ROOT I/O compression algorithms

Oksana Shadura, Brian Bockelman
University of Nebraska-Lincoln

Outline

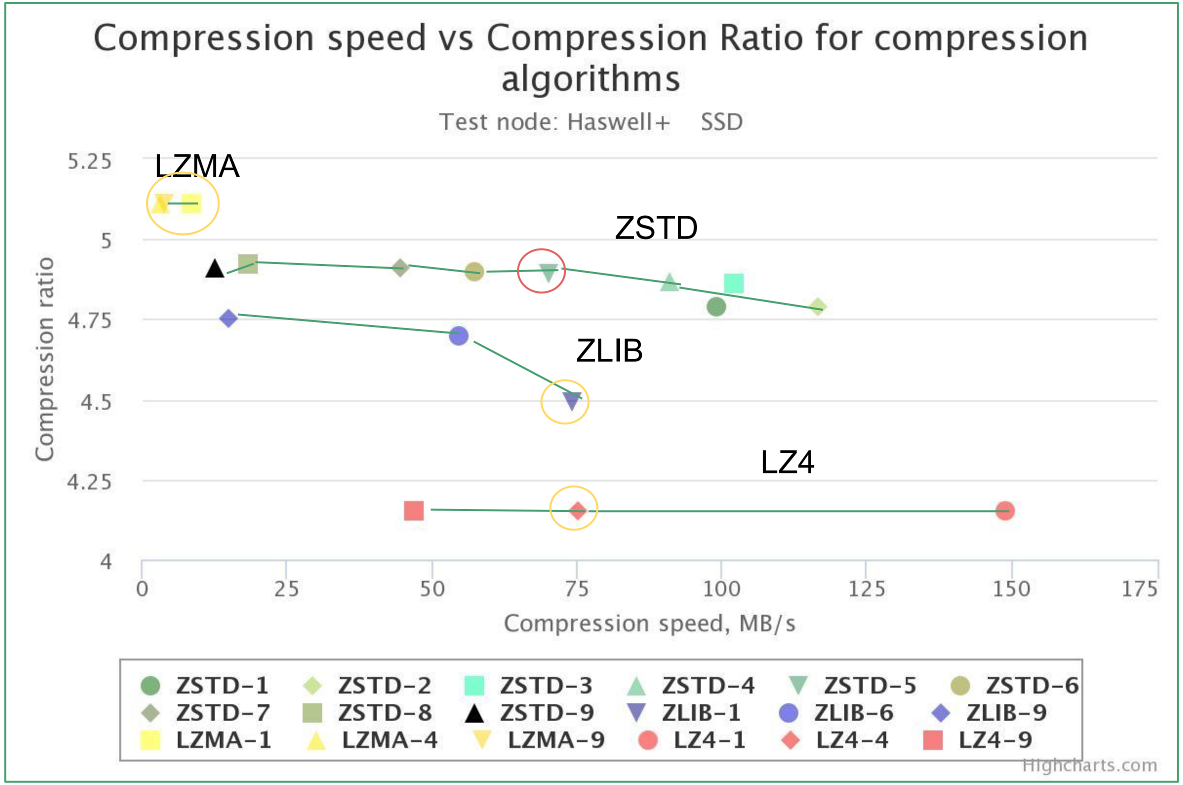
- Status update about integration of LZ4 algorithm
- Status update about integration of ZLIB* algorithms
- Status update about integration of ZSTD algorithm
- Future plans

Status of ROOT I/O builtin updates

Algorithm	ROOT built-in version	Planned Updates	Performance Improvements?
LZMA WIP [oshadura/lzma-5.2.4]	5.2.1	5.2.4	No (bug fixes)
ZLIB [oshadura/zlib-revert]	1.2.8	1.2.8 + CF	Yes
LZ4 [bbockelm/bitshuffle_integration_v1] [oshadura/lz4-bitshuffle] [oshadura/lz4-1.8.3]	1.7.5	1.8.3	Yes
ZSTD (not in master) [oshadura/zstd-default]	Previous test - 1.3.4	1.3.6	Yes

