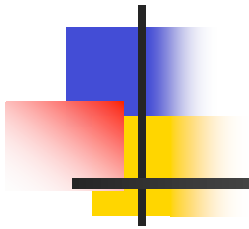


Improvements in the ROOT Cache

ROOT Users Workshop 2007

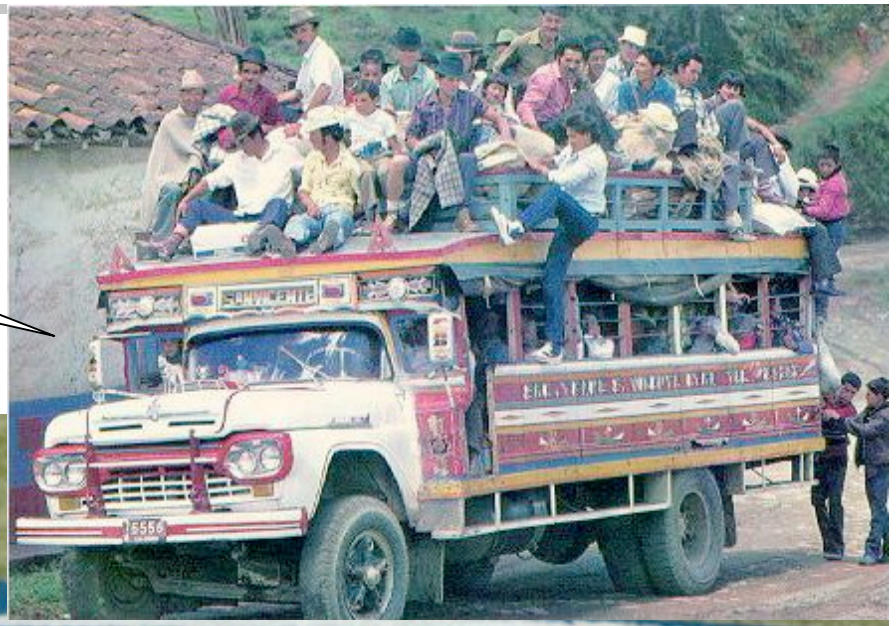
Leandro Franco
CERN

Roadmap

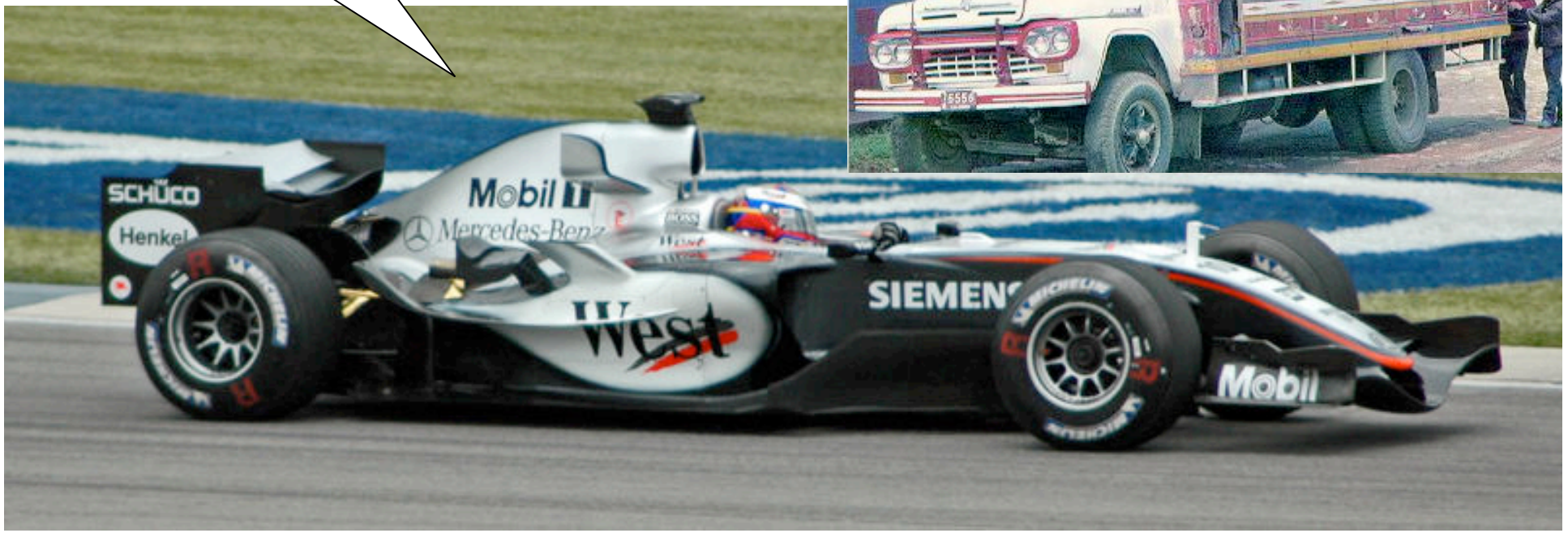
- Intelligent Pre-fetching
  - Definition of latency and bandwidth
  - Description of the problem.
  - Definition of a solution.
  - Limitations of that solution.
  - Tests and comments.
- Parallel Unzipping

Latency vs. Bandwidth

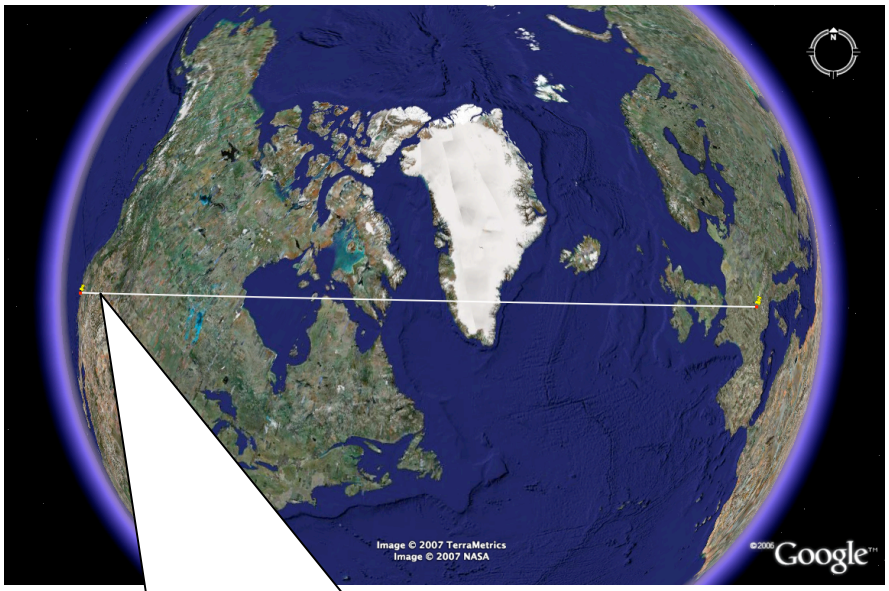
How much can you carry?



How fast can you go?



