DIANA Contributions Update

Brian Bockelman

Bulk IO, LZ4, and More

Fast IO Mode

- As previously discussed, the per-event overheads of function calls:

- This leads to the bulk IO API:
  - https://github.com/bbockelm/root/tree/root-bulkapi

- Have been (slowly) building up unit tests and coverage.

Bulk IO API

- Bulk IO API targets PODs without pointers.
  - Basically, anything we can deserialize in place. (No requirement for zero-copy.)
- Every time we load a new basket, its entire contents are deserialized.
  - So, all memory addresses are touched.

Current Work

- Current focus is on `TTreeReaderFast`. Basic pieces are there; working to expand coverage.
  - https://github.com/bbockelm/root/tree/root-bulkapi-fastread-v2
- The point of `TTreeReaderFast` is to avoid any new APIs but retain the bulk API reader's speed.
- Done through heavy use of inlining and templating: generated code should be slightly faster than the hand-written bulk API usage.
  - Why? Deserialization is done inline with iteration code. **No cache miss penalty for deserialization!**

TTreeReaderFast Example

- The code sample to the left illustrates a simple histogram booking example utilizing the `TTreeReaderFast`.
- The starred (*) lines indicate places where a Good User should do return code checks.
  - Looks much better than bulk API, but still too error prone?
  - Is there a point of introducing new serial interfaces?

```
*       TTreeReaderFast myReader("T", hfile);
*       TTreeReaderValueFast<float>
                  myF(myReader, "myFloat");
*  myReader.SetEntry(0);
   float sum = 0;
   for (auto it : myReader) {
      sum += *myF;
   }
```

Deserialization of current event is done here; inlined!

TDataFrame(Fast)?

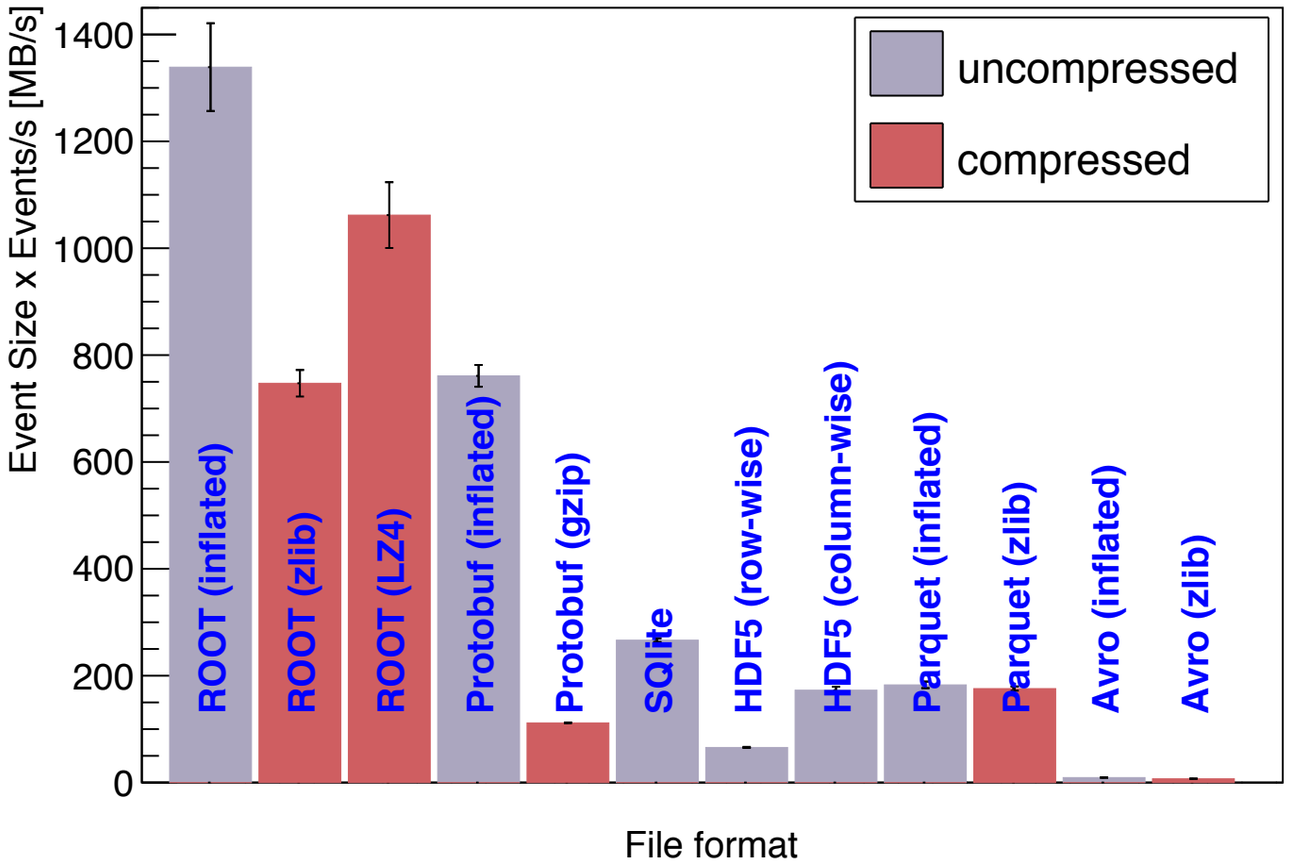
- Ideally, `TDataFrame` would be the best place for the bulk IO (or fast) APIs.
  - User specifies the semantics, `TDataFrame` has significant leeway on how to implement it.
- *Is `TDataFrame` fast enough*? Or will it only be applicable once it is JIT'd?
  - What speed regime is `TDataFrame` targeting?

