FERMILAB-CONF-12-799-CD

ROOT I/O Improvements

Ph Canal
Fermi National Laboratory, Batavia, IL, USA

E-mail: pcanal@fnal.gov

Abstract. In the past year, the development of ROOT I/O has focused on improving the existing code and increasing the collaboration with the experiments' experts. Regular I/O workshops have been held to share and build upon the various experiences and points of view. The resulting improvements in ROOT I/O span many dimensions including reduction and more control over the memory usage, reduction in CPU usage as well as optimization of the file size and the hardware I/O utilization. Many of these enhancements came as a result of an increased collaboration with the experiments' development teams and their direct contributions both in code and quarterly ROOT I/O workshops.

1. Introduction

As the Large Hadron Collider experiments are starting to take significant amount of data, some of their focus has turned to optimizing the existing infrastructure and to prepare for the upcoming computing architectures upgrades. One of the areas they have focused on is the performance of the entire chain of Input/Output, from retrieving the file from storage to recreating the data objects for processing and back to writing the output files. For the last year, the CMS and ATLAS experiments have actively participated in quarterly ROOT[1] I/O workshops, which have greatly enhanced the synergies and resulted in several improvements in ROOT I/O for all users.

Since 2005, we no longer benefit from the automatic gains made thanks to the increase in processor clock speed. Instead, the clock speed has reached a plateau. The increase in the number of transistors on a chip now translates in an increase in the number of cores rather than an increase in performance of each core. In conjunction, with a slower decline in memory price, this means that in order to benefit from the processing power of the latest chips, we need to review our software architecture and increase the amount of work that can be done in parallel within a similar memory budget. A significant amount of the effort concentrated on the I/O has thus gone towards making the I/O as parallelizable as possible both inside the same process, via threads, or when coordinating an extremely large number of processes all writing to the same output stream.

2. Performance Enhancement

Based on feedback from careful analyses of the performance characteristics of users' production code, many areas of the ROOT I/O and TTree[5] packages have been made more efficient in either runtime or memory usage. The analysis has been made using a variety of tools including igProf[2], callgrind[3][4] and AMD CodeAnalyst. For example, we improved by a factor 3 to 10, depending on the length and

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

complexity of the class name, the performance of the TTree::SetBranchAddress and TTree::SetAddress routines.

TBasket memory management went through several updates. We first dramatically reduced the amount of memory allocations induced by the management of the TBasket and TBuffer objects for each of the branches. Previously, one TBasket object and one TBuffer object and its associated memory buffer were created for each and every basket on file. We updated the code to create a single TBasket object and a single TBuffer object for each branch for the lifetime of the TTree object. The memory buffer associated with the TBuffer object is also created once and reallocated only when the buffer size is reset (for example by the AutoFlush mechanism) and when the user's data does not fit in the currently allocated memory; in this intermediary implementation this buffer size was never reduced. The same minimization has been applied to the scratch area used to read the compressed version of a basket from the file.

Further testing, including real use cases from the CMS experiment, showed that the fact that the internal buffers were never reallocated nor shrunk led to significant memory waste in the case where a small fraction of the events were much larger than the others. In this case, the buffer size would settle to accommodate those large events but the buffer were mostly empty for all the other events. This lead to several cases where the program would grow past the allowed memory budget. Another example was the case where two branches' content would alternatively be large or small. In the previous implementation, the average memory use was essentially the size of only one of the two large branches while in the new implementation the average memory use is the sum of the size of the two large branches.

To prevent this unlimited growth of the TBasket's buffer, the buffer's size is reduced if and only if the TBuffer size is greater than the following three values:

- twice the data in the current basket
- twice the average data in each basket of this branch
- twice the requested basket size (TBranch::GetBasketSize)

In this case, the size of the buffer is reduced to the maximum of the size of the data in the current basket and the average size of the baskets and the requested buffer size; the resulting size is also aligned to the next highest multiple of 512.

To further reduce the memory used by a TTree, we refactored the code used for reading and writing the TBasket data. Instead of having two buffers, one for compressed data and one for uncompressed data, for each branch, a single transient buffer holding the compressed data is now managed by the TTree object and shared by all the branches. When reading the data using multiple threads, this buffer can be made thread local.

Furthermore, we significantly reduced the overhead of the TTree::GetEntry function. In one of the case where this overhead is the most significant, namely when reading an uncompressed branch created using a leaflist, we reduced by 40% the time taken by GetEntry. To achieve this result we reduced the number of lookups by caching the current basket and its range information. We reordered the set of tests to reduce the number of conditional jumps in most frequent cases and we gave explicit hints to the compiler on which conditional branches were the most likely.

