Optimizing High-Latency I/O in CMSSW

Brian Bockelman[1]

118C Schorr Center, Lincoln NE, USA, 68588-0150

bbockelm@cse.unl.edu

Abstract. To efficiently read data over high-latency connections, ROOT-based applications must pay careful attention to user-level usage patterns and the configuration of the I/O layer [1,2]. Starting in 2010, CMSSW began using and improving several ROOT "best practice" techniques such as enabling the TTreeCache object and avoiding reading events out-of-order. Since then, CMS has been deploying additional improvements not part of base ROOT, such as the removal of the TTreeCache startup penalty and significantly reducing the number of network roundtrips for sparse event filtering. CMS has also implemented an algorithm for multi-source reads using Xrootd. This new client layer splits ROOT read requests between active source servers based on recent server performance and issues these requests in parallel.

1. Introduction

In the last two years, with the introduction of wide-area data access in CMS [3], the performance of the CMS software framework, CMSSW, has become increasingly important. In this context, we define wide-area to be:

- Round trip time (RTT) between client and server **greater than 20ms**. We assume the client and server are both on high-speed research networks often found at universities or labs. Within the same continent, the latency is typically less than 50ms; between continents, it is typically over 100ms.
- Per-TCP stream **bandwidth less than 1MB/s**. While there is typically more than 100MB/s available between client and server on an R&D network, the effective per-TCP-stream performance is greatly diminished by site firewalls and WAN packet loss.

It is likely this is the network environment encountered by users doing remote analysis from laptops, an important use case. This is also expected conditions for in the CMS AAA data federation [3]. Further, optimizations done for high-latency networks will often benefit local low-latency networks.

In this paper, we discuss various methods implemented within CMSSW to reduce the impact of running on top of high-latency networks with the Xrootd protocol [5]. Most techniques fall into one of two categories:

- Aggregating multiple I/O read requests into one client/server network interaction.
- Using multiple source servers per client.

While the techniques discussed here were done using Xrootd, they should apply to other protocols used over high-latency links, such as HTTP or remote NFSv4.

