

I/O Efficiency in HEP Data Analysis

Gerhard Brandt
DESY



Second I.N.F.N. International School on
"Architectures, tools and methodologies for developing
efficient large scale scientific computing applications"

Centro Universitario Residenziale Bertinoro
Bertinoro (FC)
22 - 27 November 2010



- Lecture
- Individual Exercises: Local Single File I/O on nodes
 - Preparation + Basic Event Loop
 - File Caching
 - File Optimisation

advanced

- *Influence of File Systems*
- *I/O Profiling with valgrind*
- *Persistent Model Complexity and Dictionaries*

- Break
- Common Exercise:

I/O on the school cluster GPFS and NFS using Torque

Look at
results
together



Feed
back



- Most applications in HEP are **write once, read often**
Therefore we focus in the Input part here

I/O Efficiency

- But we *will* also rewrite a few files and look at that

- When reading there are **bottlenecks** that limit the speed
- We'll look at some of them and remove/improve them or at least try to understand/measure them
 - Disk Mechanics
 - Network Latency
 - File Structure
 - Zipping
 - Data Interpretation / Object Building Algorithms
→ Complexity of Persistent Data Model

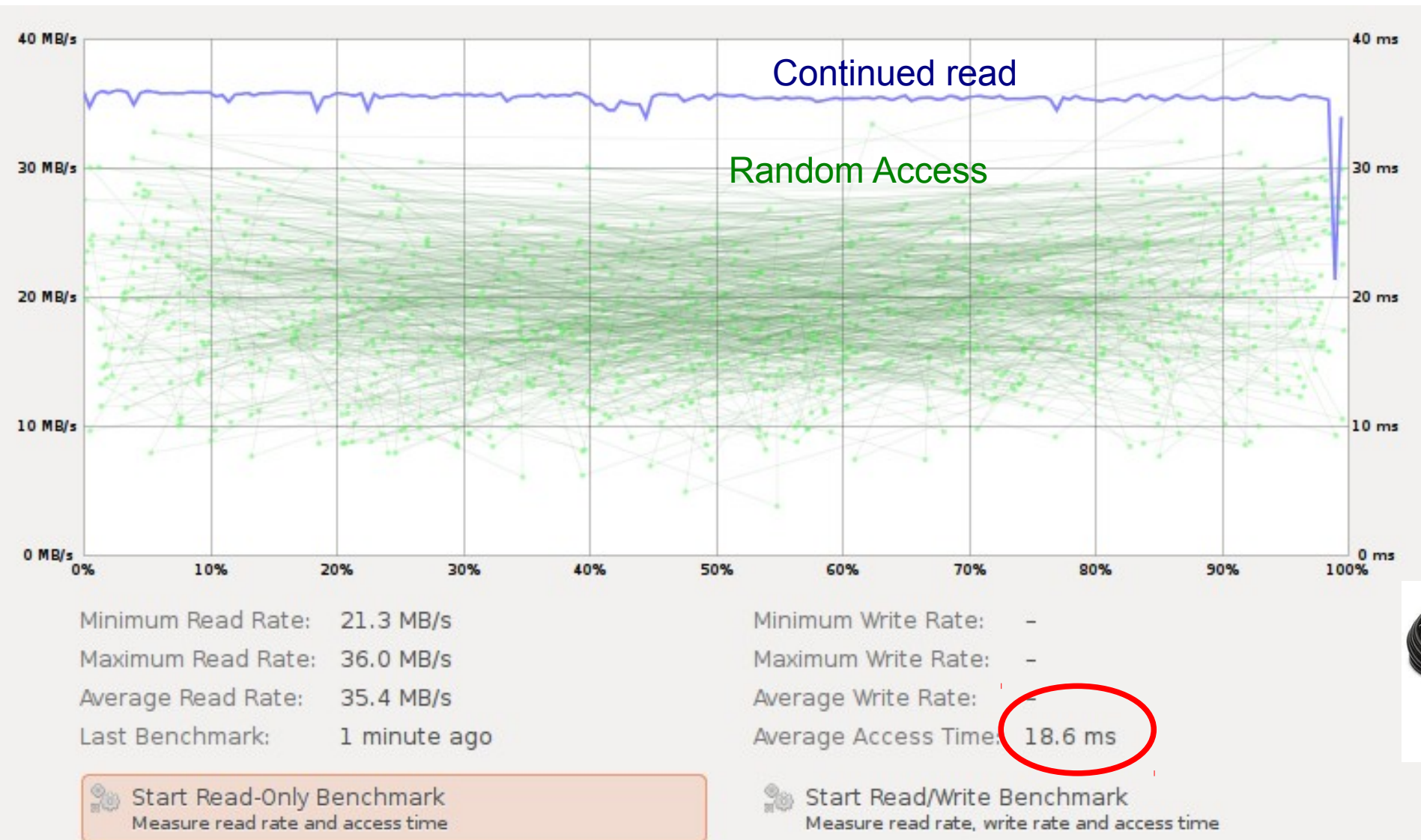
Quick Look at Some Bottlenecks ...



- Some storage benchmarks on my notebook using the Red Hat Disk Utility

External HDD connected via USB2.0

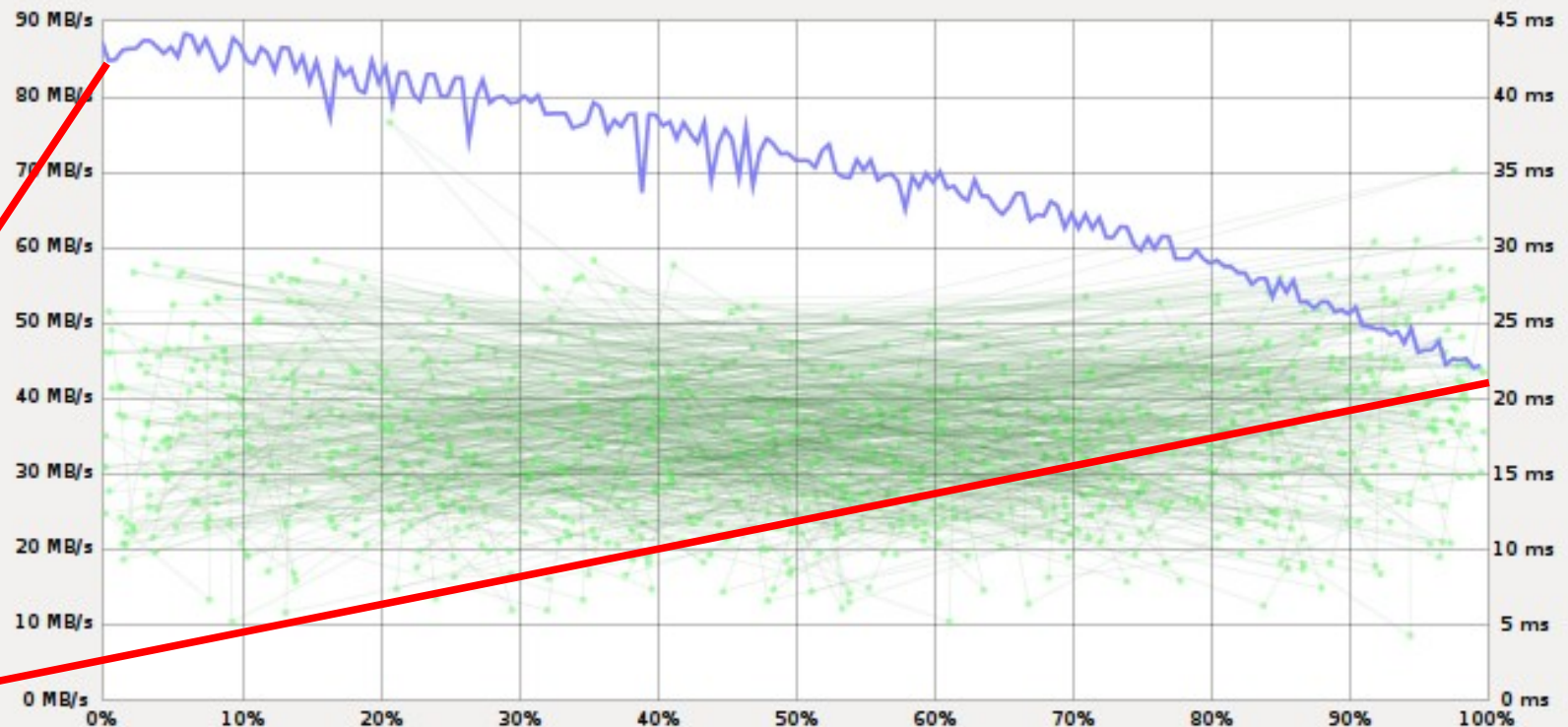
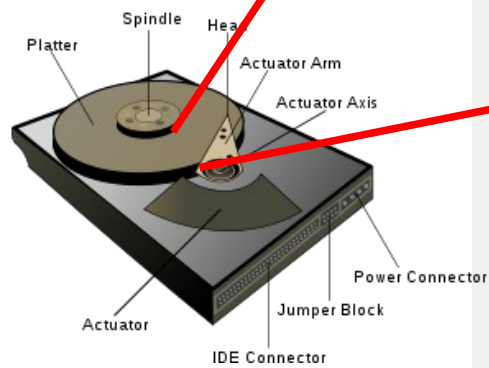
- Bottleneck: USB2.0 Speed – 35 Mb/s (*Network Bandwidth*)



External HDD via eSATA



- Bottleneck:
Disk Mechanics
40 – 90 Mb/s



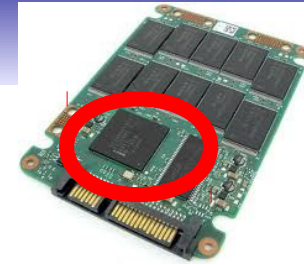
Minimum Read Rate: 43.8 MB/s
Maximum Read Rate: 88.0 MB/s
Average Read Rate: 69.9 MB/s
Last Benchmark: 57 minutes ago

Minimum Write Rate: -
Maximum Write Rate: -
Average Write Rate: -
Average Access Time: 17.1 ms

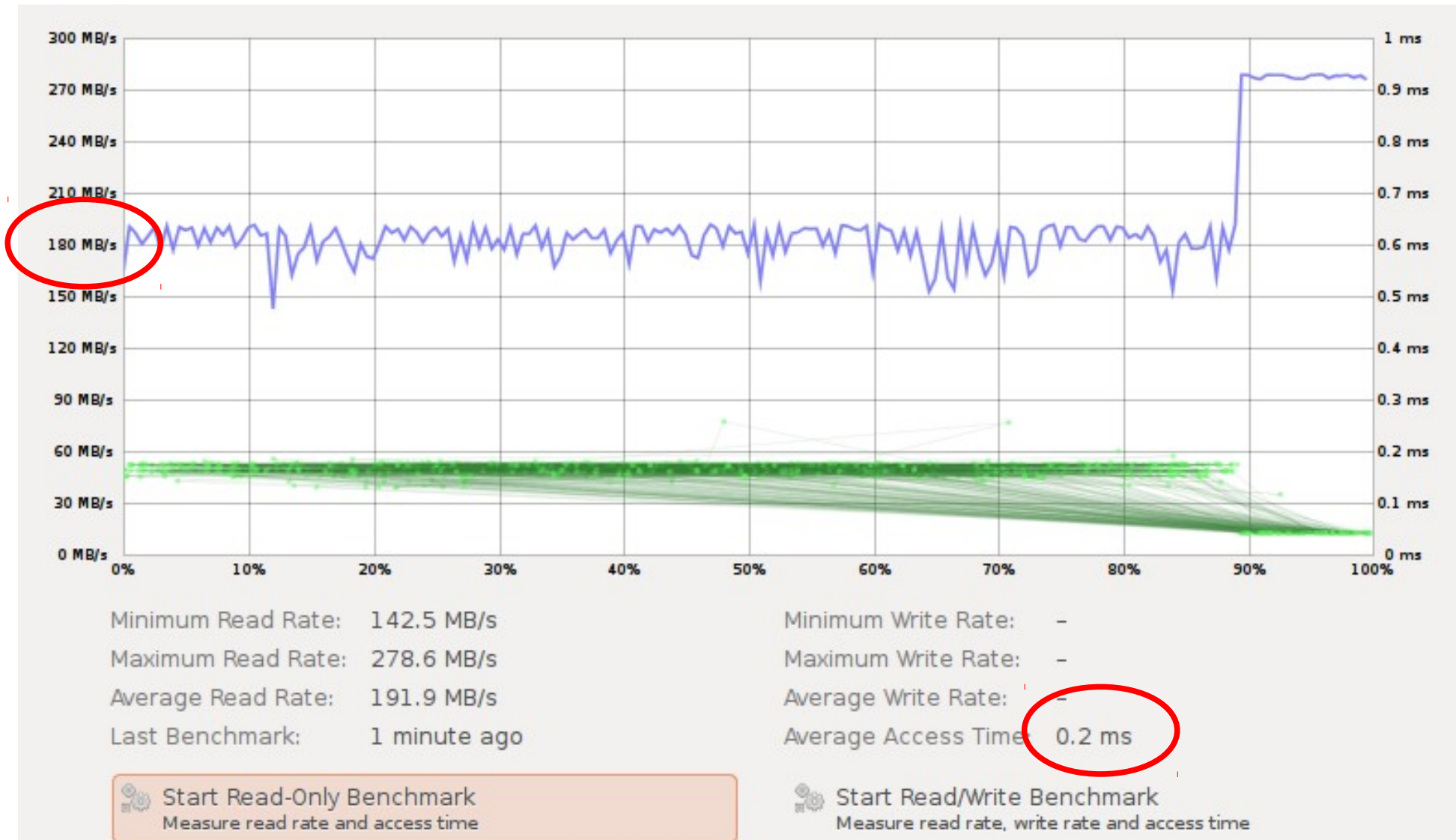
Start Read-Only Benchmark
Measure read rate and access time