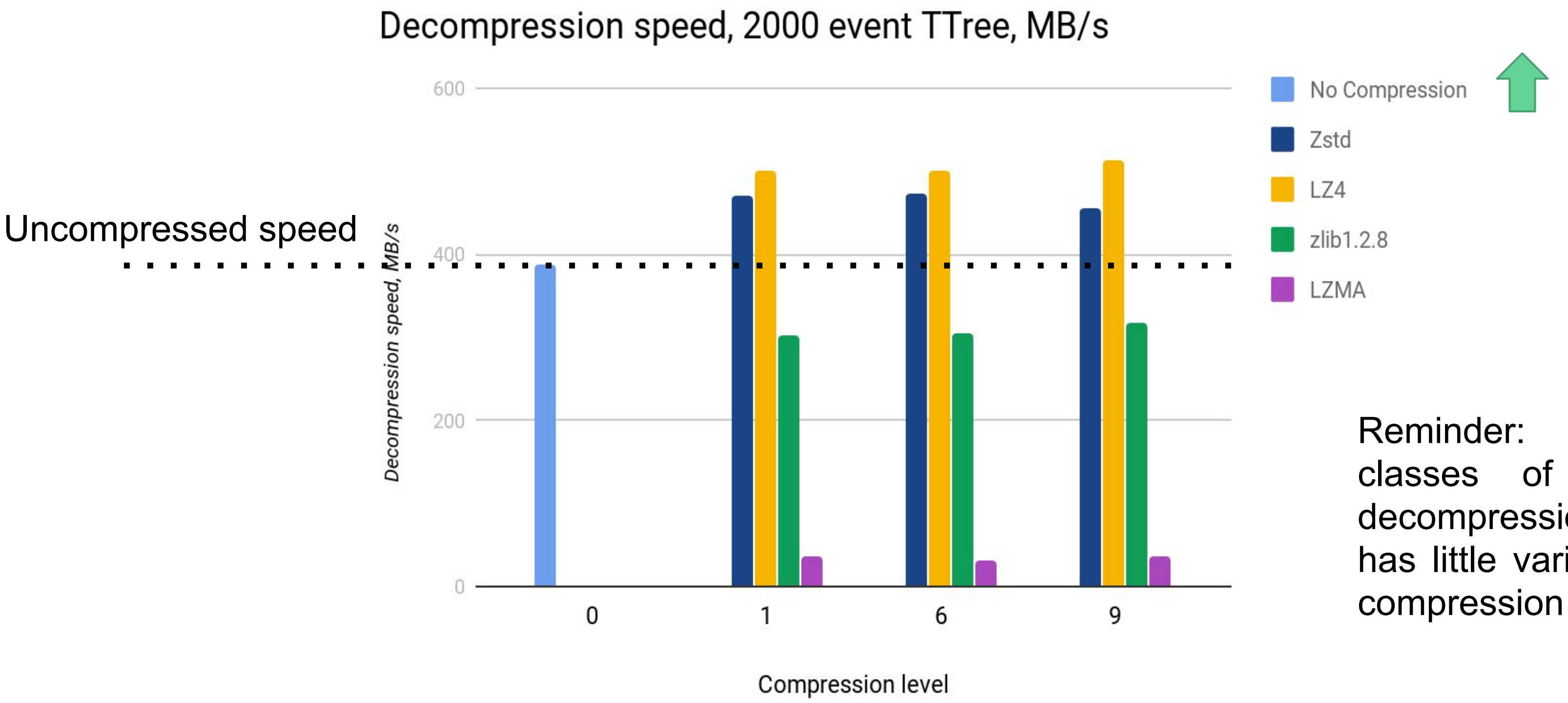
Write Tests - Write Speed and Compression Ratio

Larger is better



Larger is better Test used: roottest-io-compression-make with 2000 entries

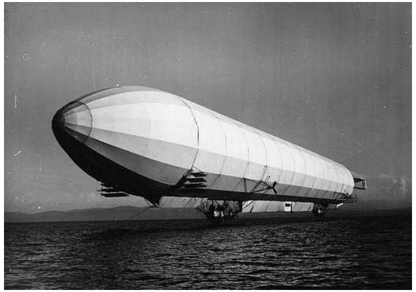
Read Speed - Compare across algorithms



Higher is better!

Reminder: for these classes of algorithms, decompression speed has little variation across compression levels.