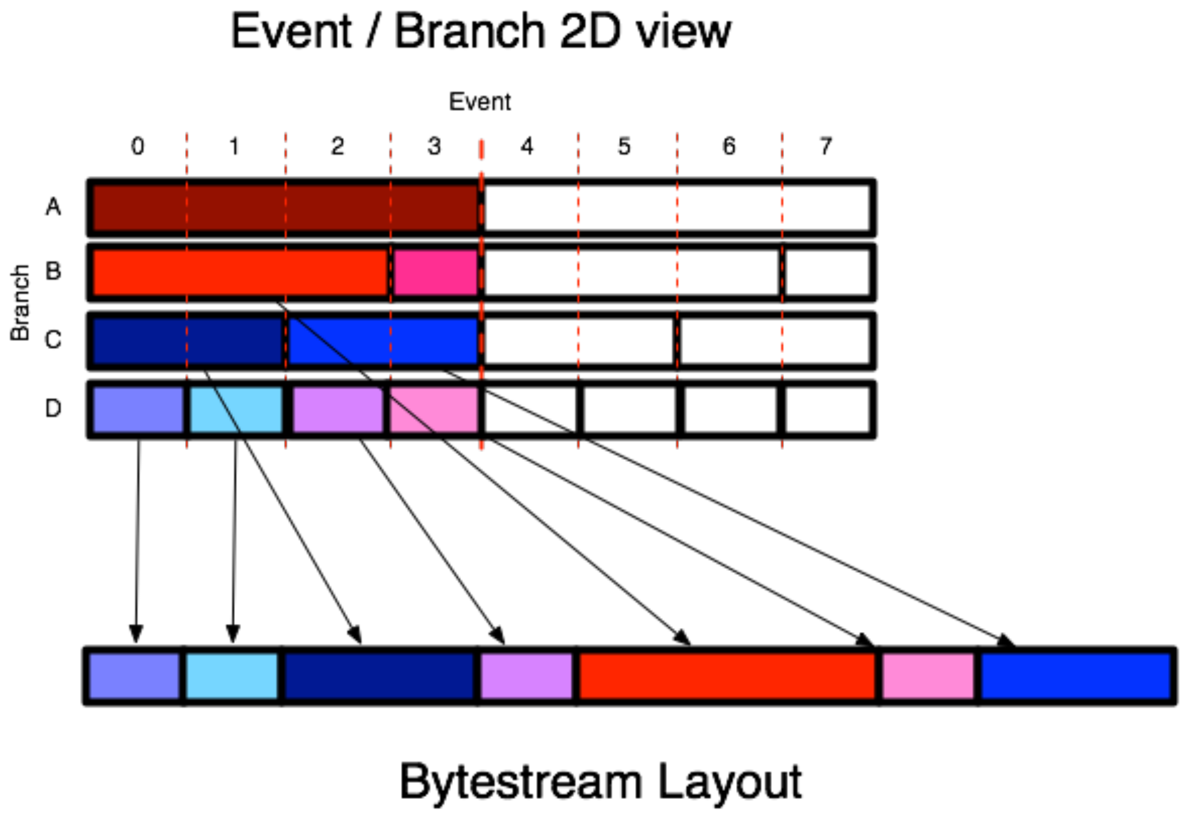

Figure 1. An example of the logical layout of a ROOT file and how it maps to a bytestream. In this example, there are 8 events and 4 branches organized into two clusters. The ROOT data is organized into “baskets” containing one or more event’s worth of data. Note that different object sizes in each branch can result in a different number of events per baskets. The colors illustrate how the baskets containing data for events and a branch map into the 1D bytestream.

2. Background

ROOT’s TTree object provides a column-like data store for the High Energy Physics (HEP) community. It is used widely in the field; collaborations not using ROOT are the exception. The data is organized into events (rows of the data store) and branches (columns). An event corresponds to a physical occurrence – such as a particle collision in an accelerator. The data in an event is organized into multiple serialized C++ objects; each branch is a named C++ object of a certain type. Each branch object in the event, which may be relatively small, is not stored separately on disk. Rather, objects are collected in buffers (one buffer per branch); when the buffer is full, the objects are compressed and written to disk. The buffer on disk is referred to as a basket. This organization is done so individual branches can be read efficiently. Under the assumption that objects in the same branch will have similar data and thus be highly compressible, the baskets are compressed before written to disk. ROOT will keep a dictionary in-memory containing the location of all baskets; this is stored with the file and used later for reading.

Buffer sizing is a difficult problem as not all objects are a fixed-size; since buffers are only written out when full, poor buffer choices can cause the objects of a single event to be spread across many parts of a file (consider the case where buffer A can hold 1000 objects of type X and buffer B can hold 1 object of type Y). To preserve some level of data locality for events, ROOT will organize the data into clusters; a set of events which are contiguous on disk. A data cluster is formed by flushing the contents of all buffers once every N events, even if they are not full. A typical value of N is 5-20, depending on event size.

Figure 1 shows a stylized representation of the 2D-view of a TTree and an example of how it may be written to disk.

If a user requests branch A from event X, ROOT will calculate the appropriate basket containing the object. If the basket is not in memory, it will read it from disk. ROOT will keep one uncompressed basket per branch in case if a user will read out additional events. So, if there are 1,000 baskets in a TTree, a user reading all branches and all events from a remote server will incur the penalty of 1,000 network round-trips. This is a prohibitive penalty, especially over high-latency links. To avoid this penalty, a built-in object – the "TTreeCache" – will cause multiple baskets for popular branches to be fetched upon request. When event X is accessed, TTreeCache will try to fetch as many clusters of data as can fit in the memory buffer, starting with the cluster containing X. If there are 1,000 baskets in a TTree – but only one cluster – theoretically, the TTreeCache could request all baskets in one network round trip.

3. Aggregating Read Requests

3.1. TTreeCache startup optimization
To operate, the TTreeCache needs a list of "popular branches" in the TTree. This can be specified by the user, but users are often unaware of the branches they are using (this information may be hidden by ROOT's abstraction layers). The default mechanism for determining the popular branches is for the TTreeCache to observe all reads for the first 20 events; any branch used during those events is declared to be popular and always prefetched.

During the first 20 event startup phase, ROOT's I/O behaves as if there is no TTreeCache at all – each basket is fetched in a separate network round trip. Over high latency links, the startup phase can cause a severe performance penalty.