Latency vs. Bandwidth



In networking, different kind of connections can have similar “physical”speeds. Therefore, latency is mostly related to distance.

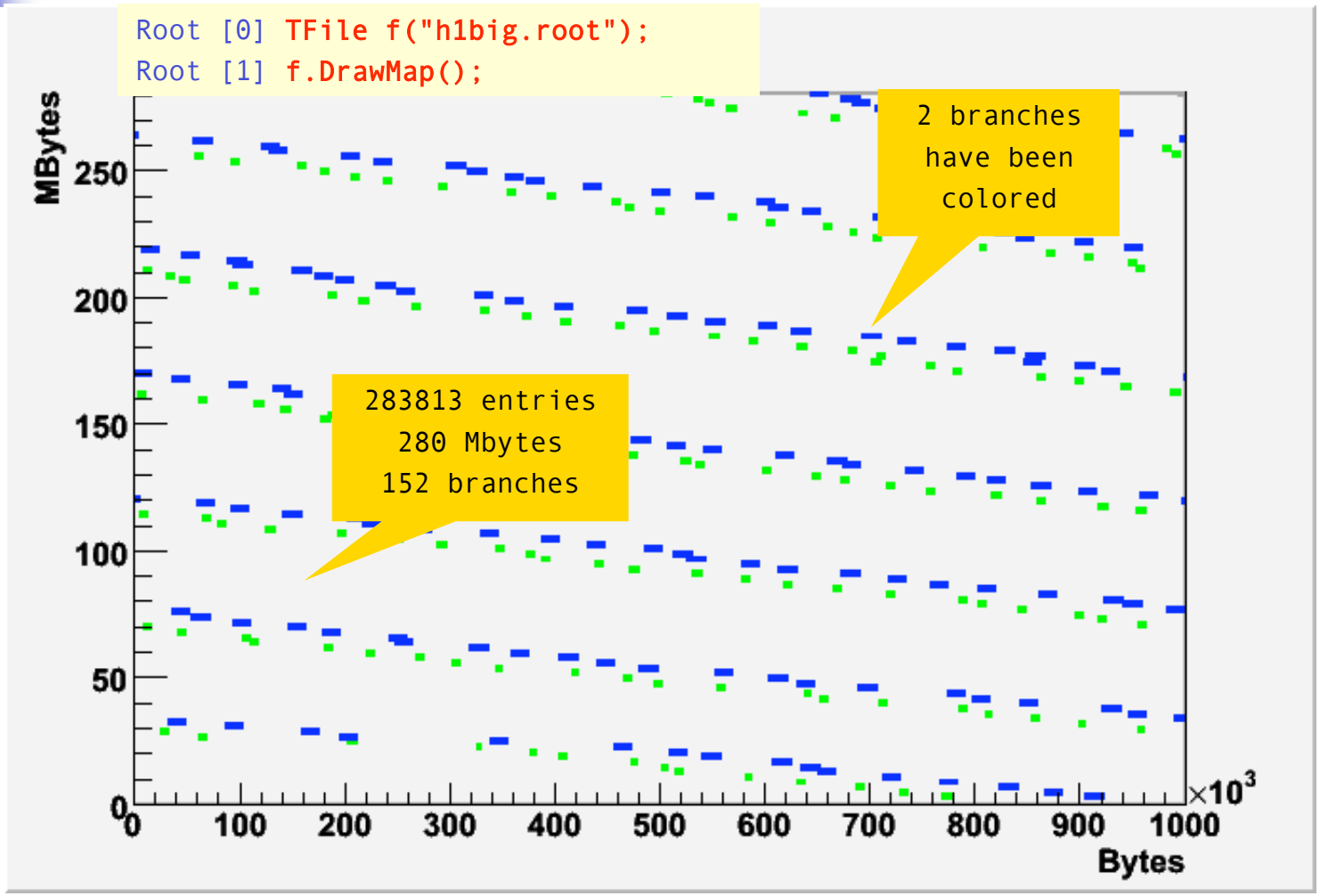
While bandwidth gives us a measure of the “diameter” of the pipe.

Problem

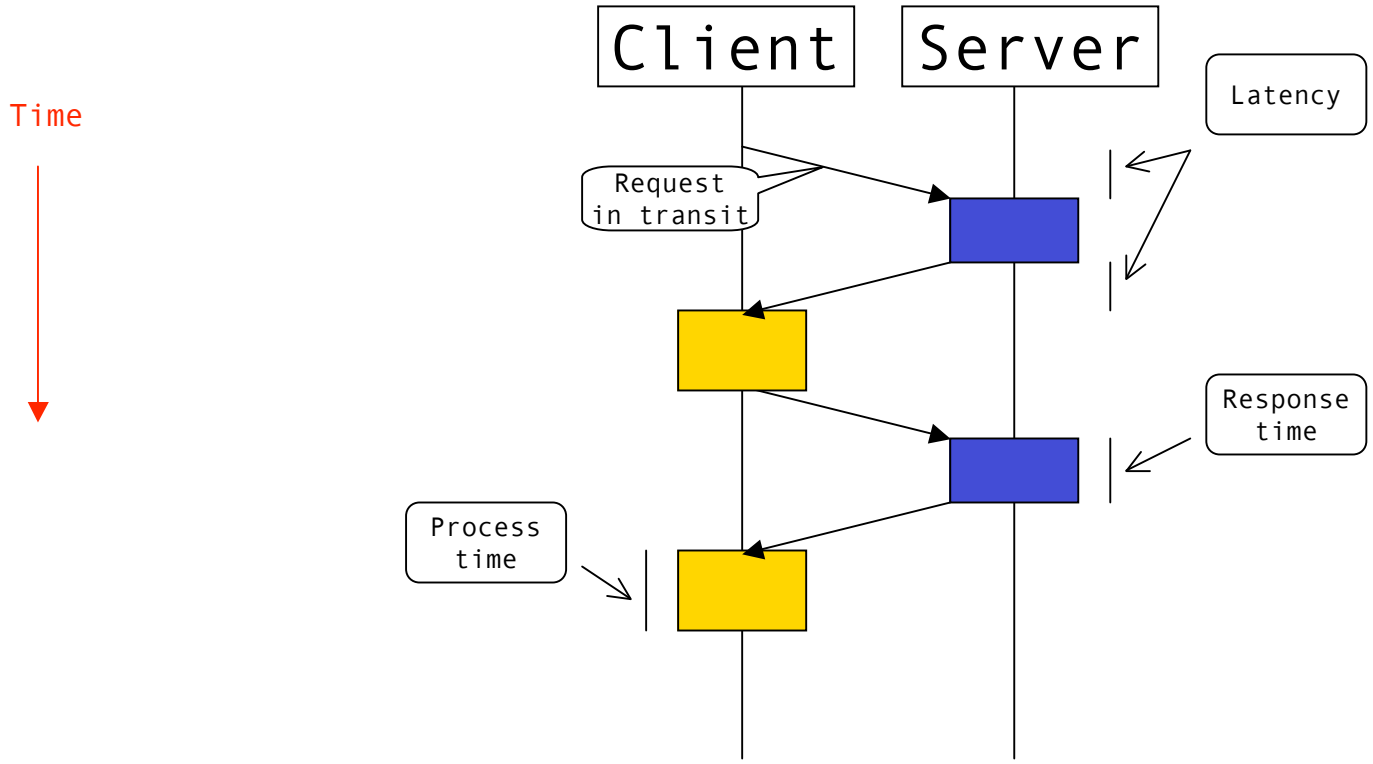
- While processing a large (remote) file the data must be transferred in small chunks.
  - It doesn't matter how many chunks you can carry if you will only read them one by one
- In ROOT each of those small chunks is usually spread out in a big file.
- The time we spend waiting for the data in transit could be a considerable part of the total time.

