XCache
the future development menu

XRootD & FTS Workshop · CNRS Rhône-Auvergne, Lyon

17 September 2026

Alja & Matevž Tadel, UCSD

OVERVIEW

What moved this year

- **Analysis facilities** — small blocks made affordable, and the serialisation we knew about for years finally measured
  - an open wound, now with numbers on it
- **Resource monitor** — running in production; two memory-safety bugs found and fixed
- **Cache control & consistency** — shipped in 6.0, working end to end over HTTP
- **Testing** — the cache has some real CI for the first time

WHAT THIS TALK IS

Progress on work already in the pipeline — written, measured or merged. Where something is unfinished it is named as such, with what it would cost.

Analysis facilities

XCache was built to pull a whole file across a WAN once and serve it from disk many times. Analysis sends it vector reads — one large request, scattered into hundreds of few-kB fragments — and at an analysis facility those requests come from a large number of jobs.

The block size and the request path were sized for the other job.

We looked at three levels of the stack: inside `XrdPfc`, in the server, and in the protocol. Each change is largely self-contained, and each solves part of the performance problem. What they cost to adopt is the last slide of this section.

Peak speed vs. sustained high load at a facility

Peak is what benchmarks run

- One client, quiet cache, nothing competing
- Proves what the chain **can** deliver — client, cache, remote storage
- Necessary. Not sufficient.

What a facility actually runs

- O(10)–O(100) concurrent analyses against one cache
- **Deployment** is decided by sustained aggregate throughput
- More users, more distinct data, competition for the disk — what stays becomes a **retention and purging** question

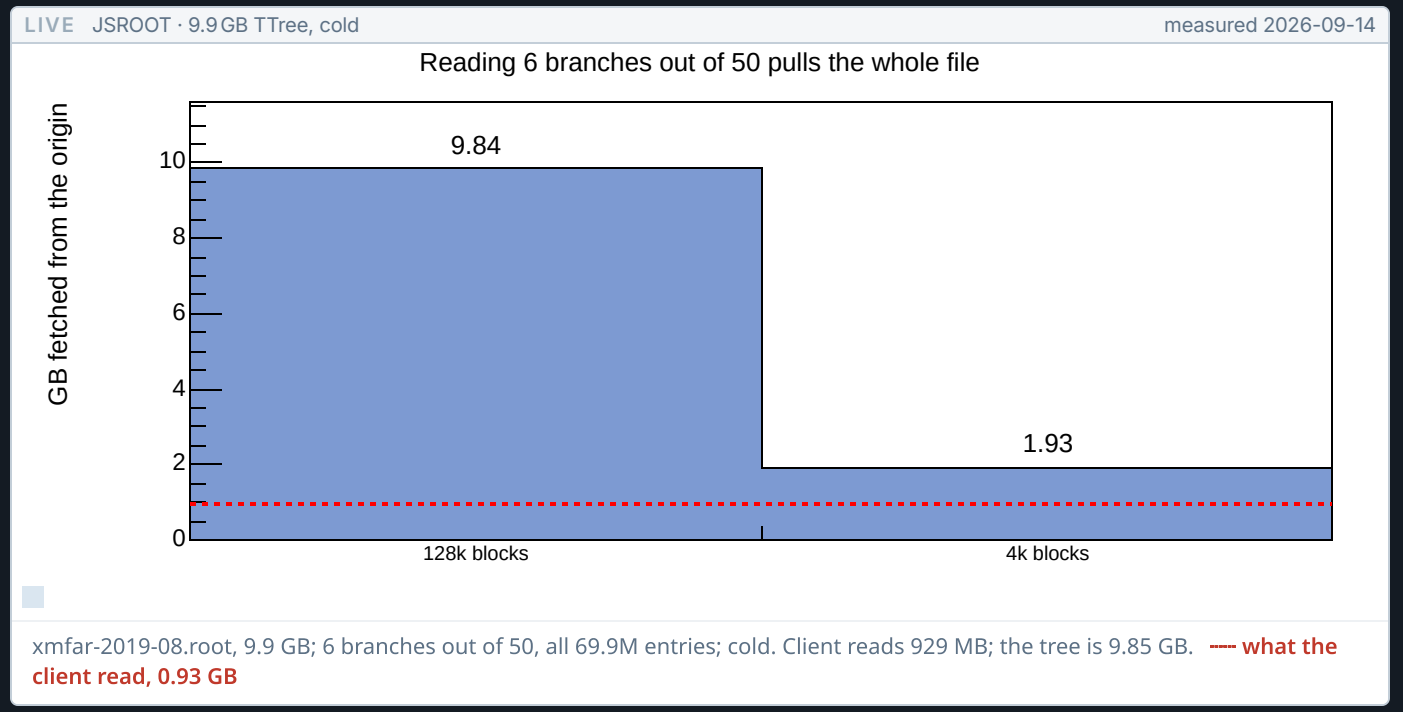
	cold, few clients	**warm, many clients**
limited by	origin RTT, request concurrency	cache disk and page cache, purge competing with fills
what helps	overlapping requests, merging adjacent blocks	fewer, larger reads — *not* more concurrency
we have	measurements	**nothing in this talk**