To avoid the startup penalty, CMS utilizes a temporary, secondary TTreeCache. The secondary cache is manually trained to read all branches for the first 20 events; during those events, the CMS framework will record which branches are utilized. Once event 21 is requested, the CMS framework will switch to the "primary" TTreeCache, train it with the branches the framework observed used, and deletes the secondary. This approach will cause an over-read for the first 20 events (as all branches are read regardless of user request), but we've found the modest increase in data read (typically, <20MB) is worth the reduction in number of network requests.

3.2. Trigger pattern optimization
One common use pattern is for user applications is to read out one set of branches – the trigger branches - for each event and, based on the contents of the data read a "trigger" may fire, causing ROOT to read out the remaining branches for the event. We refer to this as the "trigger pattern". Depending on the physics analysis, a trigger may fire infrequently; if it occurs rarely enough, it may

never do so during TTreeCache training period. In this case, only the trigger branches will be prefetched to the TTreeCache. Since the non-trigger branches are outside the TTreeCache, ROOT will fall back to its default behavior and perform one read per basket for these branches.

When the application follows the trigger pattern, reading the non-trigger branches can dominate the I/O time due to the number of individual reads. As the CMS framework already tracks the trigger branches for the TTreeCache startup optimization, we know when the user requests a non-trigger branch. Instead of passing the read request to ROOT, we create a temporary TTreeCache and train it for the non-trigger branches and a single event. The temporary TTreeCache will fetch a single basket from each branch in one network request.

3.3. Read Coalescing

When ROOT requests several baskets in one "vectored" read, it is common to see several nearly contiguous requests. For example, it may include a read at offset 500 of length 100, followed by a read at offset 700 of length 100. Because most modern file systems store data in 4096-byte-or-larger extants, the two reads will be serviced from a single I/O request to disk but incur the overhead of two read() syscalls.

The CMS framework will coalesce two nearby reads into a single, larger read issued to the storage system; any reads within 32KB will be coalesced. This will reduce the number of read() syscalls at a cost of increased bandwidth use. As read coalescing will likely cause the memory needed for buffering in the client, we allocate an additional static 256KB buffer for handling the vector read response. If more than 256KB of data is over-read through read coalescing, the CMS framework will break up the ROOT vector read into multiple vectored reads.

4. Multi-source Xrootd Client

On sufficiently high-latency links, small amounts of packet loss can cause significant delays in TCP throughput; we have observed cases where effective bandwidth for a TCP stream is less than 8Mbps on a network link with 10Gbps of capacity. A time-tested approach utilized by GridFTP [4] is to utilize multiple parallel TCP streams for a single file transfer.

Most multisource protocols optimize to copy the entire file to the client; the file is split into chunks and the chunks are distributed over different TCP streams. This approach is not usable for our case; we may be reading a small percentage of the file. The data requested by ROOT may not map neatly into pre-defined chunks. We decided to build a new multisource algorithm on top of the existing Xrootd infrastructure within the CMS framework. The algorithm uses no special Xrootd features, but the implementation does heavily utilize the fact that the Xrootd client provides an asynchronous-callback interface.

The algorithm had the following design goals:

- *Quality metric*: Determine a metric for quality of the source server; the algorithm should prefer servers with a higher quality over servers with lower quality.
- *Source discovery*: Actively balance transfers over multiple links in order to determine several high-quality sources of the file.
- *Recovery*: The client should be able to recover from transient I/O errors at a single source.
- *Do no harm*: Minimize the impact on the Xrootd service versus a single-source client. We should understand both average case and the worst case scenarios.
- *Balance*: Have the number of requests per source be proportional to source quality.

For each logical file open, the multisource algorithm maintains three sets of servers:

- *Active servers*: servers we are currently using to service reads (max of 2).
- *Inactive servers*: servers with an open file handle, but not used by default.
- *Disabled servers*: servers that have been used previously but had a fatal error.

The inactive sources and disabled sources are initialized empty. When a file is opened, the initial Xrootd server (the "redirector") will return a separate data server to handle data movement; the active sources set is initialized with this data server. Every 60 seconds, a new file open is attempted, excluding the data servers already returned. Any resulting server is added to the active sources. If there are already two servers in the active sources, the new data server is added to the inactive sources. If the redirector returns a "file not found" error, then the next file open is scheduled for 2 minutes in the future. This allows the client to eventually find all file sources.

