Journal of Physics: Conference Series **331** (2011) 072010 doi:10.1088/1742-6596/331/7/072010

# The Use of Proxy Caches for File Access in a Multi-Tier Grid Environment

**R. Brun[1], D. Duellmann[1], G. Ganis[1], A. Hanushevsky[2], L. Janyst[1], A.J. Peters[1], F. Rademakers[1], E. Sindrilaru[1]**
[1]CERN, Geneva, Switzerland,
[2]Stanford National Accelerator Laboratory, Menlo Park, USA

**Abstract.** The use of proxy caches has been extensively studied in the HEP environment for efficient access of database data and showed significant performance with only very moderate operational effort at higher grid tiers (T2, T3). In this contribution we propose to apply the same concept to the area of file access and analyse the possible performance gains, operational impact on site services and applicability to different HEP use cases. Base on a proof-of-concept studies with a modified XROOT proxy server we review the cache efficiency and overheads for access patterns of typical ROOT based analysis programs. We conclude with a discussion of the potential role of this new component at the different tiers of a distributed computing grid.

## 1. Introduction

Since the start of the LHC collider operation the LHC experiments have collected and analysed an impressive amount of data from their complex detector systems. The success of physics program with first publications emerging already a few months after the machine start has largely profited from the distributed computing infrastructure used by all experiments – the World-wide LHC Computing Grid (WLCG). Within WLCG multi-tiered computing models are implemented taking advantage of the larger computational resources than at the CERN Tier-0 alone. WLCG therefore defines also the service roles and utilises available resources from existing computing centres around the globe.

The distribution of data within the LHC grid has during the first period been dominated by scheduled replication from Tier-0 to Tier-1 sites, which take over an important role in providing custodial storage on Mass Storage Systems (MSS) deploying tapes. As data gets further replicated to Tier-2 and Tier-3 sites, additional services for workflows such as user analysis and simulation are provided to complete the High Energy Physics (HEP) data handling chain.

With the first operational experience of the data taking in 2010 the WLCG collaboration has recently held a Data Management Jamboree [1], where further optimisations to the existing data strategy have been discussed and several demonstration activities have been started, to prove the feasibility of suggested enhancements. In this paper we describe work towards a more dynamic data distribution chain exploiting the concept of transparent data caches to increase the access performance and resource efficiency of the current system. After a description of the main concepts, we compare several approaches to implement data caches at Tier-3 and Tier-2 sites and analyse the results from first performance tests. Closely related to this concept, also the general performance of wide-area user access is being studied to understand under which conditions efficient wide-area access is feasible and when client side caching would offer significant advantages.

## 2. Wide Area Access and Caching for Data File Access

HEP data analysis exposes typically less sequential file access patterns than data movements or reconstruction workflows. This is a direct result of the user application either skipping a significant amount of input events and/or skipping a significant amount of the data products available within an input event. The typical pattern exposed is therefore a moving band of I/O positions with small amounts of read data and non-monotonically rising read offsets within the input file. This pattern affects in particular (but not only) efficient data access in the wide area with the HEP data access software (e.g. ROOT [2]) for a variety of reasons:

Published under licence by IOP Publishing Ltd

- Inadequate read-ahead size used by I/O protocols
- Large accumulated latency due to many network round-trips
- Limited network bandwidth and slow increase of usable bandwidth within a TCP/IP session

The large read-ahead size (optimised for sequential access) and the simplistic read-ahead implementation led during the first deployment phase to rather low delivered bandwidth to the user application and to a large network overhead due to repeat transfers of the same data. In several cases the processing of a file required 5-10 times the total file size to be transferred.
While this situation has improved dramatically with the introduction of larger (e.g. 30 MB) vector-reads and an in-process data cache on the ROOT side (TTreeCache [3]), the cache can not be preserved across process boundaries: neither consecutive user jobs on the same node nor jobs running on other nodes can share TTreeCache information directly. We therefore propose to implement a persistent data cache, which maintains low latency access to popular data for multiple processes, e.g. consecutive execution with the similar input data on a user laptop or a larger number of concurrent processes running on a larger computing farm.