Start Read/Write Benchmark
Measure read rate, write rate and access time

Internal SSD



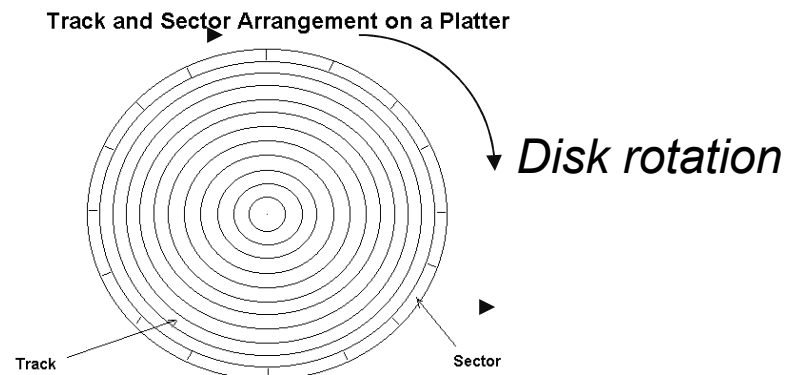
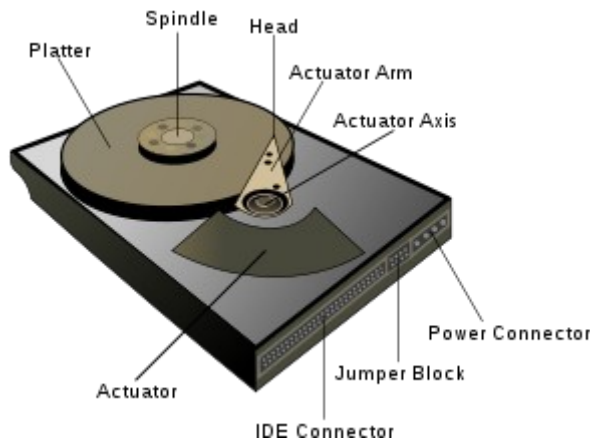
- Intel Postville X25-M on SATA II
- Bottleneck: Intel Controller – 180Mb/s / 290 Mb/s (empty part)



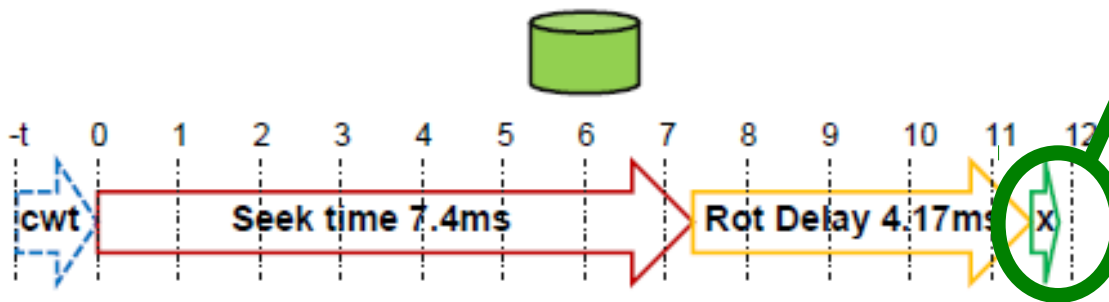
Disk Mechanics



- An I/O operation requires that the disk
 - Move the head to the right circular track (seek time)
 - Wait until the proper sector arrives (rotational delay)
 - Then transfer the data



Seagate Barracuda 180



- Only small amount of time spent on reading
- Should prevent seek time and rotational delay
- Blocks to read should be close-by on the disk
- Random access is a performance killer

Slide from ESC09 on how to fix disk latency:

- File system tries to hide disk slowness
 - Memory caching to avoid disk I/O
 - Also done in high-end disk controller caches
 - Pre-reading to keep channel utilization high
 - Done in the background to minimize impact
 - Also done in some high-end RAID disk controllers
- Offset ordering
 - Reduces seek time
 - Also done in high-end disk controllers

Requires no
change to written
file

Requires change
to written
file

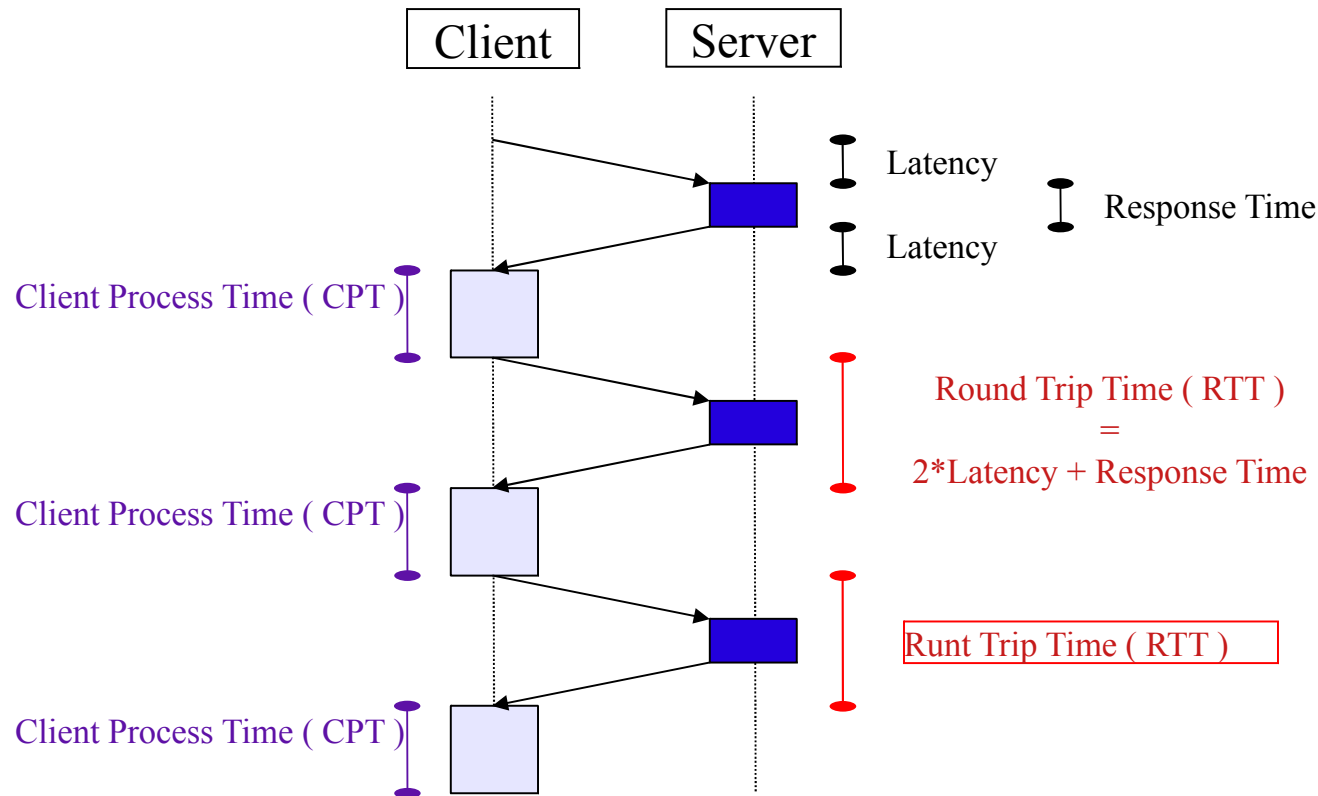
These are recurring features

- Does not only apply to disk mechanics
- We will demonstrate these features using ROOT

Remote Files and Latency



- Latency is not only an issue with disks
- The number of transactions matters in particular when reading remote files
- Every transaction adds an overhead to due network/software response latency



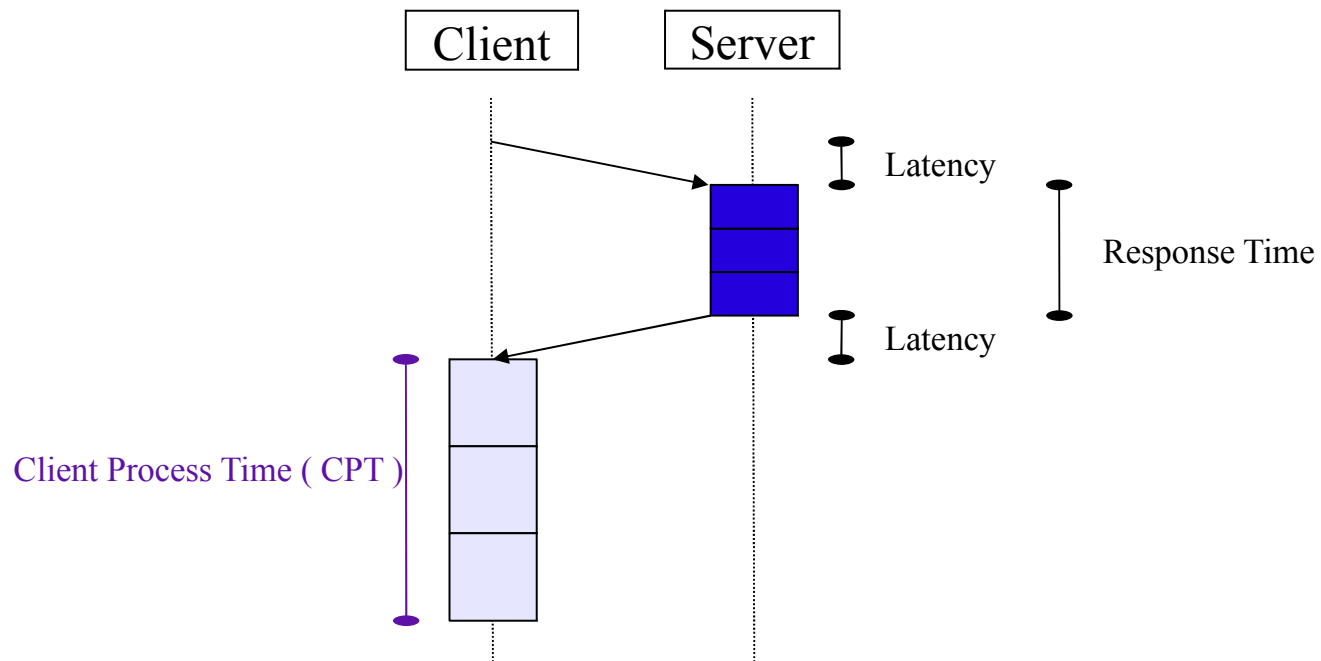
$$\text{Total Time} = 3 * [\text{Client Process Time (CPT)}] + 3 * [\text{Round Trip Time (RTT)}]$$

$$\text{Total Time} = 3 * (\text{CPT}) + 3 * (\text{Response time}) + 3 * (2 * \text{Latency})$$

How to Remove the Latency



- Solution:
 - Same principle as for disk
 - Perform only one big request instead of many small requests



$$\text{Total Time} = 3 * (\text{CPT}) + 3 * (\text{Response time}) + (2 * \text{Latency})$$

The Linux Boot Sequence



- How does it look in practice
- First I/O in the day: we boot up the computer ...
- Here's the evolution of my boot sequence 2008 – 2010
 - bootchartd collects statistics every time you boot Ubuntu Linux

2008
Hardy Heron 8.04

Jaunty Jackalope 9.04

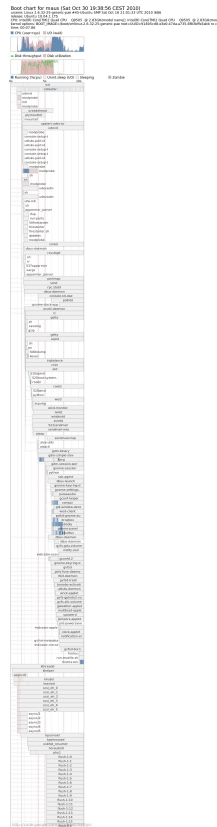
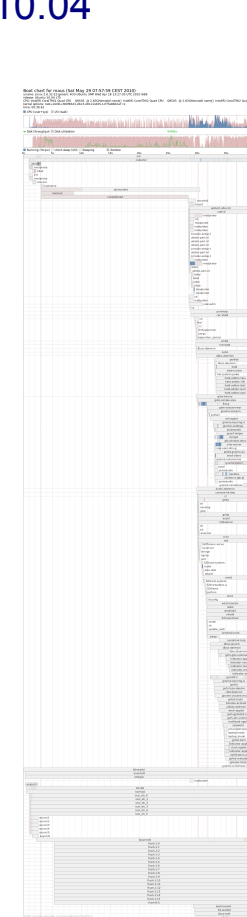
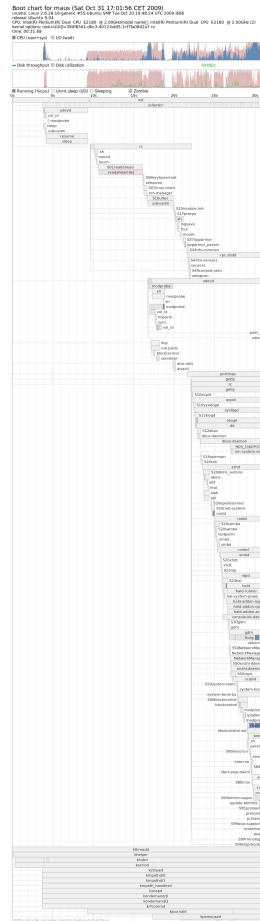
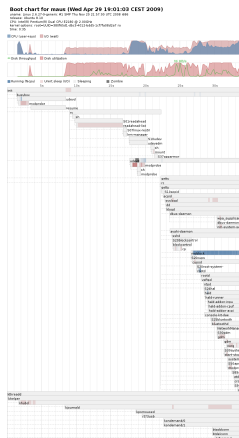
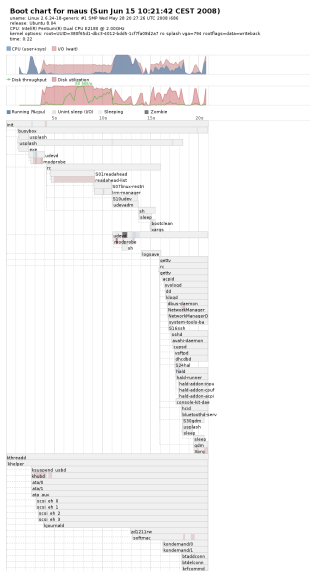
2009

