Latencies and data access.

Boosting the performance of distributed applications.

Fabrizio Furano
INFN – Istituto Nazionale di Fisica Nucleare
furano@pd.infn.it

Co-Authors:

Andrew Hanushevsky
SLAC – Stanford Linear Accelerator Center
abh@slac.stanford.edu

Peter Elmer
CERN
peter.elmer@cern.ch

Gerardo Ganis
CERN
gerardo.ganis@cern.ch

Motivation

- The typical way in which HEP data is processed is (or can be) often known in advance
- We want to better understand:
  - Which information can be useful about the data access patterns
    - Towards a single file
    - Towards a set of files accessed by an analysis job
  - Where are the major reasons which lower the data access performance
- We want to exploit this knowledge to enhance the data access performance

A simple job

- A simplification of the problem
  - A single-source job
  - Processes data “sequentially”
    - i.e, the sequentiality is in the basic idea, but not necessarily in the produced data requests
  - e.g. scanning a ROOT persistent data structure

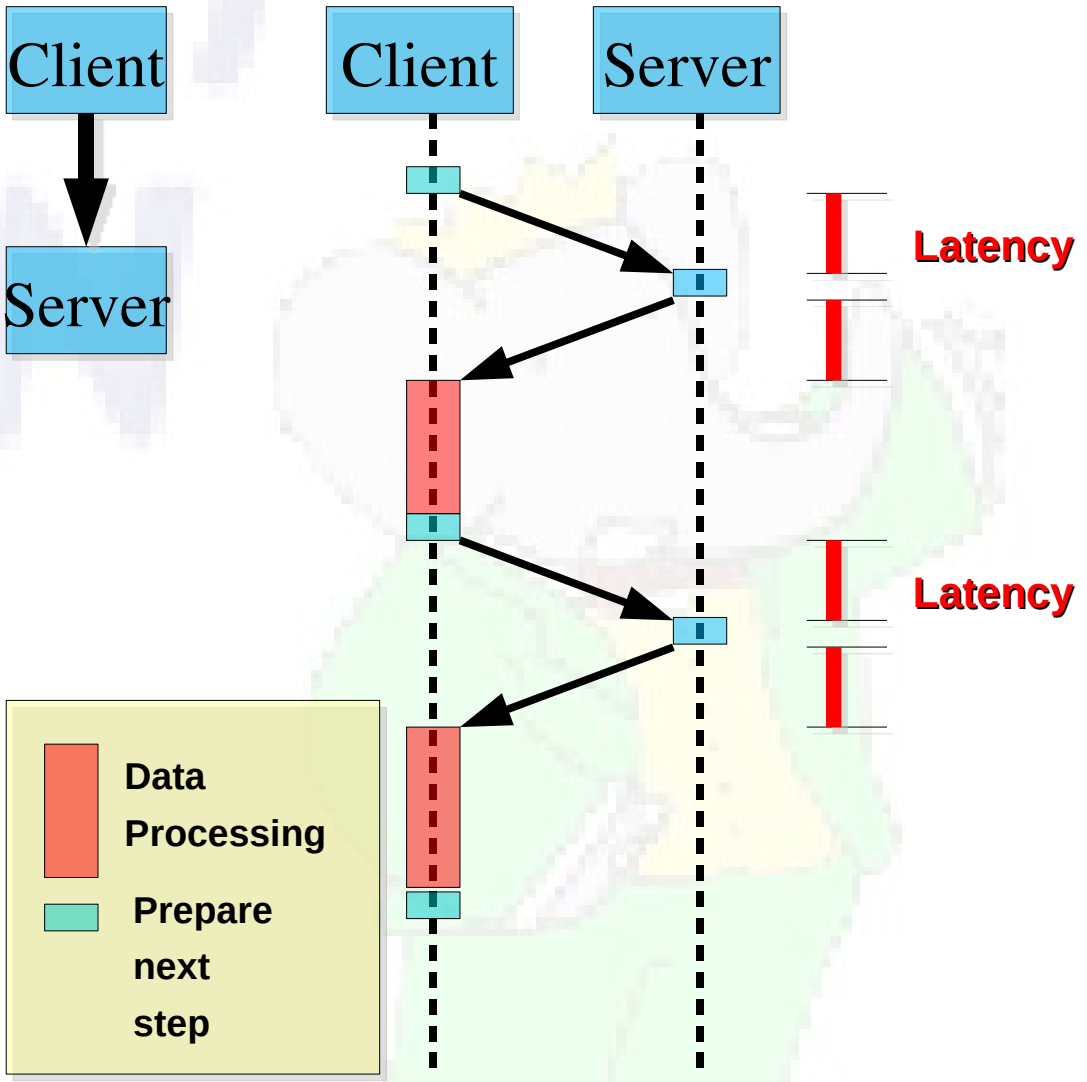
```
while not finished
  d = get_next_data_piece()
  process_data(d)
end
```

Simplified schema

We can make some simple hypotheses.