Status update about integration of LZ4 algorithm



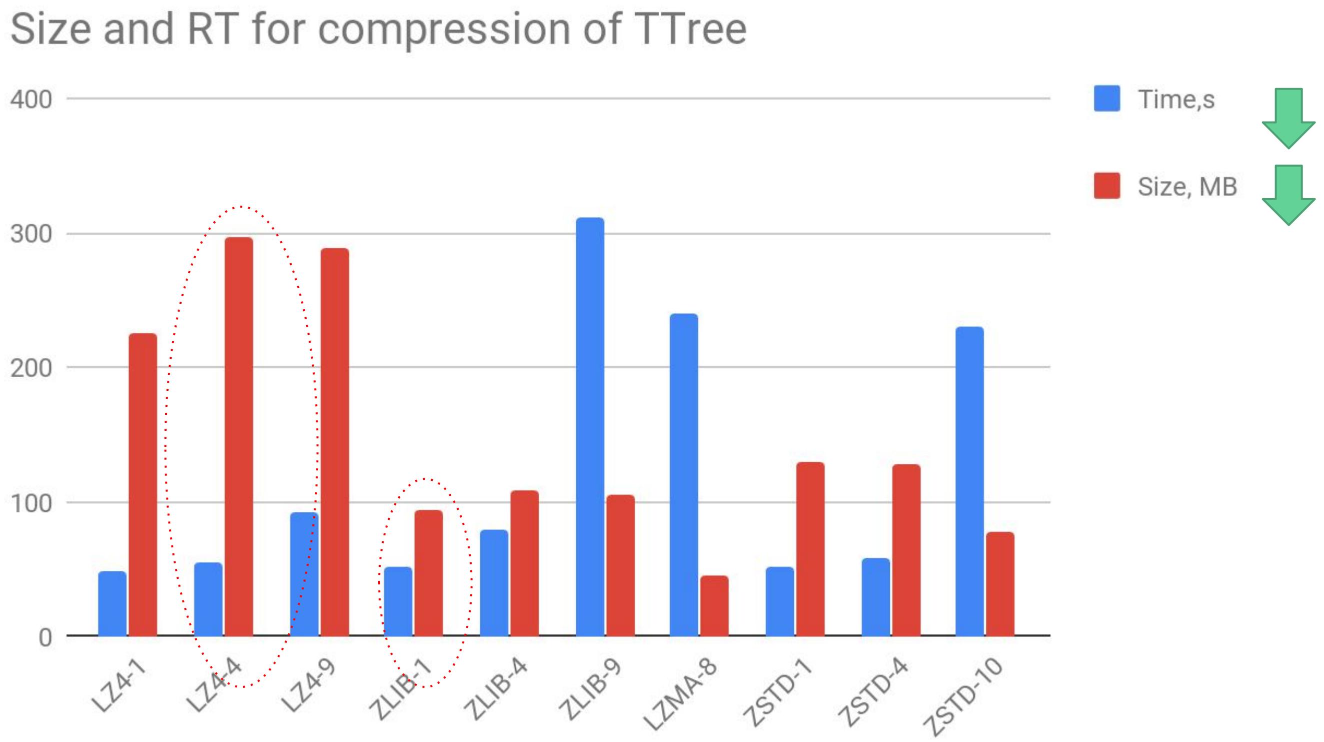
...sadly search gives no logo...but only pictures of Zeppelin LZ4 aircraft :-)

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!

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

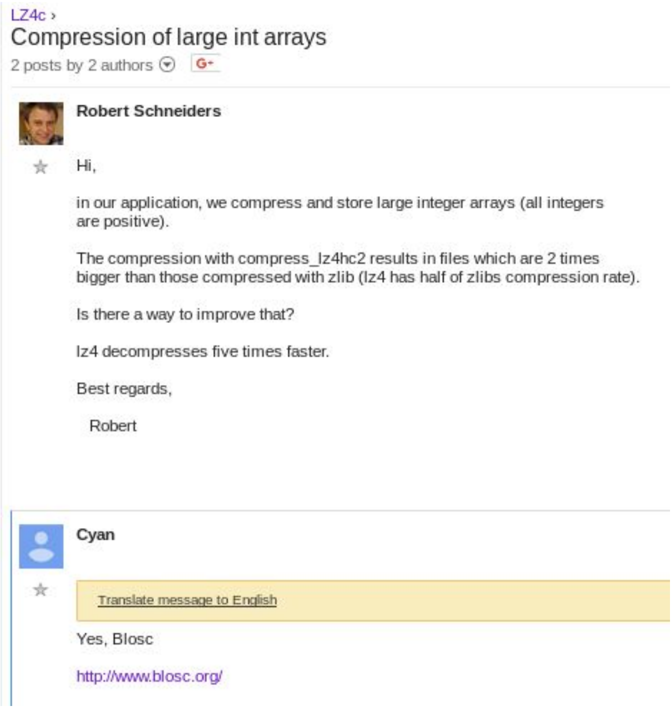


BitShuffle pre-conditioner for LZ4

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.



