FERMILAB-CONF-11-869-CD

ROOT I/O: The Fast and Furious

Philippe Canal

Fermi National Laboratory, Batavia, IL, USA

pcanal@fnal.gov

Brian Bockelman

University of Nebraska-Lincoln, Lincoln, NE, USA

bbockelm@cse.unl.edu

René Brun

CERN, Geneva, Switzerland.

Rene.Brun@cern.ch

Abstract. The increases of data size and in the geographical distribution of analysis intensify the demand on the I/O subsystem. Over the last year, we greatly improved the I/O throughput, in some case by several factors, when reading ROOT files. ROOT's improved techniques include improving the pre-existing prefetching, the automatic flushing of data buffers at regular intervals and streaming objects member-wise. These advances reduce the number of transactions with the local disk or the network. We worked in close collaboration with the Large Hadron Collider (LHC) experiments to optimize the I/O access to their use cases and to help adapt their framework to take full advantage of these advances. This presentation will describe in details these improvements and how users can benefit from them.

1. Introduction.

As their frameworks mature, the LHC experiments have an increased emphasis on performance issues both localized and large scale. In parallel, the ROOT team has started several efforts to find and remove inefficiencies in the core ROOT libraries. We analyzed several ROOT usage scenarios from the LHC experiments and found opportunities to reorganize memory allocations that decreased the total memory use and memory fragmentation. Analyzing the reading of ROOT files over wide area networks, we found solutions to decrease the time required for reading a file by several orders of magnitude; the same techniques also decrease local file access times.

2. Prefetching of the TTree data.

For storing large quantities of homogenous objects, ROOT[1] provides a specially designed container called the TTree[2]. The TTree class is optimized to reduce disk space used for storing objects and enhance access speed. A TTree can hold any kind of C++ data, including simple data types, objects, arrays and containers. When using a TTree, the objects or variables may be decomposed into simpler

Published under licence by IOP Publishing Ltd

Operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy

data types, and associated to a TBranch object belonging to the TTree. Depending on the "split-level" setting for the TTree, one or more TBranch objects are associated with each top-level object (the higher the split-level, the more TBranch objects per top-level object). Each branch has a memory buffer object called the TBasket or just 'basket'. Whenever the method TTree::Fill is invoked, the user objects are copied into these baskets. The set of user objects stored in the TTree by this call forms an entry. Once a basket is full, it is written to disk. When requested, the baskets are compressed before being written. This approach produces a smaller file than if each object was written contiguously.

Prior to ROOT v5.26, all baskets default to the same size. Accordingly, the baskets of each branch had a varying number of entries (depending on the object's original size). A branch containing one integer per entry and having a basket size of 32KB will write a basket to disk every 8000 events; a branch with entries containing a collection greater than 32KB will write a basket for every entry. Thus, the contents of a single entry with many varying-sized branches may be spread throughout a file. Due to the latency of reading non-contiguous baskets, accessing many branches of an entry is inefficient on modern disks and high-latency network access.

We introduced the class TTreeCache that prefetches a set of baskets, given an event range, a cache size, and a set of branches. It reduces by several orders of magnitude the number of I/O transactions needed to read the content of TTree.

The TTreePerfStats class is used to analyze the qualitative and quantitative benefits of various configuration of the TTreeCache. Figure 1 shows the improvements provided by the TTreeCache on a sample ATLAS data file. The graph shows the offset of reads performed by ROOT within the file versus the entry number. One can see a stair effect; each time ROOT decided to prefetch, its reads were closely clustered within the file.

