ROOT I/O Benchmarking and Client Side Efficiency Metrics

Fons Rademakers
CERN

pre-GDB, 9-Oct, 2012, LAPP, Annecy.

ROOT and Performance Measurements

- Performance benchmarking has been of primary importance since the beginning of ROOT
  - Introduction of the ROOTMARKS
- The $ROOTSYS/test directory contains a large number of benchmark programs:

```
stress.cxx                   stressMathCore.cxx
stressEntryList.cxx          stressMathMore.cxx
stressFit.cxx                stressProof.cxx
stressGUI.cxx                stressRooFit.cxx
stressGeometry.cxx           stressRooFit_tests.cxx
stressGraphics.cxx           stressRooStats.cxx
stressGraphics.ref           stressRooStats_models.cxx
stressHepix.cxx              stressRooStats_ref.root
stressHistoFit.cxx           stressRooStats_tests.cxx
stressHistogram.cxx          stressShapes.cxx
stressInterpreter.cxx        stressSpectrum.cxx
stressIterators.cxx          stressTMVA.cxx
stressIterators.h            stressVector.cxx
stressLinear.cxx
```

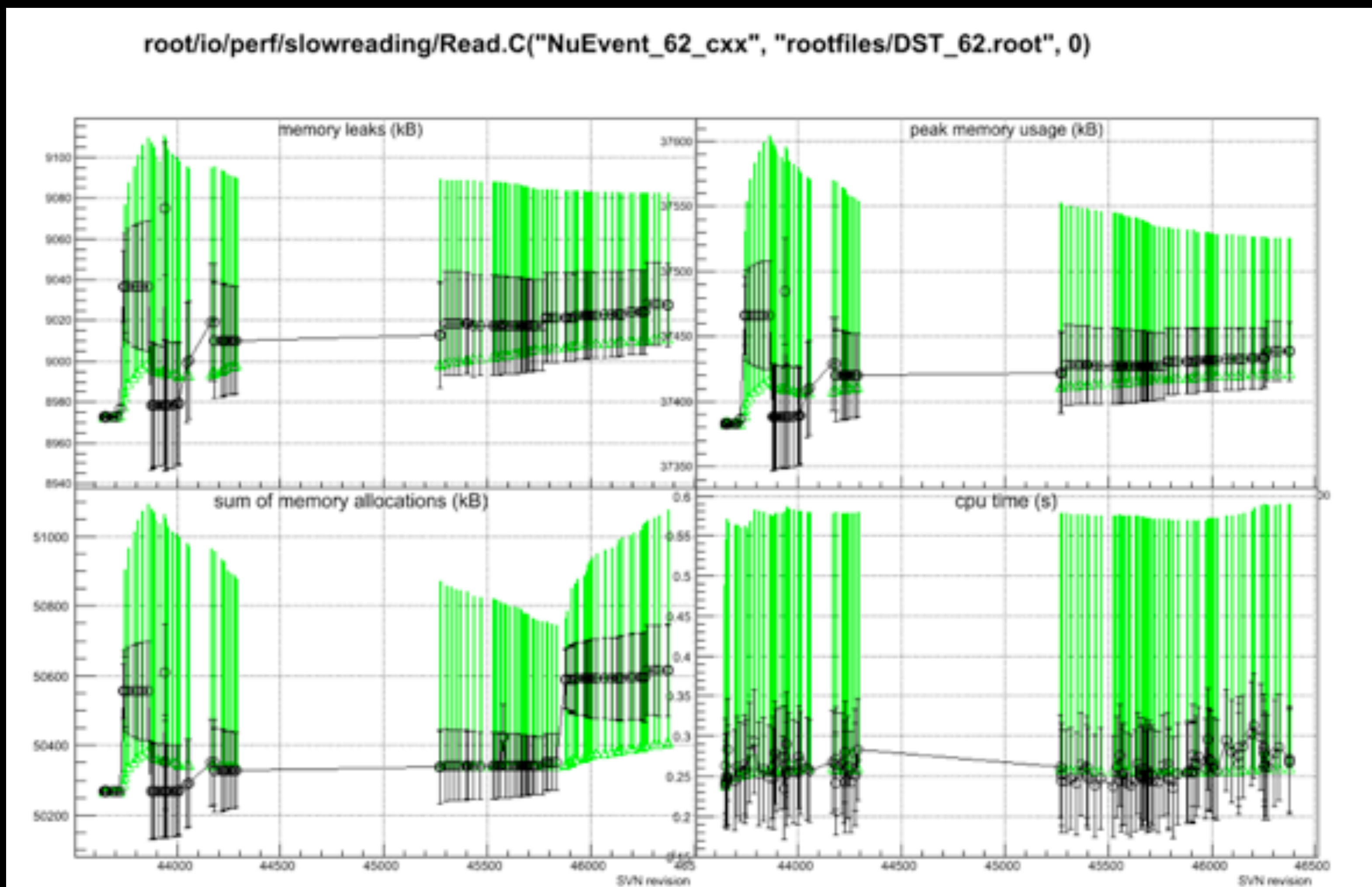
ROOTMARKS

- Output of the stress programs are a ROOTMARK number:

```
******************************************************************
*                                                                *
*          S T R E S S   H E P I X  S U M M A R Y                *
*                                                                *
*     ROOTMARKS =1339.8   *  Root5.99/01   20120509/1205         *
*                                                                *
*  Real Time =  284.1 seconds, CpuTime =  208.0 seconds          *
*  SYS: Darwin macrdm.rademakers.org 12.2.0 Darwin Kernel Version 12  *
*  SYS: 10.8.2 Mac OS X                                          *
******************************************************************
```

roottest Regression Test Suite

- In roottest suite, measure memory and CPU time
- Any few percent deviation triggers an regression test error