Hypothesis: the data are not local to the client machine
Hypothesis: there is a server which provides data chunks

1 client
1 server
A job consisting of several interactions

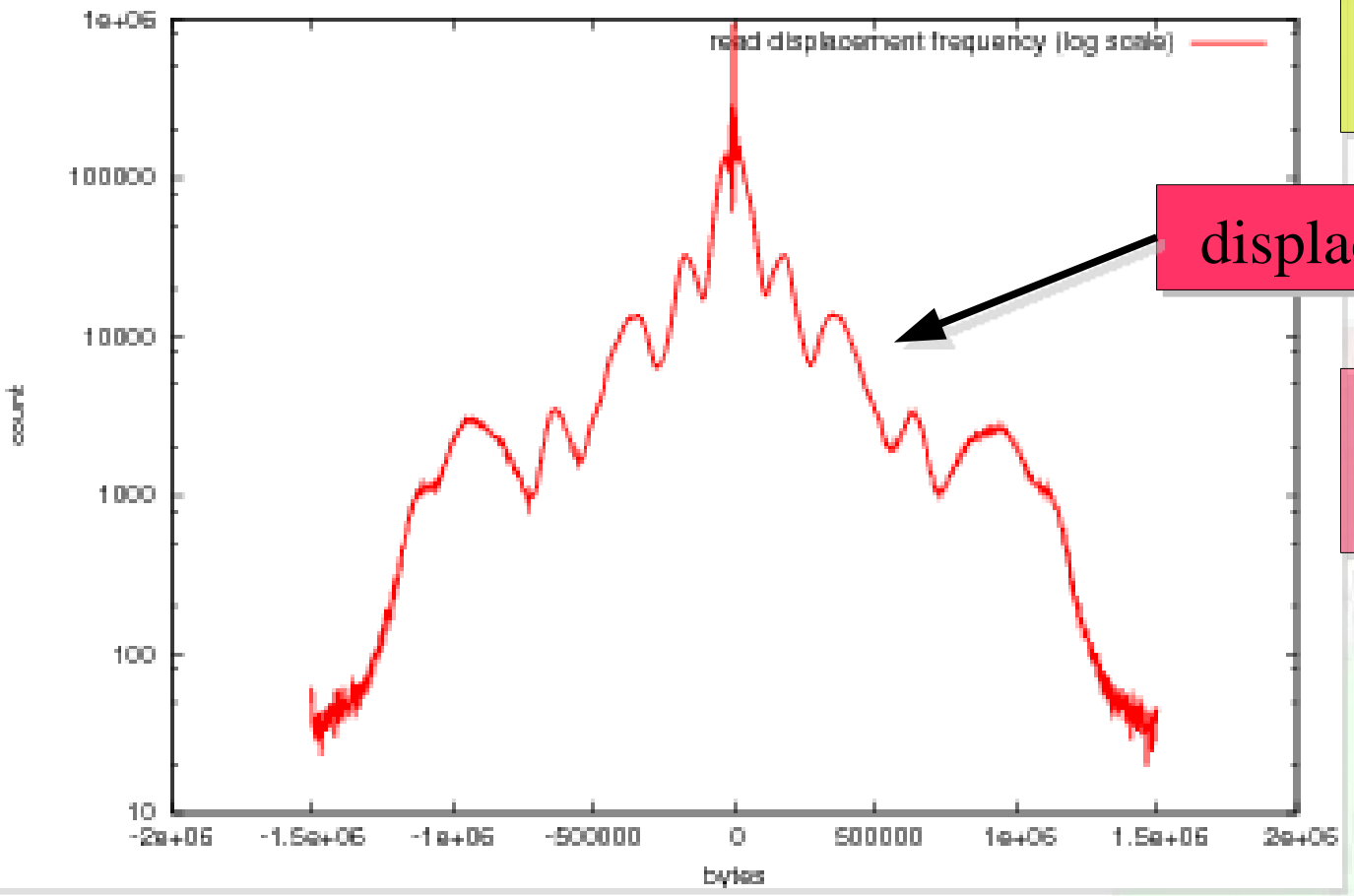


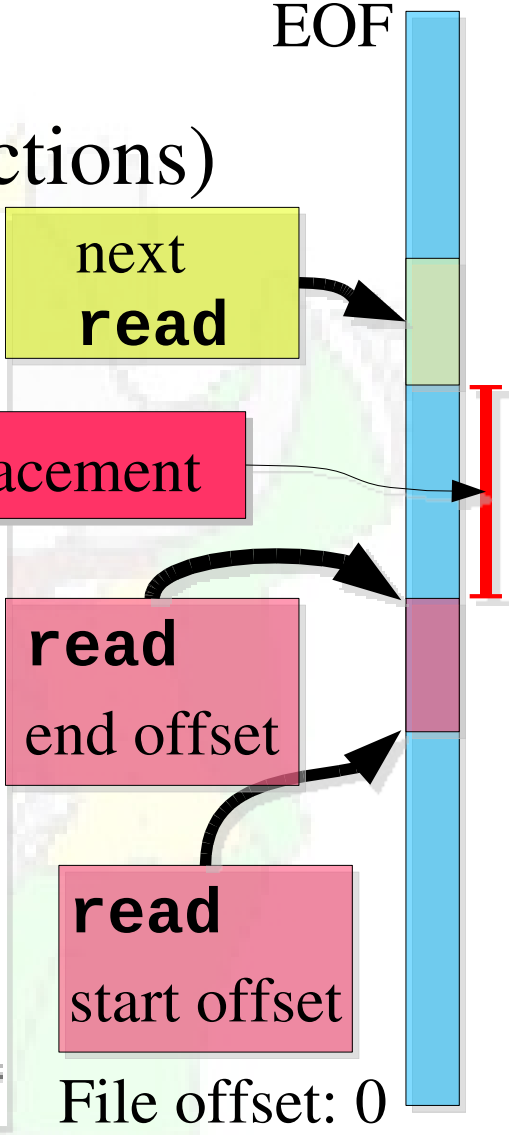
Latency

- The transmission latency (TCP) may vary
  - network load
  - geographical distance (WAN)
    - e.g. Min. latency PD-SLAC is around 80ms (about ½ of the *ping* time)
- e.g, if our job consists of 1E6 transactions
  - 160ms * 1E6 makes our job last 160.000 seconds more than the ideal "no latency" one
  - 160.000 seconds (2 days!) of wasted time
- Can we do something?