Lucid Lynx
10.04

Maverick
Meerkat
10.10

Intrepid Ibex 8.10

Karmic Koala 9.10



Boot chart for maus (Sun Jun 15 10:21:42 CEST 2008)

uname: Linux 2.6.24-18-generic #1 SMP Wed May 28 20:27:26 UTC 2008 i686

release: Ubuntu 8.04

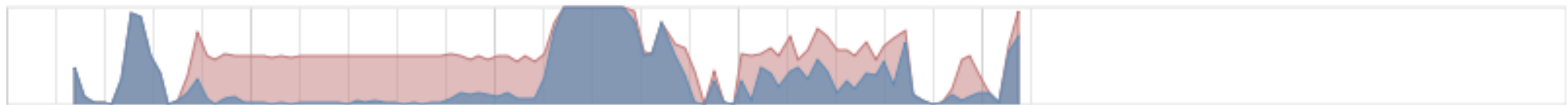
CPU: Intel(R) Pentium(R) Dual CPU E2180 @ 2.00GHz

kernel options: root=UUID=380f65d1-dbc3-4012-bdd5-1cf7fa08d2a7 ro splash vga=794 rootflags=data=writeback

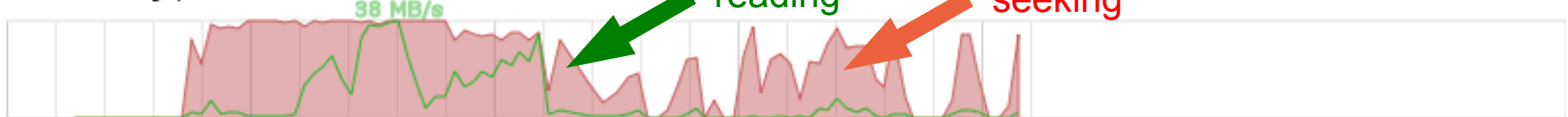
time: 0:22

time

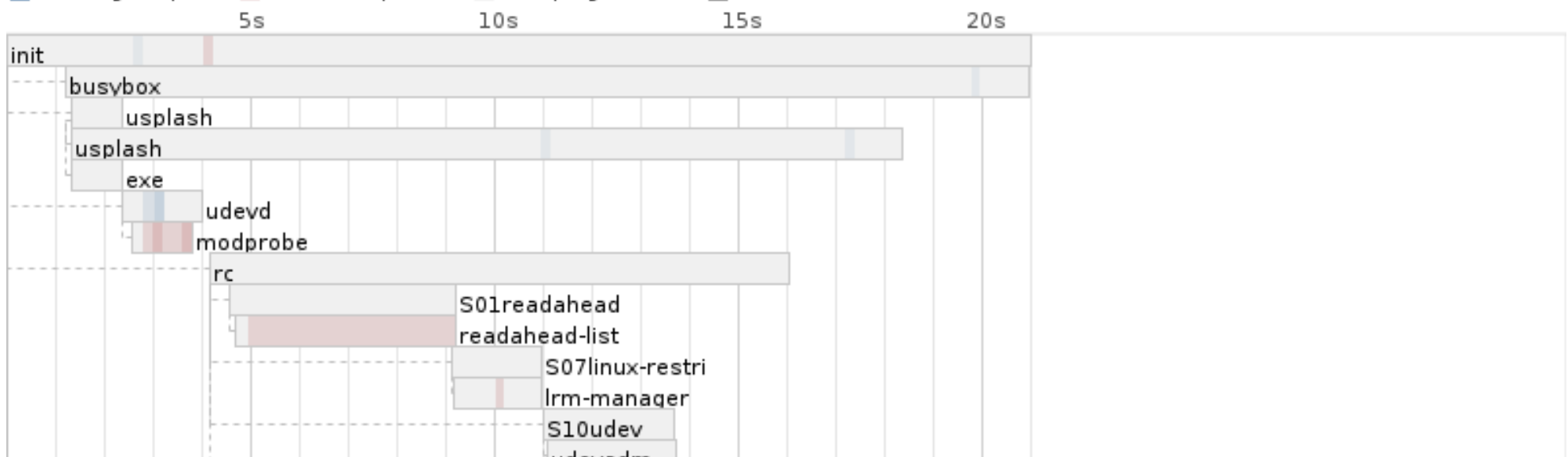
■ CPU (user+sys) ■ I/O (wait)



— Disk throughput ■ Disk utilization

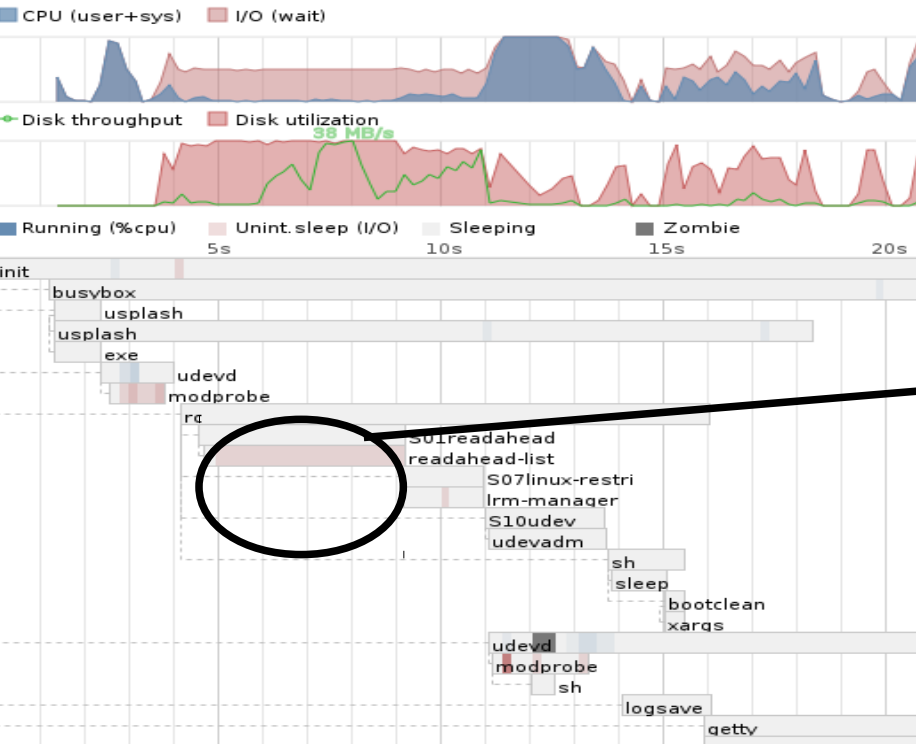


■ Running (%cpu) ■ Unint.sleep (I/O) ■ Sleeping ■ Zombie



Boot chart for maus (Sun Jun 15 10:21:42 CEST 2008)

uname: Linux 2.6.24-18-generic #1 SMP Wed May 28 20:27:26 UTC 2008 i686
 release: Ubuntu 8.04
 CPU: Intel(R) Pentium(R) Dual CPU E2180 @ 2.00GHz
 kernel options: root=UUID=380f65d1-dbc3-4012-bdd5-1cf7fa08d2a7 ro splash vga=794 rootflags=data=write
 time: 0:22

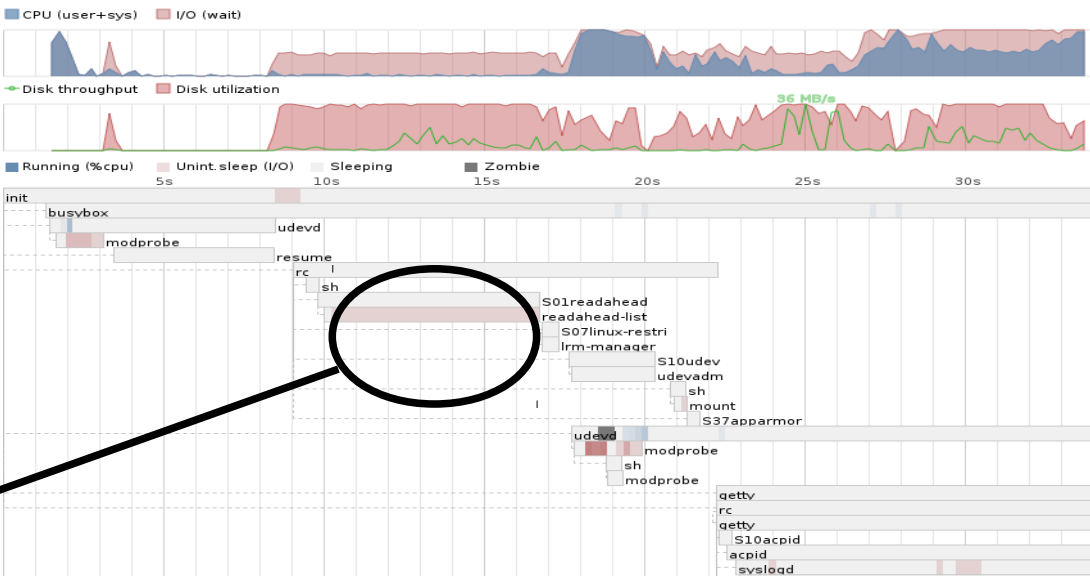


The beginning ... June 2008

- After several years notebook-only I decided to buy a desktop again and recycle some old hardware
- Cost: 200€ (Shuttle K45 + Pentium Duo)
- Boots in 22 seconds
- I/O: ~5s @ 38 Mb/s
- S01readahead – some caching going on

Boot chart for maus (Wed Apr 29 19:01:03 CEST 2009)

uname: Linux 2.6.27-9-generic #1 SMP Thu Nov 20 21:57:00 UTC 2008 i686
 release: Ubuntu 8.10
 CPU: Intel(R) Pentium(R) Dual CPU E2180 @ 2.00GHz
 kernel options: root=UUID=380f65d1-dbc3-4012-bdd5-1cf7fa08d2a7 ro
 time: 0:35



After ~1 year ... April 2009

- Still using the same cheapo hardware
- Software bloat begins to take its toll
- Boots in 35 seconds
- I/O: ~25s @ 5 Mb/s (peak: 36 Mb/s)
- Caching has become inefficient

2009



Boot chart for maus (Sat Oct 31 17:01:56 CET 2009)

uname: Linux 2.6.28-16-generic #55-Ubuntu SMP Tue Oct 20 19:48:24 UTC 2009 i686

release: Ubuntu 9.04

CPU: Intel(R) Pentium(R) Dual CPU E2180 @ 2.00GHz model name: Intel(R) Pentium(R) Dual CPU E2180 @ 2.00GHz (2)

kernel options: root=UUID=380f65d1-dbc3-4012-bdd5-1cf7fa08d2a7 ro

time: 00:31.68

■ CPU (user+sys) ■ I/O (wait)



October 2009

- bought new HDD
- I/O: ~5s @ ~5 Mb/s (Peak: 98 Mb/s)
- But faster I/O cannot help much
 - Partition fragmented
 - CPU bound at the end
- Still boots in 31 seconds!

Boot chart for maus (Fri Apr 23 16:38:36 CEST 2010)

uname: Linux 2.6.31-16-generic #44-Ubuntu SMP Fri Mar 12 15:23:03 UTC 2010 i686

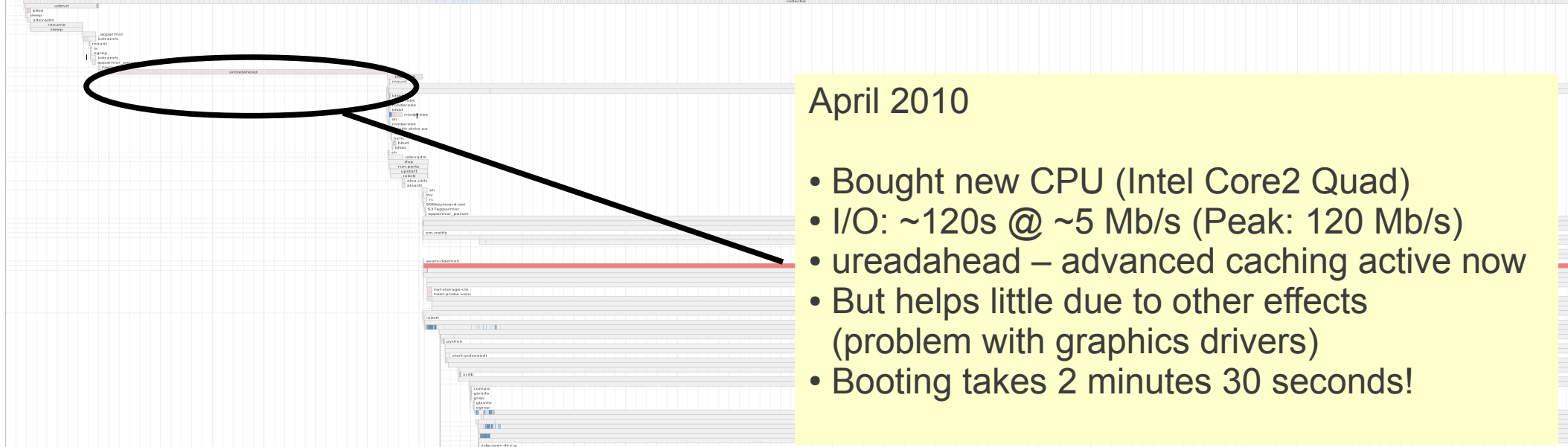
release: Ubuntu 9.10

CPU: Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHz model name: Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHz (4)

