Optimizing High-Latency I/O in CMSSW

Brian Bockelman

CHEP 2013

Disclaimer!

- This presentation assumes you know the basics of the ROOT I/O implementation.
  - We'll cover some of the basics as a refresher.
  - Not a ROOT expert? Instead of reading your email, try this link: http://iopscience.iop.org/1742-6596/331/4/042005

Why high latency?

- We are interested in improving the experience on high latency networks because:
  - We can provide end-users immediate interactive file access via high-latency networks ("the Internet").
  - This allows us to explore and adopt use cases which break data locality.
- We take "high latency" to mean greater than 50 millisecond ping time.

High-Latency is the future!

- We have seen increased interest in data federations within the WLCG.
- I thoroughly believe that this model is appropriate for HEP outside LHC.
  - It is important to identify approaches we can feed back into ROOT.
- If we continue to target smaller computing resources, departmental clusters, and individual laptops, the network will only get worse!

Test Methodology

- Performance tests shown in this presentation compare reading data locally (Nebraska to Nebraska) versus remotely (Nebraska to CERN lxplus).
  - Local reads: 25.2MB/s; ping time 0.3ms.
  - Remote read: 18.7MB/s; ping time 137ms.
- Performance tests were done by timing a simple job using our experiment framework, CMSSW. We read 1,000 events; branches read depend on two parameters:
  - Read amount: percent of branches (by volume) to read for every event. **Set to 30%**. Each event gets the same set of branches.
  - Trigger prescale: frequency at which module should read out all branches. **Set to 50**.
  - Done with *IOExerciser* module (present in recent CMSSW releases).
- Results are normalized by time it takes a job to run locally with all optimizations.

I/O Time

We use the following model*:

sum(latency + (# bytes read) / [(bandwidth) * (parallel streams)])

sum is taken over all read requests

- Minimize the time! Techniques we've used:

A. Reduce number of reads: Aggregate together multiple reads into a single network request.

B. Reduce data read: Less data = less transfer time. [This is a surprisingly small effect - we tend to be latency-bound.]

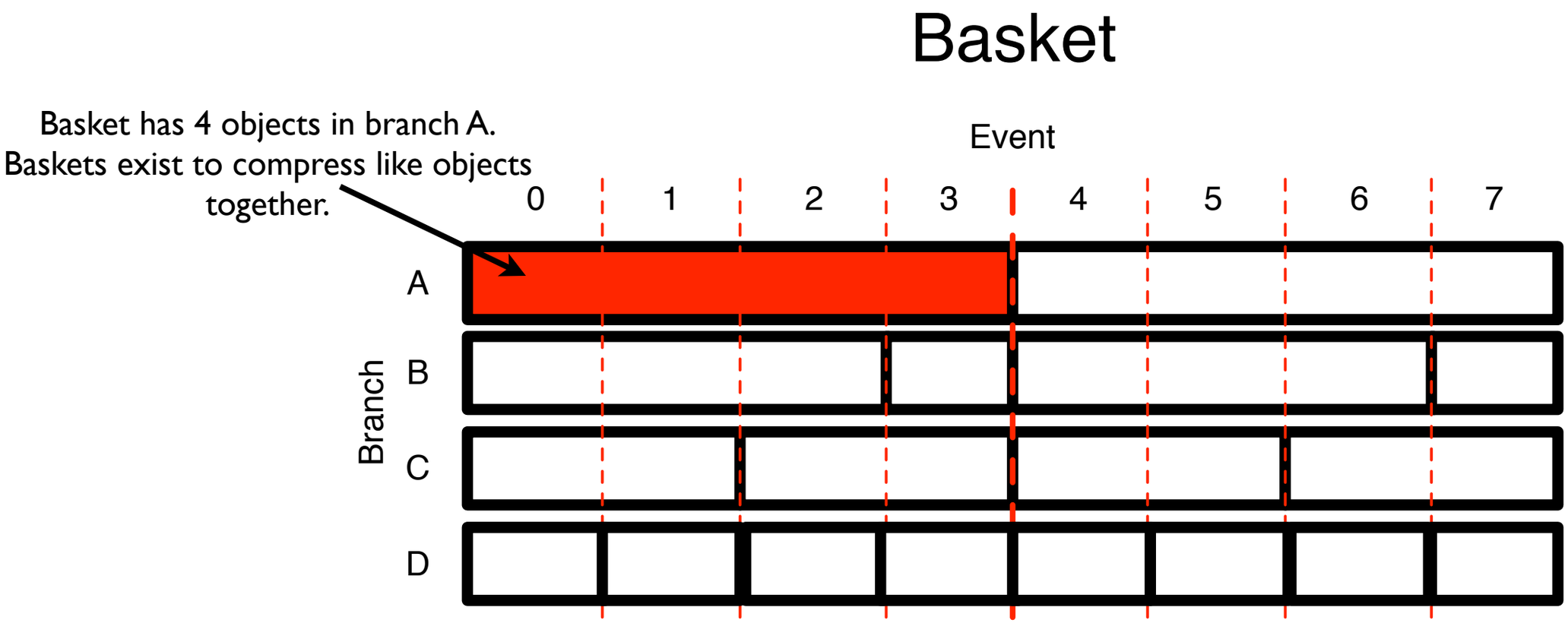
- We discuss this only in the paper, not here.