Example: a BaBar analysis

- 100 jobs taken randomly from 5000
- Study of the traced load (11e6 interactions)



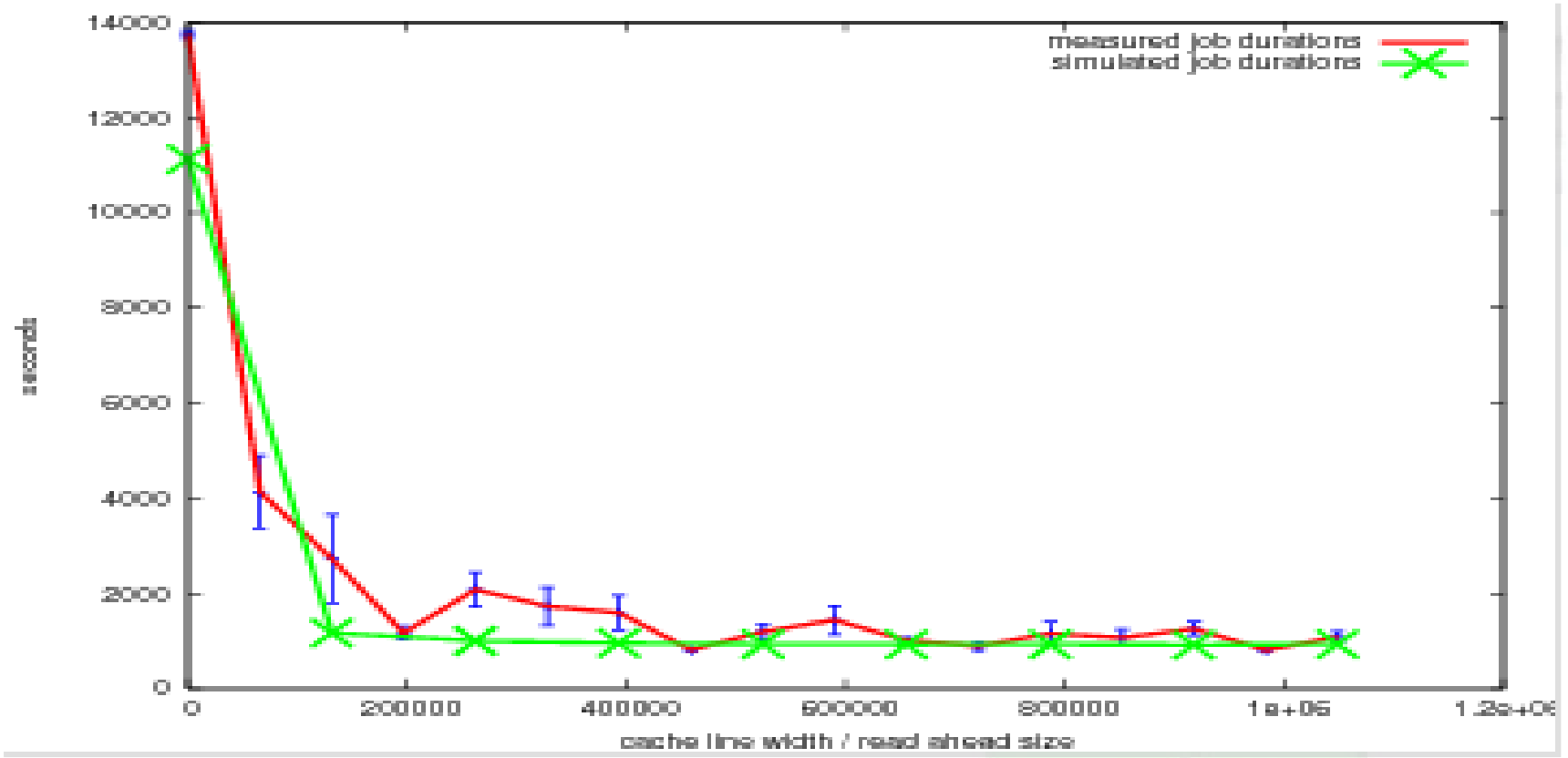


Bigger blocks: a cache ?