kernel options: root=UUID=380f65d1-dbc3-4012-bdd5-1cf7fa08d2a7 ro

time: 02:31.28

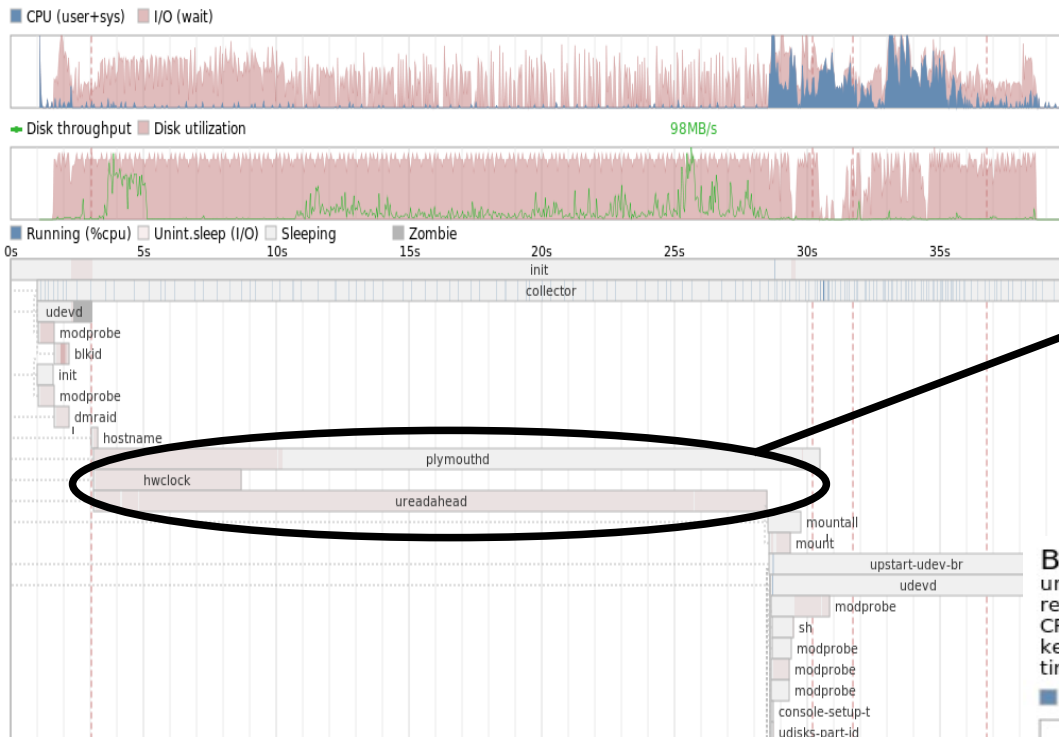
■ CPU (user+sys) ■ I/O (wait)



April 2010

- Bought new CPU (Intel Core2 Quad)
- I/O: ~120s @ ~5 Mb/s (Peak: 120 Mb/s)
- ureadahead – advanced caching active now
- But helps little due to other effects (problem with graphics drivers)
- Booting takes 2 minutes 30 seconds!

Boot chart for maus (Sat May 29 07:57:59 CEST 2010)
 uname: Linux 2.6.32-22-generic #33-Ubuntu SMP Wed Apr 28 13:27:30 UTC 2010 i686
 release: Ubuntu 10.04 LTS
 CPU: Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHzmodel name: Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHzmodel name: Intel(R) Core(TM)2 Quad CPU
 kernel options: root=UUID=380f65d1-dbc3-4012-bdd5-1cf7fa08d2a7 ro
 time: 00:36.81

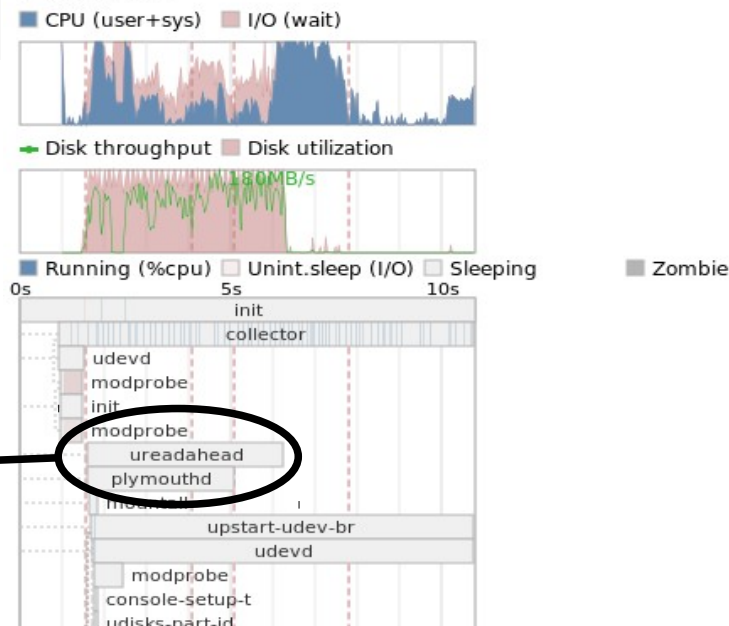


May 2010

- Problem with driver fixed
- ureadahead – cached read now biggest part of boot process
- I/O: ~27s @ ~5 Mb/s (Peak: 98 Mb/s)
- Throttled by disk fragmentation
- Still boots in 36 seconds!

Boot chart for maus (Sat Oct 30 19:38:56 CEST 2010)

uname: Linux 2.6.32-25-generic-pae #45-Ubuntu SMP Sat Oct 16 21:01:33 UTC 2010 i686
 release: Ubuntu 10.04.1 LTS
 CPU: Intel(R) Core(TM)2 Quad CPU Q9505 @ 2.83GHzmodel name: Intel(R) Core(TM)2 Quad CPU
 kernel options: BOOT_IMAGE=/boot/vmlinuz-2.6.32-25-generic-pae root=UUID=91805cd8-e3e0-47da-a
 time: 00:07.86



Recently ... October 2010

- Bought a 40 Gb SSD (Intel Postville)
- I/O: ~5s @ 180 Mb/s all the way
- Boots in <8 seconds!

Summary – Linux Boot Sequence



So what do we learn already about efficient local I/O just from booting our computer?

- Wall clock time can stay pretty much the same if other factors (CPU / bloated software) are the limiting factor → *see other lectures in this school*
- Cacheing/readahead *could* speed things up – but is no panacea
- Disk fragmentation / random access can be a major brake
 - It's not true that Linux FS don't have to be defragmented
They have to, they just can't be...
- Beyond that Hardware is the biggest factor
 - Can improve only if we have the money or political power
 - And only if the paradigm changes: HD → SSD
 - This lecture cannot help with improving these –
but it can help you find out when they are the problem

But that was just my private desktop ... now let's look at daily work ...



Times

- Walltime Overall time to do one turn on task at hand
- Time to start Waiting Time in Queue
- CPU Time Time spent processing the read data $\text{CPU/Wall} = \text{CPU Efficiency}$
- System Time Time spend reading the data
- Real Time For simplicity we assume $\text{RT} = \text{CT} + \text{ST}$
Corresponds to Walltime in the case of a single job

Size

- MB processed

Speed

- Rate: Mb/s from Disk
- Events/s Useful to compare totally different setups

Other

- Transactions The number of calls to read, or the number of network packets needed to transfer the files

When doing the excercises please document these numbers in a table/spreadsheet

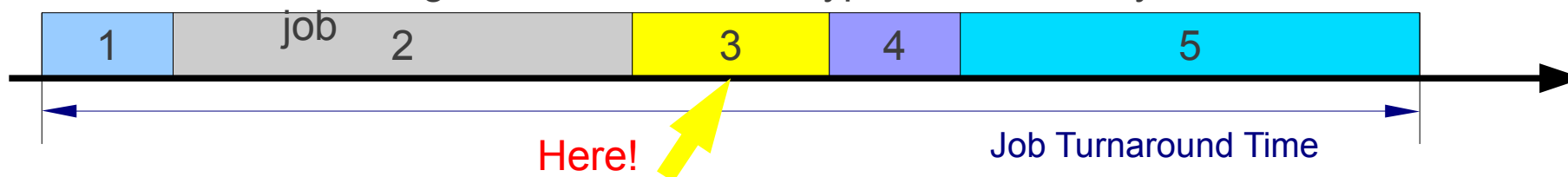
Excercise	TC Size [Mb]	RT [s]	CT [s]	Transactions
-----------	--------------	--------	--------	--------------

Where does efficient I/O matter in practice?



- Basic assumption of this lecture
 - You are running a job that reads data files from your experiment = you are doing „Analysis“
 - You want to do this as fast as possible

Waiting times involved in a typical GRID analysis



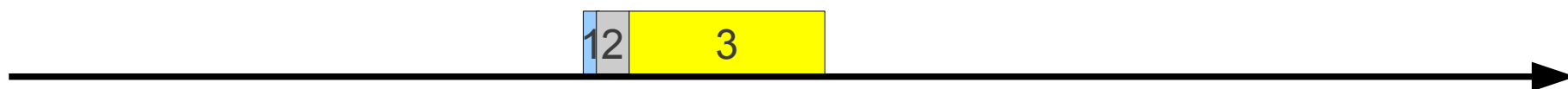
		Typical time scale (my subjective experience...)
1 Submission	Upload+brokerage	10 min.
2 TimeToStart	waiting in batch queue on site	1-10 hours
3 Running Job	← I/O going on!	10 min.
4 Postprocessing	Assembling+registering dataset, notifying user	10 min.
5 Downloading	dq2-get/DaTrl + verifying	1-10 hours

- This lecture focusses on optimising I/O during running of the job
 - We are not talking about grid upload/download times
- If you are running on the Grid I/O is typically not the bottleneck
- Still, I/O adds to the total turnaround and should be optimised

Where does efficient I/O come in?



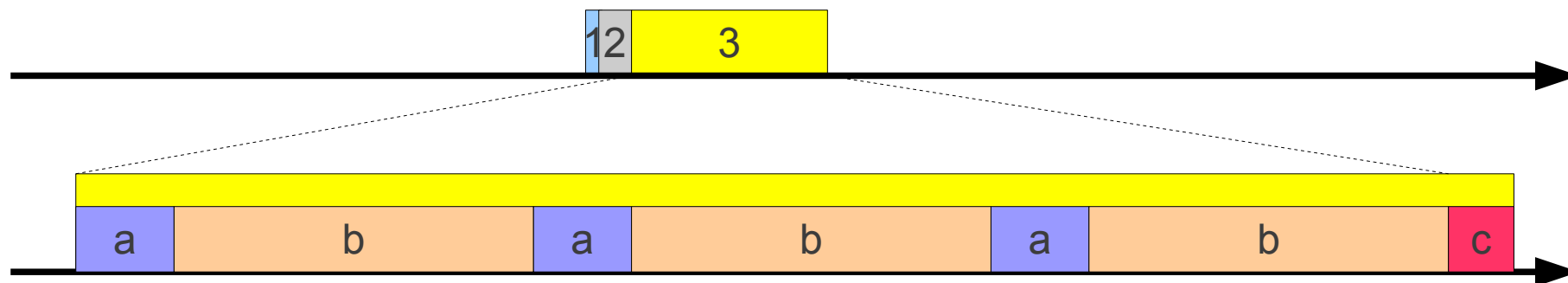
- Waiting times involved in a typical local batch job



		Typical time scale (my subjective experience...)
1 Submission		10 ms
2 TimeToStart	waiting in batch queue on site	1 s
3 Running Job	← I/O going on!	10 min.
4 Postprocessing		—
5 Downloading		—

- I/O during running of the job could dominate the job turnaround
- If you are running on the Grid you have other problems
- Still, I/O adds to the total turnaround and should be optimised

What is going on during one job



a Opening File
b Reading File
c Writing Output

Done once per file
Focus of this lecture
Done once per job in the end

- In the following we assume
 - One job = one core
 - Reading one file, or several files sequentially
 - Most typical case during analysis