Inside a ROOT Tree

```
Root [0] TFile f("h1big.root");
Root [1] f.DrawMap();
```



Network Latency



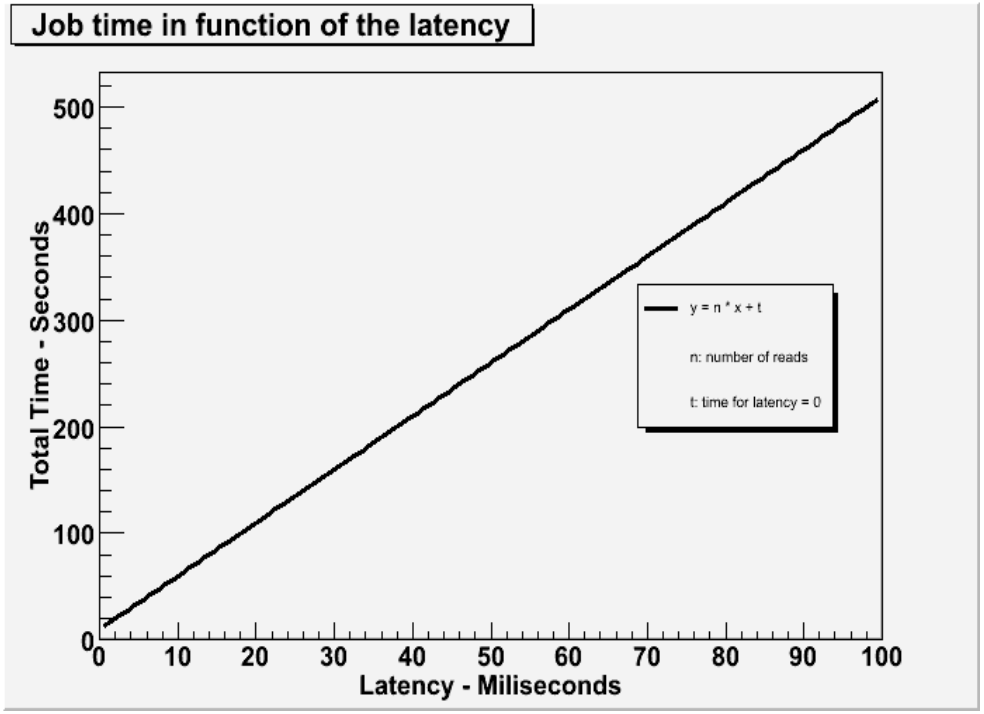
Total time = n (CPT + RT + L)

n = number of requests
CPT = process time (client)
RT = response time (server)
L = latency (round trip)

The equation depends on both variables

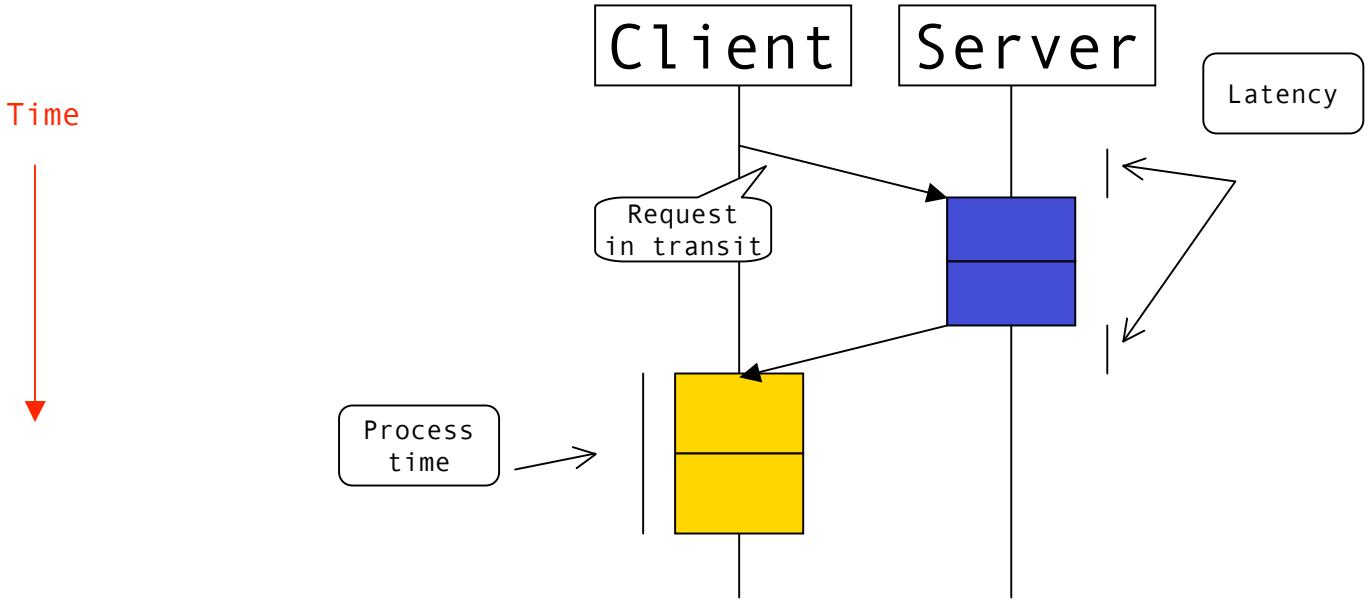
Evolution of the problem

- Time is linearly proportional to latency.
- The time needed when latency is zero is very small (Y-axis cut).
- The number of requests is the slope.



Solution

- Perform a big request instead of many small reads.