When the client issues an I/O request, the following algorithm is applied:

1. Check current quality of each source. If a currently active source has worse quality than an inactive source, swap the two.
2. Randomly swap active and inactive sources with a 1% probability per 10MB requested.
3. Split the I/O request proportionally by volume in two according to the current active source quality. The two chunks are kept as contiguous as possible; this allows the server's disks to see mostly sequential I/O.
4. Issue the two requests to the active sources. Return result to client if both succeed. If a request fails, move corresponding source to the disabled sources; immediately reissue request from one of the inactive sources. Otherwise, a C++ exception is thrown.

5. Testing Results

We have written a module, IOExerciser, which takes two parameters:

- *Branch percentage*: The percentage of branches read for each event, weighted by total number of bytes in the branch.
- *Trigger prescale*: How often all branches are read for an event. For example, a prescale of 100 means that the entire event is read once every 100 events.

We ran IOExerciser with branch percentage set to 30% and trigger prescale set to 50; 1,000 events were read per test. These settings were chosen to be realistic for user analysis but also highlight the ROOT deficiencies we have aimed to fix. Based on the monitoring gathered in [6], these appear to be parameters comparable to real-life user analysis (but more aggressive than the average analysis).

For each configuration, the following tests were done:

- *Local*: Reading data from a server in Nebraska to a client in Nebraska; a simple copy ran at 25.2MB/s and network round-trip-time is 0.3ms.
- *Remote*: Reading data from a server in Nebraska to a client in Geneva, Switzerland; a simple copy ran at 18.7MB/s and network round-trip-time is 137ms.

To reduce the number of independent variables, each test was run multiple times to make sure the requested data was resident in the server's page cache.

Results are normalized by the time it takes to read 1,000 events in the local case with all optimizations. So, a slowdown of 2 indicates that the test took twice as long as the local, optimized case. Table 2 shows the slowdown caused by removing certain optimizations compared to the fully optimized case.

Figure 3 illustrates the operations of the multi-source client. Note that servers 1 and 2 see relatively contiguous reads and overlapping I/O. For the remote case there was a 1.13x speedup by using the multisource client when reading from two servers in Nebraska. In the case where the client is in Geneva, but one source is in Nebraska and the other in Germany, the speedup was 1.51.

Table 2. Slowdowns caused by the removal of various CMS optimizations; lower values are better. Values are normalized by the time taken for the fully-optimized local read case. Note that read coalescing can harm the local read case; this optimization has cost in terms of CPU time and, for this

test case, was not able to coalesce many reads.

	Defaults	No Startup Optimization	No Trigger Optimization	No Read Coalescing
Remote	2.38	23.16	13.89	2.41
Local	1.0	1.20	1.03	0.91