C. Increase parallelism: Utilize multiple TCP streams for data transfer.

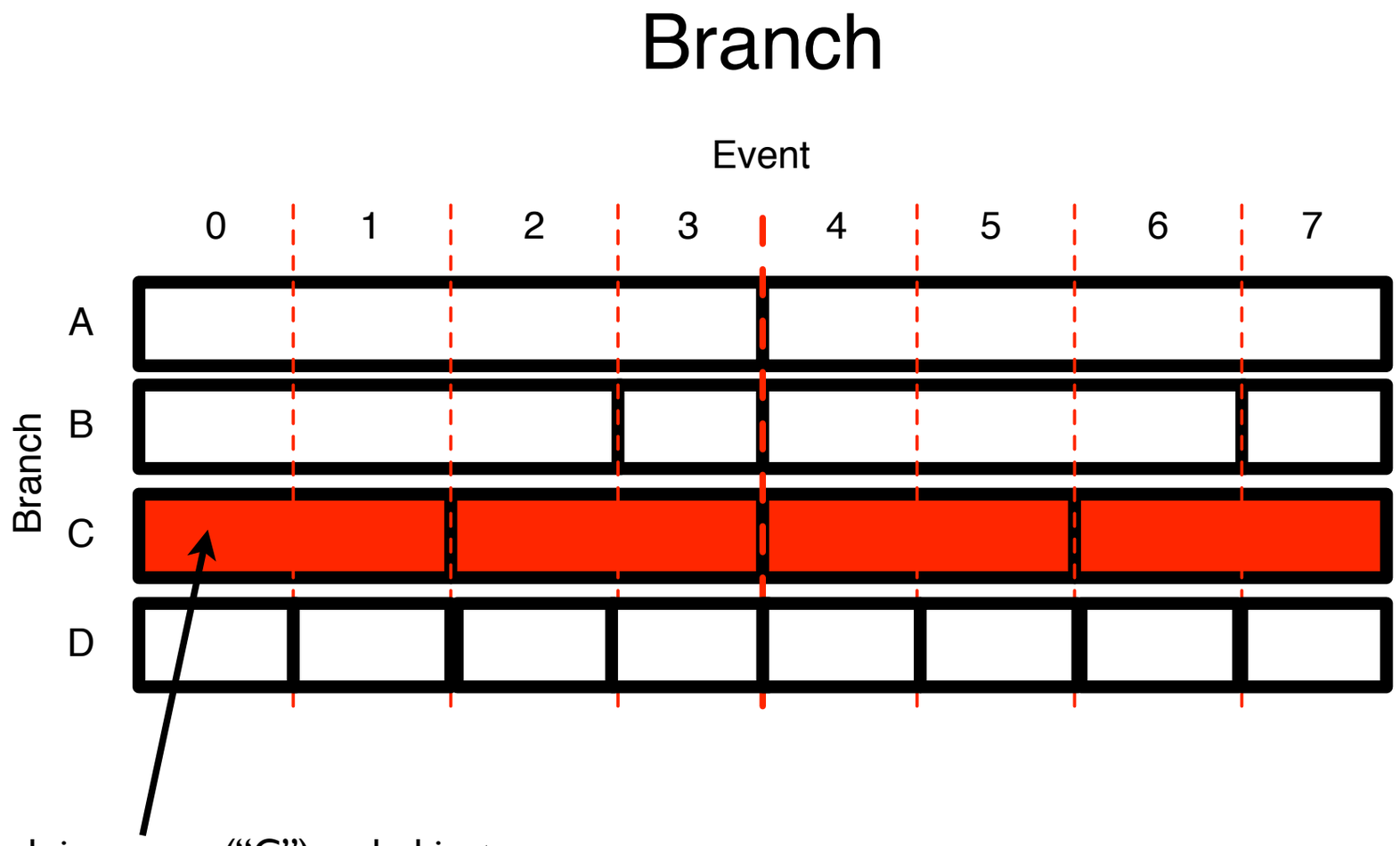
*Note we assume the limit is TCP-rate, not network capacity!