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags,
        [mode_t mode]);
```

Open file

- Independent of device
- Many options
 - **O_RDONLY**, **O_WRONLY**, or **O_RDWR**
 - How the file will be accessed
 - **O_CREAT**, **O_EXCL**, and **O_TRUNC**
 - File creation disposition
- Determine (non-) blocking I/O

- `#include <unistd.h>`
 - `ssize_t read(int fd, void *buf, size_t count);`
 - `ssize_t write(int fd, void *buf, size_t count);`
- `#include <sys/uio.h>`
 - `ssize_t readv(int fd, const struct iovec *iov, int iovcnt);`
 - `ssize_t writev(int fd, const struct iovec *iov, int iovcnt);`

Reading / writing
a single buffer

Reading / writing
a vector of buffers

Instead of testing these directly we will „wrap“ them in ROOT



open()

Examples...

- Transparent for user via Tfile::Open()
- Derived versions provide interface to various protocols used to access files:
- Dcache, http://, xrootd, ...



readv()

• xrootd

- TFile f1("root://machine1.xx.yy/file1.root")

• dCache

- TFile f2("dcap://machine2.uu.vv/file2.root")

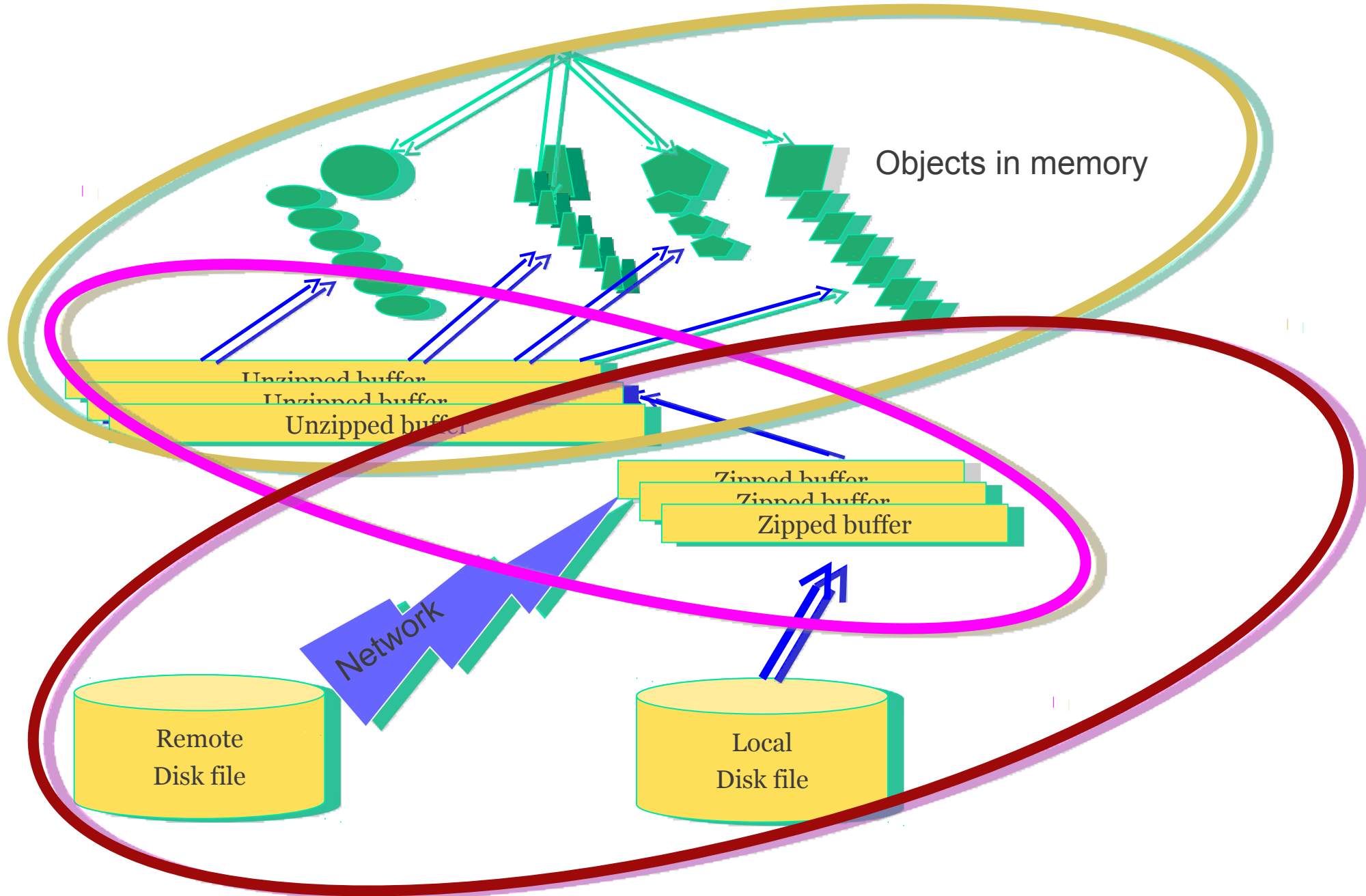
• httpd

- TFile f3(<http://something.nikhef.nl/file3.root>);
- uses a standard (eg apache2) web server

TDCacheFile::ReadBuffers()
dc_readv2()

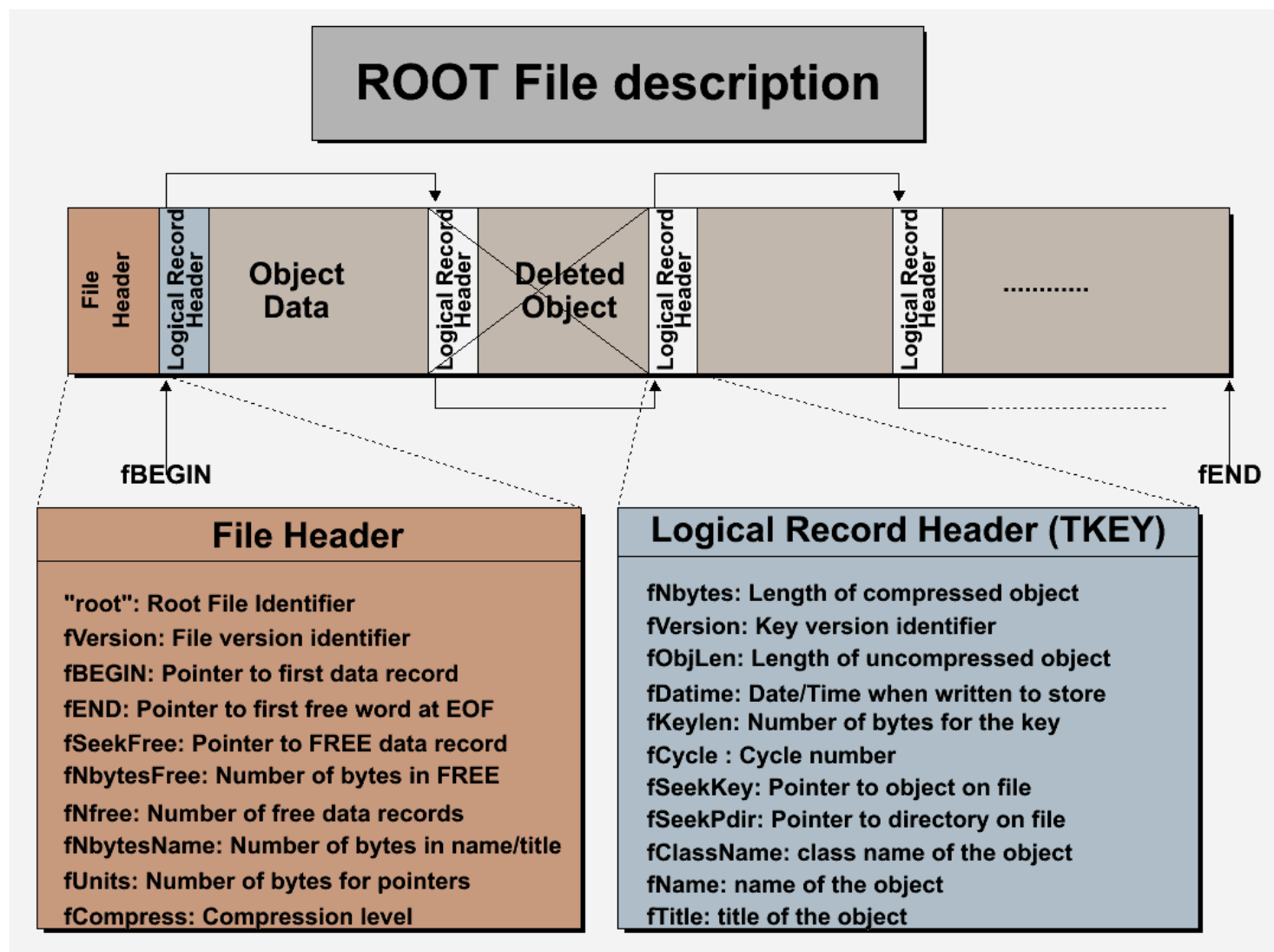
```
239 //
240 //
241 Bool_t TDCacheFile::ReadBuffers(char *buf, Long64_t *pos, Int_t *len, Int_t nbuf)
242 {
243     // Read the nbuf blocks described in arrays pos and len.
244     // where pos[i] is the seek position of block i of length len[i].
245     // Note that for nbuf=1, this call is equivalent to TFile::ReadBuffer.
246     // This function is overloaded by TNetFile, TWebFile, etc.
247     // Returns kTRUE in case of failure.
248
249     #ifdef _IOVEC2_
250
251     iovec2 *vector;
252
253     vector = (iovec2 *)malloc(sizeof(iovec2)*nbuf);
254
255     Int_t total_len = 0;
256     for (Int_t i = 0; i < nbuf; i++) {
257         vector[i].buf = &buf[total_len];
258         vector[i].offset = pos[i] + fArchiveOffset;
259         vector[i].len = len[i];
260         total_len += len[i];
261     }
262
263     Int_t rc = dc_readv2(fD, vector, nbuf);
264     free(vector);
265
266     if (rc == 0) {
267         fBytesRead += total_len;
268         SetFileBytesRead(GetFileBytesRead() + total_len);
269         return kFALSE;
270     }
271
272     #endif
```

ROOT I/O Landscape



- Not very complicated structure
- Just a list of Tkey's + object pairs
- Can look at it using `TFile::Map()`

<http://root.cern.ch/root/html/TFile.html>



- Loading TFiles

```
$ root -l test.root
```

- Shortcut for

```
$ root
```

```
$> _file0 = TFile::Open("test.root");
```

Common pitfall: files not ending with .root are interpreted as scripts

How to inspect files

File Structure:

- `$ _file0->Map()`

Logical Structure:

- TBrowser: GUI to inspect ROOT objects (among other things...)
- `$ new TBrowser`
- I prefer: List directory – all TFiles are TDirectory's
- `$.ls`

TBrowser example



- Our default test file in the TBrowser
- /storage/software/brandt/D3PD.011.root

Objects on the file
We are only interested
In the TTree „physics“

Branches
Of the Ttree

The screenshot shows the TBrowser interface with the file `D3PD.011.root` open. The left pane, titled "All Folders", displays a tree structure of folders. The right pane, titled "Contents of '/ROOT Files/D3PD.011/root'", displays a table of objects.

Left Pane: All Folders

- root
- PROOF Sessions
- /storage/software/brandt
 - batch
 - files
 - D3PDs
 - tutorial
 - batch_tutorial
- ROOT Files
 - D3PD.011.root (selected)
 - physicsMeta;1
 - physics;7
 - el_jetcone_dr
 - el_jetcone_signedIP
 - el_jetcone_ptrel
 - el_jetcone_index

Right Pane: Contents of "/ROOT Files/D3PD.011/root"

Name	Title
CollectionTree;1	CollectionTree
Lumi;1	Lumi
Schema;1	
physics	physics
physics;6	physics
physics;7	physics
physicsMeta;1	physicsMeta

14 Objects.



- TBranches

Structure on disk

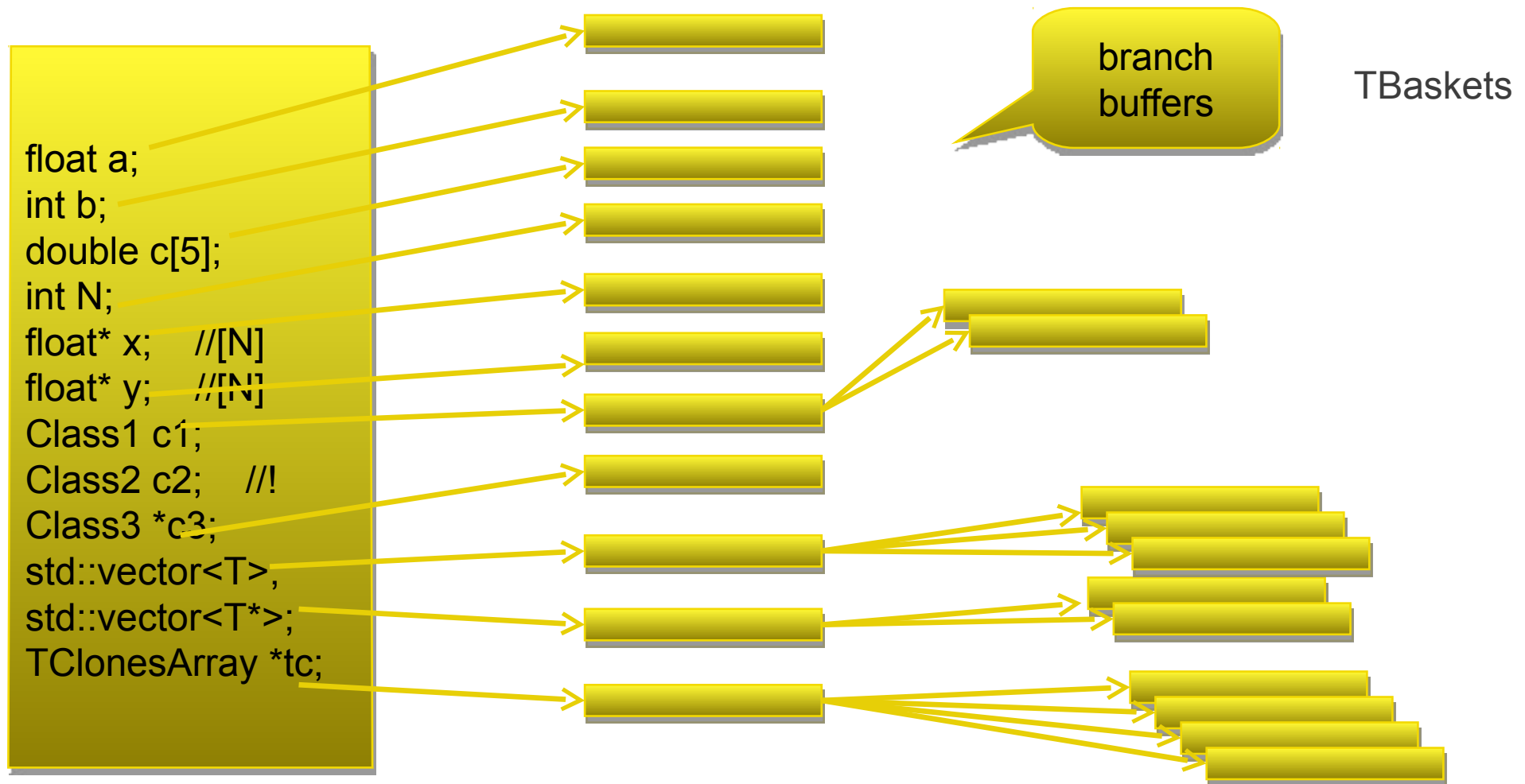
- A series of Tbaskets (buffers)
- Each basket has a certain size
Previous default: 32kb / basket
- Filled with events until full, then
 - Basket is zipped and written

Can see the baskets using
`TFile::Map()`

Where do the TBranches come from?



- ROOT does Automatic branch creation from object model
- Some options exist to influence this



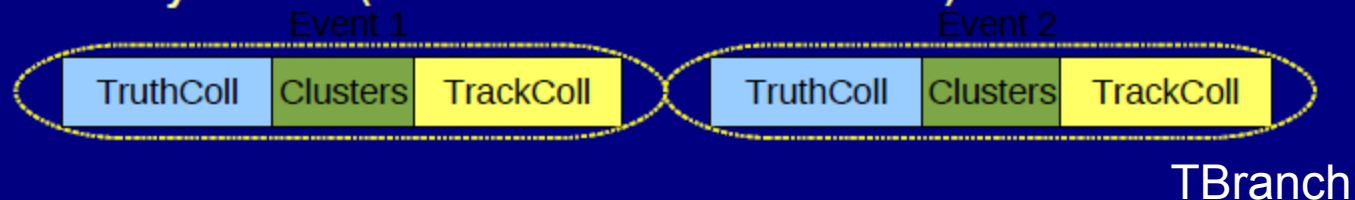
28

17 June 2010

- Physics: Events belong together
- Ordering By Event
- But sometimes we do not need to read all objects
- For I/O then better to write objects close together
- ROOT: Can do this using TTree „Splitting“
- Full Splitting:
 - Split also into members of objects
 - Then also better compression

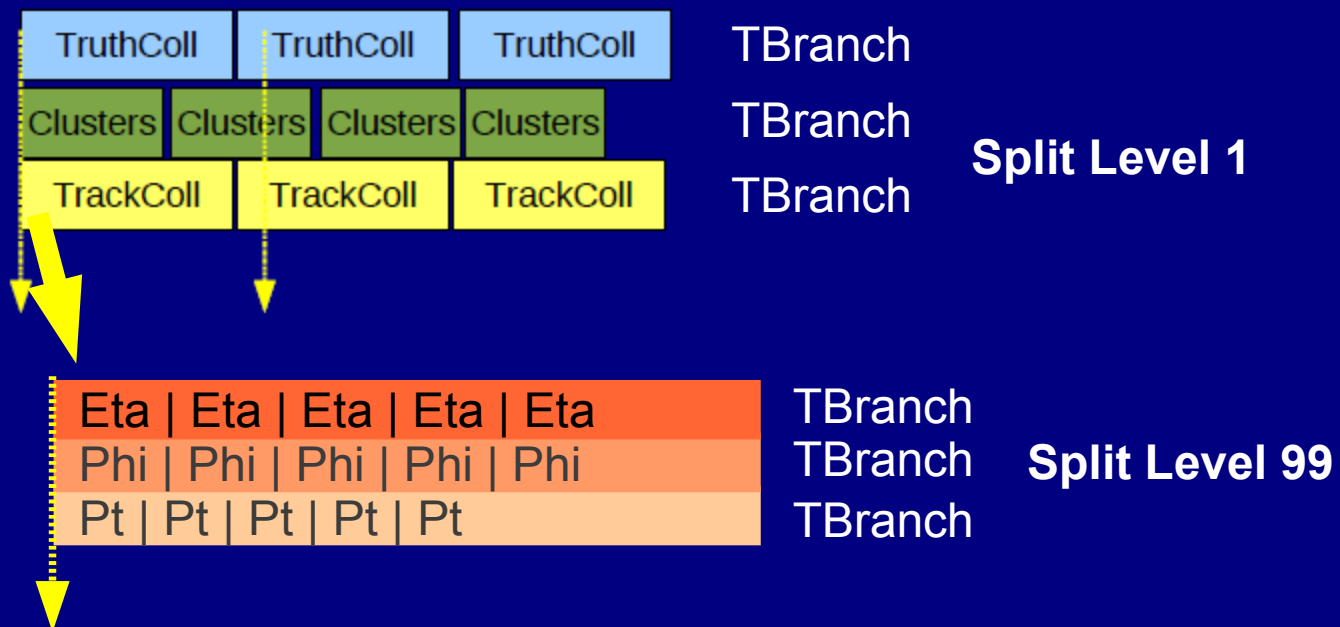
How are data objects written to disk

- By event (most Raw Data Streams) **Split Level 0**



- By object, splitting events (most ROOT files)

- Allows to read subset of event data




Recently this big difference has become less important due to memberwise streaming

- Open the file
- Read all events in an event loop