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

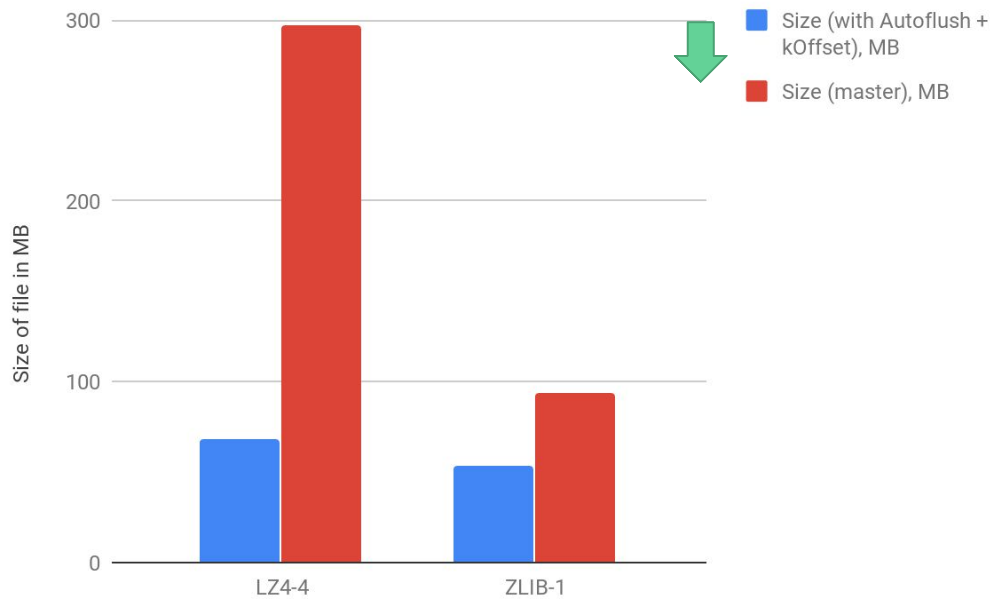
http://www.blosc.org/

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

(https://github.com/Blosc/c-blosc) is meta-compressor supporting LZ4, ZLIB, ZSTD with BitShuffle and Shuffle filters