The Shortest ROOT IO Tutorial Ever

ROOT File Layout

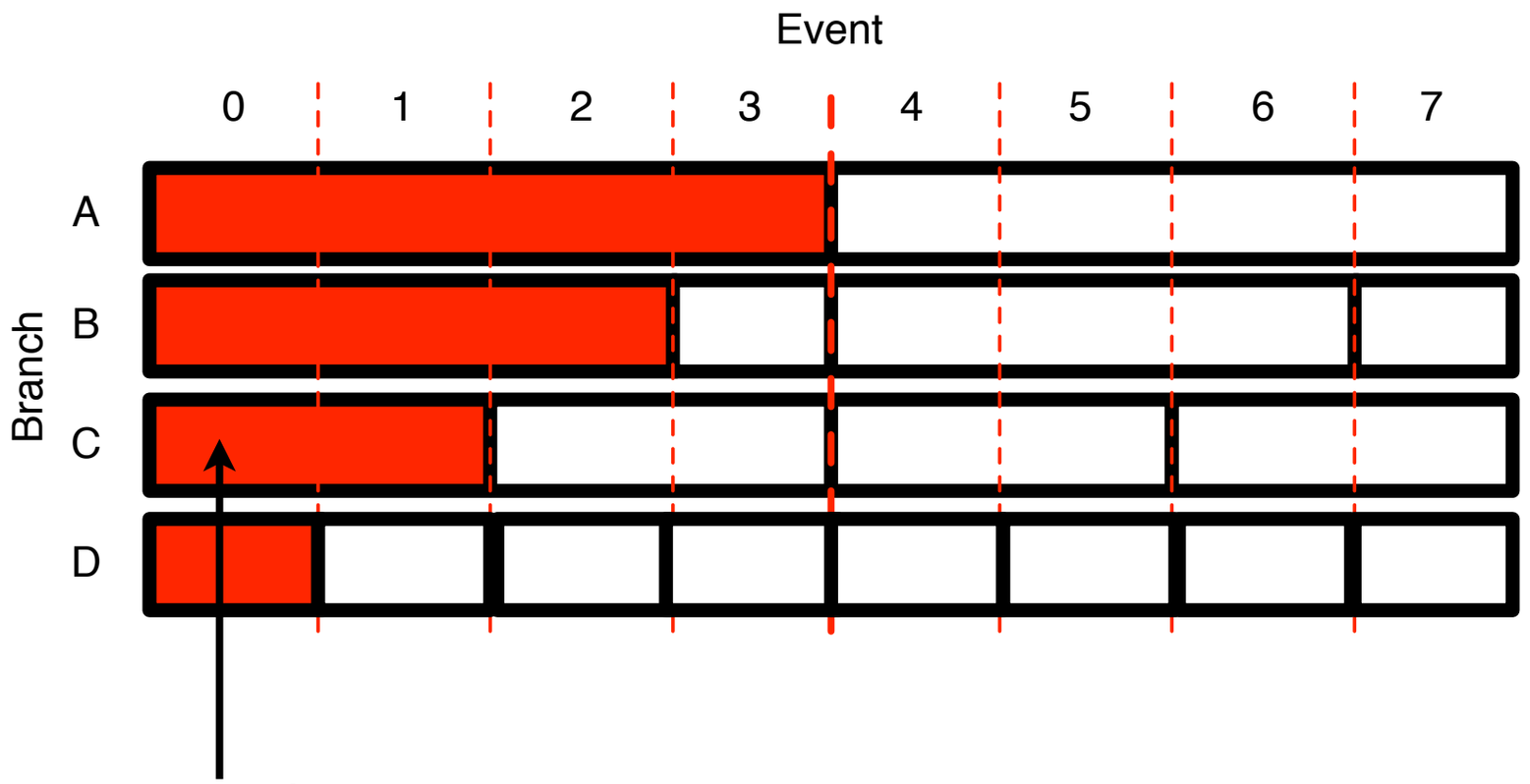


ROOT File Layout



ROOT File Layout

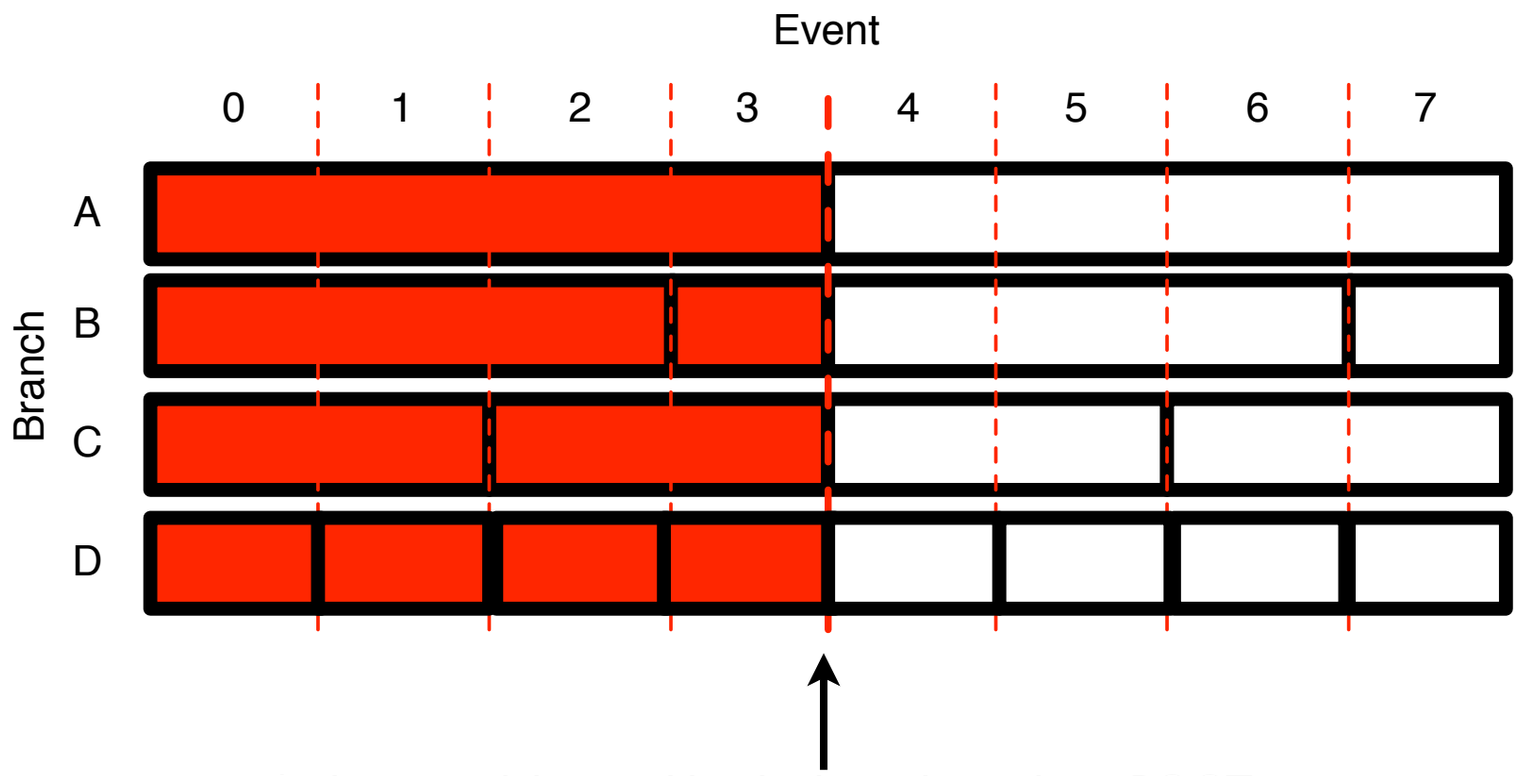
Event



To read all branches from an event, you need one basket per branch.
Baskets may cover an arbitrary number of events - hence, event content may be spread throughout a file.