```
void taodr(Int_t cachesize=10000000) {  
  
    TFile *f = Tfile::Open("D3PD.011.root");  
    TTree *T = (TTree*)f->Get("physics");  
    Long64_t nentries = T->GetEntries();  
    //T->SetCacheSize(cachesize);  
    //T->AddBranchToCache("*",kTRUE);  
  
    TTreePerfStats ps("ioperf",T);  
  
    for (Long64_t i=0;i<nentries;i++) {  
        T->GetEntry(i);  
    }  
    T->PrintCacheStats();  
    ps.SaveAs("aodperf.root");  
}
```

Cache –
Stay tuned



Simplified version of the macro we use to measure I/O performance

- I/O benchmarking: TTreePerfStat (introduced in ROOT 5.26)
- Will be the workhorse used in the exercises

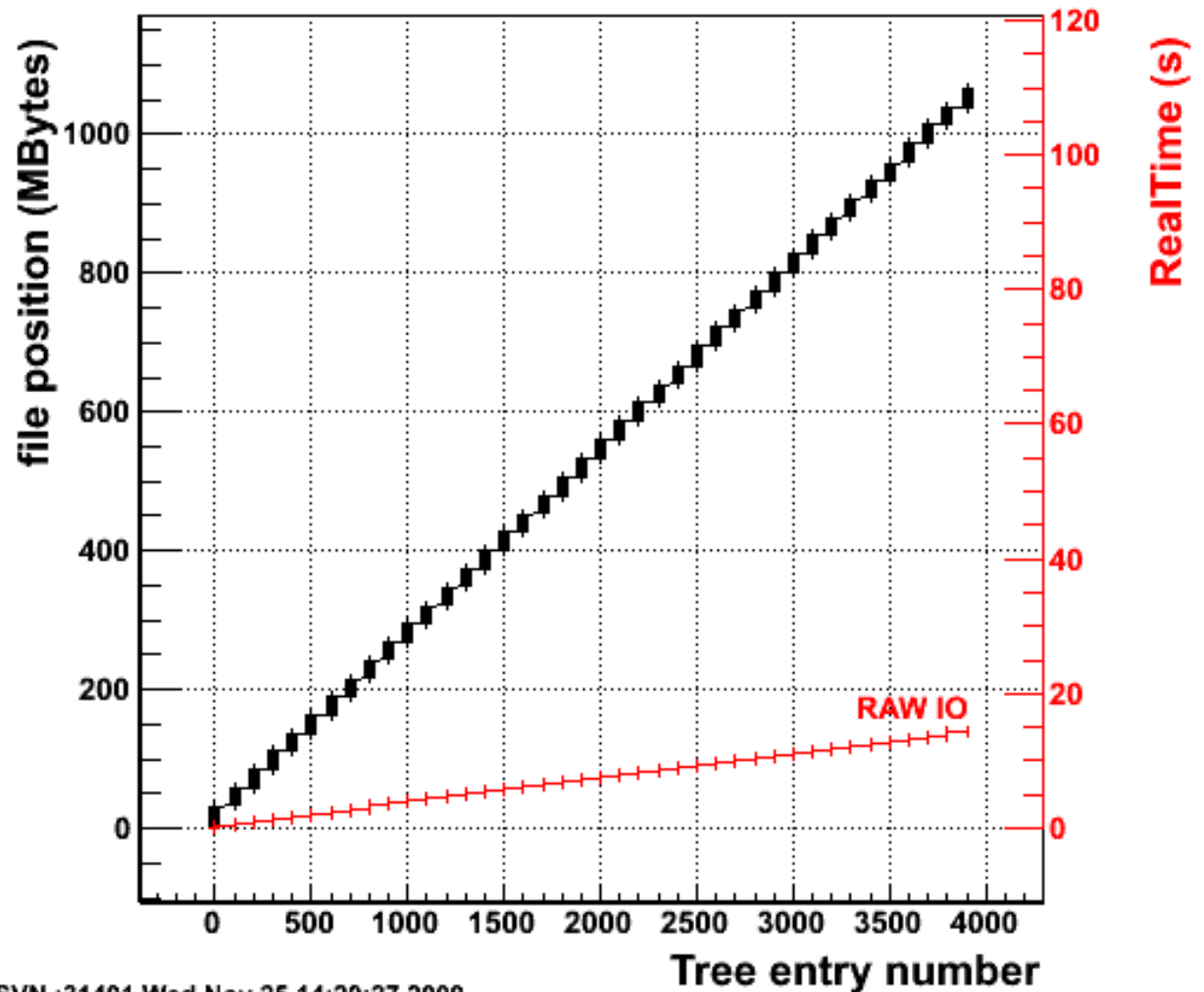
atlasFlushed.root/CollectionTree

Cache

Transactions

Real Time
CPU Time
Disk Time
Disk Mb/s

```
TreeCache = 39 MBytes
N leaves = 9705
ReadTotal = 1070.72 MBytes
ReadUnZip = 3936.2 MBytes
ReadCalls = 532
ReadSize = 2012.637 KBytes/read
Readahead = 256 KBytes
Readextra = 0.00 per cent
Real Time = 109.859 seconds
CPU Time = 95.890 seconds
Disk Time = 14.527 seconds
Disk IO = 73.705 MBytes/s
ReadUZRT = 35.830 MBytes/s
ReadUZCP = 41.049 MBytes/s
ReadRT = 9.746 MBytes/s
ReadCP = 11.166 MBytes/s
```



Darwin guest216.Inf.Root5.25/05, SVN :31401 Wed Nov 25 14:20:27 2009

Reading an old, unoptimised File

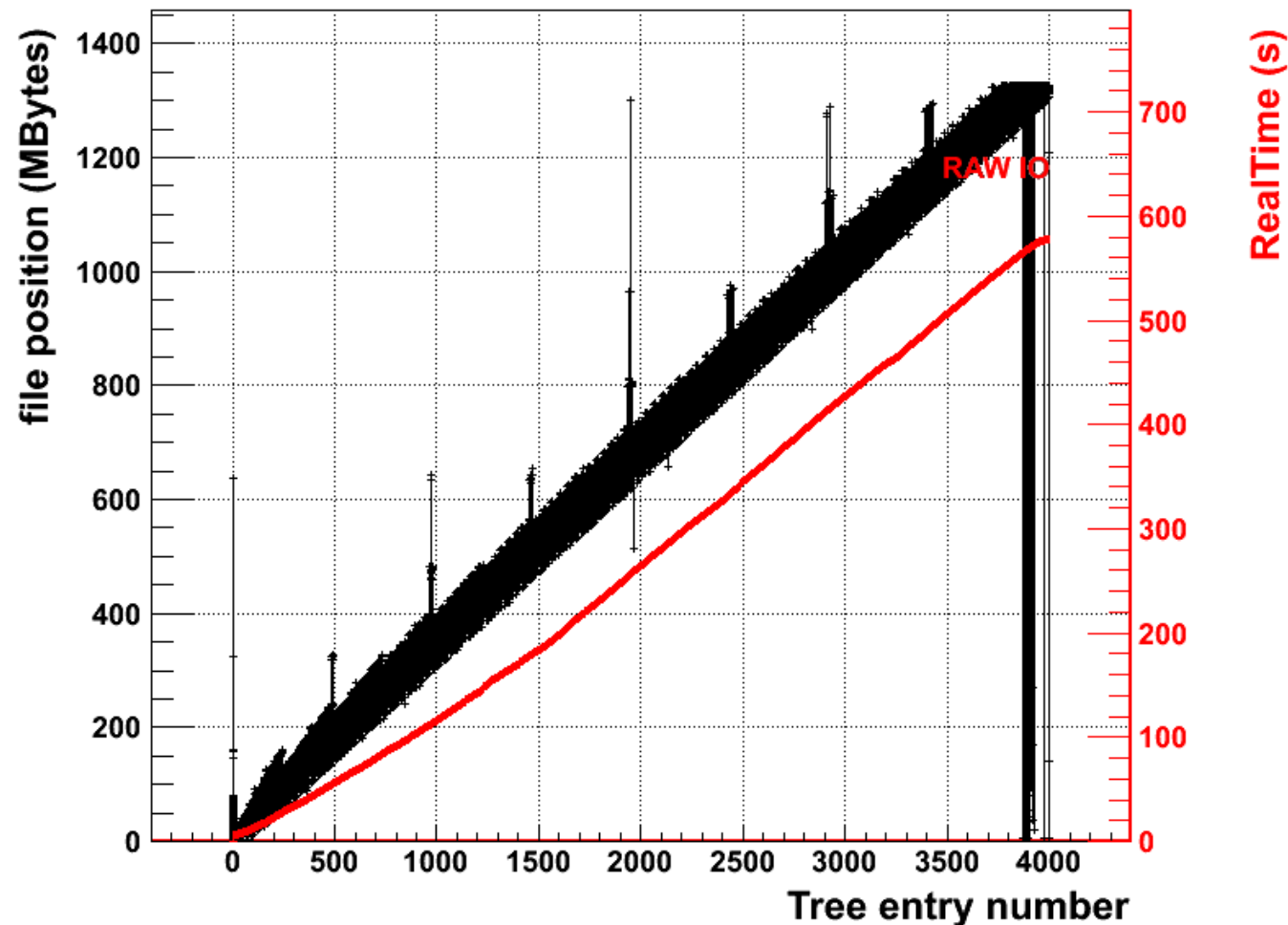


- Default was for all buffers to have the same size.
- Branch buffers are not full at the same time.
 - A branch containing one integer/event and with a buffer size of 32Kbytes will be written to disk every 8000 events.
 - while a branch containing a non-split collection may be written at each event.
- Many small reads.
- Backward seeks.
- Gap in reads.


Manual Ordering of Baskets
Is possible during fast cloning



Exercises



Enforce “clustering”:

- 
- Once a reasonable amount of data (default is 30 Mbytes) has been written to the file, all baskets are flushed out and the number of entries written so far is recorded in **fAutoFlush**.
 - From then on for every multiple of this number of entries all the baskets are flushed.
 - This insures that the range of entries between 2 flushes can be read in one single file read.
 - The first time that **FlushBaskets** is called, we also call **OptimizeBaskets**.
 - The **TTreeCache** is always set to read a number of entries that is a multiple of **fAutoFlush** entries.

No backward seeks needed to read file.

Dramatic improvement in the raw disk IO speed.

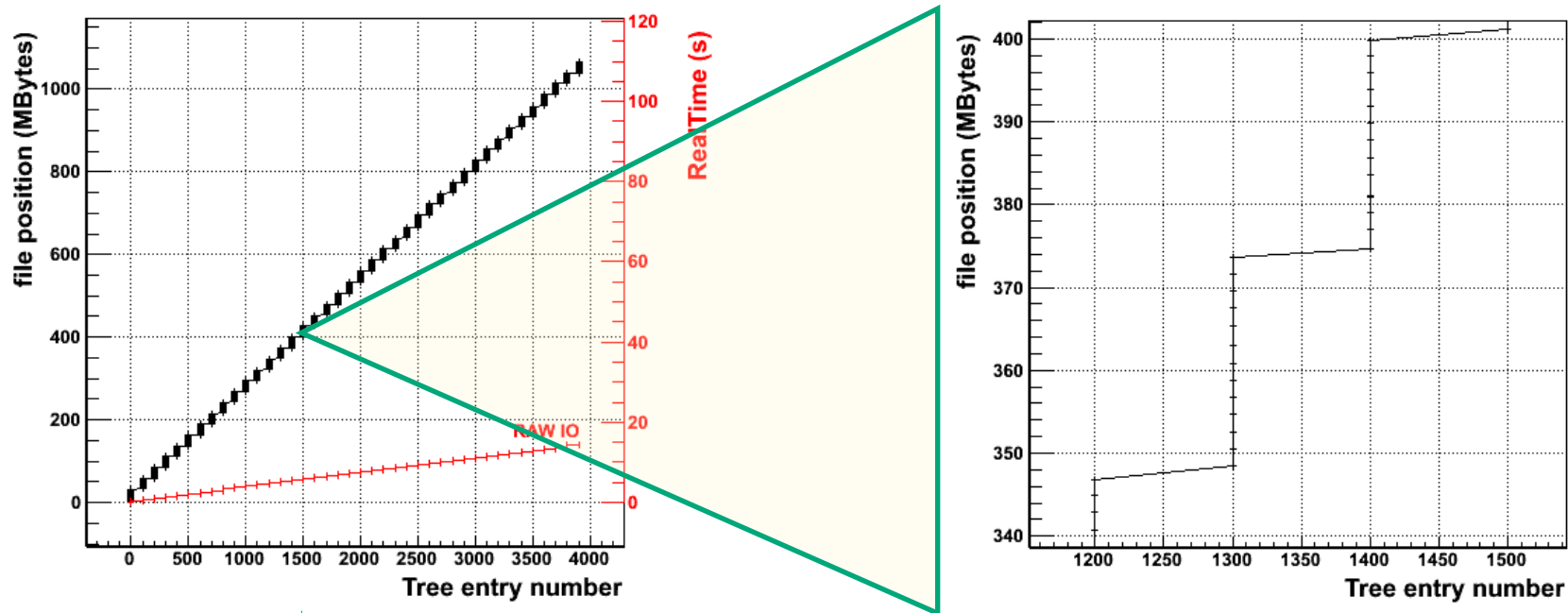
Basket Optimisation and Auto Flush



These solutions are available **only** in **v5.26** and above.

Automatically tweak basket size.

Flush baskets at regular intervals.



- 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](#).
- Can partially compensate for bad/complex persistent model
- Typical size of the [TreeCache](#) is 30 Mbytes, but higher values will always give better results.

- Done via the TTree