To share the cached information we have investigated two different implementation scenarios:

- A server on network protocol level
- A transfer fragment cache maintained by ROOT in a shared file area

### *2.1. Caching Proxy-Servers*

In this approach caching is performed on the network protocol level (e.g. XROOT [4]), which is used to obtain the next vector-read block from a file server. Instead of connecting the client application directly to the origin storage server, we introduce a proxy-server, which implements the protocol towards the client to serve its requests and forwards (using the same protocol) client requests to the origin server. The added value of this intermediate component results from the fact that all delivered vector-read blocks are being kept also on a local disk area, which is maintained by the proxy-server. On repeated requests, this proxy-server can therefore directly serve a second request for the same data from its local storage, with less latency than a connection to the original server would imply. In case of a single proxy-cache server, this would be located network-wise close to its clients. As the incoming and outgoing protocols are identical the cache is functionally transparent also hierarchical setups could be constructed to further increase cache efficiency.
This approach is directly following the approach for database data distribution using the Frontier/SQUID [5] packages, which is deployed successfully by the CMS experiment and now being evaluated also by ATLAS. On the operational side it has been shown that read-only caches are easy to setup and maintain even for sites with very limited hardware and personnel resources.

#### *2.1.1. XROOT Proxy-Cache*

As a concrete implementation of this approach we used the XROOT proxy-cache server, which maintains the data cache using one sparse file for all fragments of the original file. To allow a fast test if a particular data block is already cached, a parallel bitmap file is maintained which is mapped into the server memory and updated as additional data gets cached. This technique enables the use of cached data also in the case of only partially overlapping vector-reads. As this implementation works on the page level rather than the complete file, the client will during un-cached reads only be exposed to the latency of transmitting one request – not the latency of transmitting the full file. As the server component can easily maintain a global map of the popularity of each particular page over a larger number of user requests, it would be possible to implement caching policies, selecting only fragments for local caching that exceeding a certain minimal hit rate. Given the centrally maintained state in the proxy-server component, it should also be relatively easy to implement policies for the maximal cache size and age of cached data.

*2.1.2. HTTP and SQUID Proxy-Cache*

As second implementation of the proxy-cache approach would be even more aligned with the Frontier/SQUID setup in use for database access. In this case transfers would be based on the http protocol using the SQUID proxy-server to maintain the cache of popular data files. This would have the advantage of re-using SQUID as an established and already deployed software component. A major disadvantage though might be the resulting latency for un-cached requests. Event though sparse vector-reads can be formulated as http requests; the SQUID proxy-cache currently serves those requests by downloading the full file before serving the client with the requested data blocks. This implication of using SQUID may also have a significant adverse impact on the cache efficiency in case the majority of analysis programs only utilise a small fraction of their input data files.

*2.2. ROOT maintained Transfer Fragments*

The second approach we evaluated takes advantage of the I/O protocol abstraction inside the ROOT toolkit and implements a shared fragment cache without the need for external server components by using a shared file area. Any TTreeCache vector-read result is stored directly from the client application as an individual file in the shared area. Before issuing a new vector-read request this area is checked for the existence of a suitable fragment – therefore delivering similar latency gains as the server based setups above. This approach has the advantage of being agnostic of the particular transfer protocol and can be used without change to the I/O protocols already supported by ROOT. One disadvantage being that the caching is only provided for ROOT based programs, which is not seen as major constraint for HEP use cases. The management of the shared cache area will require though a shared file area, which can scale up to a larger number of individual small files (typical vector-read size is 30 MB – compared to input file sizes reaching several GB) and will need an additional cache eviction policy, for example based on the cache hits for a fragment by time unit. As there is no shared knowledge about the fragment popularity until a fragment is stored, it would be more difficult to implement caching policies that store cache fragments only after a certain minimal number of uses. Also the eviction of old or less popular cache fragments might need additional processing components to regularly scan the existing cache content and maintain a global cache size limit.