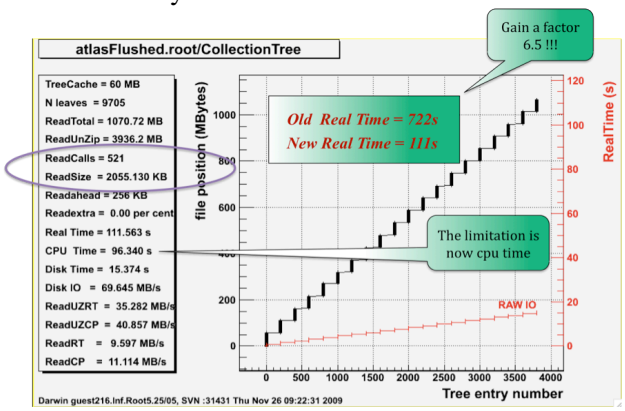

Figure 1

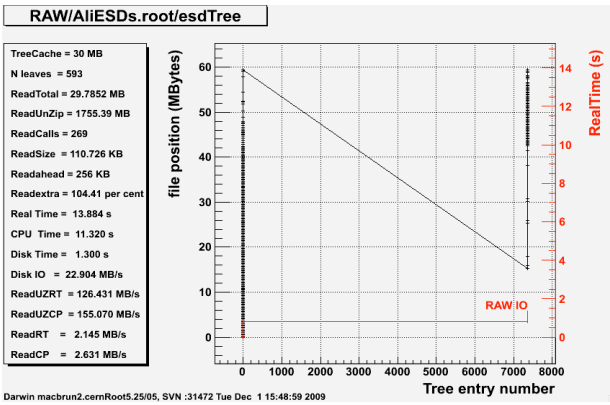

Figure 2

The TTreePerfStats results in Figure 2 show that, even when using prefetching, there were backward seeks and gaps in the reads. These prevent the Linux operating system from aggressively prefetching the file content.

3. Baskets Clustering

To reduce - and often remove - the gaps and backward seeks, ROOT v5.26 introduced enhancements to the sizing and management of each branch's baskets. Prior to v5.26, any customization was from manual tuning for a specific use pattern and data layout. As the files of the LHC experiments now commonly containing thousands of branches, tuning the I/O by hand is impractical.

In v5.26, ROOT introduced TTree::OptimizeBaskets to automatically resize the baskets. TTree::OptimizeBaskets attempts to have each branch basket hold the same number of events and reduce the total memory use. In addition, the TTree now automatically flushes its baskets to disk, creating a "cluster" of entries. After a file is created and a given amount of data (30MB by default) is buffered, all baskets are compressed and flushed to disk; the entry number is recorded. Afterward, for

every multiple of entry, the baskets are flushed. Thus, all entries within a cluster can be loaded by a single read. The frequency of this flushing can be customized by calling TTree::SetAutoFlush. The first time TTree::FlushBaskets is invoked, TTree::OptimizeBaskets is also invoked.

For clustered files, the number of entries prefetched by TTreeCache is a multiple of the flush interval. This removes the backward seeks when prefetching in files written with ROOT v5.26, increases the average read size, and results in a dramatic increase in the I/O speed from disk.

4. ROOT Collaboration with CMS.

The CMS experiment's framework provides a complex use case for ROOT I/O, often pushing the technical boundaries of what can be accomplished. This section provides a few illustrative examples how an experiment's framework may use ROOT in unexpected ways, and how the two teams constructively collaborated to provide better performance. While this section examines two potential pitfalls, we emphasize the overall scheme works remarkably well over a wide range of CMS use cases.

To generalize, the CMS framework file holds four types of information: metadata, run, luminosity section, and event information. If a file holds all CMS detector events recorded during a nominal hour of running, one would expect 1 or 2 runs, 150 luminosity sections, and 1 million events. When transitioning between lumi/runs, expensive CPU and data-lookup operations are performed. Our hypothetical file would have about 7,000 events per transition. Unfortunately, the detector event ordering does not exactly match the TTree entry ordering. Events are ordered by the time they finish processing in the high-level trigger online computing farm, and filtered according to physics contents. Thus, there are often less events per luminosity section than nominal, the detector order of events has no relation to the TTree order, and events in the same luminosity section are not necessarily stored contiguously in the TTree.

This file organization challenges the ROOT I/O optimizations discussed previously; the CMS team must have a sufficiently deep understanding of ROOT in order to balance the CPU cost of luminosity transitions with the I/O cost of reading TTrees out of order. Close collaboration between the framework and ROOT teams to build expertise and understanding is a requirement for high-performance I/O in an experiment. For CMS, the solution was a compromise – all events in a given run and luminosity sections are processed in TTree order in a file. Unnecessary (forward) skips in the TTree ordering may occur if a luminosity section is not contiguous within a file, but, as files are reprocessed in CMS, the luminosity sections tend to be contiguous, especially for user-oriented file formats.

Another performance benefit realized from the collaboration between CMS and ROOT is the number of active TTrees used during processing events. The I/O protocol implementations may maintain caches external to ROOT: in this case, ROOT will inform the underlying protocol when to read-ahead into the cache and when to empty the cache. If there are multiple TTrees being read simultaneously during event processing, they share one I/O cache: the prefetching of buffers in one TTree will cause the other TTree's buffers to be dropped. ROOT cannot prioritize the order of the requests for the underlying I/O layer; so one asynchronous request will finish before the next one is begun. This effectively makes the asynchronous requests synchronous, and decreases overall performance. With this understanding, CMS reduced the number of TTrees containing event information to one.