- A reasonable client side cache can lower the number of interactions
- The single data transfers will be larger
  - But will carry the data needed for many others
- A cache can be built with performance in mind
    - No double `memcpy` of the available data chunks
- A cache can also be pre-populated with data
    - Automatically: the client “speculates” about the near future read activity
    - App based: the app informs the communication library about its near future intentions

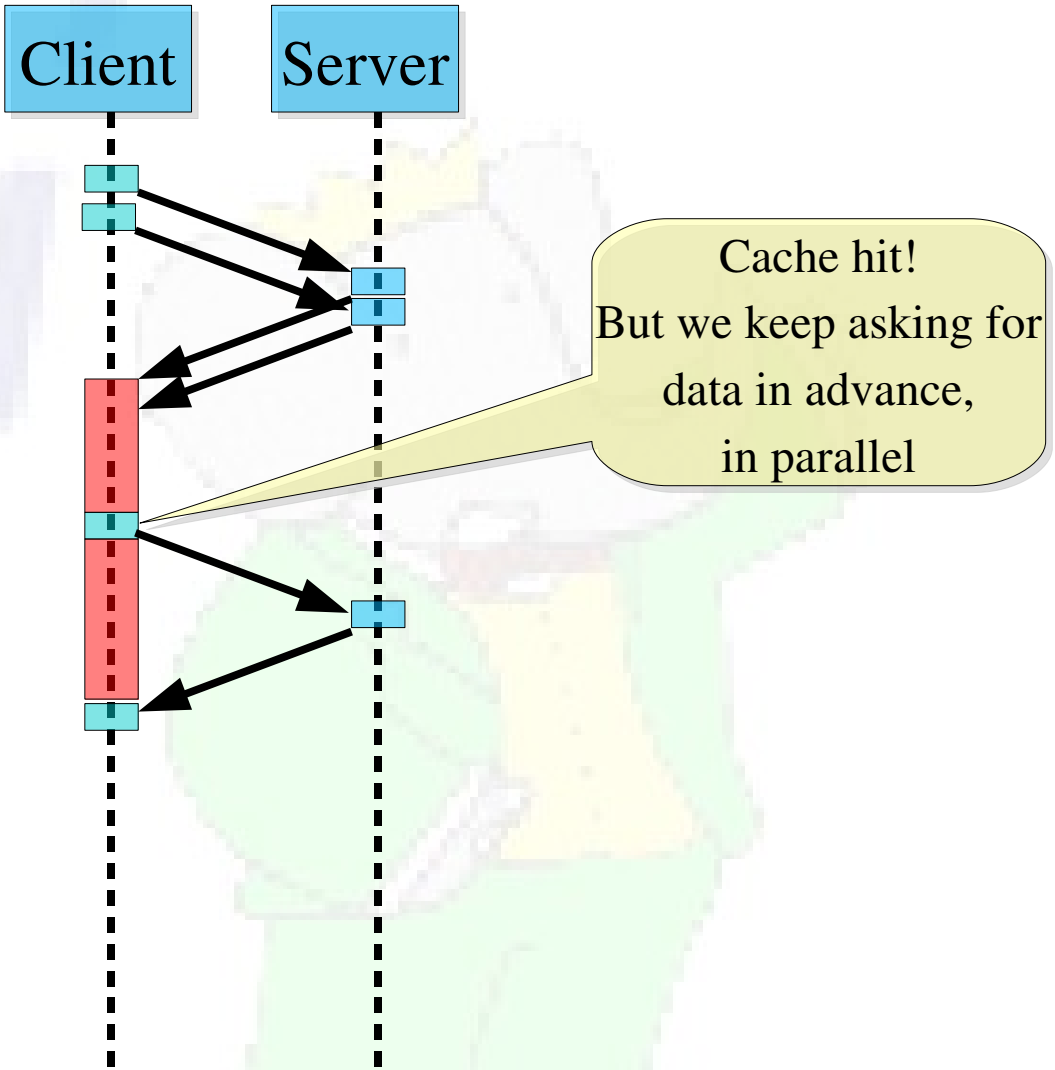
Data xfer: a verification

- Job duration vs. cache block size (0=no cache)
- Client in Padova, server at SLAC