ROOT File Layout

Cluster

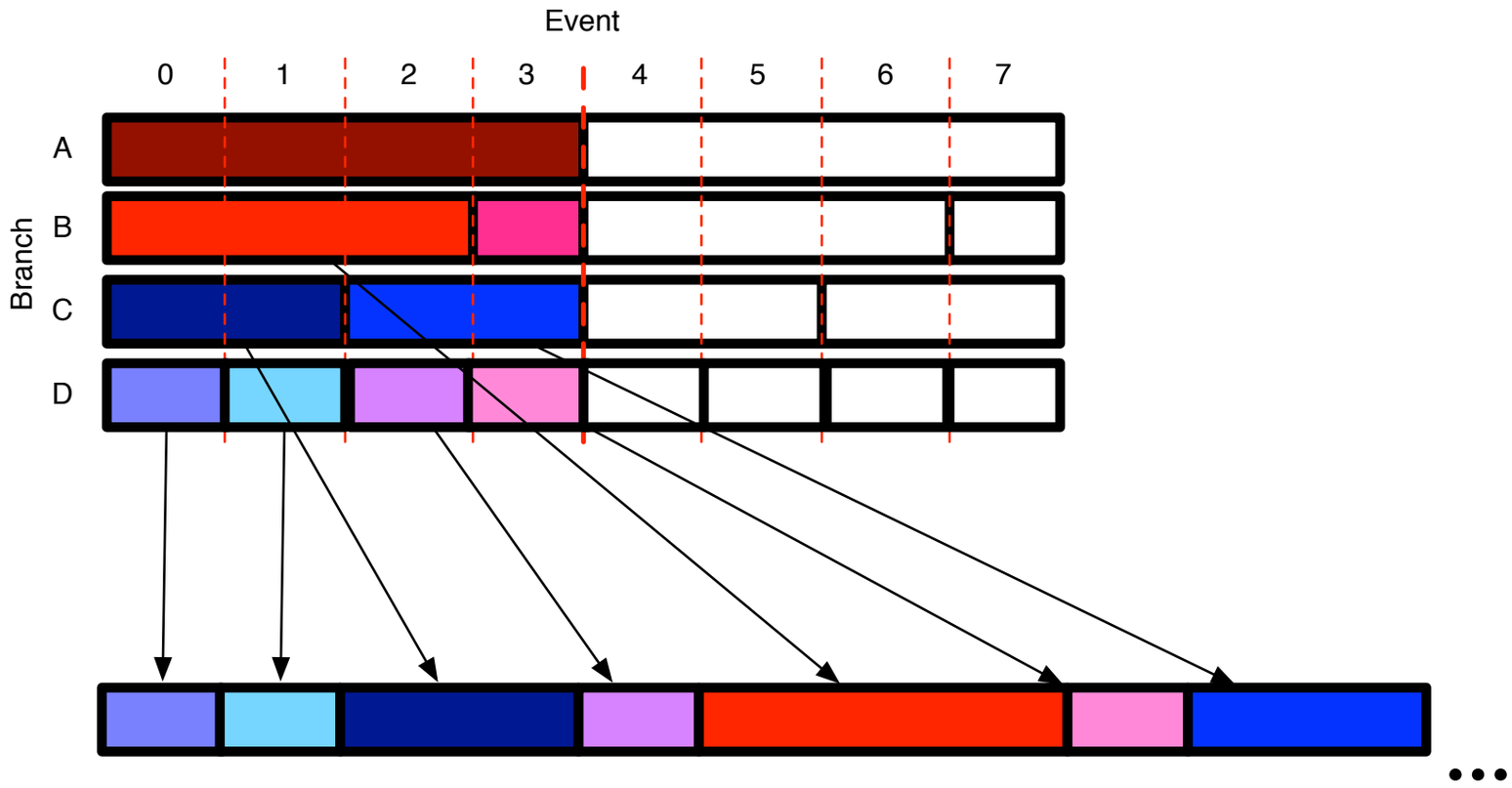


A cluster is delineated by the boundary where ROOT forced all baskets to align. This prevents events that are logically far apart (0 and 7) from sharing baskets on disk and improves event locality.

I.e., if the cluster size is set to 20MB, then ROOT will write files such that all data for any given event is within a 20MB range.

ROOT File Layout

Event / Branch 2D view



Bytestream Layout

In this example, we have 8 events and 4 branches in two clusters.
Baskets are written to disk once they are full.

TTreeCache Basics

- ROOT deserializing code will read out one basket at a time from disk. These baskets can be quite small (hundreds of bytes), causing a huge number of random reads.
- Observation / assumptions:
  - If we read out branch A of event X, then we will likely read out branch A of event X+1.
  - The set of branches read does not vary widely between events.
- The TTreeCache will observe what branches are read for the first twenty events. Afterward, it will prefetch the corresponding branch baskets for as many clusters as will fit in memory.

Why Bother?

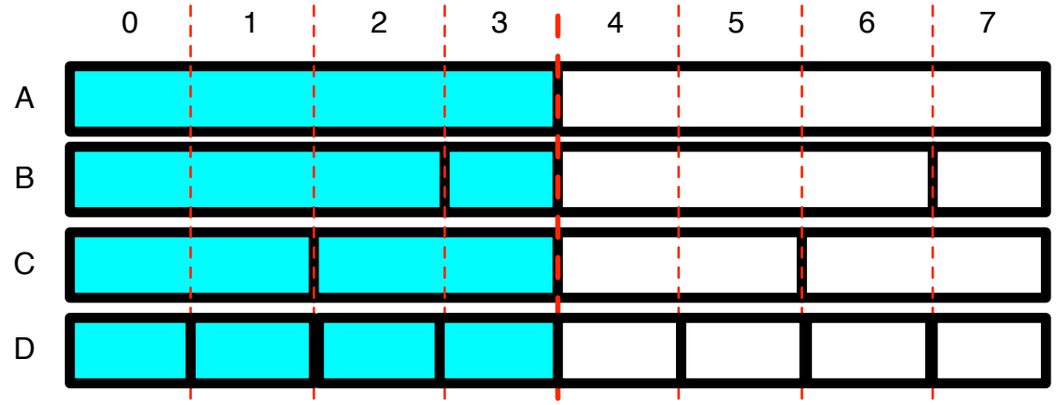
- Default behavior is one IO request (and network round trip) per basket; as there are often thousands of branches per event, this is a huge number of network round trips.
  - TTreeCache can issue hundreds or thousands of requests at once.
- Compared to the fully-optimized, single-source CMSSW:
  - Disabling TTreeCache results in a **1.7x** slowdown for the local case.
  - And a **177x** slowdown for the high-latency case.

Strategy A: Reduce number of reads

Improved TTreeCache startup

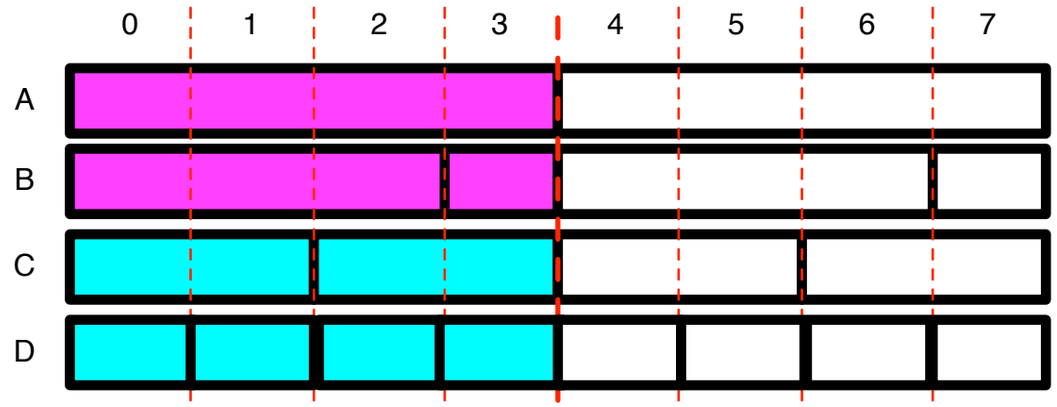
- When training, the TTreeCache reads out one basket at a time.
  - If the user reads 1000 branches on the first event, there are 1000 network round trips; on our test network, this is ~130second delay.
- We create a second TTreeCache which fetches all data for the first 20 events (or 20MB; whatever is smaller) and we separately record all branches used.
  - After the first 20 events, we manually train another TTreeCache which is used throughout the rest of the job run.
- Hence, the first 20 events typically are read with a *single* network round-trip.