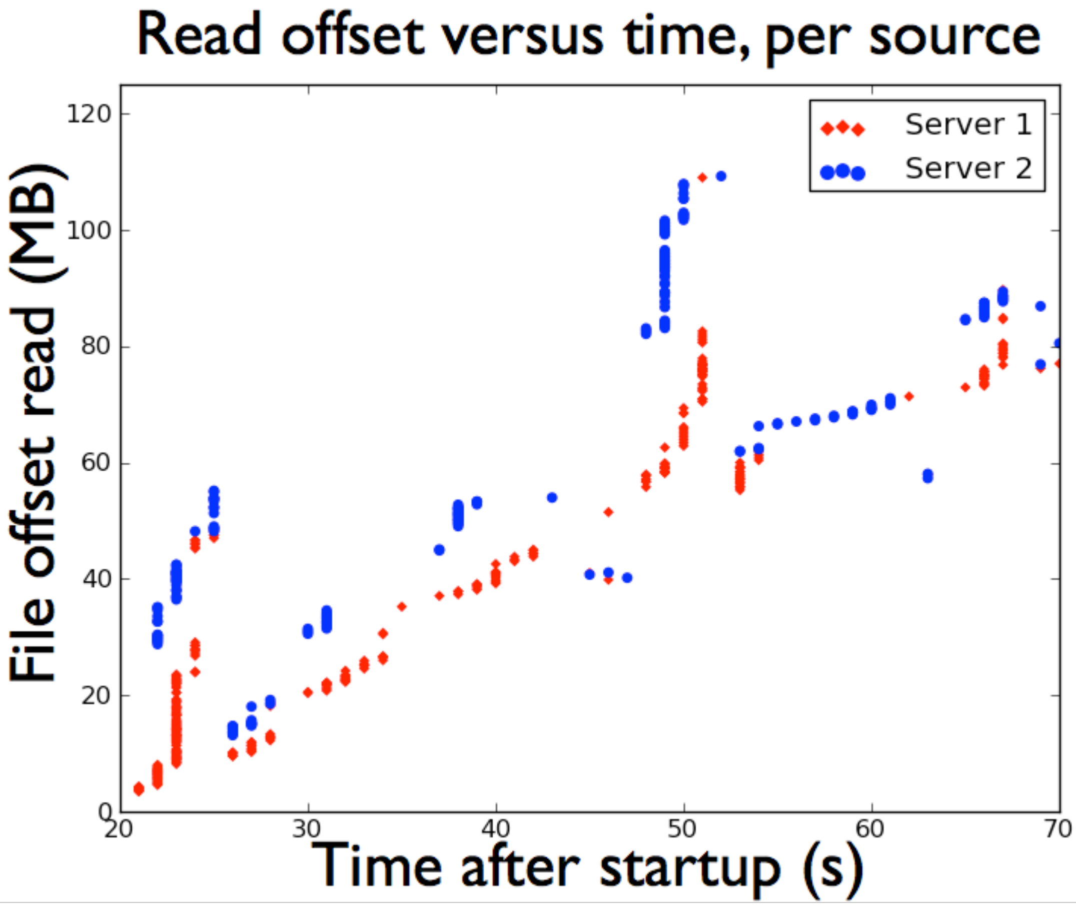

Figure 3. An illustration of the multi-source client functionality. For each read request performed by ROOT, one dot is plotted (including each individual request in a "vectored" read) at the file offset as a function of time.

6. Conclusions and Future Work

The TTreeCache startup and trigger pattern optimizations provide significant runtime improvements on high-latency links; there are even small improvements present for the low-latency case. The read coalescing and multisource optimizations do not provide significant runtime improvements in the average case. Read coalescing tends to reduce the number of read() system calls on the local system; this positive effect is not illustrated by the tests performed.

The multisource optimization provides significant improvements for the worst-case where the Xrootd redirector selects a poor initial source or the initial source quality degrades while the file is opened. We believe this "worst case" is a common-enough occurrence to use the multisource client when running jobs over a high-latency network.

Note that, in some cases for our stress test, there is less network overhead to download the file locally than have ROOT read out a subset – even with CMS's optimizations. The downsides of downloading the entire file include increasing total bandwidth needed per stream and having to buffer files on local disk; this approach has been tried in the past by CMS and subsequently mostly abandoned. In the LAN case, the factor of 2-10x increase caused by downloading the complete file is acceptable; unfortunately, there is not as much excess WAN bandwidth available. Further, CMS will occasionally produce individual files that are larger than the available disk space on the worker node, meaning the technique was impractical at several sites.

It is obvious that successful I/O experiments performed by CMS should be reviewed and submitted to ROOT so all HEP experiments may benefit. After the startup optimization was put into production for CMS for about a year, a similar implementation was included into a ROOT 5.34 patch release. We believe the trigger pattern optimization could similarly be included. The multisource client, while quite promising, will likely need more tuning in production.

7. Acknowledgements
This work was funded by the National Science Foundation grants PHY-1104664, PHY-1104549 and PHY-1104664.

References
[1] Brun R and Rademakers F 1996 ROOT - An Object Oriented Data Analysis Framework, *Proceedings AIHENP'96 Workshop (Lausanne, Sep. 1996) (Nucl. Inst. & Meth. in Phys. Res. A vol 389)* pp81-86.
See also http://root.cern.ch/.
[2] Canal C et al 2011 ROOT I/O: The Fast and Furious. *J. Phys.: Conf. Ser. 331 042005*
[3] Bauerdick L et al 2012 Using Xrootd to Federate Regional Storage *J. Phys.: Conf. Ser. 396 042009*
[4] Allcock W, Bresnahan J, Kettimuthu R, Link M, Dumitrescu C, Raicu I and Foster I 2005 The Globus Striped GridFTP Framework and Server *Proceedings of Super Computing 2005 (SC05)*
[5] Dorigo A, Elmer P, Furano F and Hanushevsky A 2005 Xrootd - A highly scalable architecture for data access WSEAS Transactions on Computers (2005)
[6] Bauerdick L et al 2012 Xrootd Monitoring for the CMS Experiment *J. Phys.: Conf. Ser. 396 042058*