```
f = new TFile ("xyz.root");  
T = (TTree*)f->Get("Events");  
T->SetCacheSize(30000000);  
T->AddBranchToCache("*");
```

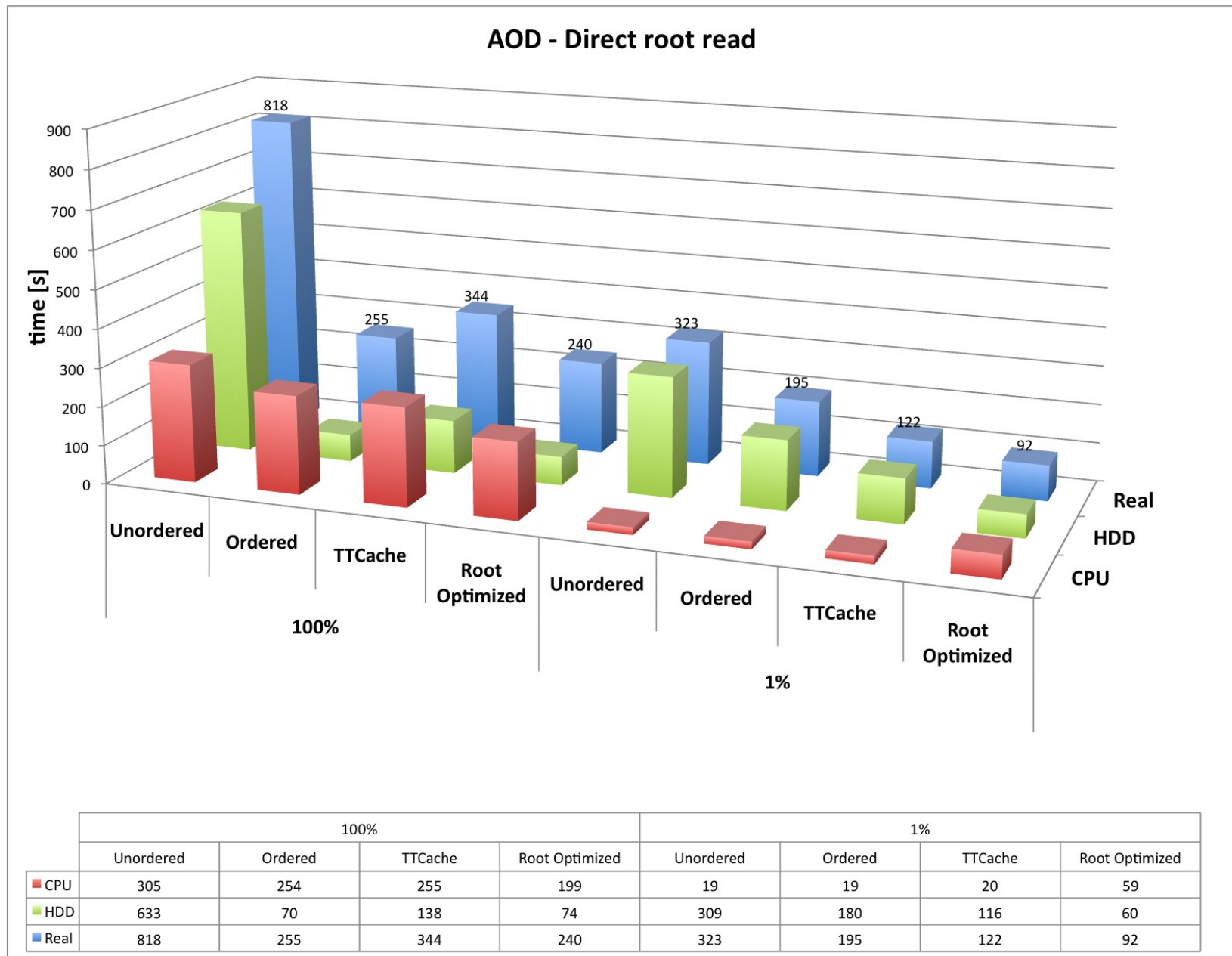
Possible Configuration

- Automatic (Learning Phase)
Learn used branches from given number of entries
- Give branches explicitly

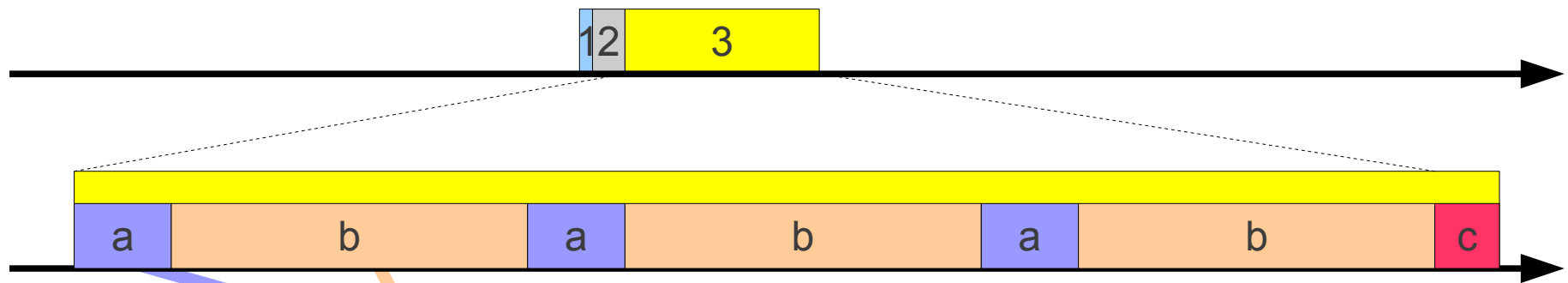
all

Just some