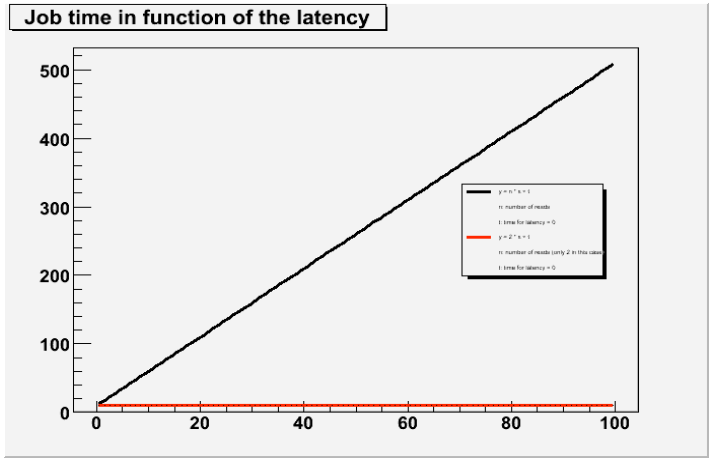
Total time = n (CPT + RT) + L

n = number of requests
CPT = process time (client)
RT = response time (server)
L = latency (round trip)

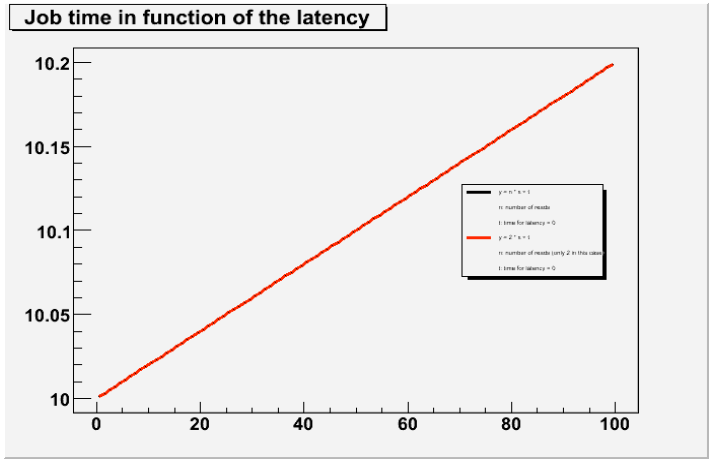
The equation does not depend on the latency anymore !!!

Performance gain

- Network latency became a constant.
  - It's almost the same as performing a local read.



- Although it's almost imperceptible, the latency is still there.

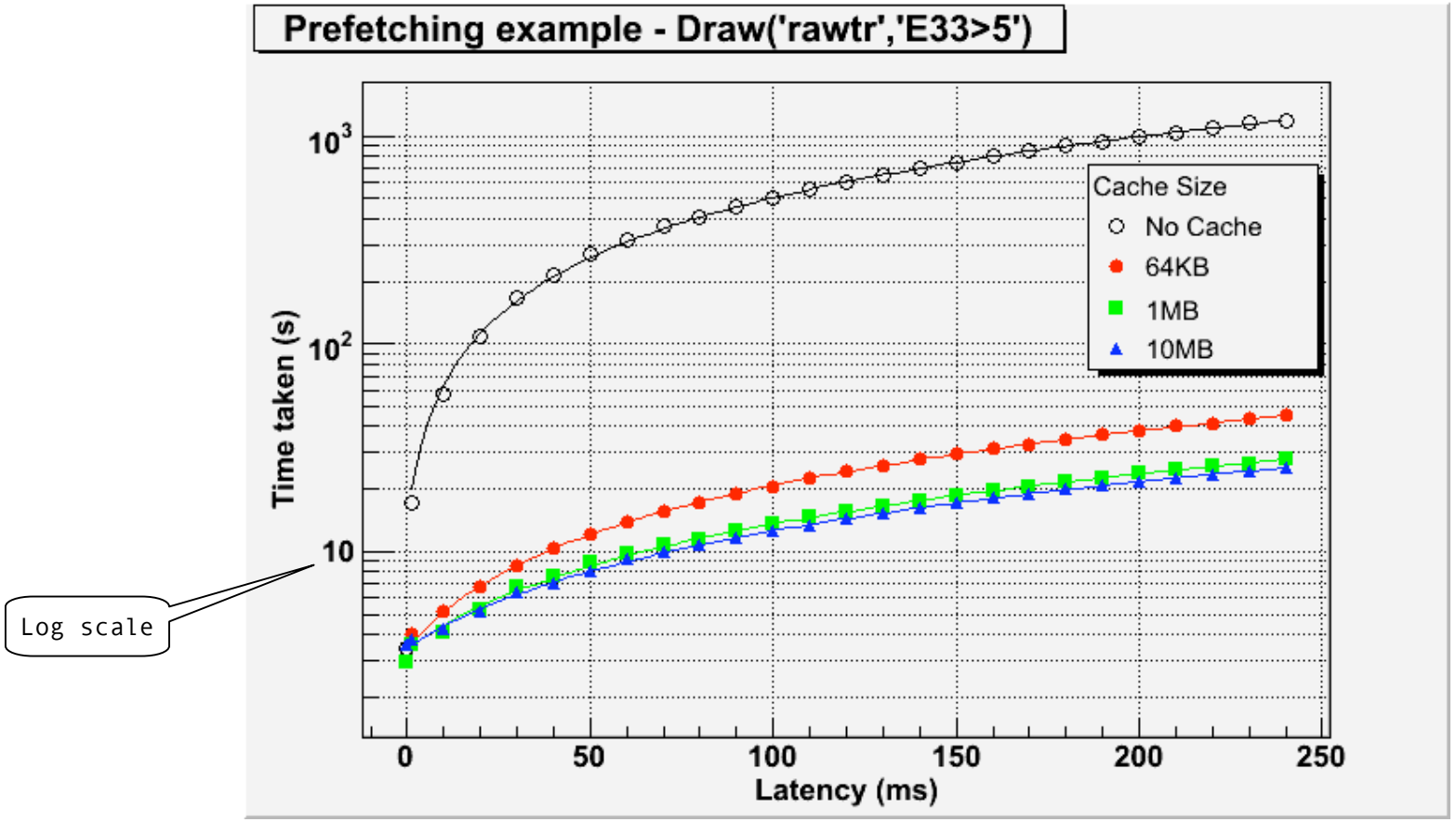


Limitation: Transfer size

- To transfer all the data in a single request is not realistic. The best we can do is to transfer blocks big enough to cause an improvement in performance.

- Let's say our small blocks are usually 2.5KB big, if we have a buffer of 256KB we will be able to perform 100 requests in a single transfer.

