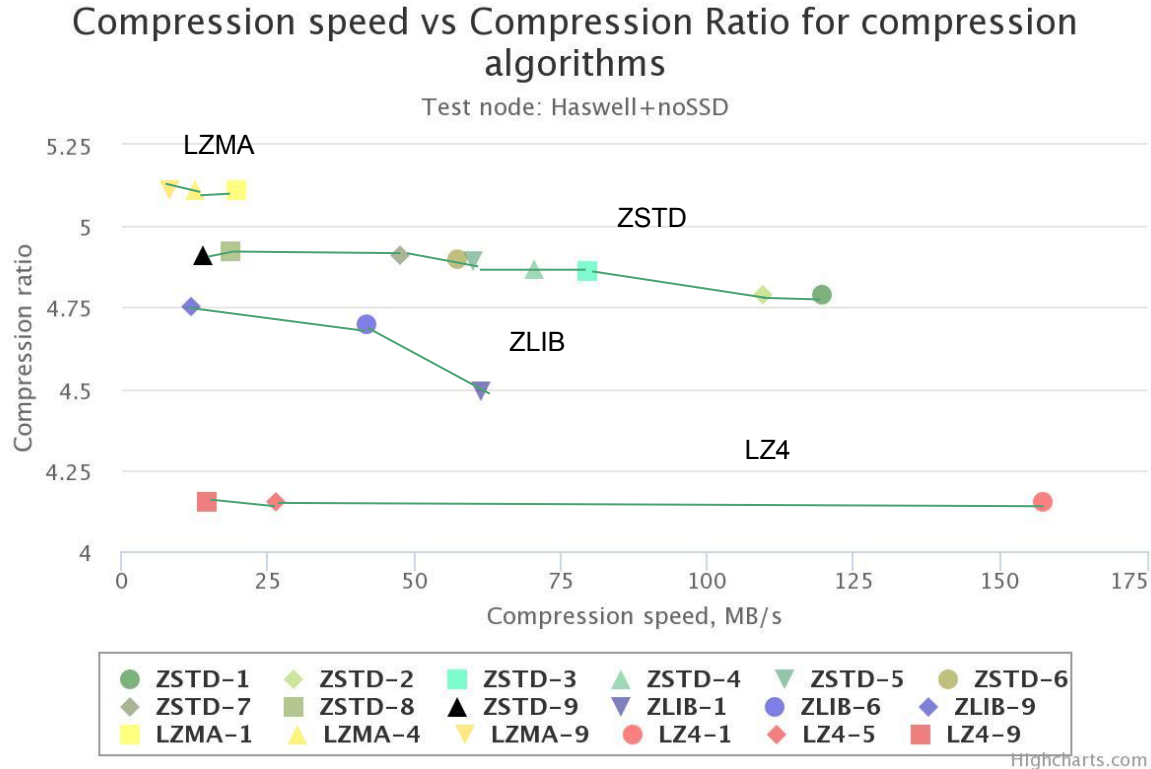
- Not actively supported, but much more performant then ZLIB





Future work: ZSTD - Haswell x 56core - no SSD

Larger is better ↑





LZ4: block compression vs streaming compression

- **"Block" API** : This is recommended for simple purpose. It compress single raw memory block to LZ4 memory block and vice versa.
- **"Streaming" API** : This is designed for complex things (compress huge stream data in restricted memory environment).

Now we are using block compression!



Streaming compression (using compression dictionaries) **is supposed to be beneficial in case of small data (small branches)**

Planned to be available in ROOT 6.16



BitShuffle pre-conditioner for LZ4

Bitshuffle is an algorithm that rearranges typed, binary data for improving compression

Plan of work:

- Determine how we should expose this functionality (separate algorithm versus special API to core/zip versus preconditioner chain).
- Switch LZ4 to streaming mode.
- BitShuffle one block at a time (into a thread-local array), then feed individual 8KB blocks to LZ4.
- Cleanup unused BitShuffle code. Remove OpenMP integration (dead code right now).
- Make BitShuffle use appropriate trampolines to pick AVX2 vs SSE2 version at runtime.
- Remove debugging statements.
- Work with Philippe to determine the best way to detect "primitive branches" - right now, that's an ugly hack.
- Implement unzip methods for LZ4.
- Remove LZMA attempt (did not result in improvements).
- Special-case the buffering of the offset array.

<https://sft.its.cern.ch/jira/browse/ROOT-9633>

Planned to be available in ROOT 6.16

(<https://github.com/Blosc/c-blosc>) is meta-compressor

supporting LZ4, ZLIB, ZSTD with BitShuffle and Shuffle filters

LZ4c >

Compression of large int arrays

2 posts by 2 authors



Robert Schneiders

★ Hi,

in our application, we compress and store large integer arrays (all integers are positive).

