I/O Advances in CMSSW

Brian Bockelman
Presented by Giacinto Donvito

The Setup

- CMSSW uses ROOT for its underlying I/O layer.
  - CMSSW is a generic framework used for everything from Online to analysis. There are dozens of use cases and even more IO patterns
- We take the constant stream of innovations from the ROOT team and adopt (or adapt) them to fit our needs.

This presentation covers some recent work with ROOT I/O in CMSSW.

Outline

- CMSSW I/O use cases.
- Additions in CMSSW over bare ROOT.
- Recent Work in ROOT and CMSSW.
- Sneak peak on the future.

CMSSW Use Cases

- CMSSW must support almost all CMS software needs. Typically, a use case can be characterized by:
  - Percentage of branches it reads out.
  - Whether the branches used varies from event-to-event.
- We can't enumerate the use cases, so we can't optimize per-user; additionally, some of our data tiers have users with conflicting needs. One must beware of optimizing one user at the cost of another.

ROOT I/O

- By selecting the split-level (how many buffers a single object is broken into), we can optimize a file for reading whole events or just a small subset of branches.
  - If we optimize for reading an entire event at a time, we would write all data in an event contiguously in a file.
  - If we think users will read a small subset of the branches, we want to write branches contiguously on disk.
  - There's a continuum of settings between "event-based" and "branch-based"; newer ROOT versions make this somewhat moot - more later.

CMSSW Data Tiers

- Even though each there are many use cases, we can guess the most common use cases of each data type and optimize accordingly.
  - RAW: Smallest number of branches as possible; no reasonable user would want a small subset of one event.
  - RECO: Medium number of branches; about the same size as RAW per-event, and usages are extremely varied (in terms of # of branches read and skim efficiency).
  - AOD: Largest number of branches, highest split level. Smallest events in terms of kilobytes. Heavily used for analysis with a small subset of branches;used for skims.

CMS Read Ordering

- It’s convenient and useful for CMSSW to read events in the same order they came out of the detector.
  - For example, it is expensive to load conditions data to switch between runs. We don’t want to do that often.
  - Because of the asynchronous buffering and merging in online, they are not in “detector order” in our files.
- ROOT is optimized to read the events in the order they are written in the file; “file order”.

CMS Read Ordering

- Reading out in “detector order” was a disaster.
  - We continuously thrashed the OS and ROOT buffers for files where “file order” and “detector order” were drastically different.
- As “file order” is too expensive due to conditions, we had a compromise.
  - We read a complete run/lumi combination at a time, in “detector order” for that run/lumi. Within a run/lumi, we read the events in “file order”.
  - Assuming there are few run/lumi combos in a file, this prevents most thrashing. It also tends to sort the files for the next level of processing, which is important for the “higher layers” where there are many lumis.

CMS File Layout

- The focus of all ROOT's optimization *and testing* efforts is the case where there is one TTree for Event data.
  - I.e., asynchronous prefetching is a FIFO. If there are multiple event TTrees, the asynchronous prefetch become synchronous, as one TTree has to wait for the previous to finish its entire prefetch.
  - It's fine to have multiple TTrees per file; you just want that's read for every event.
  - Don't fight the tide! CMS has slowly migrated to one Events TTree, and has found many advertised optimizations suddenly work better.

Prefetching with TTreeCache

- ROOT provides a mechanism, TTreeCache, for managing prefetch.
- TTC keeps a list of active branches (either learned automatically or set by the framework).
- TTC will prefetch all the buffers for these branches for the next N events, in one I/O operation, if possible.
  - N is selected by ROOT so the prefetch will be about 20MB of size (configurable).
- When the entire prefetch buffer has been used, ROOT will throw it away and fill it with the next N events.

TTreeCache in CMSSW

- CMS took a long time to adopt TTreeCache because:
  - Only one TTC can be active per file per TTree. If you read multiple TTrees with the same cache, it will thrash.
  - We had the statistics to show the TTC was making performance worse, but it took awhile to figure out why.
    - Statistics tell you when something is wrong, but not necessarily how to fix it!
    - We mistakenly thought TTC just didn't work!

Lesson: Work with the Experts until you're sure it works!