NOTE — BEFORE THE NUMBERS

Every measurement in this talk is a **single client on a quiet cache**. Measuring the loaded regime needs SSD or NVMe and a modern NIC — our boxes are ten years old. Please help :)

Reading 6 branches out of 50 pulls the whole 9.9 GB file

- Basket sizes in a compact analysis format are a few kB
  - e.g. DAOD_PHYSLITE, NANOAOD
- **The client batches them into vector reads of 200–600 elements** — one large, badly fragmented request
- At `pfc.blocksize 128k` the cache rounds every fragment up to a 128 kB block



WHY THE 10.6× OVER-FETCH IS NOT A CONSTANT

Every 128 kB window holds a wanted basket, so the whole file gets fetched — that ratio is just file size over bytes read, and climbs if you read fewer branches. **At 4 kB it is basket size against block size: 2.08× whatever you select**, and 26% faster than 128 kB despite more requests. How much you over-read depends on the branches, and on the file — basket sizes and split mode. This is one simple ROOT file, from the Gled xrootd monitoring.

Why small blocks were unaffordable

- The client sends **one vector read** of 200–600 elements
- The cache takes it apart and issues **one request per block** — at 4 kB that is **470 714** requests for the tree on the previous slide
  - measured with `pfc.cschk net` and with `off`: identical, and **no vector read ever left the cache**

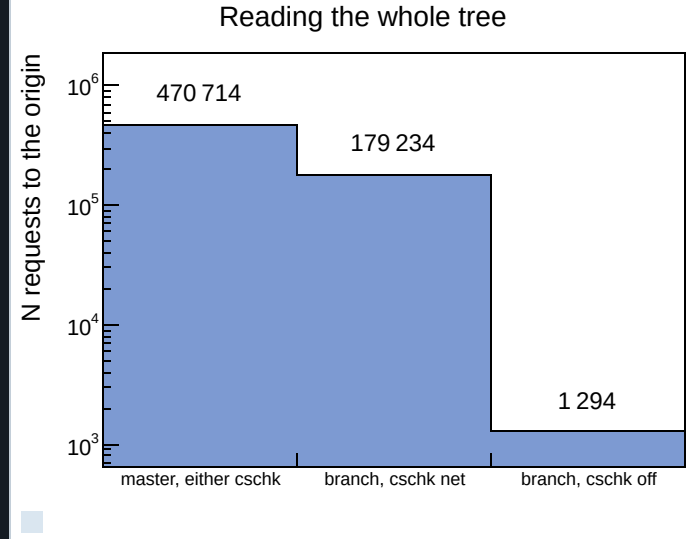
> **THERE IS NO GOOD VALUE FOR `PFC.BLOCKSIZE`**
> 64, 32, 16 kB are all available — every setting trades over-fetch against request count and none is good at both. The block size is fixed when the file enters the cache, before anyone knows how it will be read.