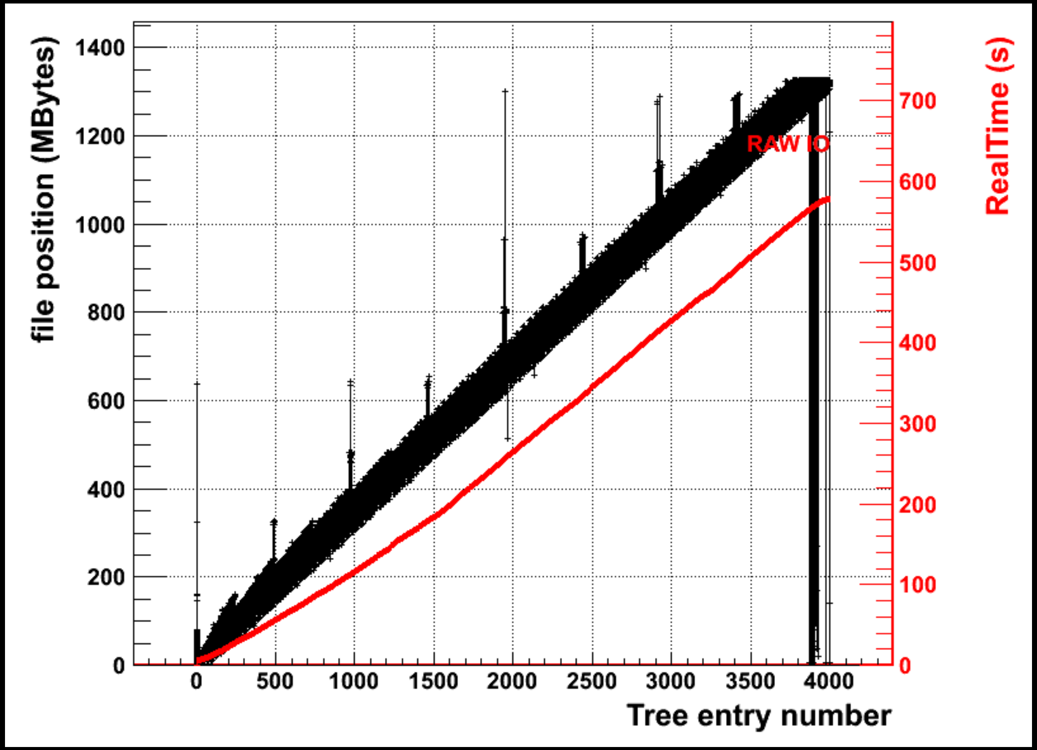
Client Side I/O Performance Analysis

- Monitor TTree reads with TTreePerfStats

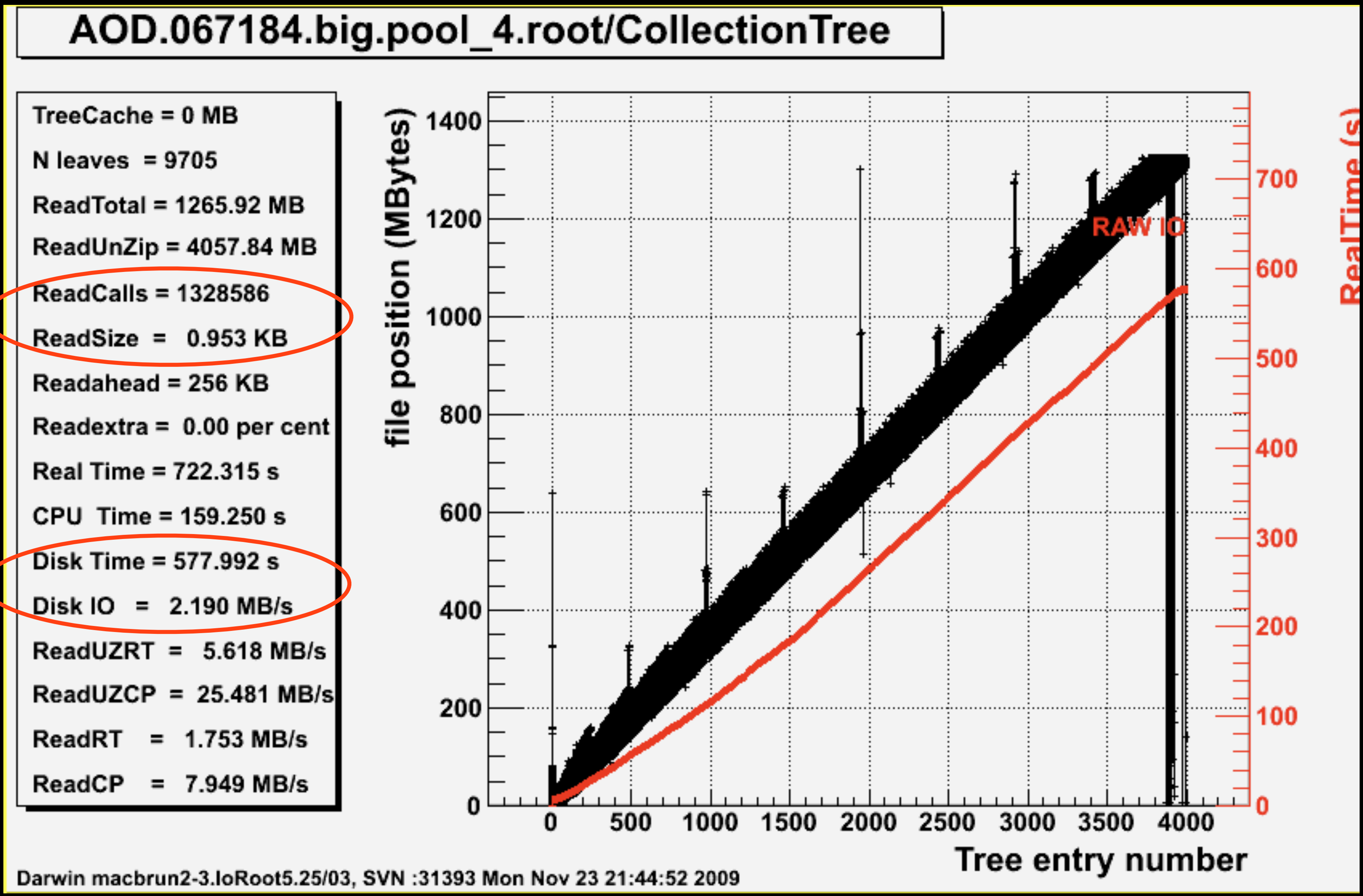
```
TFile *f = TFile::Open("xyz.root");
T = (TTree*)f->Get("MyTree");

TTreePerfStats ps("ioperf",T);

Long64_t n = T->GetEntries();
for (Long64_t i = 0;i < n; ++i) {
   GetEntry(i);
   DoSomething();
}
ps.SaveAs("perfstat.root");
```

TTreePerfStats

- Visualizes read-access
- x-axis: tree entry number
- y-axis: file offset
- y-axis: real time