5. Optimization of the ROOT I/O streaming engine.

5.1. Memberwise Streaming

When storing a homogenous collection of objects in ROOT, the user can select between object-wise and member-wise streaming. When streaming object-wise, each object in the collection is streamed with all its data members contiguous on disk. When streaming member-wise, the collection's instances of a given data member are stored contiguous on disk. For example, suppose a class has data members x,y and z, and we stream a collection of three objects of this type: (1, 2, 3). With

object-wise streaming, the first three data written will be from object 1: (x_1,y_1,z_1); then, all the values of the object 2: (x_2,y_2,z_2); and finally object 3: (x_3,y_3,z_3). With member-wise streaming, the order will be the x values: (x_1,x_2,x_3), then the y values: (y_1,y_2,y_3), and finally the z values: (z_1,z_2,z_3).

Objectwise Streaming Memberwise Streaming

Member-wise streaming is used in a TTree whenever a collection is split; when the collection is not split, object-wise streaming was used prior to v5.26.

Starting in version v5.26, member-wise streaming is the default for streaming all collections. This resulted in better data compression and faster de-streaming of the data. We evaluated the impact of moving to member-wise streaming using 5 different CMS data files

- *cms1.root*: An older RECO file using split-level 99.
- *cms2.root*: A recent non-split RAW data file.
- *cms3.root*: A recent non-split RECO file
- *cms4.root*: An example lepton-plus-jet analysis in a format known as a user PAT-tuple (split)
- *cms5.root*: An example AOD (analysis object data) file. It is not split; the objects are a strict subset of the RECO objects.

We rewrote all these files using the v5.26 basket-clustering algorithm and using both member-wise streaming and object-wise streaming. Tables 1 and 2 show CPU time used to completely read the file, including loading libraries. When testing, the file was always pre-loaded to the OS page cache (as we are evaluating CPU time). The number of events read was varied to normalize the CPU time to about 10 seconds. Each file was read independently from the experiment's data processing framework using a library generated with TFile::MakeProject.

The files written in member-wise streaming mode are between 2% and 10% smaller compared to their object-wise counterpart. The CPU time for reading member-wise files is 12% lower for split files and 30% lower for non-split files. Thus, the improvement warranted switching the default mode to member-wise.

Table 1: Split Files

File name	Memberwise	Size	Cpu time to read
cms1.root	No	17.5 GB	10.55s ± 0.15 (2200 entries)
cms1.root	Yes	16.8 GB	9.12s ± 0.08
cms4.root	No	1.47 GB	10.18s ± 0.19 (2500 entries)
cms4.root	Yes	1.43 GB	9.24s ± 0.06

Table 2: Non Split Files

File name	Memberwise	Size	Cpu time to read
cms2.root	No	1.65 GB	10.95s ± 0.05 (1000 entries)
cms2.root	Yes	1.53 GB	8.20s ± 0.05
cms3.root	No	0.780 GB	10.59s ± 0.05 (700 entries)
cms3.root	Yes	0.717 GB	8.29s ± 0.08
cms5.root	No	1.55 GB	10.20s ± 0.17 (700 entries)
cms5.root	Yes	1.40 GB	8.09s ± 0.08

5.2. Optimizing the ROOT I/O streaming engine.

After improving the performance of reading data from the disk or network, the current bottleneck for streaming an object is now CPU time required for ROOT to turn the compressed basket into valid C++ objects. This streaming is implemented by the TStreamerInfo class.

5.2.1. *Analysis of the Issues*. Prior to version v5.28, the CPU intensive, inner most part of the StreamerInfo based ROOT I/O code[3], was centred around a generic **`switch`** statement used via a large function template for several cases: single object, collection of pointers, collection of objects, caching mechanism for schema evolution, and split collection of pointers. This implementation was chosen to improve code localization and reduce duplication.

After analyzing the assembler produced by optimizing compilers, we noticed many initializations (and resulting memory fetches) were done without need. Some initializations were intentional consequences of hand optimization, while others were unintentional and due to the overly aggressive compiler. Sometimes the compiler optimization resulted in reordered code with more memory fetches than necessary. Many **`if`** statements were necessary to tweak the implementation's performance, to avoid the proliferation of cases in **`switch`** statements, and to avoid code duplication. To stay generic, the code could only access the content of collections using **`operator[]`** in order to leverage function overload and being able to use the same template code for all the supported cases. For the collection proxy case (which includes STL containers), the **`operator[]`** was also very generic with the same function implementation, littered with **`if`** statements used for both direct access and for iterative containers. This reliance on an **`operator[]`** also prevented the efficient use of the **`next`** operator when looping over associative collections (e.g. std::map) with a direct access operator. As the same block of code supported both reading a single object and reading the content of a container, every single object read went through a spurious single-iteration loop.