Planned to be available in ROOT 6.16

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);
```

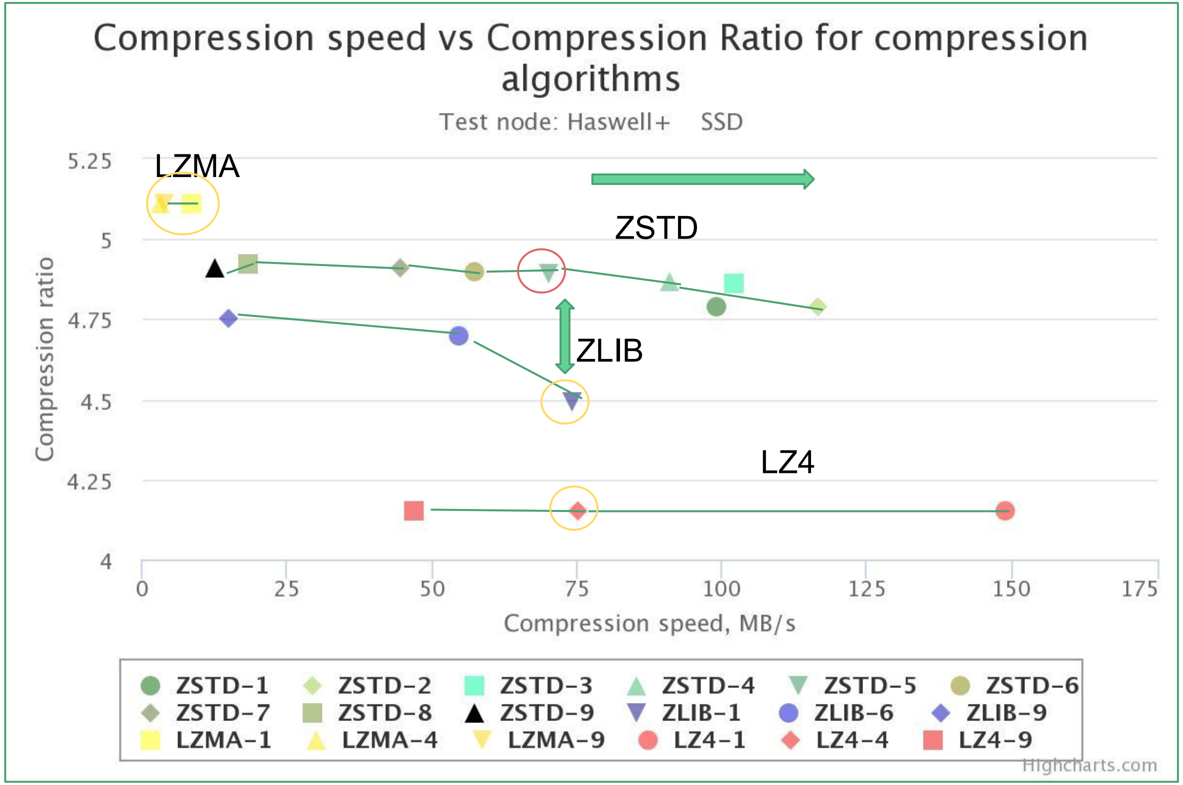
Note: ROOT older than 6.12 cannot read files written with kGenerateOffsetMap.

Status update about integration of ZSTD algorithm

Zstandard

facebook

Write Tests - Write Speed and Compression Ratio

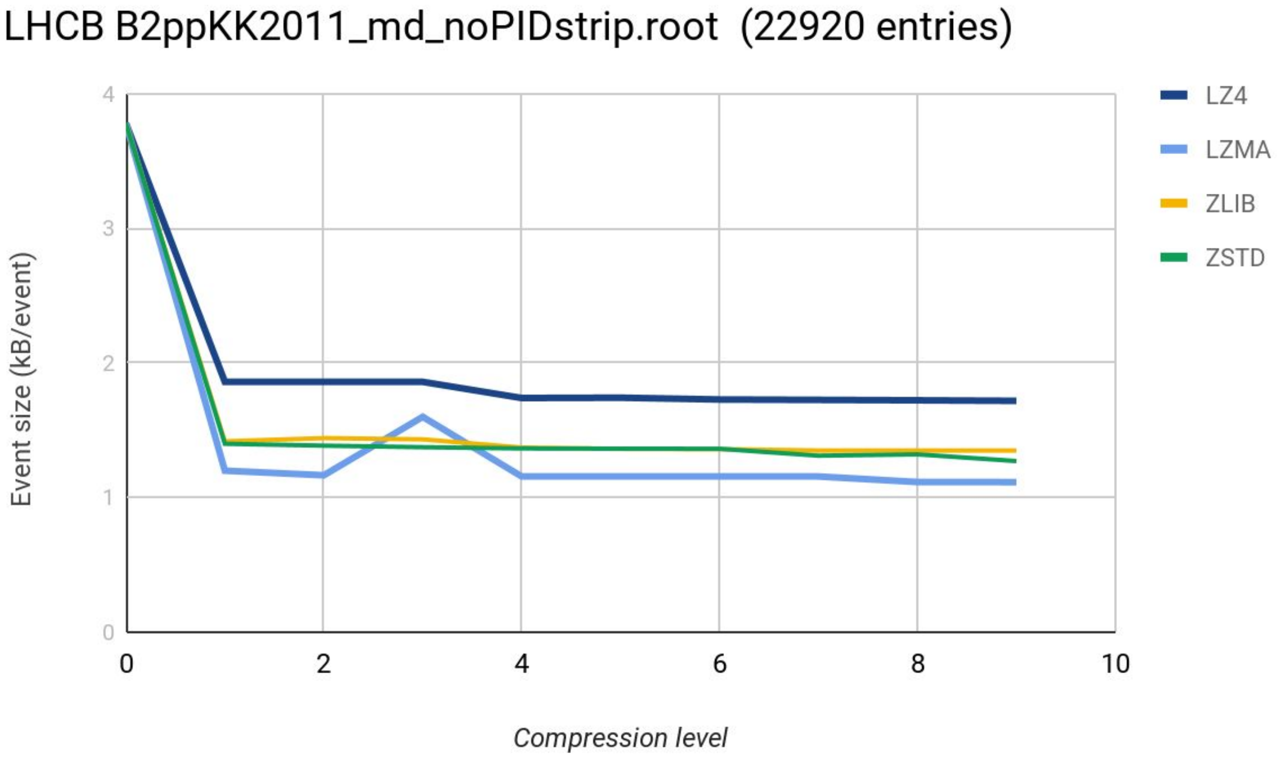


Larger is better

Larger is better

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

LHCB



LHCB B2ppKK2011_md_noPIDstrip.root (22920 entries)
Event size (kB/event)
0
1
2
3
4
0
2
4
6
8
10
Compression level
LZ4
LZMA
ZLIB
ZSTD

Status update about integration of ZLIB* algorithms

ZLIB-CF vs. ZLIB

