Bulk IO Update

Brian Bockelman

Pondering Analysis

- ROOT has high overheads for simple objects!
- Consider a TBranch that consists of a single float.
- Each object read by ROOT results in:
    - Virtual calls (TBranch::GetEntry, TBasket::PrepareBasket, TLeaf::ReadBasket, TBuffer::ReadFloat).
    - Function pointer call (TBranch::fReadLeaves).
    - Plus all the CPU overhead of bounds checking and error condition checking.
        - E.g., for every float we read from the same basket, we check for different possible basket layout formats.
- Additionally, there is significant work at basket boundaries.

ROOT Bulk IO

- **Proposal**: For simple branches, introduce a “fast path” that requires *one* library call per basket.

- The fast path will only support the latest ROOT format and a strict subset of branch types.

- User code is responsible for falling back to more flexible “slow path”. **By design, we want the difference between “fast” and “slow” to be explicit.**

    - Concerned that poorly implemented auto-fallback will cause user to accidentally lose performance.

New “User-facing” APIs

- This low-level API is necessarily messy! Will talk about high-level API next.

- `Int_t TBranch::GetEntriesFast(`
  `Long64_t` **`entry`**`,`
  `TBuffer &`**`user_buf`**`,`
  `unsigned` **`flags`**`);`

- Returns the number of events deserialized, or -1 on failure.

- Object of type T can be found using:

```
*reinterpret_cast<T*>(user_buf.GetCurrent())[idx]
```

“User-facing” API

- `Int_t TBranch::GetEntriesFast(`
  `Long64_t` **`entry`**`,`
  `TBuffer &`**`user_buf`**`,`
  `TBuffer &`**`offset_buf`**`,`
  `unsigned` **`flags`**`);`

  - Returns the number of events deserialized, or `-1` on failure.

  - The object for event `idx` can be accessed via:

```
*reinterpret_cast<T*>(
    user_buf.GetCurrent() +
    offset_buf.GetCurrent()[idx] );
```

Preliminary Results

- Wrote a microbenchmark that writes 100M floats to a ROOT file; can read it back via either "standard" (TTreeReader) or "bulk" APIs.
  - 3.5s to read with TTreeReader.
  - 0.75s to read with low-level bulk APIs.
  - **Approximately a 4.7x speedup!**
- *Note*: microbenchmark is meant to showcase bulk API speeds.

The “flags” argument

- `flags` is meant to allow for alternate behavior. Current idea:
  - `kRaw`: For objects that can deserialize in-place, return the serialized buffer.
    - Users can then invoke deserialization routine immediately before reading the event. **Why?**
      - Even with the bulk APIs, to perform a byte-swap, we must sequentially scan the entire buffer twice: once inside ROOT (to perform byteswap) and once in user code. **Implies we read data from memory into cache twice!**
      - If `(deserialize + user code)` is combined into one streaming memory access, then we can save time.

NOTE: Since flags are runtime; will likely switch these to a compile-time technique.

Raw Buffers and Users

- With our float microbenchmark, deserializing in user code reduces runtime from 0.75s to 0.62s.
    - **Important result**: there is no measurable CPU cost byte swap if byte swap is done inside user code.
- Taking in estimates of startup overhead into account (0.23s), the **total speedup is 8.3x**.
    - **CAVEAT**: benchmarking in this manner is fairly inaccurate, especially as the total runtime is so small. It's reasonable to say "a lot faster" but an improved testbed is warranted.
- *Unfortunate it's a really bad idea to ask users to actually do this coding themselves!*

TTreeReaderFast

- Consider sample TTreeReader code:

```
TTreeReader myReader("T", hfile);
TTreeReaderValue<float> myF(myReader, "myFloat");
Long64_t idx = 0;
Float_t sum = 1;
while (myReader.Next()) {
   sum += *myF;
}
```

  - **Observation**: unlike most other places in ROOT, `TTreeReaderValue<float>` provides compile-time guarantees about the object type.

- **Idea**: write a `TTreeReaderFast` class that manages the TBuffer.

  - **myReader.Next()** could be inlined by compiler, avoiding function calls unless a new basket is needed.

  - For certain types, ***myF** would invoke the appropriate deserialization code (`kRaw` mode).

Next Steps

- **High-level interface**: Work to hide all low-level interfaces behind a "TTreeReaderFast" facade.
    - Possibly make the existing interfaces private / internal.
- **Zero-copy interface**: If `TFile` was extended with mmap-compatible interfaces, we could avoid memory copies.
- Continue to expand object types and branches that can use the bulk IO API.
    - Next up: variable-length arrays.
- **More micro-benchmarks**!
    - Particularly, how much of this improvement is lost as we "add in realism"?