5.2.2. *Possible Solutions*. Continuing to use template functions by customizing the implementation of the **`switch`** cases (depending on the 'major' case) would remove unnecessary loops in the case of single objects. It would also allow the removal of many of the **`if`** statements.
However, this implementation would not allow optimization for a specific collection type (i.e., an STL vector) because they are 'hidden' behind the collection proxy concept, which allows one single abstract interface for all collections. The large **`switch`** statements would still be present, preventing many compilers from performing proper optimization.

Alternatively, ROOT could have developed a solution where, as part of the class dictionary[4], a set of functions would have been provided for each class. Such a solution would be incomplete; it would not support the case of classes that need to be emulated (when the classes are part of a ROOT file but are not defined in any loaded C++ libraries).

5.2.3. *Selected Solution*. In ROOT v5.28, the large **`switch`** statements were replaced by a set of customized function calls. The required functions are selected during a StreamerInfo's compilation and are recorded using function pointers.

One advantage of this approach is that new or specialized streamer implementations may be added easily. For example, we have customized the streaming in the following cases:

- Single objects (no more unnecessary loops).
- Loop with known number of iterations for statically-sized vectors of pointers.
- Loop with simple increment (this case includes vectors and all emulated collections).
- Loop using an iterator for a compiled collection.

This technique was also applied to a few other places in the ROOT I/O implementation, including in **`TClass::Streamer`**, as shown in Example 1 and 2. The initial value of **`fStreamerImpl`** is **`StreamerDefault`** a function whose sole purpose is to properly set **`fStreamerImpl`** via the call the function **`Property`**. The advantage of this delayed initialization is to guarantee that the choice of implementation is made with the exact same information as the **`switch`** statement had.

We also removed **`if`** statements that can be resolved by examining class layouts. We are able to strip out some of the functions by explicitly inlining inner functions in the outer functions. The outer loop is now much simpler and can be overloaded in the various TBuffer implementations. This removes code that was needed only to support the writing of ROOT files in XML format or in an SQL database.

This technique increases code duplication, which is mitigated by the fact that our streaming code has been very stable for many years. This stability gives us good confidence that we will not have to modify (for functionality at least) the code and thus reducing the maintenance cost associated with this duplication. To battle duplication in the code written by the developers, smaller functions are combined by using function templates.

```
void TClass::Streamer(void *object, TBuffer &b) const
{
   // Stream object of this class to or from buffer.
   switch (fStreamerType) {
      case kExternal:
      case kExternal|kEmulated: { ...; return; }
       case kTObject: { ...; return; }
Etc.
```

Example 1: Inner code before the optimization.

```
inline void Streamer(void *obj, TBuffer &b) const {
   // Inline for performance, skipping one function call.
   (this->*fStreamerImpl)(obj,b,onfile_class);
}
```

```
void TClass::StreamerDefault(void *object, TBuffer &b) const
{
   // Default streaming in cases where Property() has not yet been called.
    Property(); // Sets fStreamerImpl
   (this->*fStreamerImpl)(object,b);
}
```

Example 2: Inner code after the optimization

As of v5.28, the optimization has been applied to the general infrastructure of the read engine and the most common use cases; this includes the streaming of numerical types. When using these enhancements, we see a time reduction of about 20% on several ATLAS and CMS sample data files.

6. Conclusion

Over the last year, the main focus of the ROOT I/O team has been to significantly enhance the I/O performance both in terms of CPU and real time. We have consolidated the code by leveraging old and new tools to reduce the number of defects, including Coverity, Valgrind and traditional test cases, often provided by users via the ROOT forum and the bug tracking system. Thanks to our collaboration with the LHC software framework teams (as highlighted by CMS), these enhancements resulted in substantially more efficient use of the available CPU resources for end-users.

References

[1] R. Brun and al., "ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization", Computer Physics Communications; Anniversary Issue; Volume 180, Issue 12, December 2009, Pages 2499-2512.

[2] The ROOT Team, "TTree", http://root.cern.ch/download/doc/12Trees.pdf

[3] ROOT version 5.26 implementation of the TStreamerInfo class:
http://root.cern.ch/root/html526/TStreamerInfo.html
http://root.cern.ch/viewvc/tags/v5-26-00/io/io/src/TStreamerInfoReadBuffer.cxx?view=markup

[4] How to generate a ROOT dictionary, http://root.cern.ch/drupal/faq#n676