## 3. Performance Studies in Local and Wide Area

To evaluate and compare the above proposals we have setup a test infrastructure in the wide-area and local area network. The setup consisted of a source data server (XROOT and HTTP) at BNL connected via the optical-private-network (OPN) to CERN (round trip time of ~100ms). At CERN we used a second machine running an XROOT proxy-cache serving ROOT clients via a 10GB (RTT 0.5ms). The configuration with the cache at CERN and data source outside was picked for convenience, as the majority of the contributors were present at CERN. A realistic deployment scenario however would eg use Tier-0,1,2 as source server and provide the cache at higher tier sites (eg Tier-3) close to the majority of analysis client processes.

To obtain the baseline performance of our setup we used the Iperf [6] tool to measure the network performance of single and multiple stream transfer. We also used the xrdcp command to obtain similar measurements with the XROOT protocol. The results obtained are shown in table 1.

For comparison we note that typical analysis jobs at the current state of the experiment frameworks and ROOT are able to consume 1-4 MB/s. A rate of 10 MB/s can be achieved with a pure ROOT application reading from a local disk. The target value of 10 MB/s for the pure remote I/O chain (eg network, XROOT, ROOT) would therefore be desirable also for remote WAN access in order to insure that client jobs can fully utilise the available CPU resources.

*3.1. Results with the XROOT Proxy-Cache*

In table 2 we have collected the performance measurements obtained with native ROOT (TTreeCache enabled) - this time utilising a proxy-cache between client and data server. The values obtained for the cold cache (first read) shows that the cache implementation can almost reach the performance of a

remote read and exposes only minimal overhead. The measurement for the hot cache (second read) shows a significant performance increase of almost a factor three. In this case the performance reaches almost the full performance (97%) of a local read at the site of the client. The proxy-cache hence does not introduce a significant penalty but provides significant benefits in the WAN environment for repeat reads. The second part of table 2 details the possible benefit of the sparse cache implementation. If only two branches of the benchmark file are being used, then the cache can provide an additional performance gain. In this case only 3.9% of the total file volume is cached and the real time for accessing the cached data falls to less than a second (to be compared to the initial read duration of 104 seconds). This clearly shows the performance potential of sparse caching.

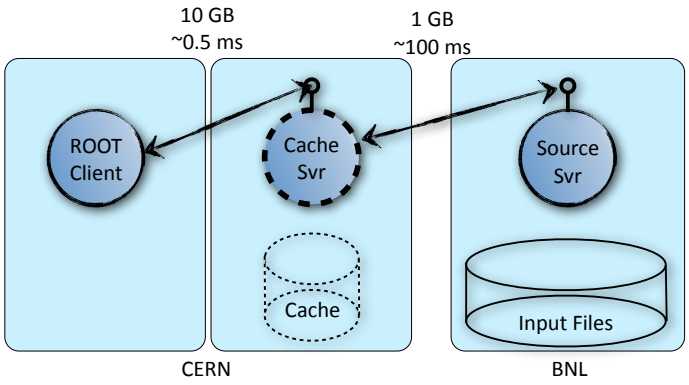


**Figure 1.** Network Setup between CERN and BNL and Data Flow.

**Table 1.** Network and protocol performance in the WAN setup.

| Protocol | Bandwidth measured | Settings / Comments |
|---|---|---|
| iperf | 5 MB/s (50 MB/s) | 1 Stream (10 Streams) |
| xrdcp | 3 MB/s (5MB/s) | before (after) optimisation of transfer granule size, 1 stream |
| ROOT client | 3-4 MB/s | atlasFlushed.root + TreeCache<br>no experiment framework code involved. |