The compression with compress_lz4hc2 results in files which are 2 times bigger than those compressed with zlib (lz4 has half of zlibs compression rate).

Is there a way to improve that?

lz4 decompresses five times faster.

Best regards,

Robert



Cyan

★

[Translate message to English](#)

Yes, Blosc

<http://www.blosc.org/>



Future plans:

- Re-enable LZ4 as a default compression algorithm
- Merge ZLIB-CF developments in ROOT master
- Improve I/O interfaces for easier switching between compression algorithms
- Further work on optimization of compression algorithms in ROOT (using of compression dictionaries and etc.)
- Introduce the automatized performance benchmarking of ROOT I/O



- This work was supported by the National Science Foundation under Grant ACI-1450323

Acknowledgements

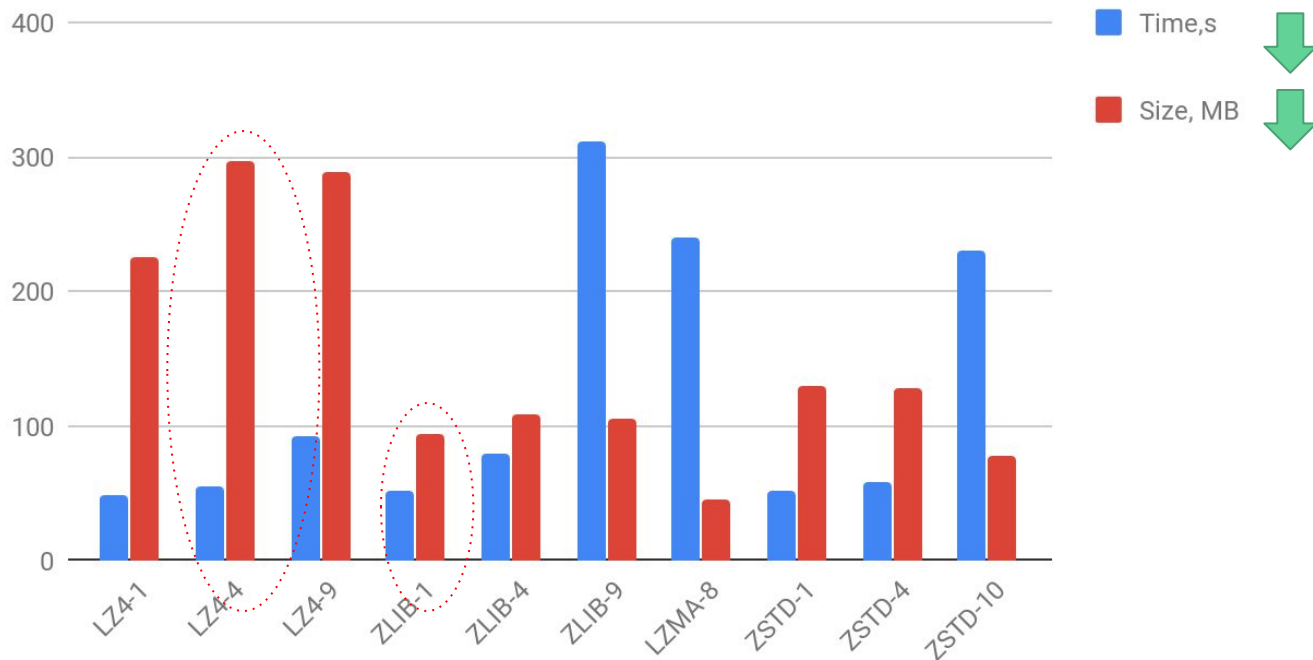


Thank you for your attention!