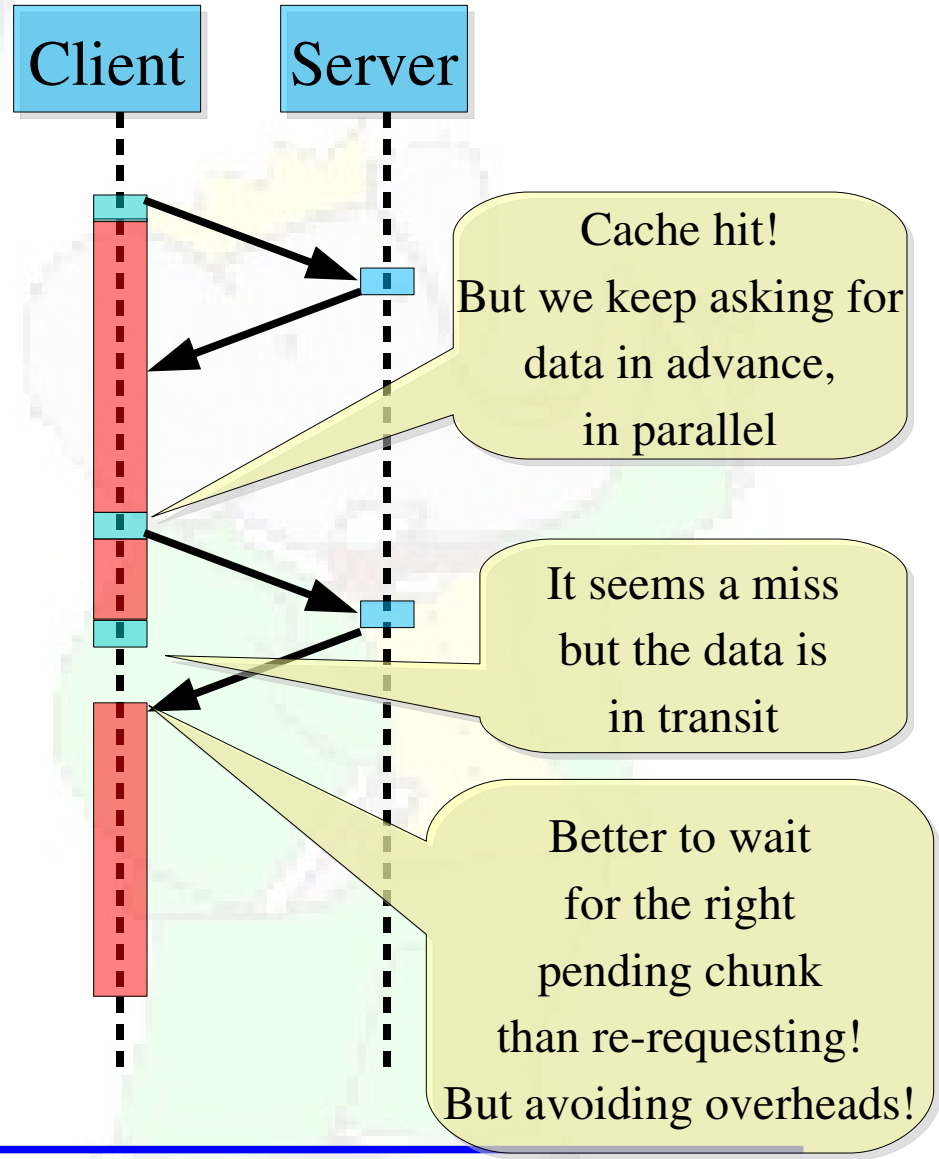
Prefetching

- Read ahead
  - the communication library asks for data in advance, sequentially
- Informed prefetching
  - the **application** informs the comm library about its near-future data requests

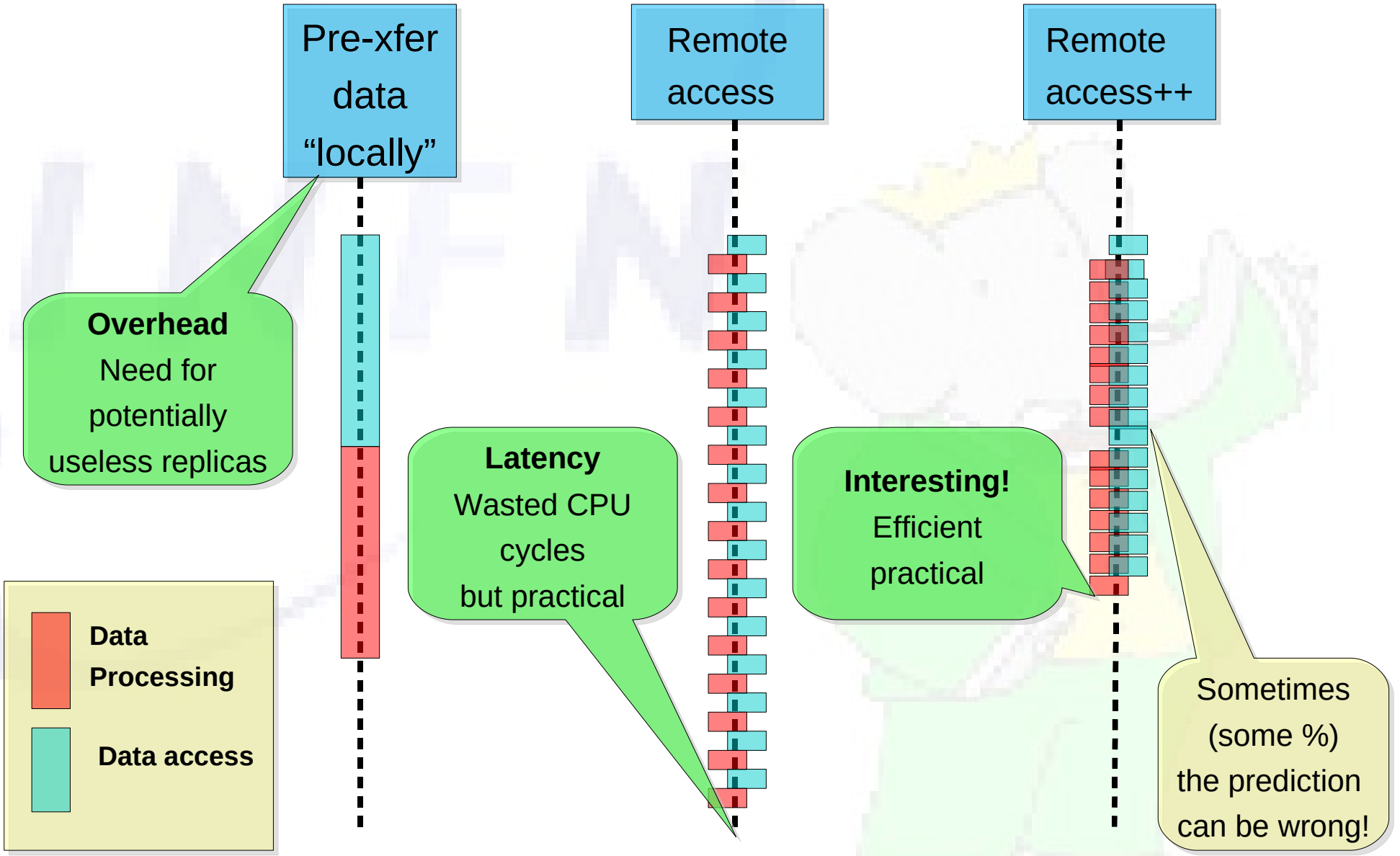


More than a cache

- Trouble: data could be requested several times
- The caching has to keep track of pending xfers
- Originally named "Cache placeholders" (Patterson)
  - But there are better ways...