Block runs — one vector read in, one vector read out

WRITTEN, NOT MERGED

- `pfc.blocksize` set the on-disk unit, the accounting granularity **and** every network request — one knob, three jobs
- A **block run** is a set of adjacent blocks fetched as one request
  - `pfc.iosize` caps a run, `pfc.iogap` bridges short holes
- A 2 MB read was 512 requests at 4 kB blocks; now it is two, at the 1 MB `iosize` cap. Merging alone: **470 714 → 179 234** requests, with checksums still on
- **readVs stay readVs**: runs are contiguous extents, so the incoming readV is block-padded into the outgoing one — **1 294 requests**, needs `cschk off`, `kXR_pgread` has no vector form

LIVE JSROOT · 9.9 GB TTree, cold log



xmfar-2019-08.root, 9.9 GB; 6 branches out of 50, all 69.9M entries; cold. 929 MB to the client and 1.93 GB fetched in every case.

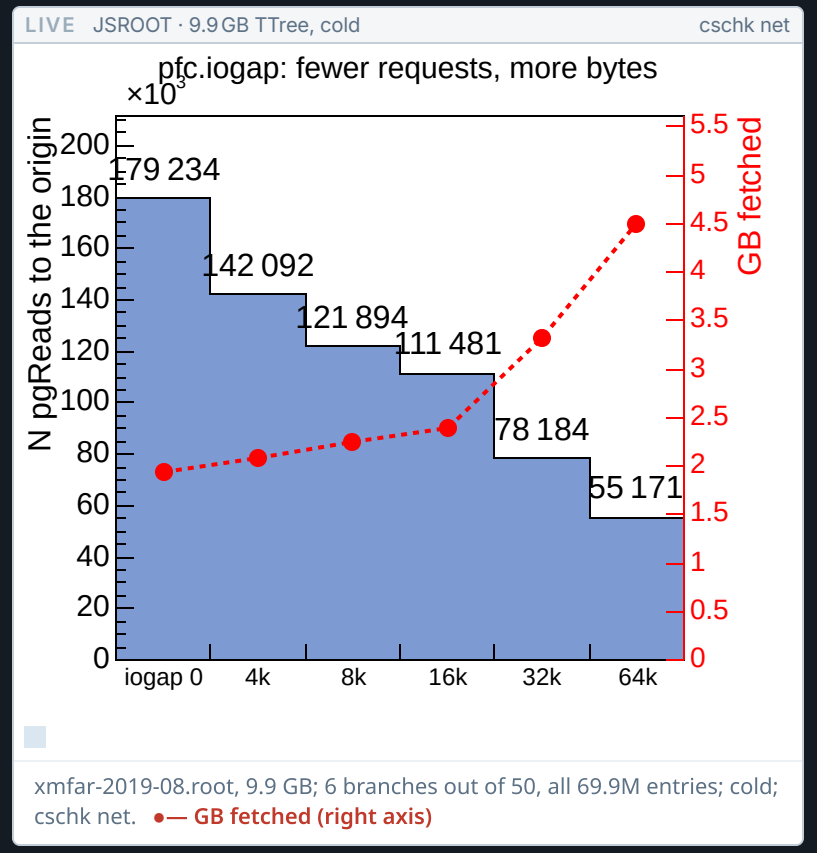
364× FEWER REQUESTS THAN MASTER, SAME WALL-CLOCK

Reading from the UCSD redirector, **San Diego → UCSD at 9 ms**, with a CPU-bound client: request count is free. What this buys is **origin load**.