We reordered the set of tests to reduce the number of conditional jumps in the most common cases. We leveraged the ability to give hints to the processor on whether a branch is more likely to be taken or not, hence improving the quality of the branch predictions.

We optimized the use of the TTreeCache memory by insuring that it is completely used even when the clusters are smaller than the memory allocated to the cache.

3. Multi-Threading

Some of the existing ROOT interface inherently requires the notion of a current or global directory and file. For example, TObject::Write's stores the object data in the TDirectory object located at the

value of global variable gDirectory. This global variable gDirectory can be updated either by direct manipulation or by calls to the TDirectory::cd function.

When this global variable is shared amongst many threads, each thread needs to put a lock on the variable between the time it is set and the time it is consumed. In typical applications this is essentially the whole life of the thread where gDirectory is set at the beginning and Write is called toward the end. This time can be reduced by explicitly (re)setting the value before it is used. However this still results in longer than necessary locks and thus higher serialization of the code. It also requires the user to explicit take and release locks making the code more difficult to write and maintain.

To completely alleviate these concerns, we chose to modify the semantic of gDirectory and gFile. Instead of being a global variable shared by all the threads, we made both variables thread local. When a new thread is started, the initial value of gDirectory points to the TROOT singleton (gROOT) and any modification of the value gDirectory by a thread is only seen by this thread.

This model still does not directly support sharing TFile amongst threads (i.e. a TFile must be accessed from one and only one thread). Whenever a TFile's control is passed between threads, the code must explicitly reset gDirectory to another value or there is a risk for this gDirectory to point to a stale pointer if the other thread deletes the TFile object since a TFile deletion will only affect the value of the local gDirectory.

3.1. Asynchronous Prefetching

To further improve the gain in process completion time, we added to the TTreeCache the ability to asynchronously read ahead from the file. The default TTreeCache behaviour is to fetch in one single read from the file, all the data needed for a block of entries. The TTreeCache can either learn automatically or be explicitly told which subset of the TTree's branches is used and should be read as part of the block read. Whenever the TTree is requested to load an entry that is outside the range of entries currently cached, a block read is triggered. Without asynchronous prefetching, the processing thread is then blocked until all the data for the block has been read from the file.

When the asynchronous prefetching is enabled, a separate thread will fetch the TTreeCache blocks that are likely to be needed next. In the meantime, the main thread can continue the normal data processing.

In order to enable prefetching the user must set the rootrc variable TFile.AsyncPrefetching.

```
gEnv->SetValue("TFile.AsyncPrefetching", 1);
```

In addition the result of the prefetches can be cached on the local disk by setting the local cache directory in which the file transfer can be saved. For subsequent reads of the same file the system will use the local copy of the file from cache. To set up a local cache directory, a client can use the following commands:

```
TString cachedir = "file:/tmp/xcache/";
// or to use xrootd on port 2000:
// TString cachedir = "root://localhost:2000//tmp/xrdcache1/";
gEnv->SetValue("Cache.Directory", cachedir.Data());
```

4. New Features

4.1. Automatic support for more than one TTreeCache per file

Using a TTreeCache has become one of the essential components for efficient TTree read operation. It is currently not yet used by default since under some read patterns, the current algorithm might lead to a decrease in performance. One of the manual steps has been the setup needed to cache two TTree

from the same file, as there could only be one active TTreeCache at a time for a given TFile object. In particular this was limiting in the case of many experiments' production file that contained two or more large TTree. In these cases, in order to benefit from the TTreeCache behaviour for both TTree, the experiment's code had to explicitly set and reset the TFile's TTreeCache before and after each TTree I/O operation.

An alternative has been to use two distinct TFile objects reading the same physical file and to access each TTree from its own TFile. We have implemented a simpler solution to the problem by updating the TFile and TTreeCache code to allow for more than one TTreeCache being active at the same time on a single TFile. TTree::SetCacheSize(Long64_t) no longer overrides nor deletes any existing cache. The operations that set and get information about a TFile's cache now takes an extra argument, which is the TTree (for example) pointer that the cache is associated with. The TFile keeps a mapping between the cache object and the object, usually a TTree, to which the cache is associated. Each cache is completely independent.

Previously, the intent was to only have a single cache per TFile in order to allow for the coordination of all the reads and tries to eliminate any possibility of reading twice the same disk area. However since the different TTree could have very different access pattern and are always accessed independently, it is very difficult to gather enough information from one TTree while processing the other to accurately and usefully predict the next block. With independent caches, the worst-case scenario is the rare occurrence of two large TTree that are strongly intertwined in the file and being read in sync. In this extreme case either the TTreeCache will have to issue more reads than absolutely necessary and over-read each block.