Jira issue: ROOT-8465

SIMD ZLIB-CF

ZLIB

≠

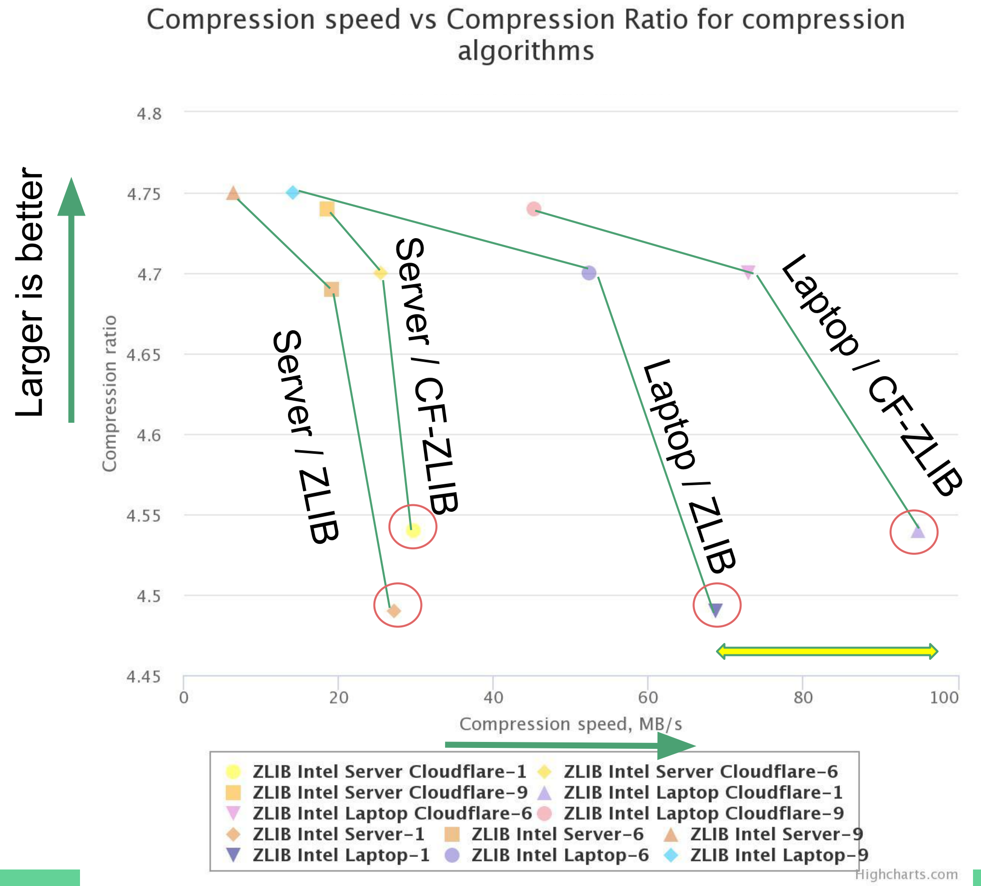
ZLIB-CF without SIMD part is not equivalent to ZLIB

https://github.com/oshadura/root/tree/zlib-revert

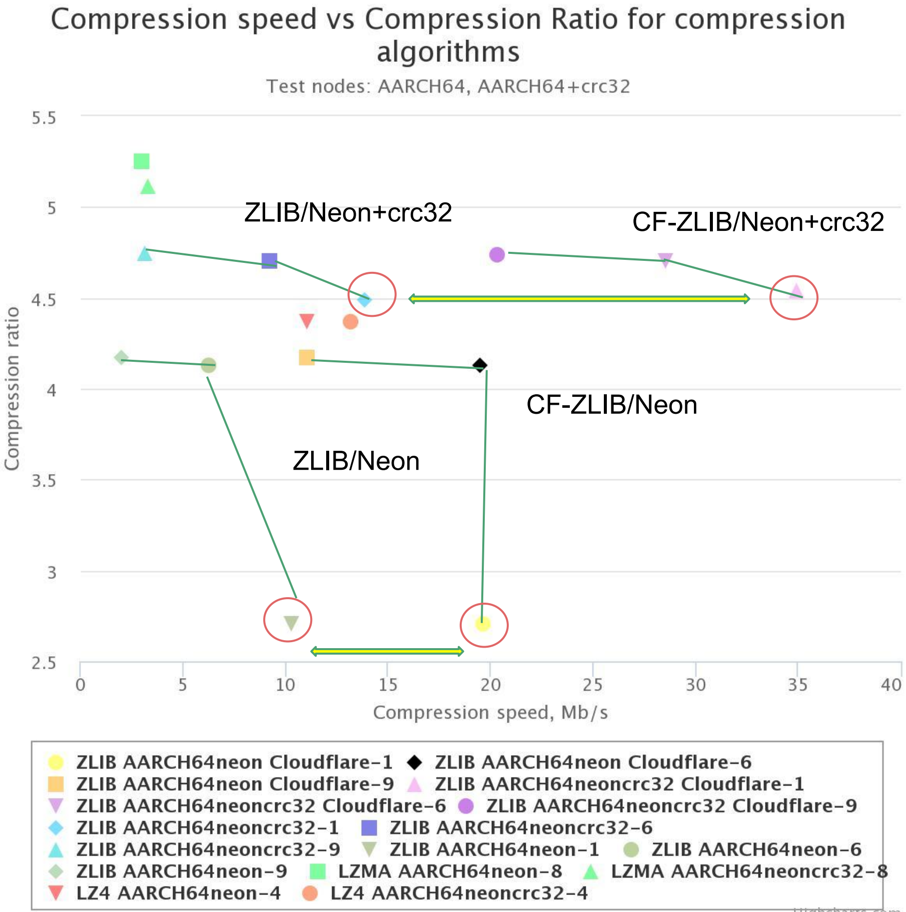
Future work: Cloudflare ZLIB vs ZLIB - Intel Laptop/Intel Server(2000 events)

Note: small dynamic range for y-axis.

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



Cloudflare zlib vs zlib - AARCH64+CRC32 HiSilicon's Hi1612 processor (Taishan 2180 oshadura@hwei-2180-ol-06 - 20 events)



- 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%

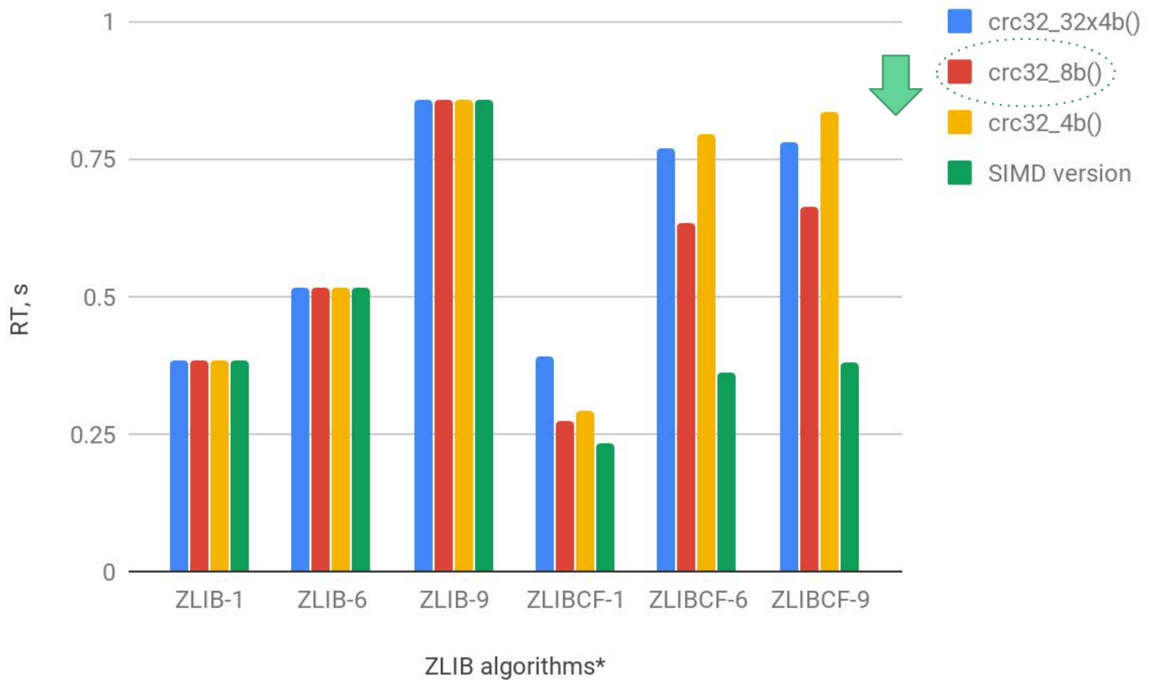
ZLIB-CF:SIMD CRC32 issue