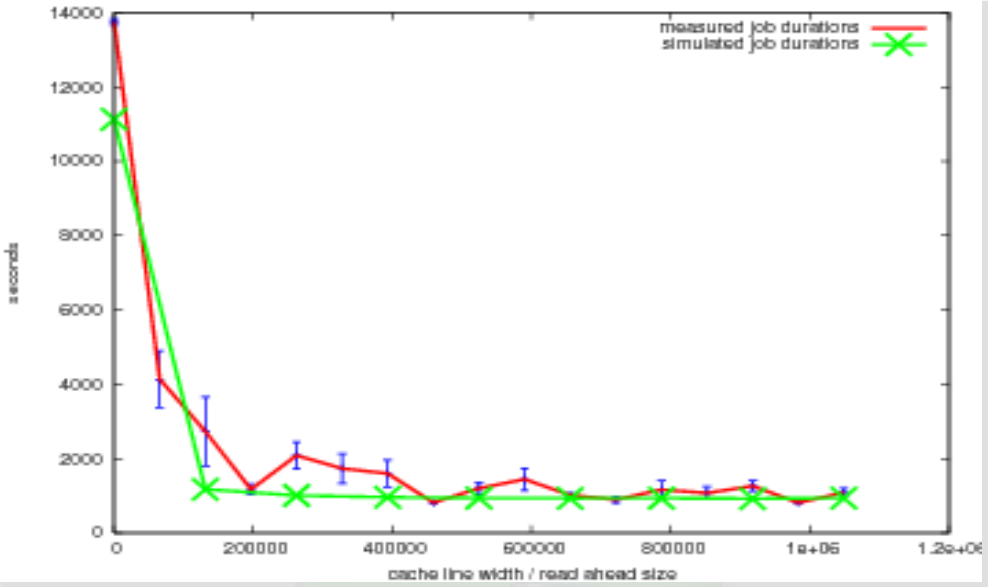


Intuitive scenarios



Question

- Nice words but...
- Can we effectively get an advantage in the real world?