4.2. Variable TTree cluster size

A TTree cluster is a set of baskets containing all the data for an integral number of entries that will be read in a single I/O operation by the TTreeCache. Each TTree has a different cluster size depending on user configuration and data content. Previously, when merging files, the resulting TTree was assuming the cluster size of the first file. Since on disk the physical clusters were not aligned on this cluster size, this resulted in attempting to read misaligned set of baskets.

When merging files, the TTree now records the cluster size of each of the input TTree, resulting in optimal I/O throughput when reading the merged file.

4.3. LZMA Compression

ROOT I/O now supports the LZMA algorithm to compress data in addition to the ZLIB compression algorithm. LZMA compression typically results in smaller files, but takes more CPU time to compress data. For example, on reconstruction data, a good setting is to use the compression level 4, which can reduce the file size by about 20% for an increase of CPU time of less than 10% compared to ZLIB level 7, while higher compression level increase the CPU time more than they decrease the file size.

There are three equivalent ways to set the compression level and algorithm supported by the classes TFile, TBranch, TMessage, TSocket, and TBufferXML. For example, to set the compression to the LZMA algorithm and compression level 4.

```
1. TFile f(filename, option, title);
   f.SetCompressionSettings(ROOT::CompressionSettings(ROOT::kLZMA, 4));
2. TFile f(filename, option, title,
           ROOT::CompressionSettings(ROOT::kLZMA, 4));
3. TFile f(filename, option, title);
   f.SetCompressionAlgorithm(ROOT::kLZMA); f.SetCompressionLevel(4);
```

The compression algorithm and level settings only affect compression of data after they have been set. `TFile` passes its settings to TTree's branches when the branches are created. This can be overridden by explicitly setting the level and algorithm for the branch. These classes have the following methods to access the algorithm and compression level.

```
   Int_t GetCompressionAlgorithm() const;
   Int_t GetCompressionLevel() const;
   Int_t GetCompressionSettings() const;
```

If the compression level is set to 0, then no compression will be done. All of the currently supported algorithms allow the level to be set to any value from 1 to 9. The higher the level, the larger the compression factors will be (smaller compressed data size). The tradeoff is that for higher levels more CPU time is used for compression and possibly more memory. The ZLIB algorithm takes less CPU time during compression than the LZMA algorithm, but the LZMA algorithm usually delivers higher compression factors.

The header file core/zip/inc/Compression.h declares the function "CompressionSettings" and the enumeration for the algorithms. Currently the following selections can be made for the algorithm: kZLIB (1), kLZMA (2), kOldCompressionAlgo (3), and kUseGlobalSetting (0). The last option refers to an older interface used to control the algorithm that is maintained for backward compatibility. The following function is defined in core/zip/inc/Bits.h and it set the global variable.

```
   R__SetZipMode(Int_t algorithm);
```

If the algorithm is set to kUseGlobalSetting (0), the global variable controls the algorithm for compression operations. This is the default and the default value for the global variable is kZLIB.

4.4. TClonesArray

To increase the performance when filling and writing TClonesArray collections, we introduced `TClonesArray::ConstructedAt` which always returns an already constructed object. If the slot is being used for the first time, it calls the default constructor otherwise it returns the already constructed object. In addition, if a string is passed as the 2nd argument to the ConstructedAt function, ConstructedAt will call Clear(second_argument) on the object before return it. This allows replacing code like:

```
for (int i = 0; i < ev->Ntracks; ++i) {
   new(a[i]) TTrack(x,y,z,...);
   ...
   ...
}
...
a.Delete(); // or a.Clear("C")
```

with the more efficient:

```
for (int i = 0; i < ev->Ntracks; _++i) {
   TTrack *track = (TTrack*)a.ConstructedAt(i);
   track->Set(x,y,z,....);
   ...
   ...
}
...
a.Clear();
```

This reduces the CPU time and memory fragmentation when the class contained in the TClonesArray, for example TTrack, allocates memory in its constructor. With the old techniques, the call to destructor and constructor of the contained class (TTrack) was required. Consequently at each iteration and for each element of the array, we had to allocate, construct and then delete all the resources held by each element.