Backup slides

Example from ROOT Forum: arrays of Int_v stored in branches of ROOT TTree

Size and RT for compression of TTree



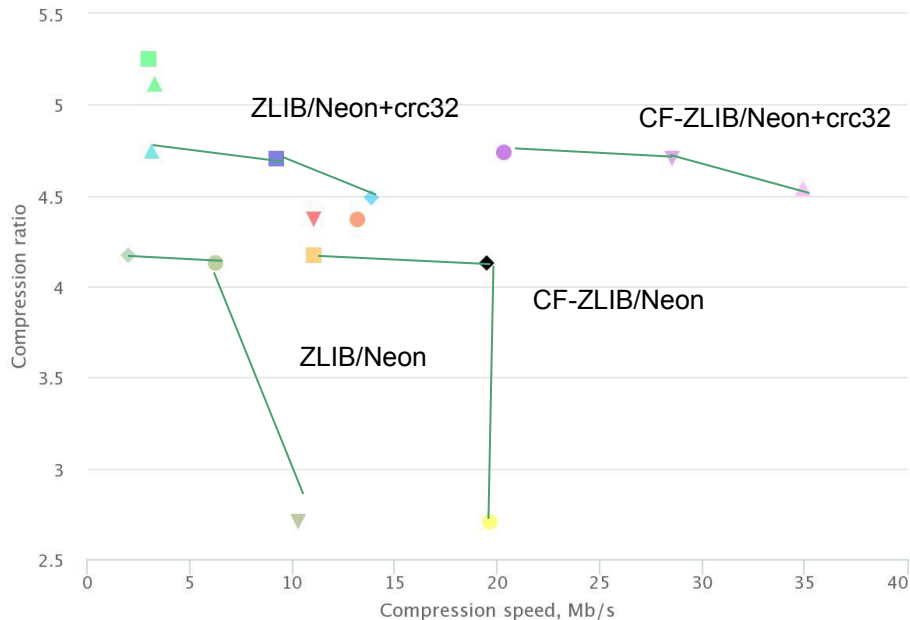
Status of updates

Algorithm	ROOT	Updates	Performance
LZMA	5.2.1	5.2.4	No (bug fixes)
ZLIB	1.2.8	1.2.8 + CF	Yes
LZ4	1.7.5	1.8.2	Yes
ZSTD (still not integrated)	Previous test - 1.3.4	1.3.5	Yes

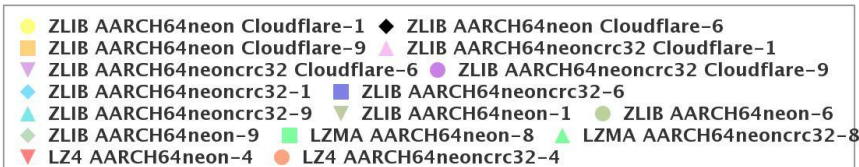
Cloudflare zlib vs zlib -AARCH64+CRC32 HiSilicon's Hi1612 processor (Taishan 2180)

Compression speed vs Compression Ratio for compression algorithms

Test nodes: AARCH64, AARCH64+crc32



- Significant improvements for aarch64 with with Neon/CRC32
- Improvement for zlib Cloudflare comparing to master for:
 - ZLIB-1/Neon+crc32: -31%
 - ZLIB-6/Neon+crc32: -36%
 - ZLIB-9/Neon +crc32-9: -69%
 - ZLIB-1/Neon: -10%
 - ZLIB-6/Neon: -10%
 - ZLIB-9/Neon: -50%



ZLIB-CF:SIMD CRC32 issue

CRC32 of 1 GByte	published by	bits per iteration	table size	time	throughput	CPU cycles/byte
Original	<i>(unknown)</i>	1	-	29.2 seconds	35 MByte/s	approx. 100
Branch-free	<i>(unknown)</i>	1	-	16.7 seconds	61 MByte/s	approx. 50
Improved Branch-free	<i>(unknown)</i>	1	-	14.5 seconds	70 MByte/s	approx. 40
Half-Byte	<i>(unknown)</i>	4	64 bytes	4.8 seconds	210 MByte/s	approx. 14
Tableless Full-Byte	<i>(sent to me by Hagai Gold)</i>	8	-	6.2 seconds	160 MByte/s	approx. 18
Tableless Full-Byte	found in "Hacker's Delight" by Henry S. Warren	8	-	6.3 seconds	155 MByte/s	approx. 19
Standard Implementation	Dilip V. Sarwate	8	1024 bytes	2.8 seconds	375 MByte/s	approx. 8
Slicing-by-4	Intel Corp.	32	4096 bytes	0.95 or 1.2 seconds*	1050 or 840 MByte/s*	approx. 3 or 4*
Slicing-by-8	Intel Corp.	64	8192 bytes	0.55 or 0.7 seconds*	1800 or 1400 MByte/s*	approx. 1.75 or 2.25*
Slicing-by-16	based on Slicing-by-8, improved by Bulat Ziganshin	128	16384 bytes	0.4 or 0.5 seconds*	3000 or 2000 MByte/s*	approx. 1.1 or 1.5*
Slicing-by-16 4x unrolled with prefetch	based on Slicing-by-8, improved by Bulat Ziganshin	512	16384 bytes	0.35 or 0.5 seconds*	3200 or 2000 MByte/s*	approx. 1 or 1.5*



We will test "Slicing-by-16" for compression level 1,2,3,4,5 (fast compression)
and "Slicing-by 8" for compression level 6,7,8,9 (slower compression)

ZLIB-CF: ROOT performance on a branch without SIMD