**Table 2.** Proxy-cache efficiency and performance impact.

| | Realtime | Bandwidth | Cached Fraction | WAN / LAN Access |
|---|---|---|---|---|
| all branches | | | | |
| 1st read - cold cache | 288s | 3.7 MB/s | 0% | 28% |
| 2nd read - hot cache | 109s | 9.7 MB/s | 100% | 97% |
| two branches | | | | |
| cold cache | 104s | 0.32 MB/s | 0% | |
| hot cache | 0.9s | 35 MB/s | 3.9% | |

To measure the cache performance for real-life experiment use cases, we used in addition the following applications:

- a D3PD job using ATHENA 15.6.12
- a single collection extraction job using ATHENA 16.0.1

In the case of the D3PD job we observed that 99% of all 4kB data pages of the original file have been requested by the application. Sparse caching is hence not providing significant benefits compared to full file caching. The observed behaviour does depend on the detailed configuration by the physics working-group and we also note that the used ROOT version (5.22h) does not perform the automatic basket optimisation [3], which is expected to significantly increase the colocation of selected data.

For the use case of single collection extraction, which requests from a 4GB input file only 70MB of data in order to store an output collection of 17MB, we have measured with 4kB pages a cache volume of 270MB. Also here we obtain a significant gain in storage efficiency. Only 6% of the input file storage volume is required in the sparse page cache.

The two examples above confirm the expectation that the variation in selectiveness between different types of user jobs can be quite significant. Only a statistical analysis of the combined access patterns in a larger number of real user jobs would allow taking quantitative conclusions on the effectiveness of sparse caching with respect to full file caching.

As the experiment frameworks are now providing the option of remote LAN access (rather than full file copy to the worker node) we expect that this analysis can be performed in the next months, as soon as larger samples of user jobs with remote access become available.

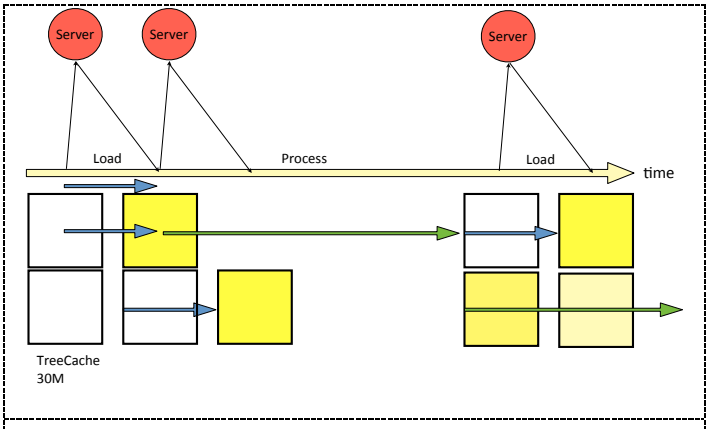


**Figure 2a.** Asynchronous pre-fetching of buffers.

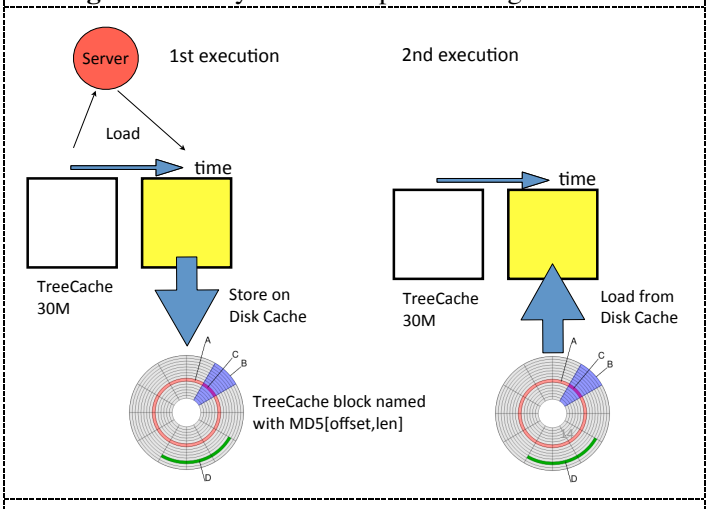


**Figure 2b.** Caching of TTreeCache buffers

*3.2. Results with a ROOT managed cache*

A second set of tests has been performed using the approach of caching vector-read fragments directly from the ROOT client process. For this we implemented in ROOT an additional processing thread, which issues asynchronously multiple vector-read requests to the storage server. This functionality allows overlapping the application processing time for a given fragment with the I/O latency in obtaining the