Simulated network latency



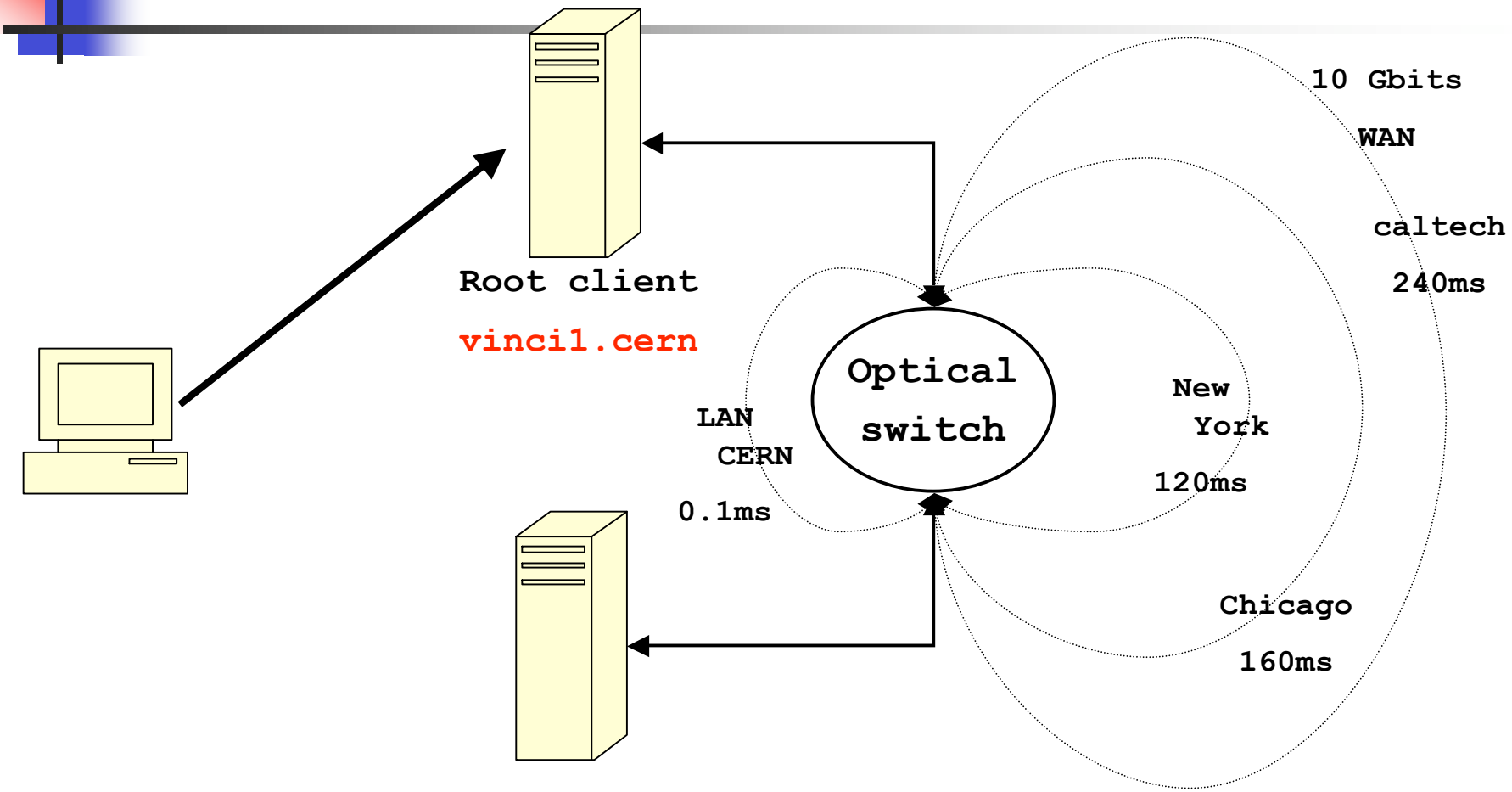
Real measurements

- The file is on a CERN machine connected to the CERN LAN at 100MB/s.
- A is on the same machine as the file (local read)
- B is on a CERN LAN connected at 100 Mbits/s and latency of 0.3 ms (P IV 3 Ghz).
- C is on a CERN Wireless network at 10 Mbits/s and latency of 2ms (Mac duo 2Ghz).
- D is in Orsay; LAN 100 Mbits/s, WAN of 1 Gbits/s and a latency of 11 ms (PIV 3 Ghz).
- E is in Amsterdam; LAN 100 Mbits/s, WAN of 10 Gbits/s and a latency of 22ms (AMD64).
- F is connected via ADSL of 8Mbits/s and a latency of 70 ms (Mac duo 2Ghz).
- G is connected via a 10Gbits/s to a CERN machine via Caltech latency 240 ms.
- The times reported in the table are realtime seconds

client	latency(ms)	cachesize=0	cachesize=64KB	cachesize=10MB
A	0.0	3.4	3.4	3.4
B	0.3	8.0	6.0	4.0
C	2.0	11.6	5.6	4.9
D	11.0	124.7	12.3	9.0
E	22.0	230.9	11.7	8.4
F	72.0	743.7	48.3	28.0
G	240.0	>1800.0	125.4	9.9