CRC32 of 1 GByte	published by	bits per iteration	table size	time	throughput	CPU cycles/byte
Original	*(unknown)*	1	-	29.2 seconds	35 MByte/s	approx. 100
Branch-free	*(unknown)*	1	-	16.7 seconds	61 MByte/s	approx. 50
Improved Branch-free	*(unknown)*	1	-	14.5 seconds	70 MByte/s	approx. 40
Half-Byte	*(unknown)*	4	64 bytes	4.8 seconds	210 MByte/s	approx. 14
Tableless Full-Byte	*(sent to me by Hagai Gold)*	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-x" to replace intrinsic-based CRC32 calculation!

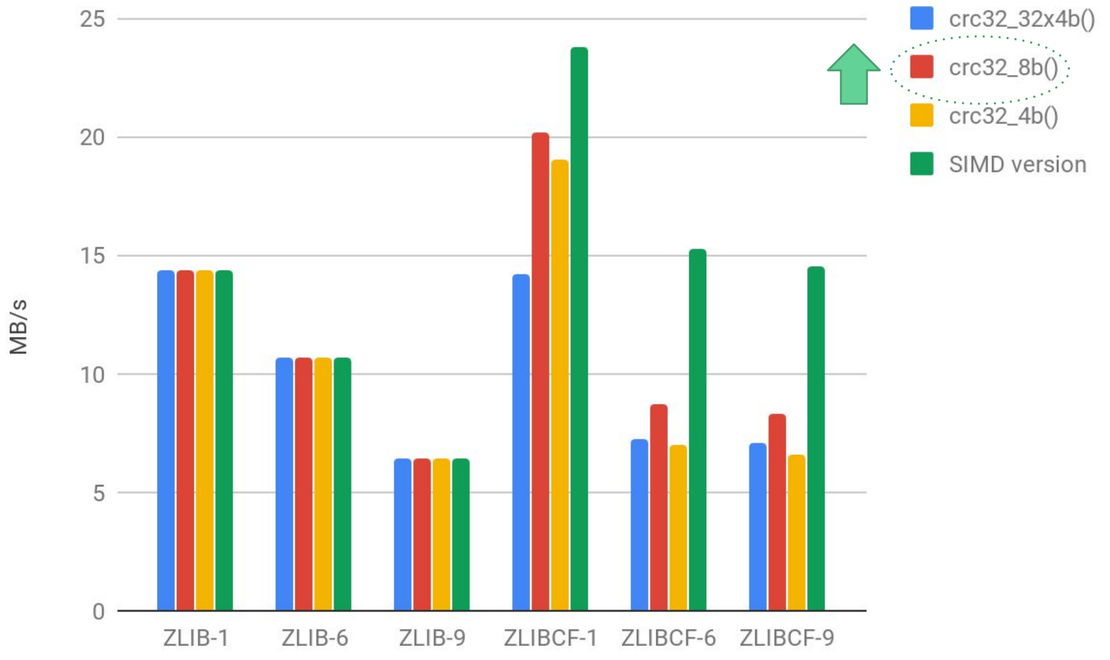
**https://create.stephan-brumme.com/crc32/*

ZLIB-CF: ROOT performance on a branch without SIMD (20 events)



Note: crc32_16b not shown as it is significantly slower in all cases.

ZLIB-CF: ROOT performance on a branch without SIMD



- **We need to sacrifice in space:** in non-SIMD v. files **<u>are 8% bigger versus ZLIB 1.2.8</u>**, while in SIMD case they **<u>are 8% smaller then ZLIB 1.2.8</u>**
- We are winning in RT: **compression speed is 30% faster in non-SIMD case and 60% faster in SIMD case!**
- **Decompression is a bit faster, but not significantly!**

Note: ZLIBCF-9 compression speed is comparable to ZLIB-1!

Future plans:

- Re-enable LZ4 as a default compression algorithm
    - Add bitshuffle filter
    - Enable streaming mode
    - Enable dictionary support
- Merge ZLIB-CF developments in ROOT master
- Decide on fate of ZSTD

Thank you for your attention!