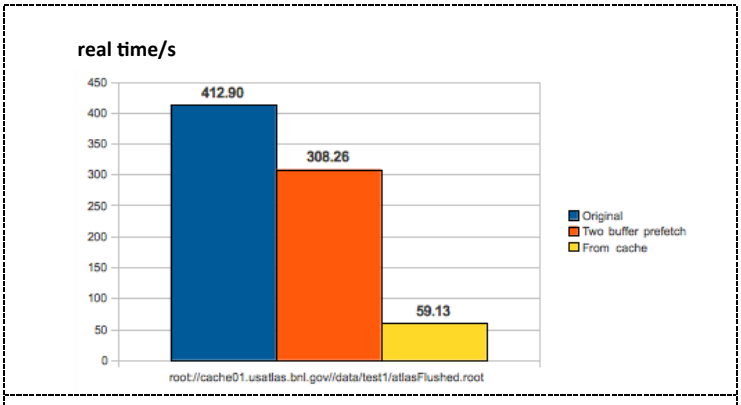


**Figure 3.** Gain due to pre-fetching and caching.

next fragment (see Figure 2a). This basic setup has been further extended to maintain already obtained fragments in a disk cache that is accessible to one or more client applications (see Figure 2b).

With this prototype we measured the impact of caching performance summarised in Figure 3. As show here already enabling asynchronous access provides in this case a real time saving of some 25%. This value depends on the ratio between CPU and I/O latency to process a given amount of data and could grow up to 50% in case both component are of equal size.

## 4. Conclusion

Both caching approaches tested have shown significant performance advantages in comparison to remote access over the WAN. Prototype implementations for both approaches exist but additional work on the operational aspects (automatic cache eviction, maximum cache size) are still required before an effortless deployment could take place at eg tier 3 sites.
In addition we have shown that asynchronous pre-fetching on the ROOT TTreeCache level can provide additional performance benefits for applications with are not fully I/O bound and that this mechanism can be implemented in a generic way independent of the remote access protocol used (eg xroot, http). The concept of sparse caching (rather than full file caching) has shown significant advantages for certain use cases but for a more quantitative evaluation the selectiveness of different user jobs needs to be folded into the analysis. We expect to obtain this information during the next months of running with remote access clients.

## Acknowledgments

The authors would like to thank Daniel van der Ster (CERN, IT-ES group) for providing us with the ATLAS analysis jobs used in our tests. We would also like to thank Michael Ernst and John Hover for providing over several months a remote test infrastructure at Brookhaven National Laboratory.

[1] Jamboree on Evolution of WLCG Data & Storage Management, 16-18 June 2010, *Amsterdam, The Netherlands, http://indico.cern.ch/conferenceDisplay.py?confId=92416*
[2] Brun R et 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.*
[3] Canal Ph, Bockelman B and Brun R, ROOT I/O: The Fast and Furious, CHEP 2010, Taipei, Taiwan, In these proceedings
[4] The Scalla/xrootd Team, The Scalla Software Suite: xrootd/cmsd, http://xrootd.slac.stanford.edu/
[5] Dykestra D, Scaling HEP to Web Size with RESTful Protocols: The Frontier Example, CHEP 2010, Taipei, Taiwan, In these proceedings
[6] Tirumala A, Qin F, Dugan J Ferguson J and Gibbs K, Iperf, http://dast.nlanr.net/Projects/Iperf/