One query to a 280 MB Tree I/O = 6.6 MB

Interesting case: Client G

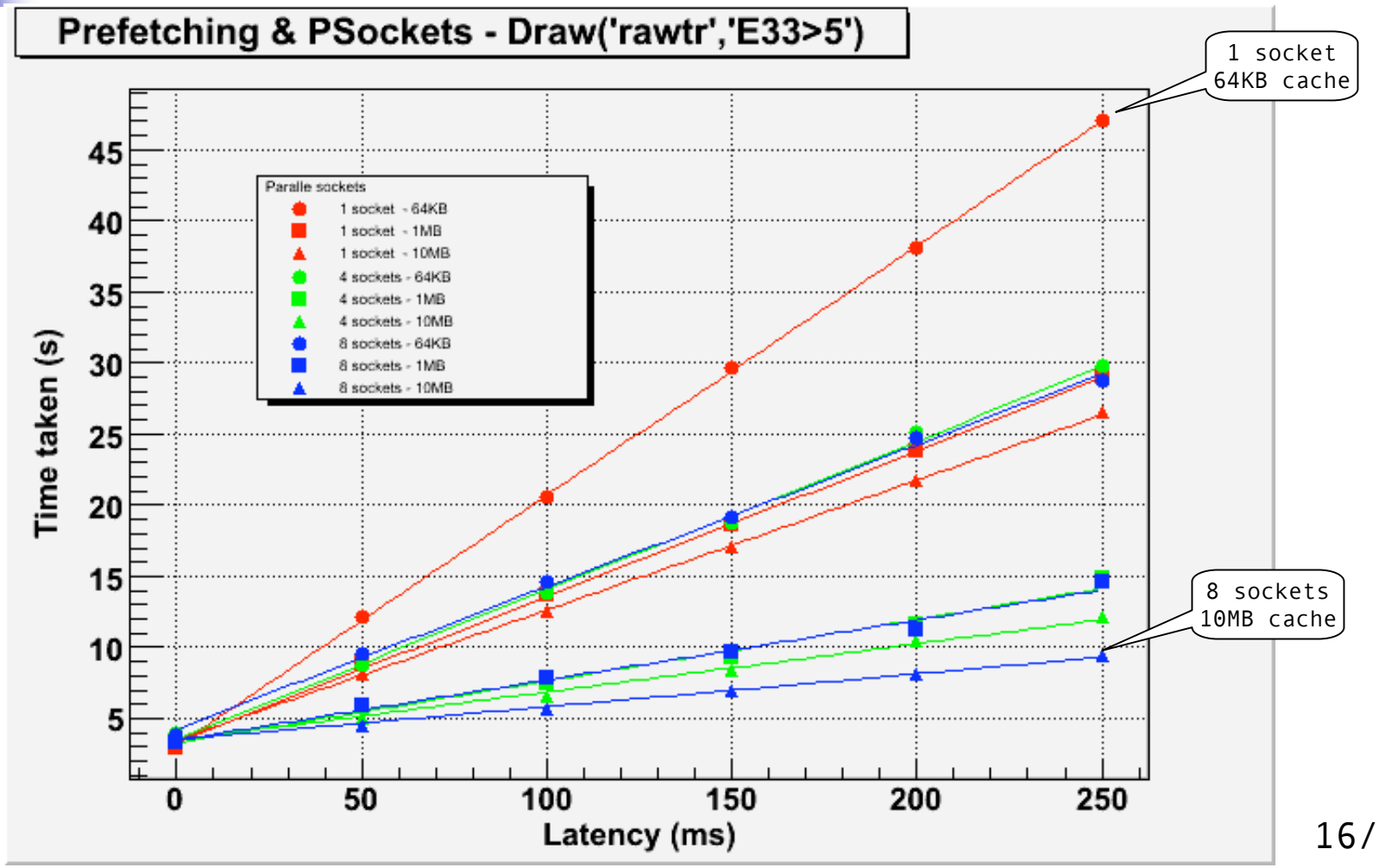


Many thanks to Iosif Legrand who provided the test bed for this client.

Client G: Considerations

- The test used the 10 GBits/s line CERN->Caltech->CERN with TCP/IP Jumbo frames (9KB) and a TCP/IP window size of 10 Mbytes.
- The TTreeCache size was set to 10 Mbytes. So in principle only one buffer/message was required to transport the 6.6 Mbytes used by the query.
- However the TTreeCache learning phase (set to 10 events) had to exchange 10*3 messages (one per branch) to process the first 10 events, ie 10*3*240ms=7.2 seconds
- As a result more time was spent in the first 10 events than in the remaining 283000 events !!
- Further work to do to optimize the learning phase. In this example, we could process the query in 2.7 seconds instead of 9.9
- We must also reduce the number of messages when opening a file (in theory, only one should be needed).

Maxing it out: Parallel Sockets



Maxing it out: Parallel Sockets

- It is a good option for fat pipes (long delay/bandwidth).
- Performance increases with the size of the cache but a big cache may not be worthwhile for small queries.
- Available in different transfer servers like rootd and xrootd.

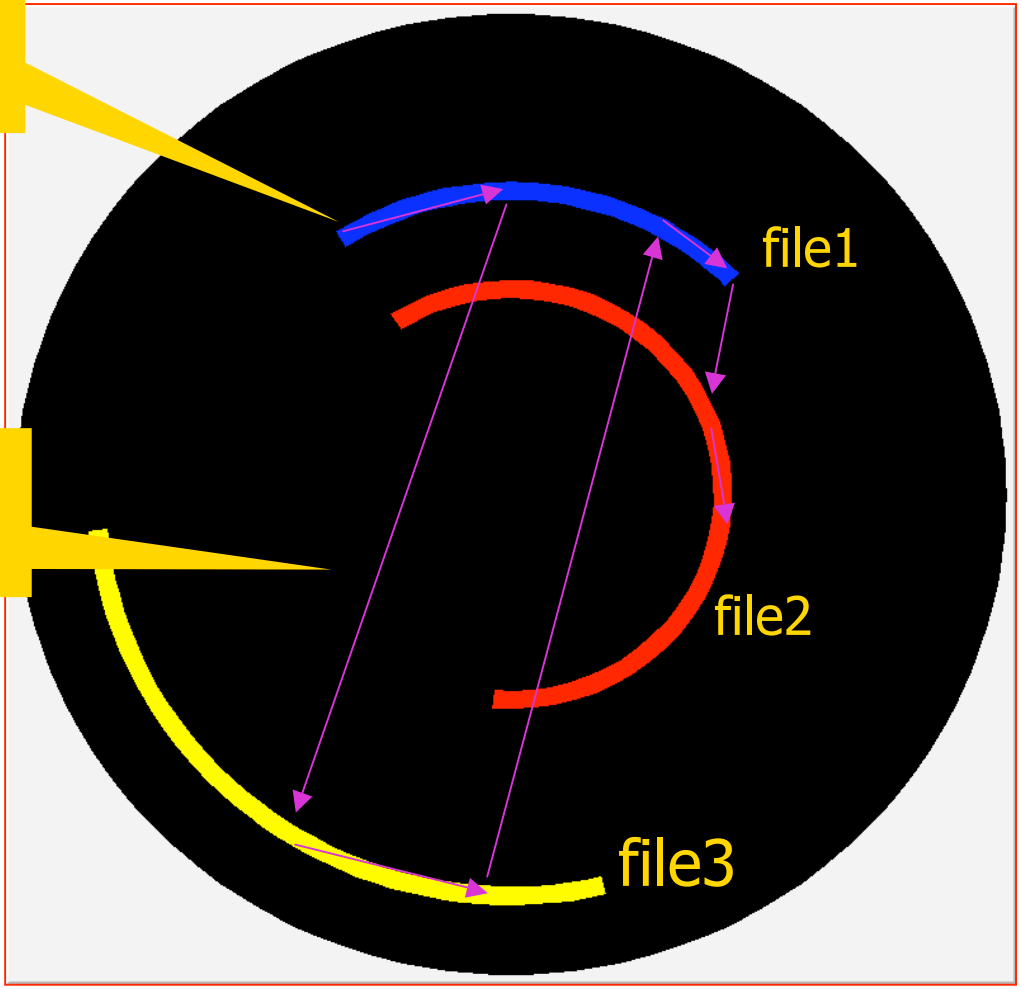
What about disk latency?

- Network latency is only one component in the delay of a request.
- Reading small blocks from disk in the server might be inefficient.
- Seeking randomly on disk is bad. Better to read sequentially if you can.
- Multiple concurrent users reading from the same disk generate a lot of seeks (wasteful!)
- These considerations are less important in a batch environment, but absolutely vital for interactive applications.

Disk latency

Disk latency is small when seeking in a file being read sequentially

But time to seek between 2 files may be large (> 5ms)



Conclusions

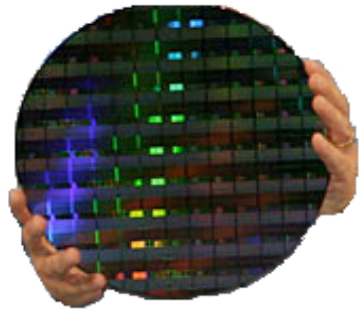
- Data analysis can be performed efficiently from remote locations (even with limited bandwidth).
- The modifications have been done in rootd, xrootd(thanks to Fabrizio and Andy) and http (by Fons). But it's not difficult to port to other servers.
- In a addition to network latency, disk latency is also improved in certain conditions (especially if this kind of read could be done atomically)

Conclusions

- Still some work to do to fine tune the learning phase and opening files.
- Possible new ideas like asynchronous readings (available on xrootd) or reading-ahead could improve the performance even further.
- Parallel unzipping? …

Check out the second part of the presentation :)