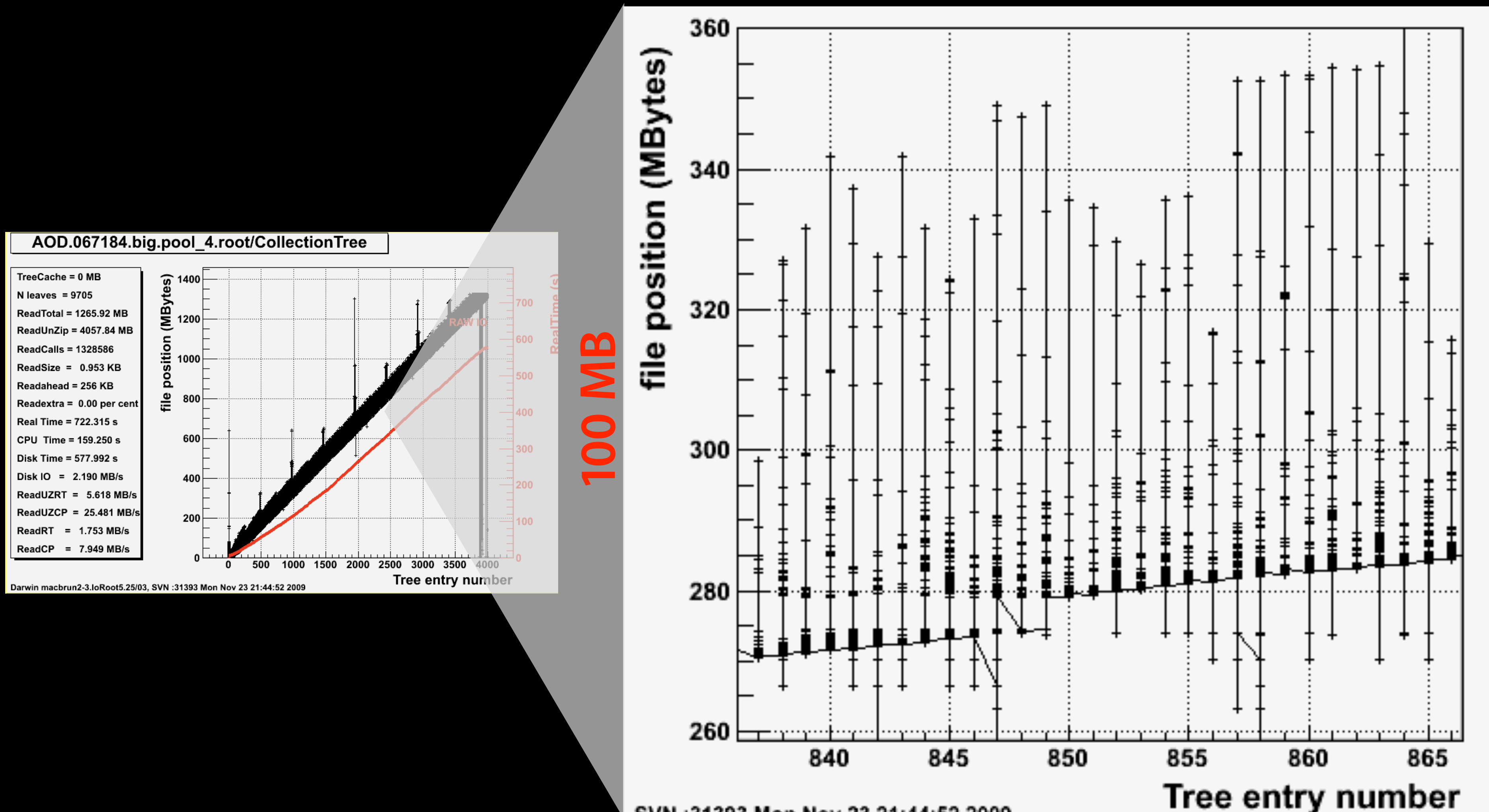


Non Optimal File Layout

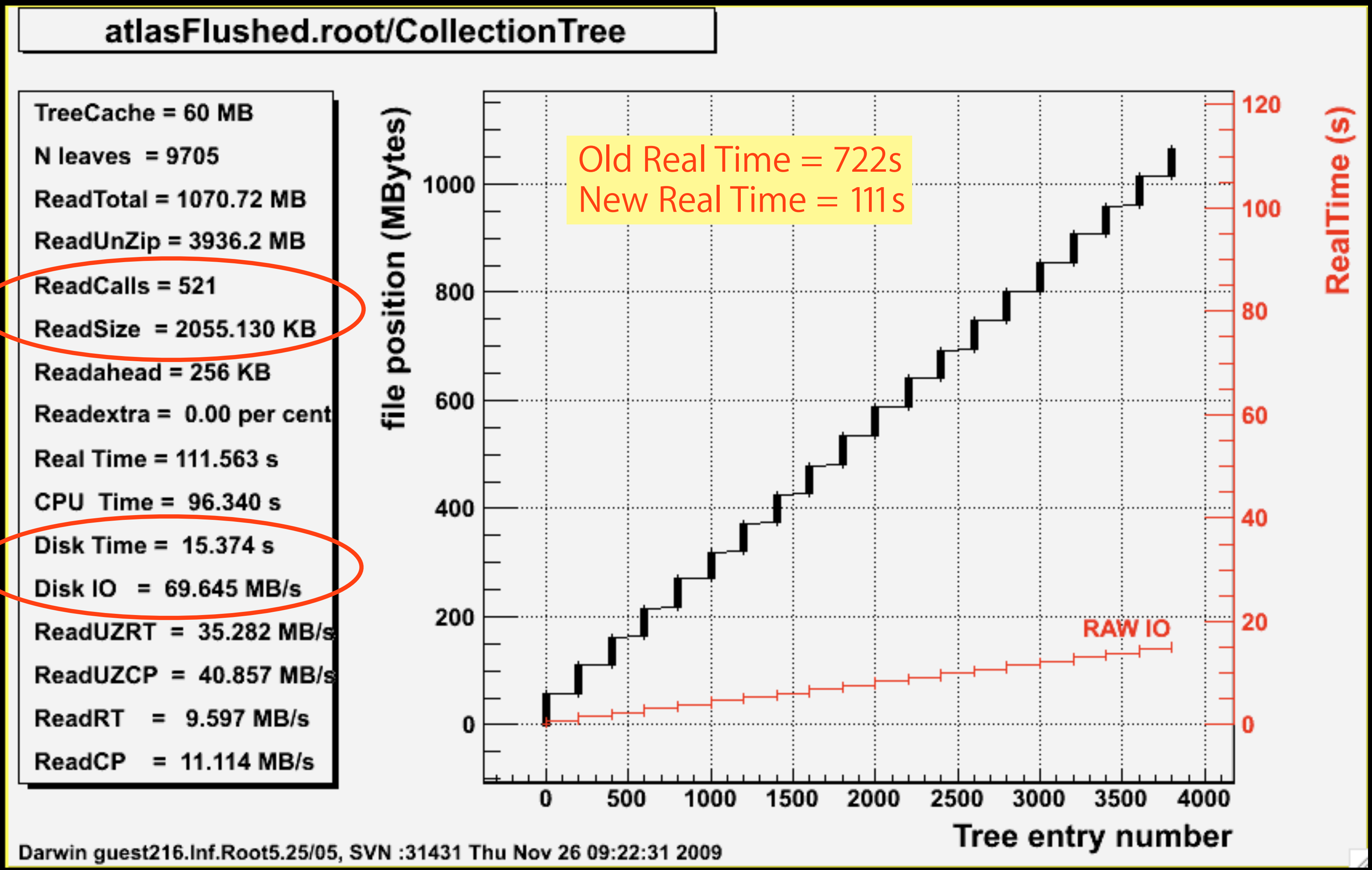


Performance measurements made using the TTreePerfStats class

Overlapping Reads



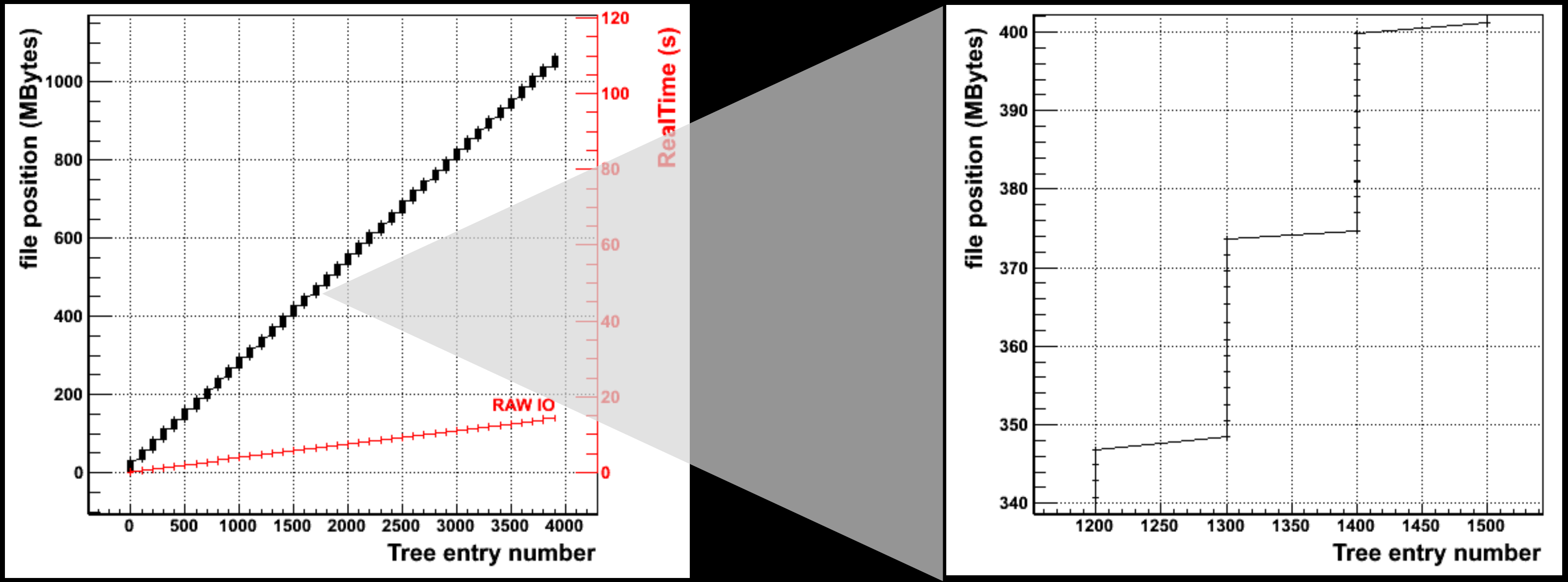
Optimized File Layout



OptimizeBaskets and FlushBaskets

- Solutions, enabled by default:
  - Tweak basket size
  - Flush baskets at regular intervals

Available in v5.26.00



OptimizeBaskets

- The TTree::OptimizeBaskets() method is a new function that will optimize the buffer sizes taking into account the population in each branch
- Tunes the branch buffer size
- Without this tuning branches containing the same event are scattered in the file
- You can call this method on an existing read-only Tree to see the diagnostics

FlushBaskets

- The TTree::FlushBaskets() method was introduced in 5.22 but called only once at the end of the filling process
- In version 5.26 this method is called automatically when a reasonable amount of data (default 30MB) has been written to the file
- The first time that TTree::FlushBaskets() is called, we also call TTree::OptimizeBaskets()