Startup TTreeCache



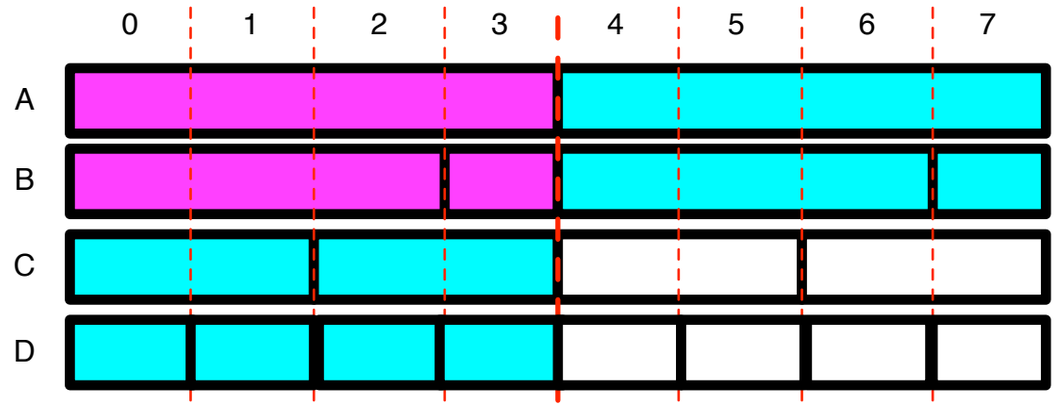
Read all baskets in first cluster using startup TTreeCache

Startup TTreeCache



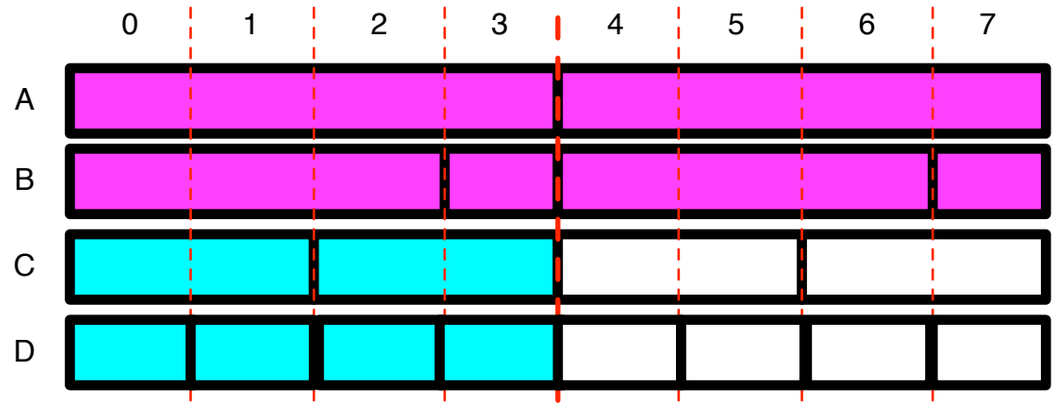
User reads from branches A & B
for first cluster

Startup TTreeCache



Regular TTreeCache prefetches branches A & B for second cluster

Startup TTreeCache



User reads from branches A & B
for second cluster.

Total reads: 2
Default behavior: 4 reads.

Startup Cache Performance

- For local reads, removing the startup cache causes a 1.03x slowdown.
- For remote reads, removing this cache causes a 13.8x slowdown.
- The performance difference is all from the training period. For remote reads, when removing the cache, the first 100 events take 8 minutes. The next 100 events take six seconds.

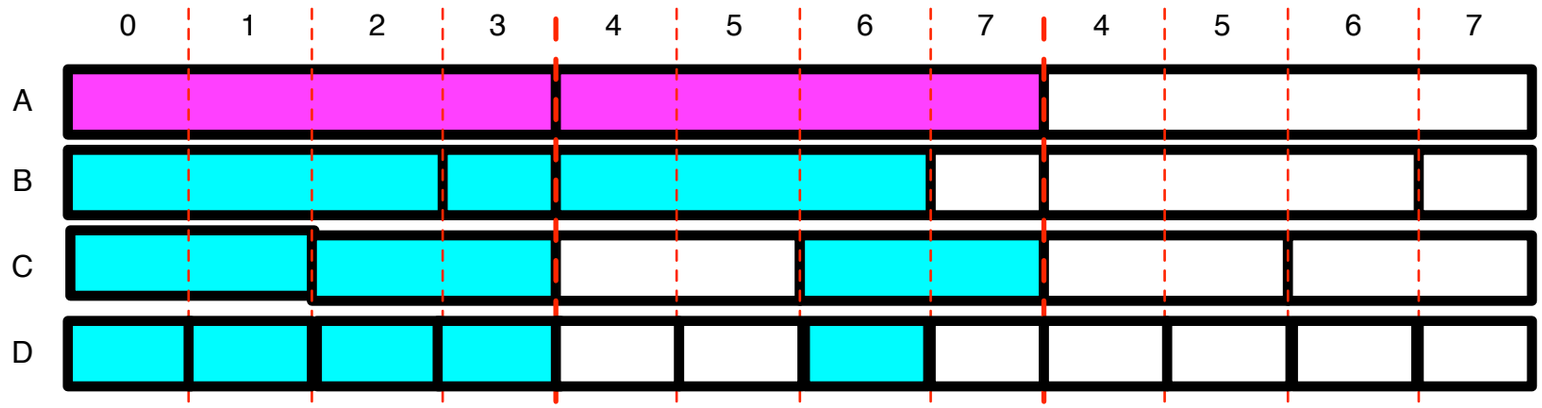
Trigger Pattern Optimization

- It is common for an analysis to read branches X, Y, and Z for each event; then, based on the contents of those branches, read out additional branches.
  - We cannot do prefetching for this case - the code has no way of knowing what will be used!
  - ROOT's default behavior is to do one I/O per basket if the user accesses a branch not in the cache.
- We again use a secondary TTreeCache: whenever we notice a cache miss will happen for the primary TTreeCache, we switch to the other TTreeCache, which reads all the missing baskets for the event.
  - How do we determine the missing baskets? The first time the "trigger" occurs, we prefetch all branches and record which ones were used.