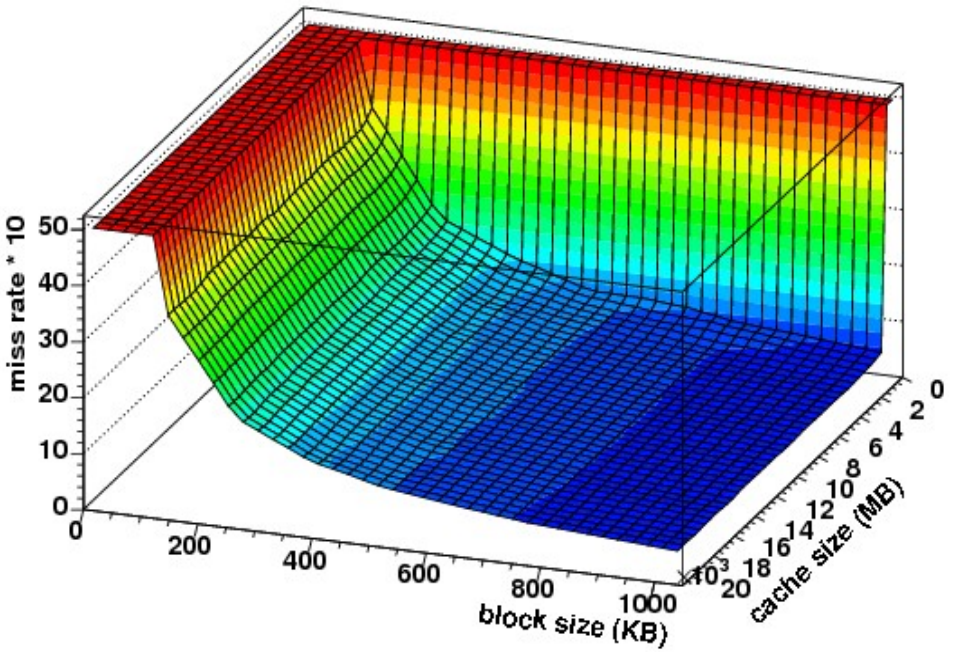


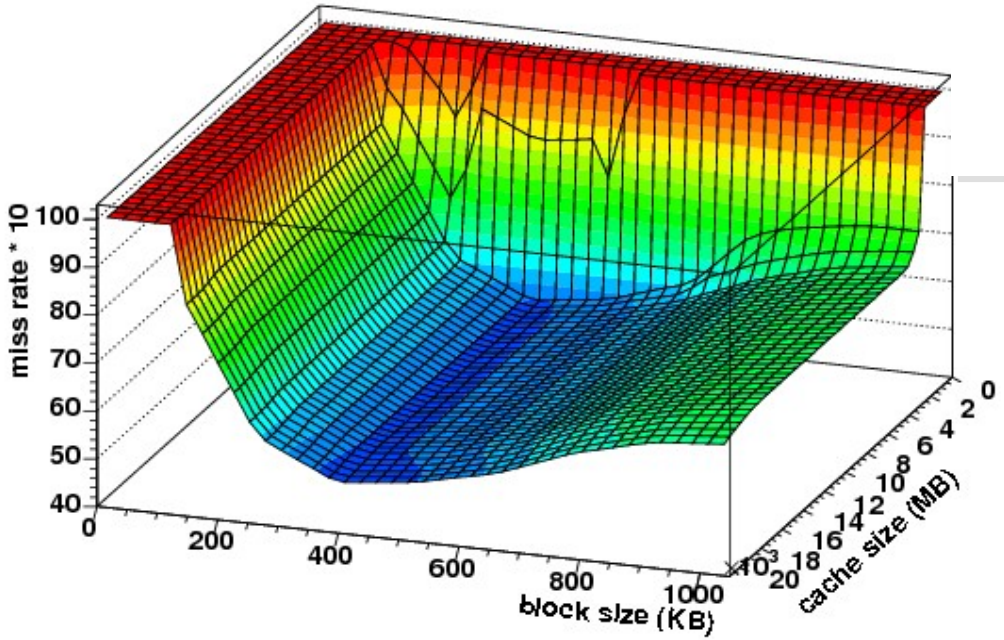
A simulation

- Input: a BaBar analysis data access trace
  - Reasonable intervals for:
    - network latency
    - cache block size
    - cache size
    - read ahead/cache block size
  - A simple xrootd-based data serving cluster
- Output:
  - miss rates
  - final analysis duration (compr. the processing time)

Miss rates

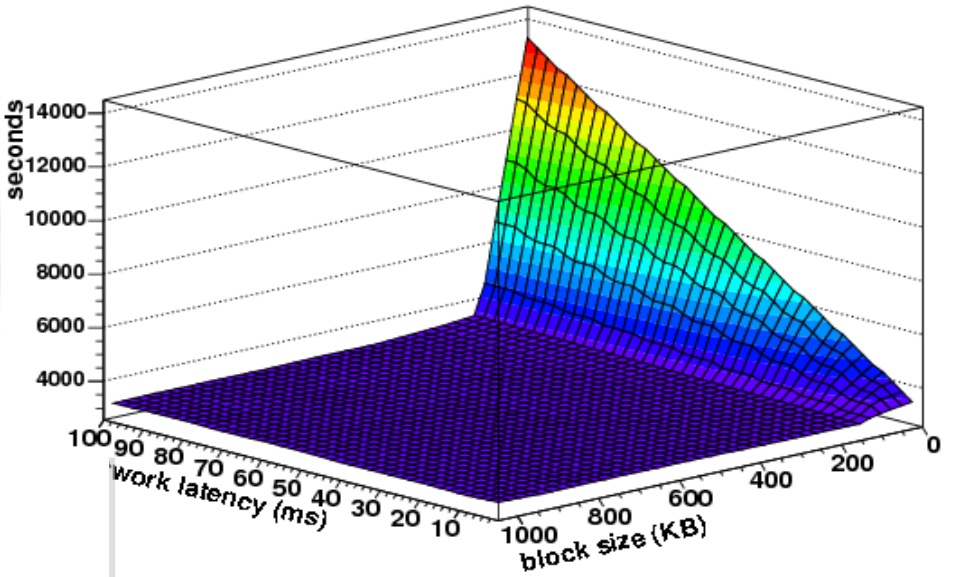
With async read ahead

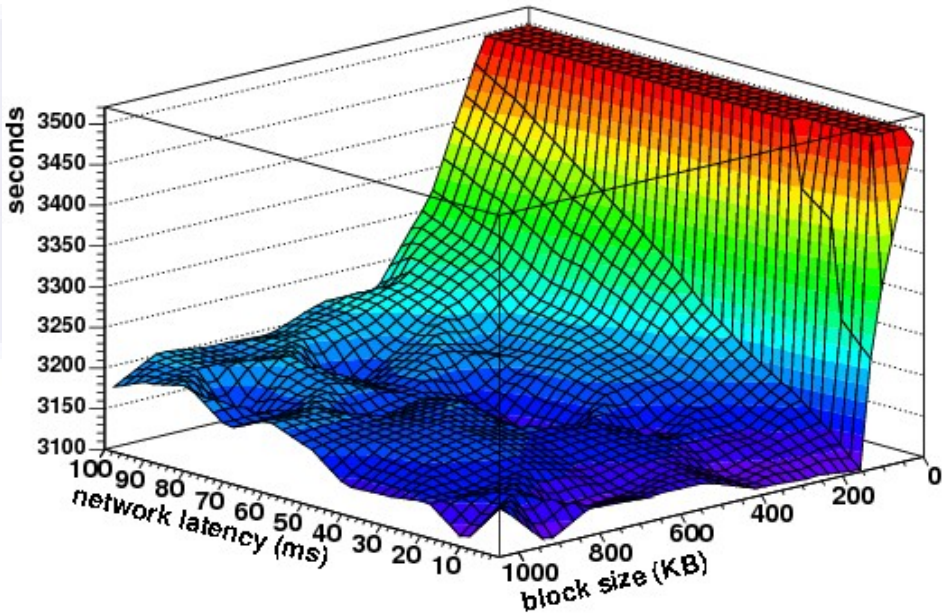




Without async read ahead

Job duration





Resource access: open

- An application could need to open many files before starting the computation
- We cannot rely on an optimized struct of the application, hence we assume:

```
for (i = 0; i < 1000; i++)
   open file i

Process_data()

Cleanup and exit
```