Parallel Unzipping



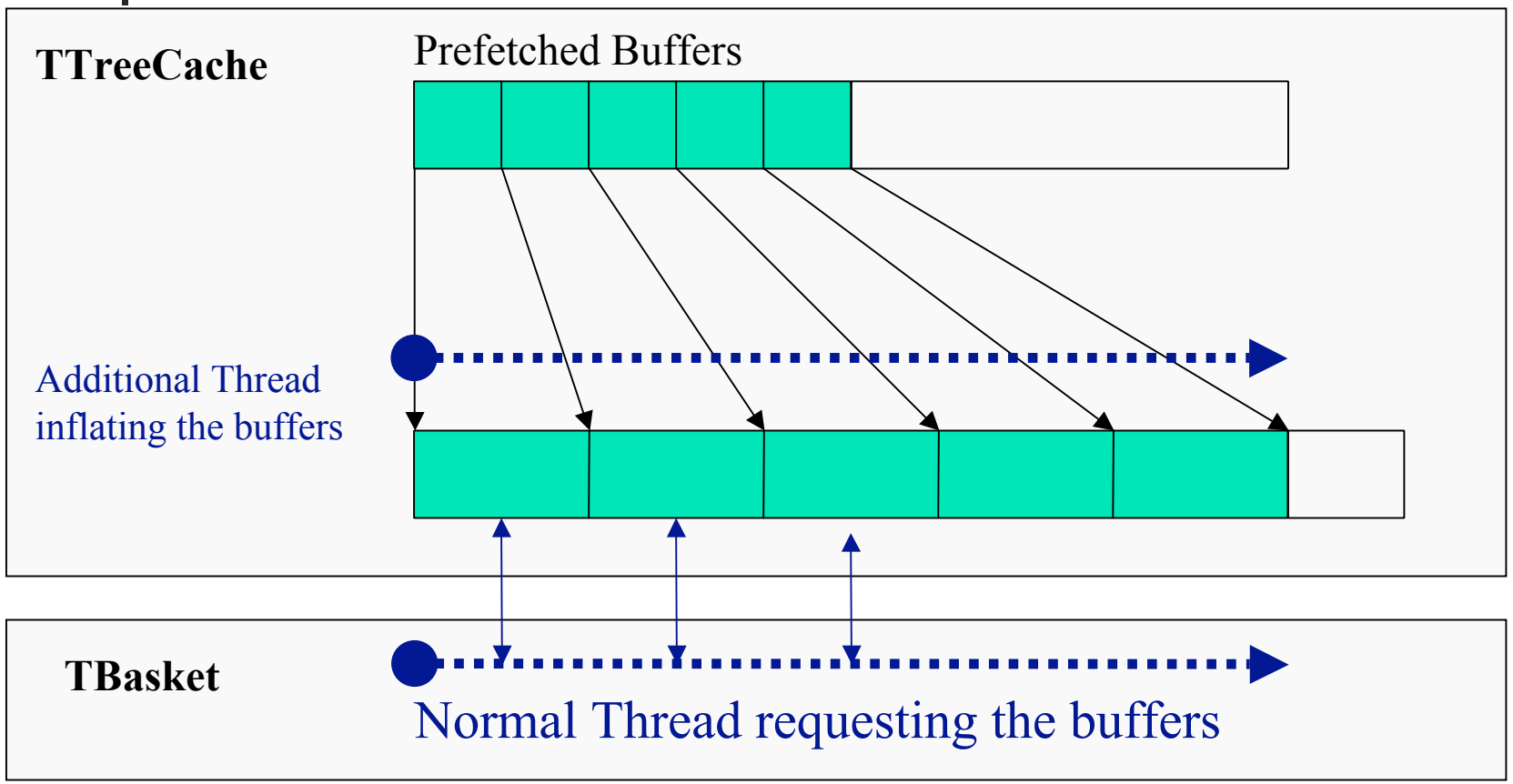
Thinking about those 80 core machines…

Parallel Unzipping

- Since most of the ROOT objects are compressed, why couldn't we inflate them in advance by using an additional core?

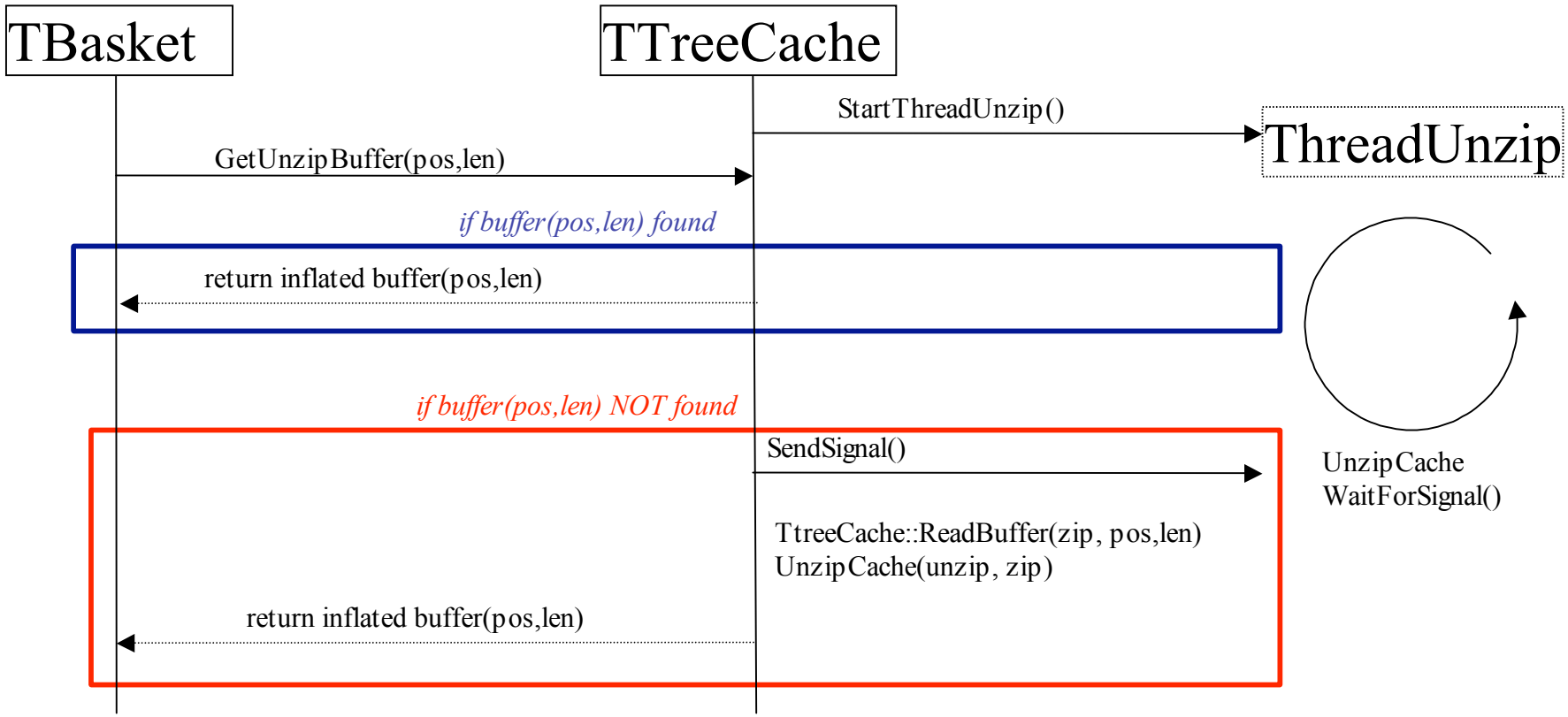
- This is only feasible if we have at hand the future buffers to be inflated, which has become possible since the introduction of the pre-fetching and caching mechanisms.