Trigger pattern optimization

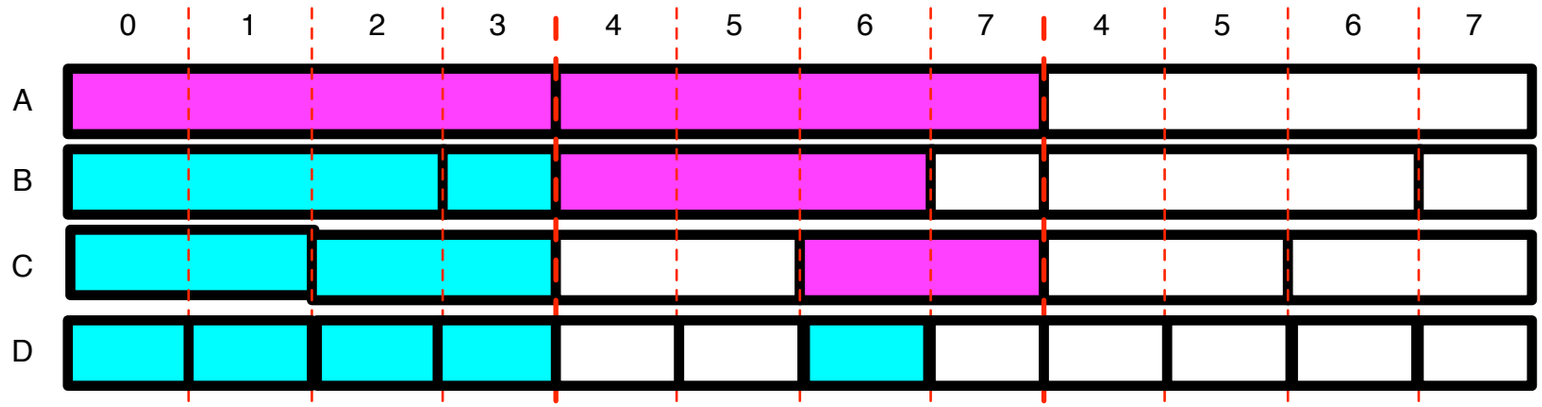
Suppose branch A is our physics trigger;
CMSSW notices object 6A is requested

Trigger pattern optimization



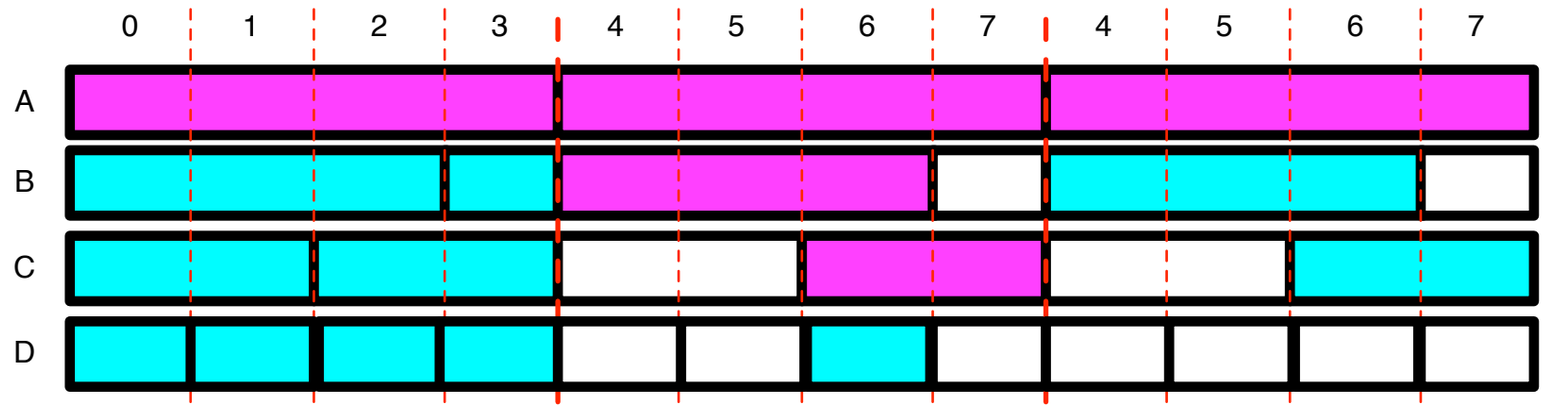
This is our first cache-miss.
Trigger cache prefetches all branches for event 6.

Trigger pattern optimization



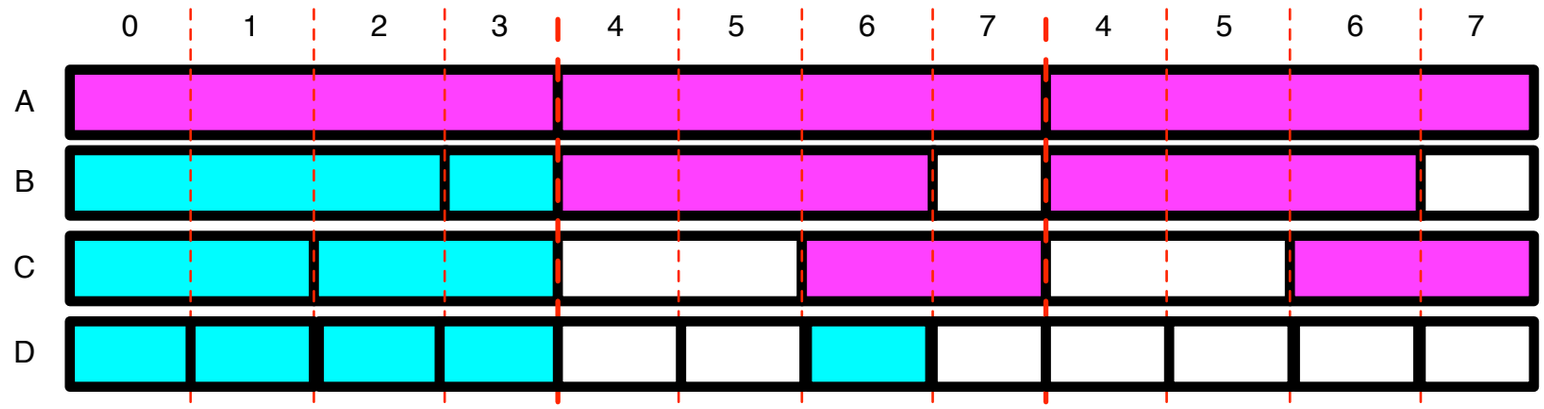
User reads branches B and C of event 6.