5. Parallel Merging

5.1. Merging User Objects.
We introduced a new explicit interface for providing merging capability. If a class has a method with the name and signature:

```
Long64_t Merge(TCollection *input, TFileMergeInfo*);
```

It will be used by a TFileMerger object, and thus by PROOF, to merge one or more other objects into the current object. If this method does not exist, the TFileMerger will use a method with the name and signature:

```
Long64_t Merge(TCollection *input);
```

The object `TFileMergeInfo` can be used inside the `Merge` function to pass information between runs of the `Merge`. In particular it contains:

```
   TDirectory  *fOutputDirectory;
   // Target directory where the merged object will be written.
   Bool_t       fIsFirst;
   // True if this is the first call to Merge for this series of
object.
   TString      fOptions;
   // Additional text based option being passed down to customize
   // the merge.
   TObject     *fUserData;
   // Place holder to pass extra information.  This object will be
   // deleted at the end of each series of objects.
```

The default in `TFileMerger` is to call `Merge` for every object in the series (i.e. the collection has exactly one element) in order to save memory (by not having all the object in memory at the same time). However for histograms, the default is to first load all the objects and then merge them in one go; this is customizable when creating the `TFileMerger` object.

5.2. Enabling Parallel Merging

With the tools currently available, if an application is run on many worker nodes each producing some output and all the data needs to be merged into a single file, the I/O time can be much larger than the processing. In this kind of scenario, each work node is processing and storing their part of the output to a local disk. Once the processing is finished, the worker will copy its output file upstream to a server. As soon as all the workers have uploaded their output, the server will then read all those output files and write the single merged output file.

For example, using disks rated at 100MB/s, if each worker produces 100MB in about 100 seconds and if there are 1000 workers nodes, the upload, which will have to be serialized on the server side, would take around 1000 seconds to finish. The merge itself would take another 2000 seconds at minimum (to read the input and write it back to the single file). This would result in a job taking 100 seconds for processing but 3000 seconds for I/O!

In such a scenario, minimizing and overlapping the I/O as much as possible is absolutely necessary. When relying on parallel merging, rather than writing to local disk the worker node starts uploading to the server as soon as some nominal fraction of their data is available. When receiving the data, the server merges it directly into the final output file, hence saving the unnecessary reads and writes of the previous solution. This also allows the uploads to be done almost completely in parallel with the processing, in our example this upload is then finished in just 100 seconds (assuming sufficient network bandwidth to the server). Since the workers start writing to the output file in parallel, it also only takes a total of 1000 seconds to write the output file. With the parallel merge, our example can be reduced from 3100 seconds down to 1000 seconds.

To support the implementation of parallel merging facilities, we introduced two new classes. TMemFile implements a completely in-memory version of the TFile class. TParallelMergingFile is a TMemFile that connects to a parallel merger server (possibly just a separate thread in the current process) which, on a call to Write, will upload its content and reset the TTree objects and any objects of classes with a ResetAfterMerge function are reset.

A TParallelMergingFile is created whenever the option "?pmerge" is passed to `TFile::Open` as part of the file name. For example:

```
   TFile::Open("mergedClient.root?pmerge","RECREATE");
   // For now contact localhost:1095
   TFile::Open("mergedClient.root?pmerge=localhost:1095","RECREATE");
   TFile::Open("rootd://root.cern.ch/files/output.root?pmerger=pcanal:p
assword@locahost:1095","NEW")  ;
```

The tutorials treesClient.C and fastMergerServer.C demonstrates the use of TMemFile to create any type of server. We introduced the tutorials `parallelMergerClient.C` and the temporary tutorials `parallelMergerServer.C` to demonstrate the parallel merging itself. `parallelMergerServer.C` is the prototype of the upcoming parallel merger server executable.

6. Conclusion

In conjunction with the experiments, ROOT I/O has continued to improve and to migrate towards the upcoming highly parallel computing architectures. Beyond the many performance improvements or other new features, we have introduced a significantly better way to store in a few files the results of many parallel processes or threads. The first building blocks of a large-scale parallel merging facility has been put in place.

References

[1] R. Brun and al. 2009 "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] Eulisse G, Tuura L 2004 IgProf profiling tool, *Proc. Computing In High Energy And Nuclear Physics (CHEP 2004), Interlaken, Switzerland, 2004*

[3] Nethercote N and Seward J. Valgrind 2007 A Framework for Heavyweight Dynamic Binary Instrumentation" *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007).*

[4] Weidendorfer J, Kowarschik M and Trinitis C 2004 A Tool Suite for Simulation Based Analysis of Memory Access Behavior. *Proceedings of the 4th International Conference on Computational Science (ICCS 2004), Krakow, Poland, June 2004.*

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