```
T->SetCacheSize(cachesize);  
if (cachesize != 0) {  
    T->SetCacheEntryRange(efirst,elast);  
    T->AddBranchToCache(data_branch,kTRUE); // Request all the sub branches too  
    T->AddBranchToCache(cut_branch,kFALSE);  
    T->StopCacheLearningPhase();  
}
```



Recap: What is going on during one file



a) Opening TFile

- Read Header
- Read StreamerInfo
 - „Instructions“ how to deserialize objects

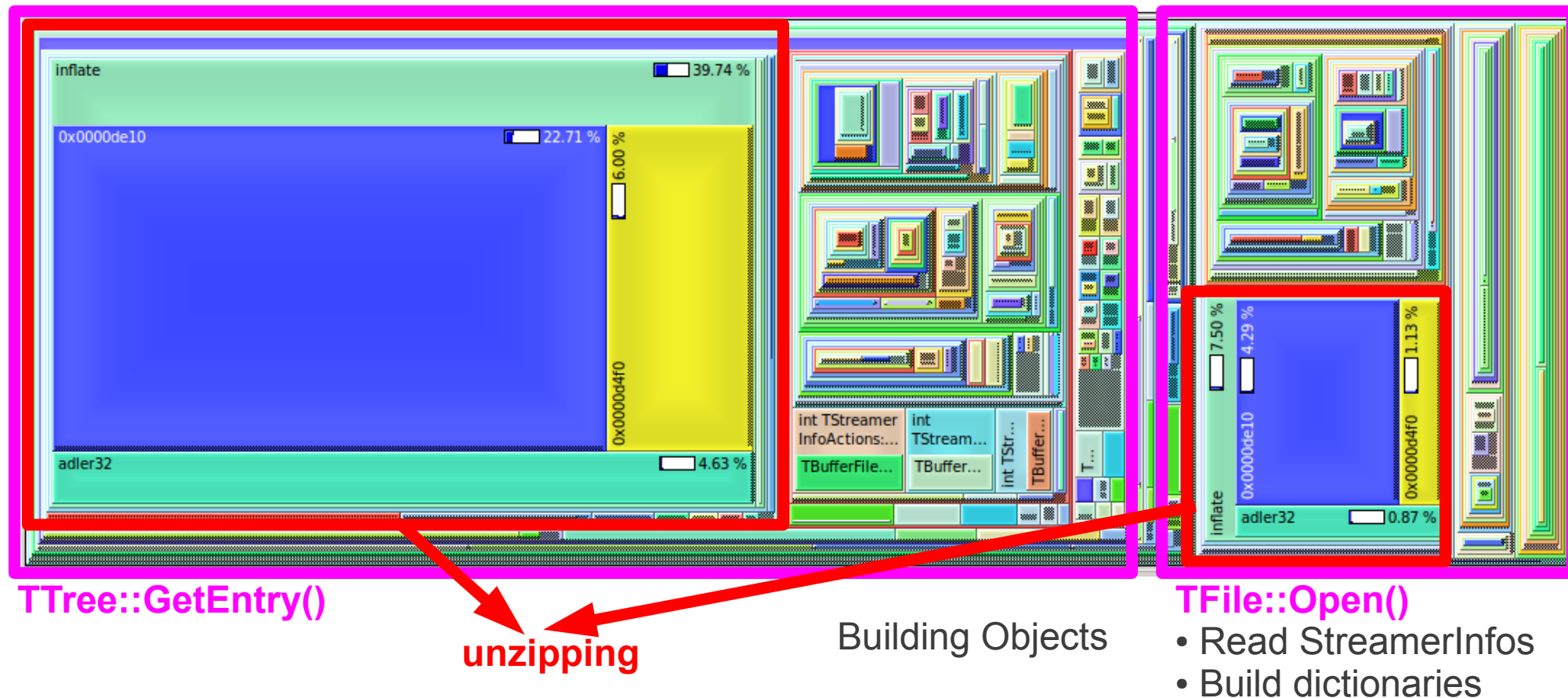
b) Reading TFile

- Read buffers (TBaskets)
- Unzip buffers
- Deserialize objects

What happens when reading a TFile/TTree?



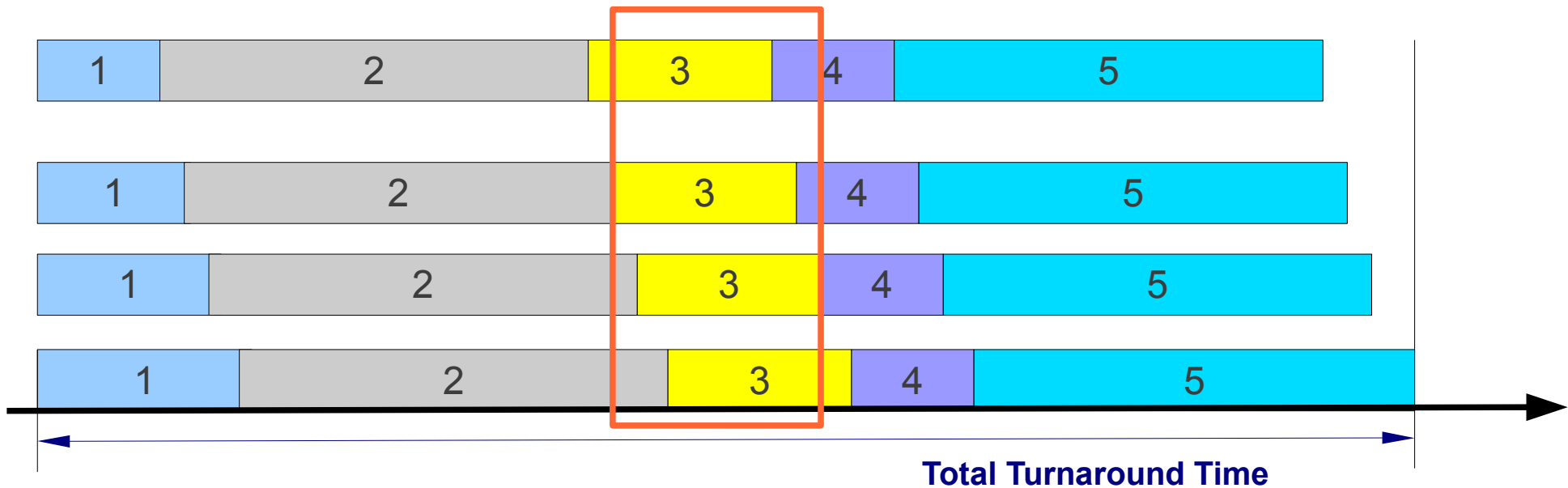
- Load file from SSD – CPU bound (98% CPU Efficiency)
- Callee Map in valgrind when looping on Ttree::GetEntry() for 1000 Events



- ~20% Opening file is non-negligible overhead
 - use large files instead of many small ones – merge those!
- ~50% Unzipping – usually 1 – 20%
 - (don't try to use unzipped files next time ...)
- Rest is taken by building EDM
- Nota bene: We are not actually doing anything with the data yet, just loading + building objects in memory!

And now for something more complex ...

Reading Many Files with Many Jobs



- Assuming all jobs run on same site, parallel I/O will be going on
- Also the case for local batch systems or use of multiple cores (eg. PROOF, AthenaMP)
 - We will be able to test this using the setup provided by the school

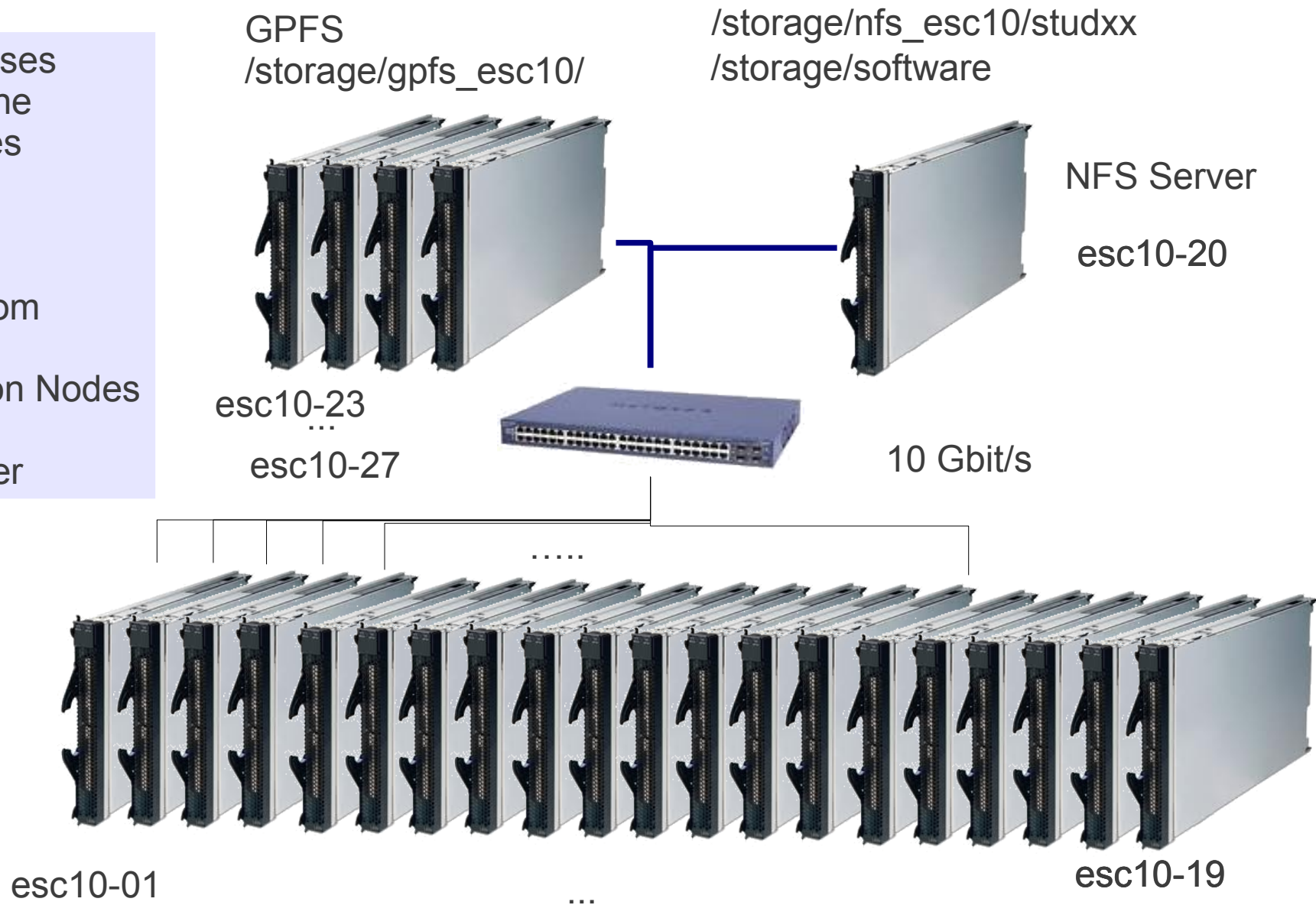
Infrastructure at this school



In the exercises we will map the I/O capabilities of the cluster set up for us:

Read Files from

- Local Disk on Nodes
- NFS Server
- GPFS Server



- Using a few simple scripts to profile a cluster
- Pick a set of files stored on a particular file system
- Send many reading jobs to the farm to read the files
- The reading jobs write out several statistics with time stamps.
Using this it is possible to correlate the behaviour of different jobs
- The integral will show the behaviour of the total cluster

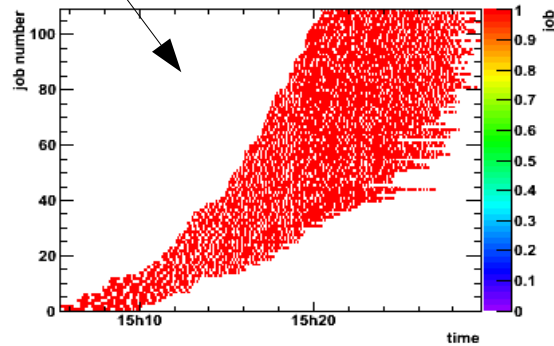
Example output for batch tests

Jobs running (red)
Or waiting/not running (white)

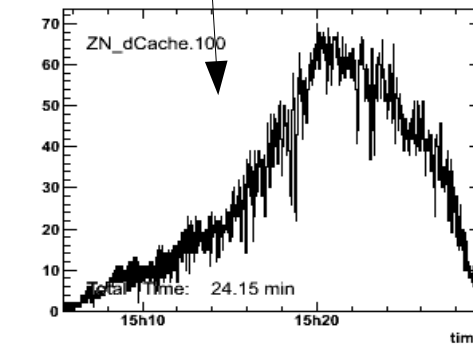
Jobs running at same time

Average CPU
Fraction

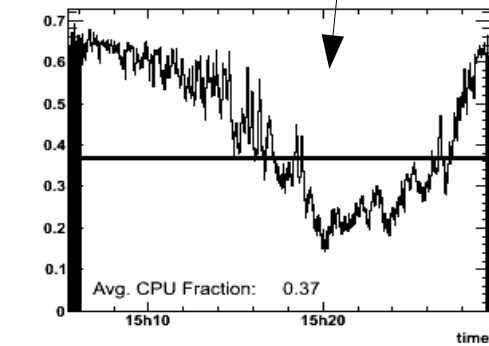
Files being Processed



Number of Jobs Reading Files at the Same Time

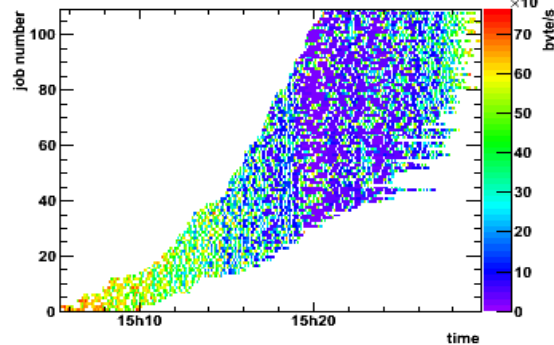


Average CPU Fraction

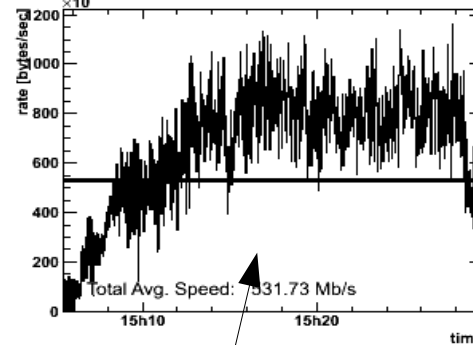


Job

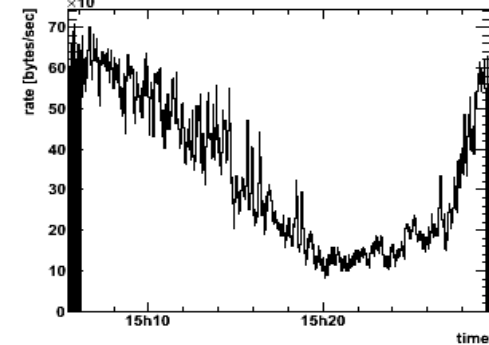
Raw I/O Rate per File



Total Raw I/O Rate



Average Raw IO Rates per Running Job



Time

- Same layout for every plot
- 2D plots show behaviour of individual jobs
- 1D plots show integral

Total I/O rate

- I/O can be achieved by identifying and eliminating one bottleneck after the other
- The order in which you are most likely to encounter I/O bottlenecks:

- Latency

Fix:

- Read big chunks only – use caching
- Do sequential file access – use ordered or optimised files

- Bandwidth

Fix:

- Have to improve hardware

- CPU

Fix:

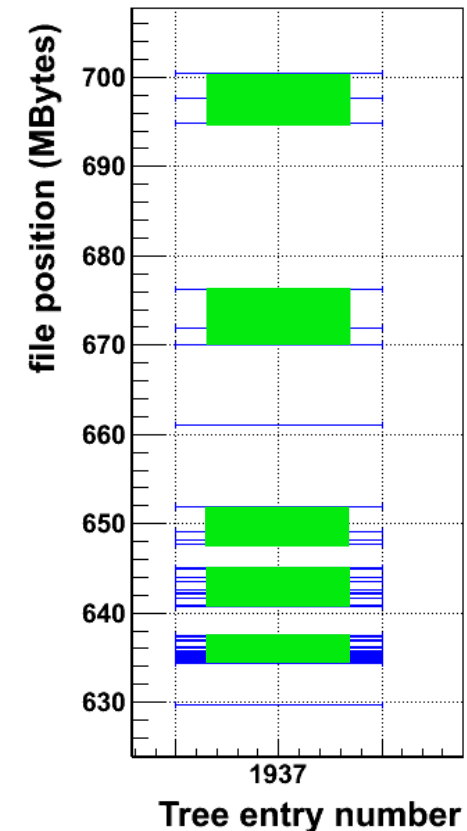
- Keep a simple file format / persistent model

BACKUP

The Readahead Cache



- The **readahead cache** will read all non consecutive blocks that are in the range of the cache.
- It minimizes the number of disk access. This operation could in principle be done by the OS, but the fact is that the OS parameters are not tuned for many small reads, in particular when many jobs read concurrently from the same disk.
- When using large values for the **TreeCache** or when the baskets are well sorted by entry, the **readahead cache** is not necessary.
- Typical (default value) is 256 Kbytes, although 2 Mbytes seems to give better results on Atlas files, but not with CMS or Alice.



- Used for split collection inside a *TTree*.
- Now the *default* for streaming collections even when not split.
- Better compression, faster read time.

Results for CMS files,

- ✦ some fully split
- ✦ some unsplit