Trigger pattern optimization



Trigger fired for event 6;
Last time, only branches B and C were used.
We only pre-fetch those.

Trigger Pattern Optimization



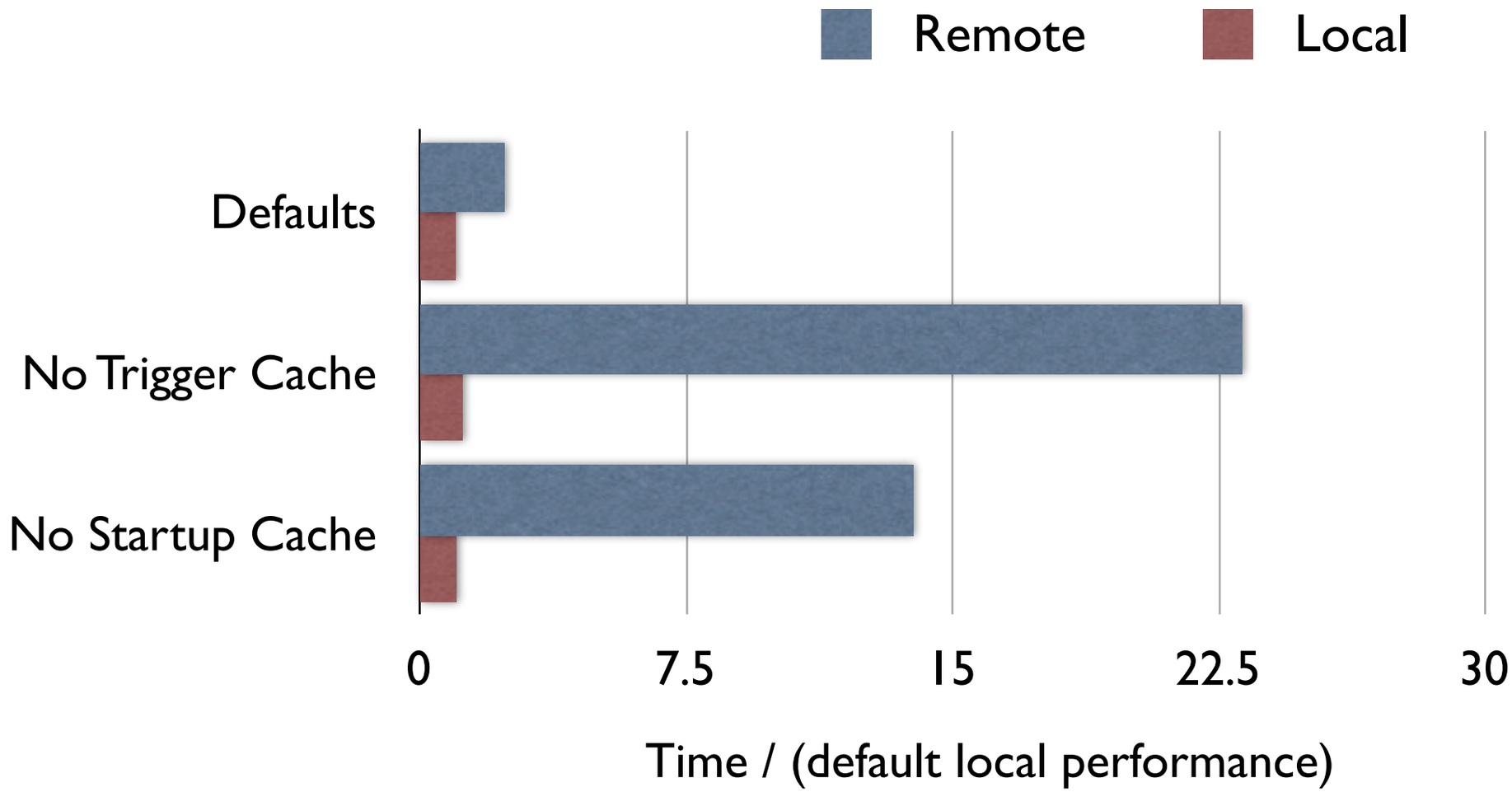
Total reads: 5

ROOT default reads: 7

Trigger Cache

- Disabling the trigger cache incurs a 1.2x slowdown on the local network (compared to the normal CMSSW).
- Disabling the trigger cache incurs a 23x slowdown on the remote network.

Summary - Avoiding Network Round Trips



Not shown: ROOT defaults (no TTreeCache) reading remotely is 177x slower than CMSSW's defaults reading locally!

Strategy C: Increase Parallelism

Multi-source I/O

- Multisource I/O is the heart of the improvements for Run II of the LHC.
- Currently limited to Xrootd protocol; techniques used would also apply to other protocols which have read pipelining or vector reads.
- Implementation strongly depends on having an asynchronous client and C++11

Why not Bittorrent?

- Most multisource protocols copy the entire file to the client.
  - Files are broken into large-ish chunks; each source downloads some subset of the chunks.
- This is not usable for us: we may be reading a small percentage of the file.
  - The pieces we read don't necessarily fit neatly into equal-sized chunks.
  - We may not have space or local I/O bandwidth to buffer the file on local disk.

Multisource IO

- Design goals:
  - *Quality metric*: Determine a metric for quality of the source server; the algorithm should prefer servers with a higher quality.
  - *Source discovery*: Actively balance transfers over multiple links in order to determine several high-quality sources of the file.
  - *Recovery*: Recover from transient I/O errors at a single source.
  - *Do no harm*: Minimize the impact on the source site versus a single-source client. Understand both average case and the worst case scenarios.
  - *Balance*: Have the number of requests per source be proportional to source quality.

For the impatient who cannot possibly wait until the CHEP paper is published:
https://github.com/bbockelm/cmssw/blob/multisource-xrootd-v3/Utilities/XrdAdaptor/doc/multisource_algorithm_design.txt

Basic Design - State

- Throughout the lifetime of the file object, three sets are maintained:
  - *Active servers*: servers we are currently using to service reads (max of 2).
  - *Inactive servers*: servers with an open file handle, but not used by default.
  - *Disabled servers*: Servers which have been used previously but had a fatal error.

