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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>ROOT built-in version</th>
<th>Planned Updates</th>
<th>Performance Improvements?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZMA</td>
<td>5.2.1</td>
<td>5.2.4</td>
<td>No (bug fixes)</td>
</tr>
<tr>
<td>WIP [oshadura/lzma-5.2.4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZLIB</td>
<td>1.2.8</td>
<td>1.2.8 + CF</td>
<td>Yes</td>
</tr>
<tr>
<td>[oshadura/zlib-revert]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LZ4</td>
<td>1.7.5</td>
<td>1.8.3</td>
<td>Yes</td>
</tr>
<tr>
<td>[bbockelm/bitshuffle_integration_v1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[oshadura/lz4-bitshuffle]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[oshadura/lz4-1.8.3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZSTD (not in master)</td>
<td>Previous test - 1.3.4</td>
<td>1.3.6</td>
<td>Yes</td>
</tr>
<tr>
<td>[oshadura/zstd-default]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Write Tests - Write Speed and Compression Ratio

Compression speed vs Compression Ratio for compression algorithms

Test node: Haswell+ SSD

Larger is better

Test used: roottest-io-compression-make with 2000 entries
Reminder: for these classes of algorithms, decompression speed has little variation across compression levels.

Higher is better!
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 TTTree

Size and RT for compression of TTTree
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 pre-conditioner 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
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
Write Tests - Write Speed and Compression Ratio

Compression speed vs Compression Ratio for compression algorithms

Test node: Haswell+ SSD

Larger is better  Test used: roottest-io-compression-make with 2000 entries
LHCB B2ppKK2011_md_noPIDstrip.root (22920 entries)
Status update about integration of ZLIB* algorithms
ZLIB-CF vs. ZLIB

Jira issue: ROOT-8465

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.
Significant improvements for aarch64 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

<table>
<thead>
<tr>
<th>CRC32 of 1 GByte</th>
<th>published by</th>
<th>bits per iteration</th>
<th>table size</th>
<th>time</th>
<th>throughput</th>
<th>CPU cycles/byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>(unknown)</td>
<td>1</td>
<td>-</td>
<td>29.2 seconds</td>
<td>35 MByte/s</td>
<td>approx. 100</td>
</tr>
<tr>
<td>Branch-free</td>
<td>(unknown)</td>
<td>1</td>
<td>-</td>
<td>16.7 seconds</td>
<td>61 MByte/s</td>
<td>approx. 50</td>
</tr>
<tr>
<td>Improved Branch-free</td>
<td>(unknown)</td>
<td>1</td>
<td>-</td>
<td>14.5 seconds</td>
<td>70 MByte/s</td>
<td>approx. 40</td>
</tr>
<tr>
<td>Half-Byte</td>
<td>(unknown)</td>
<td>4</td>
<td>64 bytes</td>
<td>4.8 seconds</td>
<td>210 MByte/s</td>
<td>approx. 14</td>
</tr>
<tr>
<td>Tableless Full-Byte</td>
<td>(sent to me by Hagai Gold)</td>
<td>8</td>
<td>-</td>
<td>6.2 seconds</td>
<td>160 MByte/s</td>
<td>approx. 13</td>
</tr>
<tr>
<td>Tableless Full-Byte</td>
<td>found in “Hacker’s Delight” by Henry S. Warren</td>
<td>8</td>
<td>-</td>
<td>6.3 seconds</td>
<td>155 MByte/s</td>
<td>approx. 19</td>
</tr>
<tr>
<td>Standard Implementation</td>
<td>Dilip V. Sarwate</td>
<td>8</td>
<td>1024 bytes</td>
<td>2.8 seconds</td>
<td>375 MByte/s</td>
<td>approx. 8</td>
</tr>
<tr>
<td>Slicing-by-4</td>
<td>Intel Corp.</td>
<td>32</td>
<td>4096 bytes</td>
<td>0.05 or 1.2 seconds*</td>
<td>1050 or 840 MByte/s*</td>
<td>approx. 3 or 4*</td>
</tr>
<tr>
<td>Slicing-by-8</td>
<td>Intel Corp.</td>
<td>64</td>
<td>8192 bytes</td>
<td>0.55 or 0.7 seconds*</td>
<td>1800 or 1400 MByte/s*</td>
<td>approx. 1.75 or 2.25*</td>
</tr>
<tr>
<td>Slicing-by-16</td>
<td>based on Slicing-by-8, improved by Bulat Ziganshin</td>
<td>128</td>
<td>16384 bytes</td>
<td>0.4 or 0.5 seconds*</td>
<td>3000 or 2000 MByte/s*</td>
<td>approx. 1.1 or 1.5*</td>
</tr>
<tr>
<td>Slicing-by-16 4x unrolled with prefetch</td>
<td>based on Slicing-by-8, improved by Bulat Ziganshin</td>
<td>512</td>
<td>16384 bytes</td>
<td>0.35 or 0.5 seconds*</td>
<td>3200 or 2000 MByte/s*</td>
<td>approx. 1 or 1.5*</td>
</tr>
</tbody>
</table>

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 are 8% bigger versus ZLIB 1.2.8, while in SIMD case they are 8% smaller than ZLIB 1.2.8.
- 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!