Read pattern replay with data integrity check & write latency hiding

David Smith (david.smith@cern.ch) on behalf of IT-DI-LCG, UP team.

8 July 2019, pre-GDB XCache

Overview

- Motivation
- Proposed projects
  - Read replay
  - Write buffer or cache
- Read replay
  - Features considered
  - Development & status
- Write buffer or cache
  - Requirements & analysis
- Summary

Motivation

- Two projects were proposed in the context of DOMA Access activity
  - There is a model of using of tier-2 without traditional storage element, but with an xcache, plus dedicated storage for the region:
    - How is xcache & its hardware likely to react to a given access pattern at a site?
      - Testing can act as a stress test for xcache
    - Desirable to support remote write
      - Less scratch space required (critical for many cores)
      - However impact of remote writes (disadvantage): could this be reduced?

Proposed projects

- Read replay:
  - Observe an xcache on intended production hardware under a plausible load pattern, e.g. from a time & file read access log
  - Verify data integrity

- Writable buffer or cache:
  - In addition to the xcache: maybe at the same endpoint, but not an integrated feature of xcache
  - To reduce latency for an application when writing xroot

Read replay: overview

- Major features considered:
  - Would like to use large dataset, but avoid need to use production storage for data source
    - Use algorithmically generated data as sources
      - No need for large amount of storage space
      - delivery can be checked by recalculating expected data

Read replay: operation

- To create the data access schedule and pattern
  - Assume a time, filename and filesize record of actual historic open()s exist for a site
- However must indicate how the file access will be done (assume this level of detail is not available in logs)
  - A compromise between
    - reproducing exact access pattern
      - probably not possible without re-running the applications or logging much more access information, and
    - a plain full sequential access
      - may be very different from actual load

Read replay: configuration

- File & Access profiles
  - Match each file to a file profile
  - Also provide access level profiles

…
10:03:45 /dteam/file1 450MB
10:03:47 /the/file2 1.2GB
…

File Profile 1

Filename Match RE
50% Access Profile 1
50% Access Profile 2

File Profile 2

Filename Match RE
100% Access Profile 2

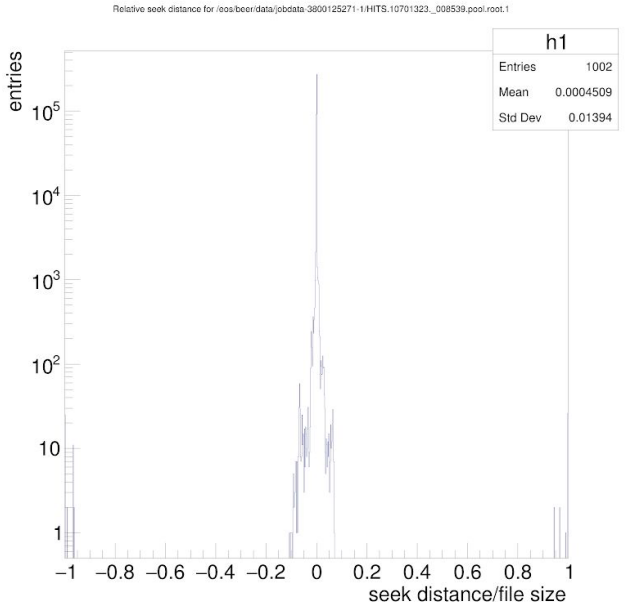
Access profile 1

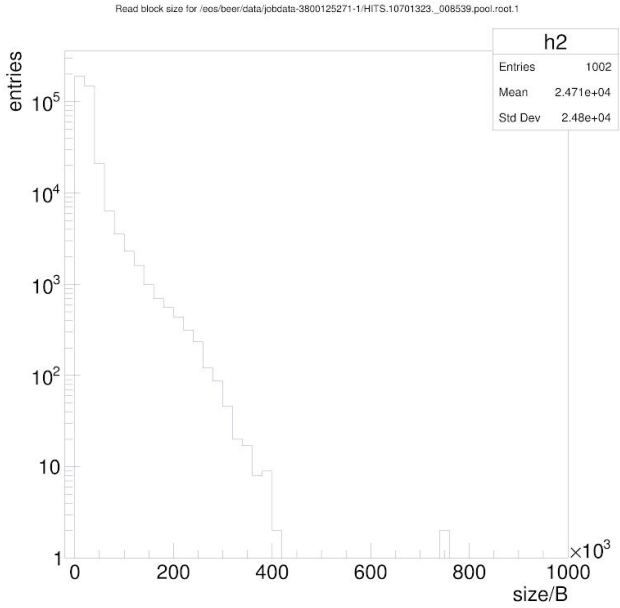
Frac bytes read (may be >1)
Read rate (MB/s)
PDF of read req size
PDF of seek distance