Problem: a mean of only 5 seconds for locating/opening a file will take 5000 seconds for the app to start processing

Parallel open

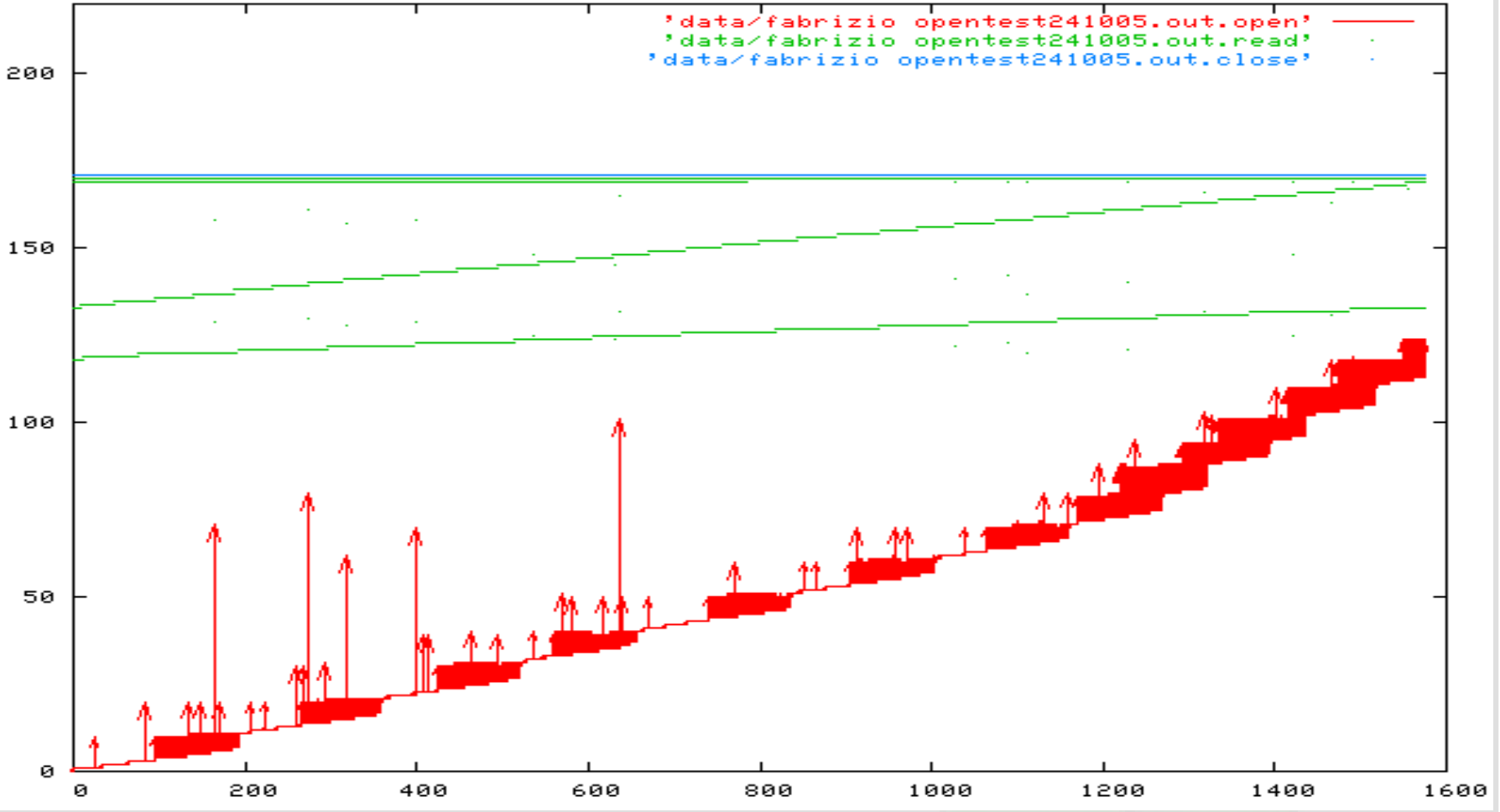
- Solution (XrdClient):
  - An open() call returns immediately
    - The request is treated as pending
    - A pool of threads takes care of the pending opens
    - The opener is put to wait ONLY when it tries to access the data
- Solution (ROOT 5.?):
  - The ROOT primitives (e.g. Map()) semantically rely on the completeness of an open() call
  - The parallel open has to be requested through

```
TFile::AsyncOpenRequest(filename)
```

Parallel open results

Test performed at SLAC towards the “Kanga cluster”

1576 sequential open requests are parallelized transparently



Conclusion

- Implementation platform: **xrootd**
  - No-copy caching is in XrdClient and ROOT 5
    - Together with read ahead and “informed prefetching”
  - Parallel Open is there also
  - Cache placeholders (optimization) are almost there
  - Multistream transfers are on the way
  - Purpose (other than raising performance):
    - To be able to give alternative choices for (temporary) data or replica placement
    - To move towards computing models where some phases can rely on remote data
      - Ev. for backup or fault tolerance purposes (if the nearest location breaks, fallback to another one with little or no overhead)