- The frequency to call TTree::FlushBaskets() can be changed by TTree::SetAutoFlush()
- Thanks to TTree::FlushBaskets() there are no backward seeks anymore (for files written with 5.26).

What is the TreeCache

- It groups into one buffer all blocks from the used branches
- The blocks are sorted in ascending order and consecutive blocks merged such that the file is read sequentially
- It reduces typically by a factor 10000 the number of transactions with the disk and in particular the network with servers like httpd, xrootd or dCache
- The typical size of the TreeCache is 30 Mbytes, but higher values will always give better results

ROOT Optimizations for WAN

- Load phase (where data is fetched from an SE into the TreeCache) is
  - Short for LAN transfers
  - Significant for WAN transfers (latency, bandwidth)
- Gain in WAN by asynchronous (double buffering) transfer technique
  - Independent of access protocol (xrootd, httpd, etc)
- In addition local file caching
- And site proxy server
- Implemented in v5.30 by Elvin Alin Sindrilaru (fellow IT-DSS)

Pre-fetching and Caching Summary

- Asynchronous pre-fetching has been demonstrated as an efficient way to improve the cpu/RT efficiency of analysis applications
  - Allows to use every synchronous protocol in asynchronous mode
  - Allows to proxy caching of TreeCache blocks on any ROOT supported file storage
    - TreeCache transforms sparse/random access into sequential local access
  - Integrated in ROOT v5.30, activated using rootrc flag:
    - TFile.AsyncPrefetching: yes

Benchmarking Programs

- roottest comes with several scripts that can be used for benchmarking
  - io/perf/userdatasets/readfile.C

for example the following session:

```
root > .x readfile.C("atlasFlushed.root","",60000000,-1,1,0.33)
```

will use a 60 MBytes cache, reading all branches and only 1/3 of entries

Conclusions

- ROOT 5.26 came with a drastically optimized Tree buffer sizing and writing algorithm
- ROOT 5.30 comes with optimized WAN access using double buffered file transfer and local file caching
- After 17 years of developments, we are still making substantial improvements in the I/O system thanks to the many use cases and a better understanding of the chaotic user analysis
- Performance monitoring and benchmarking are essential to understanding how to achieve more performance improvements