Access profile 2

Frac bytes read (may be >1)
Read rate (MB/s)
PDF of read req size
PDF of seek distance

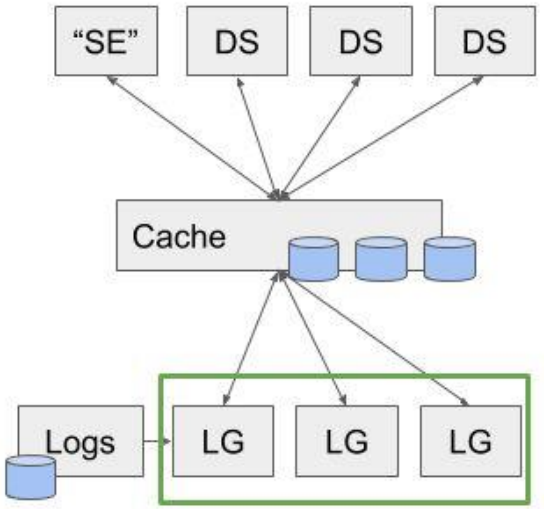
Read replay: Access example





File	*Type*	*Size/GB*	*Read / Len (single process)*	*Read / Len (MP=8)*
HITS.10701323._008539.pool.root.1	LowPt MinBias	3.2	3.05	3.39
HITS.10701323._008545.pool.root.1	LowPt MinBias	3.2	3.04	3.38
HITS.10701323._008550.pool.root.1	LowPtMinBias	3.1	3.04	3.37
HITS.10701323._008552.pool.root.1	LowPtMinBias	3.2	3.01	3.15
HITS.10701323._008556.pool.root.1	LowPtMinBias	3.2	3.03	2.60
HITS.10701323._008557.pool.root.1	LowPtMinBias	3.2	2.84	2.22
HITS.10701335._002545.pool.root.1	HighPtMinBias	2.1	0.24	0.26
HITS.12071736._001213.pool.root.1	Signal	1.1	0.99	1.03

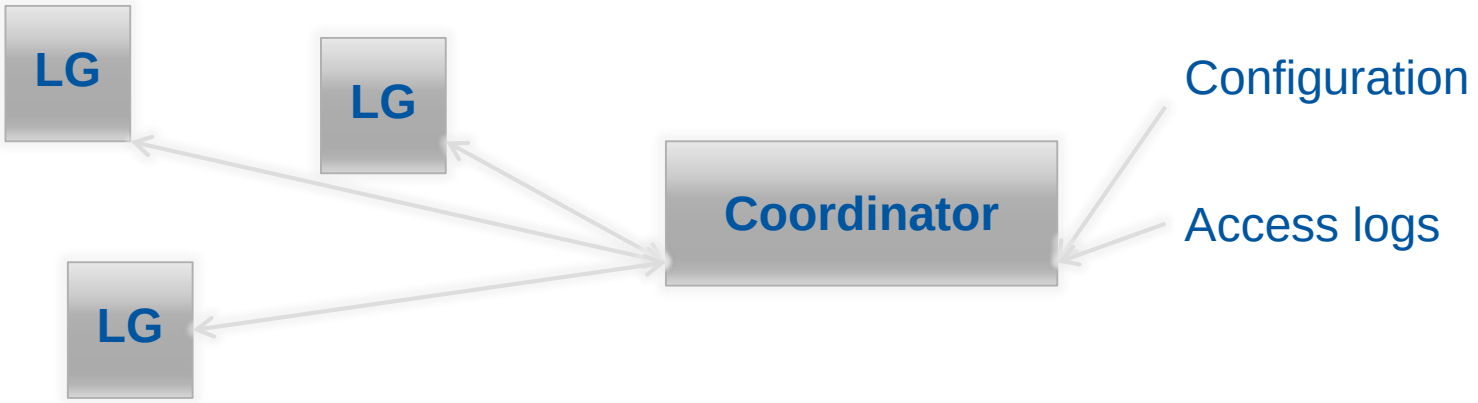
Read replay: Components



- Datasource (DS)
  - A custom xrootd Oss plugin
  - Scalable (stress testing)
- Load Generator (LG)
  - Application that uses xroot client API (XrdCl) to issue writes to the cache
  - Scalable (stress testing)