Basic Design - Source Discovery

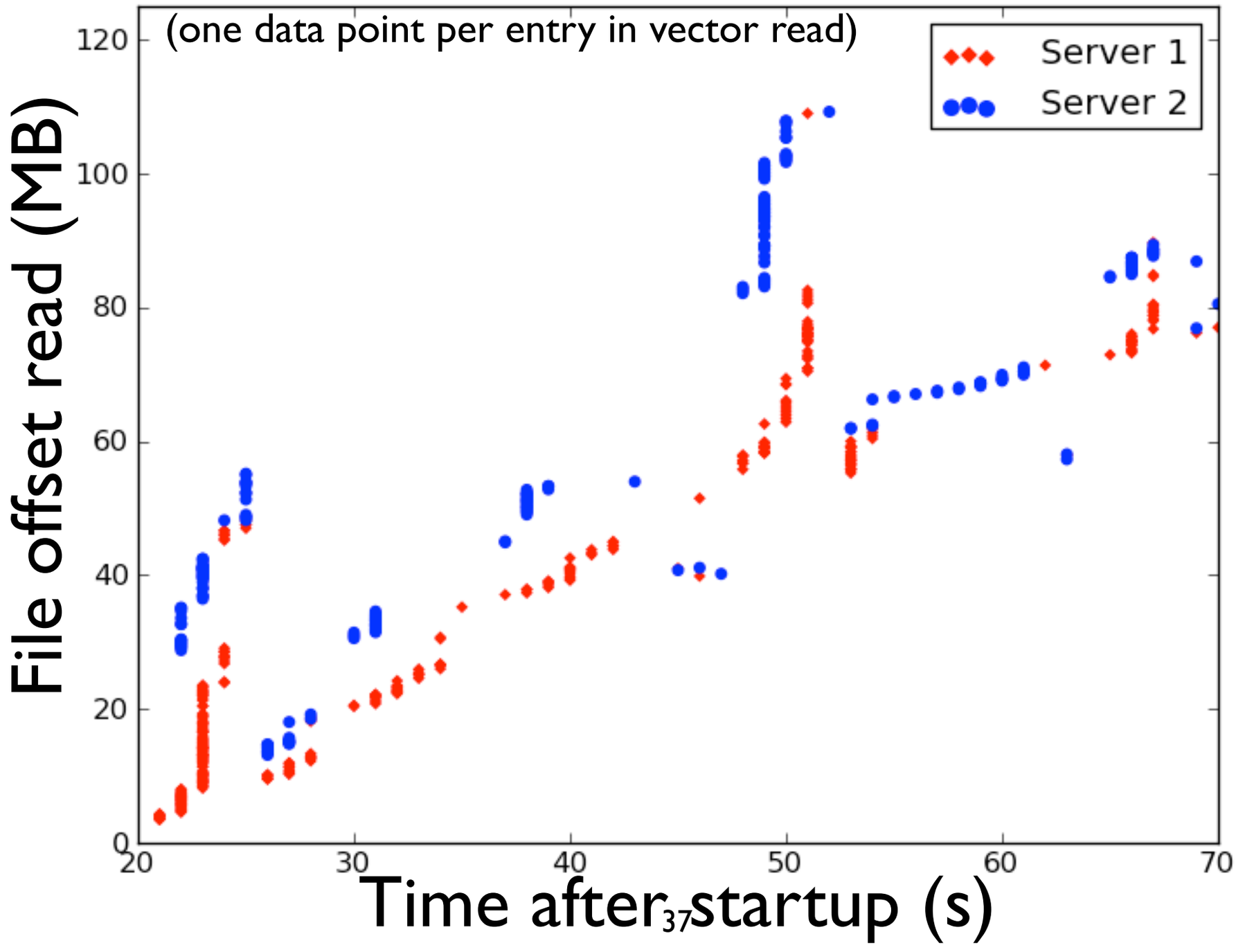
- Inactive sources and disabled sources are initialized empty.
- The active sources set is initialized with the data server returned by the redirector during the initial file open.
- Every 5 seconds, a new file-open is attempted (the client requests the redirector to exclude currently-open sources); the resulting server is added to the active sources. If there are already two servers in the active sources, it is added to the inactive sources.
  - If the redirector returns a file not found, then the next file-open probe is scheduled for 2 minutes in the future.

Basic design - IO request

- For each I/O request:
  - Check current quality of each source.
    - If a currently-active source has worse quality than an inactive source, swap the two.
    - Randomly swap active and inactive sources with a 1% probability per 10MB.
  - Split the I/O request proportionally by volume in two according to the current active source quality.
    - Request is split in two nearly-contiguous chunks; this allows the server to see sequential I/O if possible.
  - Issue the two requests to the active sources. Return result to client if both succeed.
    - If a request fails, move corresponding source to the disabled sources. Immediately reissue request from one of the inactive sources if possible. Otherwise, throw an exception.

Multisource Illustration

Read offset versus time, per source



Multisource Performance

- If the redirector picks the optimal site for the single-source case, there is no performance improvement.
- If the redirector picks a sub-optimal site, we see large improvements by using multi-source:
  - For a client at CERN, reading from Nebraska and DESY, the speedup is 1.37 compared to reading from Nebraska-only.
  - Single-source reading from DESY has the same speed as multi-source reading from DESY and Nebraska.
- If a server stops working - or its performance drops significantly compared to the other source - multisource will stop using it.
- So - multisource client basically shields us from poor redirector choices and poorly-performing servers.
  - It does not fundamentally make things faster, especially as we are not TCP-loss-limited on the test connection (Nebraska-to-CERN).

Conclusions

- We believe acceptable performance over high-latency links enables new use cases for HEP and decreases time-to-science.
- Default ROOT IO (regardless of whether TTreeCache is used) does not perform acceptably over high-latency links for our test case.
  - We have additional mechanisms to make CMSSW performance acceptable.
  - An obvious future step is to contribute these ideas back into the ROOT IO core.
- The multisource client allows us to avoid the worst problems within the infrastructure while the job is running - whether they are issues from poor redirection choices or poor server performance.
  - For cases where server performance and selection are optimal, it is performance-neutral.
  - As we do not assume anything is optimal, we hope this will be an important advance for CMS in Run2.