Description



As we can see, the additional thread has to be ”faster”
or the normal thread won't find the buffers in the cache

Calling sequence

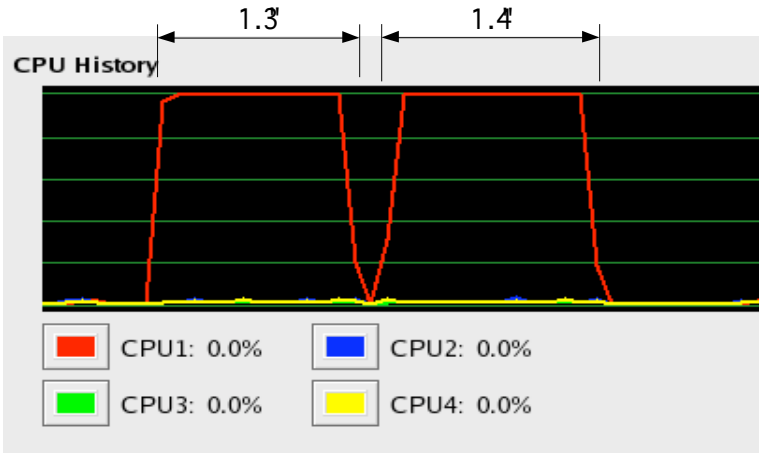


Implementation Ideas

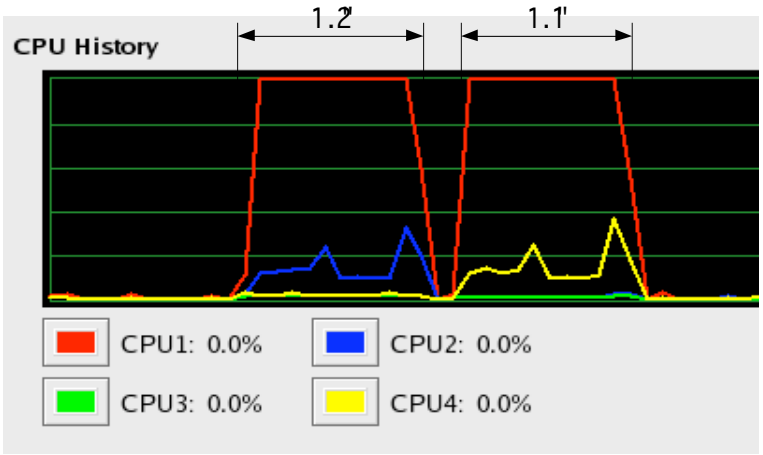
- An immediate approach is to try to start the unzipping before the actual end of the reading.
- This presents several synchronization problems.
- The idea can be generalized to a bounded-buffer (or circular producer-consumer buffer) problem, which should reduce the number of cache misses.
- Another good point is being able to use the data directly from the cache (reducing memory consumption).

Load balance

- Two sequential runs of the test.
- We can easily see that only one processor is used at any point of the analysis

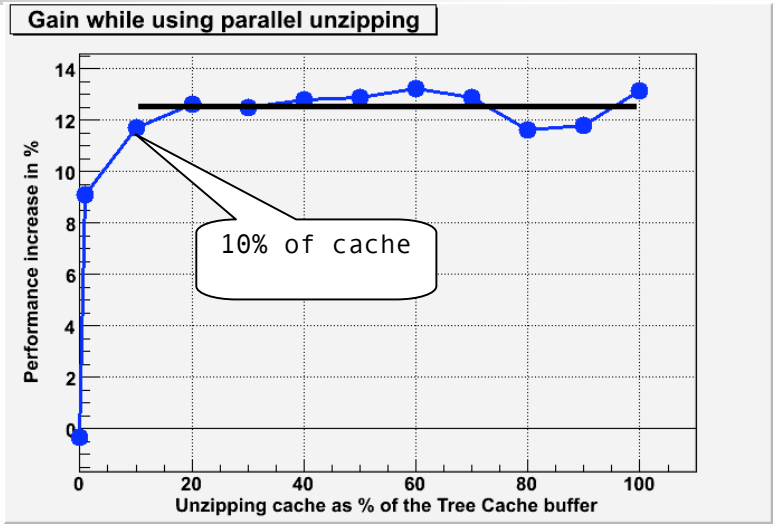


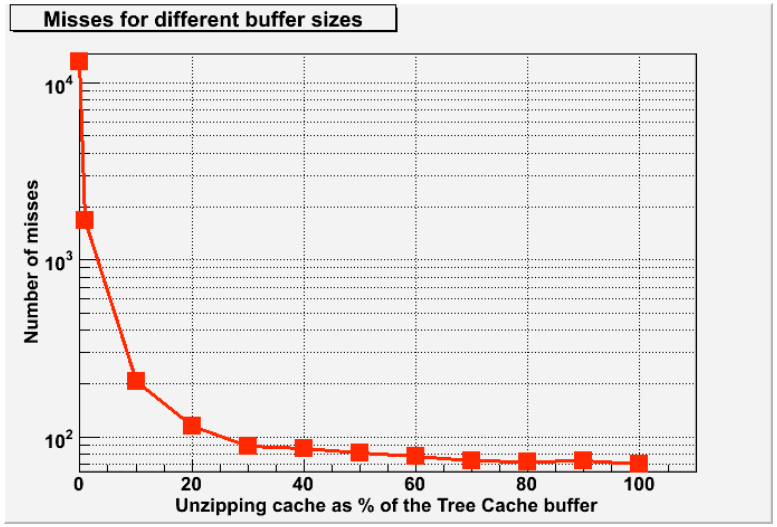
- These are the same two runs of the test when we activate the parallel unzipping.
- And the small peaks represent the second thread trying to inflate the future buffers.



How far ahead should we unzip?

- As we see here the unzipping cache doesn't have to be very big.
- In fact, a ”pre-unzipping” of only the future 1% of the cache gives us a performance improvement close to 9%.
- We see that after 10% in size, the gain is relatively constant so we have decided to leave the unzip cache size with this value by the default.





Conclusions

- Multi-core programming is trendy ;)
- It's the only way to speed up programs when the CPU speed remains constant.
- Parallel unzipping gives us the advantage of reading compressed data without any perceptible lost in performance.
- Lastly, this is an area with high potential where we can explore to see what other things can be done in parallel.