Read replay: Components II

- Load Generator (LG)
  - Decided on a coordinator & worker model for flexibility and more centralised configuration



Read replay: function

- Intended use is to start a number of load generators
  - On WNs, cloud resources, dedicated machines
  - Load generators are given the address of the coordinator
- Coordinator
  - Supplies profiles and filenames, computes and schedules when to start transfers
  - Participating load generators are offered files due to start within a short period; LG try to claim files if they have capacity

Read replay: development

- Xrootd Oss for datasource: currently functioning
  - For generating pseudo file content using MurmurHash3: the hash produces a 16 byte digest, hashes arbitrary length input message along with a *seed* (32bit value)
  - File content is considered to be a series of 512 byte sectors, and each sector contains 32, 16 byte sub-sectors: The subsector content is then hash of
    - (sector# | h0), seed=subsector#
  - Thus the content of the file can be calculated at arbitrary offset without calculating preceding sectors
  - h0 is a 16 byte value, the hash of the filename, with seed=0
  - File length is encoded in a suffix of the filename

Read replay: datasource

- e.g. /directory1/file100_1048576_0

- Is generated with content based on the hash of “file100”. The length is assumed to be 1048576 bytes. (e.g. using xrdcp to copy the file gives a file of specified length)

- Observed performance of datasource is about 500MB/s per core on the test machine.
  - e.g. saturate a 10Gb/s link with ~2 cores.

Read replay: LG and coordinator

- Communication between the two is with gRPC and protobuf for encoding.
- Coordinator basic functions written:
  - Calculates the schedule of file opens based on a start time and optional time-scaling factor.
  - Sends offers to load generator of upcoming files.
  - Collects claim requests, and sends assignments based on arbitrated requests
  - Offers an RPC for fetching the configuration (file and access profiles).

Read replay: To do

- About half of the coordinator done
  - Need to consumer for input time & file access data
- Most of the load generator to write
  - (RPC client, generation of read and seeks according to relevant PDF and read rate, XrdCl integration)

- Some system testing established on gitlab.cern.ch, as part of continuous integration
  - using a Kubernetes cluster on openstack
  - deploy pods for datasource, xcache, and a client.
  - Will add LG & coordinator and perhaps multinode datasource.

Write buffer or cache

- (Buffer -> data not kept locally after sending to remote, cache -> available until file close)
- Features considered required:
  - Upstream storage selected on a per file basis
  - File available upstream after close() succeeds
    - Desirable to have smallest delay after closing file until it becomes available upstream
  - Support only create of new file (not open of existing file)
  - Authorization and access protocol to support: at least x509, xrootd
  - General operations must be supported:
    - Opening read/write (e.g. creating and reading from open file); seeking; support for ranges never written; support for ranges written multiple times

Write buffer or cache

- Analysis so far
  - Have looked at the write pattern from 3 experiment tasks
  - Output file is written, no reads: mostly sequential
  - e.g. on a 5.7 GB root file:
    - 4 seeks
    - 98 more bytes than the final file size are written
    - Few bytes are overwritten, few bytes are never written
    - At 20ms RTT to the storage the increase in wall time compared to local write was increased 10% for most impacted task
- A writable cache facility is planned within XDC project
  - Initial discussion with the XDC team (spring time)
- Have had some discussion with xrootd & xcache team for ideas (last year)
- Also useful discussions with others (thank you)

Summary

- Shown two projects
  - Xcache read-pattern-replay with artificial data and data delivery integrity check
    - Under development
  - A buffer or cache to hide write latency
    - Probably will be a writable endpoint at the xcache (no development intended within xcache itself, to avoid extra complexity of the existing component or disturbing xcache development)