CMSSW I/O Additions

- CMS has implemented its own set of plugins for TFile which interface to the various protocols seen in the LHC (dcap, rfio, POSIX, etc).
  - These override the ones found in ROOT.
  - But also implement a few things not in ROOT.

I/O Addition: Statistics

- A typical CMSSW developer has access to their desktop and CERN.
  - Meaning they have little insight to how CMSSW is affecting a far-away T2.
  - While CPU patterns are likely the same at each site, I/O patterns vary greatly.
- Good statistics are essential to isolate problems to the I/O system. They're a necessary starting point for devs to debug.

Statistics

- CMSSW provides the max/min/average time for each I/O call (read, write, open, seek, etc).
- It also provides the total number of calls and the volume of data read/write.
- ROOT has added most of these in 5.26 as well as a histogram; we are investigating combining the two sets of statistics.

I/O Addition: Lazy-Download

- Users can always find a way to do a crazy IO pattern, no matter how sane the defaults are.
- CMS's lazy-download mode will divide the file into 128MB chunks, and download the chunk to local disk the first time it is accessed. Care is taken to guarantee the local disk space is released when the process exits. Totally transparent to the user.
- This makes the load on the storage servers fairly predictable - always 128MB reads, never more than 100% of the file read out.

I/O Addition: POSIX Pre-fetching

- POSIX has a little-used function called “fadvise” which requests the OS to perform prefetching based on given I/O patterns.
- CMS started using this 2 years ago, but care must be taken because fadvise is *blocking* if you use it at a high rate.
- ROOT has recently added this in 5.27.

Recent ROOT Work

- In older versions of ROOT, one must chose between efficiency for jobs reading out 100% of the data and jobs reading 10% of the data.
  - In the 10% case, we write branches contiguously, meaning a single event might be scattered across 100's of MB in the file.
  - In 5.26, ROOT started to auto-flush all baskets every 30MB (adjustable). It also adjusts the memory buffer for each branch so each buffer holds approximately the same number of events.

Reclustering in ROOT

- What does this do for us?
  - People reading whole events can do this by reading 30MB "clusters" at a time, which is pleasant for modern hardware systems.
  - People reading a small number of branches still make all their reads inside one "cluster" at a time.
    - This and buffer-resizing prevents constant backward-seeks, as buffers are not scattered throughout the file.

Sneak Peak Of the Future

- WARNING: the next few slides contains forward-looking statements and personal opinions of what I think will happen in the next few years.
- THESE DO NOT REPRESENT OPINIONS, PLANS, OR INTENTIONS OF CMS OR ROOT.

Adaptive Cache

- The current algorithm for determining if a branch should be in the cache is naive.
    - Once the branch is in the cache, it stays there for the rest of the job.
- I have a few ideas about a per-branch statistics-based approach
- Rather than implement my own cache, I would like a callout mechanism so each framework could adopt it to their own needs.

Double-Buffering

- Currently, whenever a cache gets used, computation pauses to refill the cache.
  - Best case is asynchronous protocols, where the "wait time" for the first buffers to arrive is hopefully minimal.
- If we kept 2 buffers, then we could use one while asynchronously filling the other, eliminating the (possibly long) pause while we wait for the next set of IO.
- ROOT seems interested - I'm excited about this one!

Improved Merging

- CMS has always used the ROOT fast merging capability.
  - Unfortunately, fast merging does not use the TTreeCache due to historic reasons.
  - TTreeCache with a reclustered file would make fast-merging near-optimal.
- I'm confident the fix will come quickly from the ROOT team.

Conclusions

- CMS has successfully built its computing on top of ROOT I/O. I’ve found the following rules of thumb:

1. Have only one TTree that’s read per event. This is what the ROOT devs test with.
2. Make sure you collect I/O statistics for debugging. Otherwise your framework developers will not be aware of how the software is affecting your storage, and will code accordingly.

Conclusions

- Continued:

3. If not avoidable, make sure skipping backwards in the TTree is very rare.
4. Prefetching is the difference between storage system life and death. Make sure the TTreeCache is effective for your jobs.
5. The new ROOT "reclustering" technique increases data locality and hence performance in most cases. Use it as soon as possible.