LZ4 Merged! 🎉

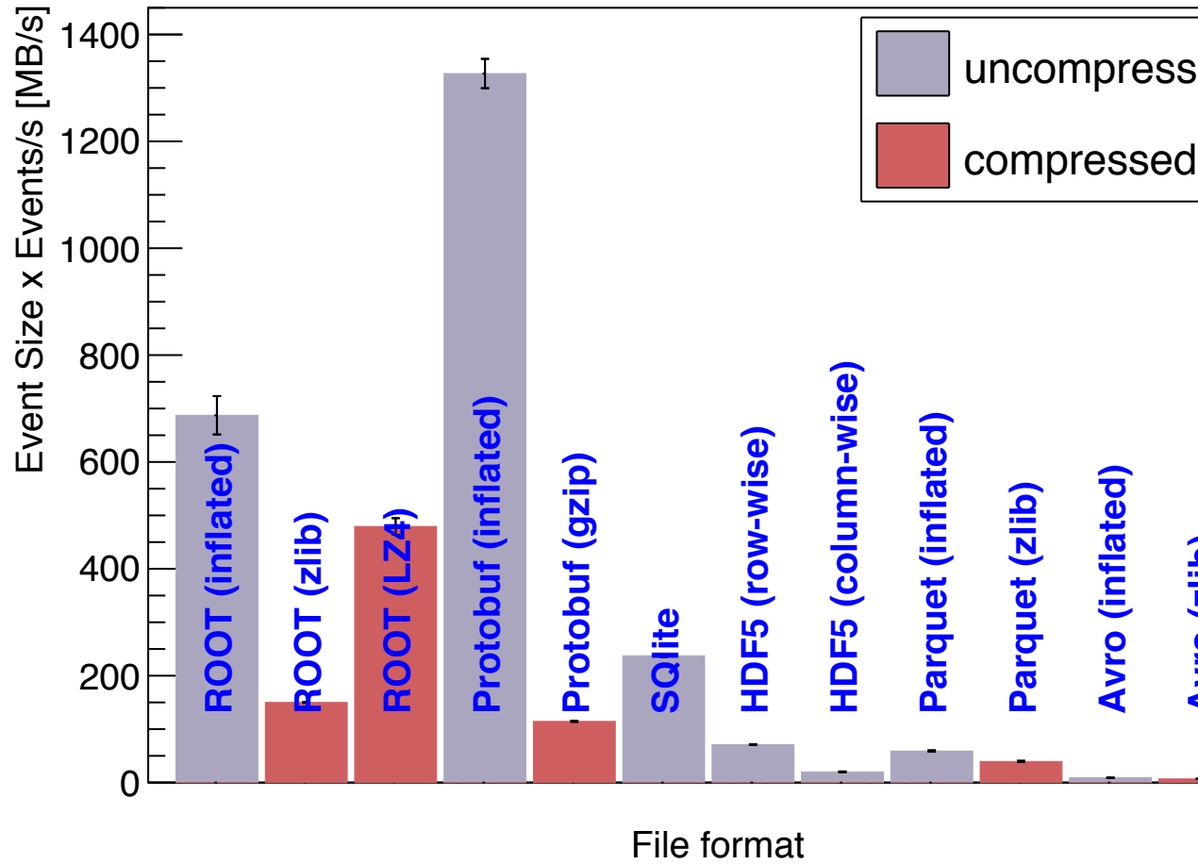
- Was quite the journey to decipher - then fix - LZ4 performance tests.
  - Summary of problems: subtle cmake issue caused optimization flags to be dropped. Effectively we were testing with “-O0 -g” - an enormous performance hit.
- Performance, short version: on test dataset, 12% larger files than zlib, 95% of uncompressed read performance.
- LZ4 TODO: more test coverage, get working on Windows (LZ4 ships a separate set of build files for Windows).

Comparison Plots (from Jakob Blomer)

Quick Plot



Read ntuple



Protobuf “wins” only in the uncompressed case!

Making a difference

- LZ4 helps illuminate the usefulness of the bulk IO API.
- Table to right shows time required to read 100M floats, in seconds.
- Existing ROOT API dominates costs except for LZMA.
- Significant speedup from using LZ4 for bulk or fast reader API!

Compression Algorithm	Normal API	Bulk API	Fast Reader
LZMA	4.40	2.87	2.81
ZLIB	3.24	1.19	1.08
LZ4	3.19	0.58	0.52
None	3.16	0.57	**0.49**

Ease of ROOT Contributions 🎉

- Last IO workshop, had a list of suggestions to make contributions easier.

- A BIG THANK YOU to CERN SFT for hitting many of these! Switch to GitHub and a public CI has removed *significant* friction to contributions.

- Remaining requests:

  - Slack group for ROOT devs? (Or Mattermost or Gitter or … anything low-latency for discussions)

  - Post Docker images for relevant Linux build platforms?

Goals for the Week

- Discussions to have:
  - LZ4 by default? <~ CMS may be interested in backport?
  - `pip install PyROOT` <~ may be more practical than imagined. Experience to share from HTCondor.
    - Most opportune places to put bulk IO work?
    - Next targets for parallelization of IO?
- Use opportunity to hack on code with Philippe!
  - Try to get bulk IO interfaces into something that is mergeable.
  - Pick 1-2 bugs covering areas of ROOT IO I am not familiar with and try to fix them.