`pfc.iogap` is a workaround, and it shows

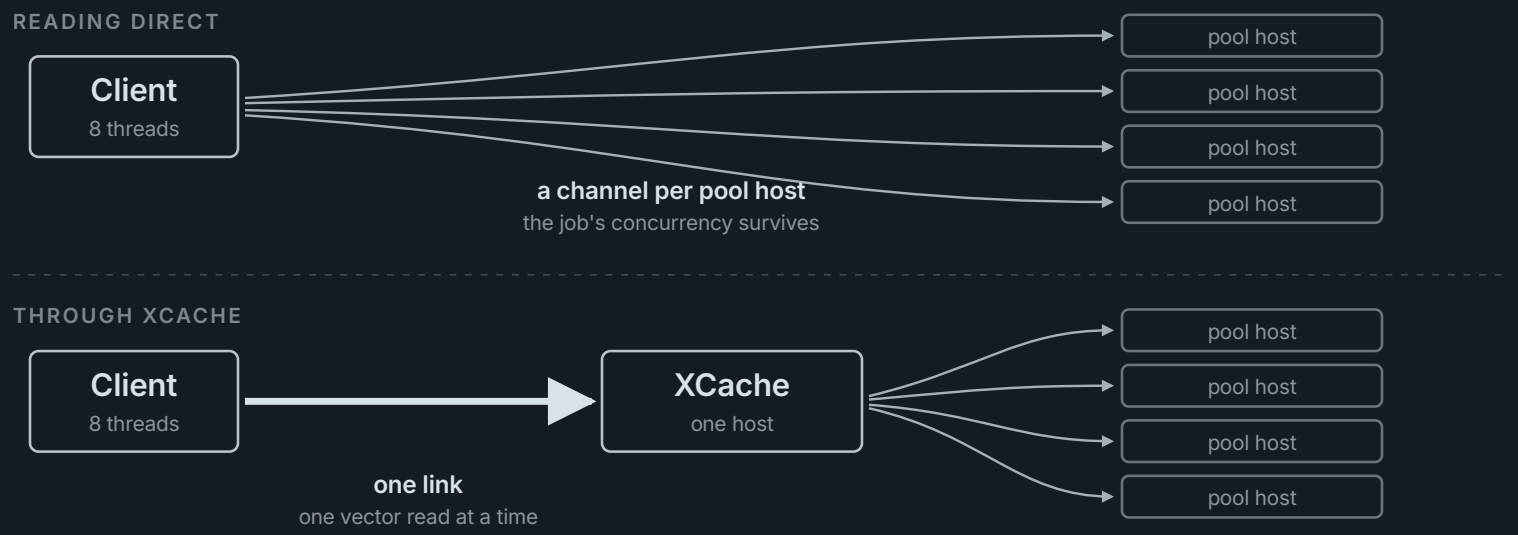
- With `cschk net` there is no vector form, so the only way to cut requests is to make each bigger — `pfc.iogap` bridges short holes, fetching blocks nobody asked for
- **4 kB: 179 234 → 142 092 pgReads** for 1.93 → 2.08 GB; 8 kB takes it to 121 894 for 2.25 GB
- **Between 16 and 32 kB** the bytes start climbing faster than the requests fall
- `iogap 8k` is also the fastest of the six — 170.8 s against 175.9 s at `iogap 0`

> WITH READV AVAILABLE IT IS PURE LOSS
>
> **1294 requests at `iogap 0` and at 8 kB alike** — identical — for 17% more bytes and 4% more time. `pfc.iogap` was introduced because `pgRead` has no vector form.



Concurrency — the part XCache cannot fix — it is already fully async

- A cache is one host, and XrdCl pools on (user, host, port) — so a job gets **one socket**
- **That socket carries one vector read at a time** — the protocol thread is parked inside `XrdSfsFile::readv()` until the response is complete, and that call has no async form
- `do_Read()` offloads on the request's `pathid`; **`do_ReadV()` ignores it**
- ⇒ the job's concurrency arrives on one socket and is serialised **above** **`XrdPfc`, not in it**



WHAT THAT COST

Cold AGC ttbar, 9 files, 8 RDF threads, **San Diego → FNAL: 210.8 s** through the cache, **36.6 s** direct — slower than no cache.

Fix the server, or rewrite every URL

BOTH IMPLEMENTED AND MEASURED

Async `do_ReadV`

- A server-side path so a socket can carry more than one vector read
  - **no wire change**: the protocol already allows it
- The **concurrency** needs no client change — measured with stock XrdCl
- One open, cold, **San Diego → FNAL, ~60 ms**, 67.1 MB, 16 in flight: **9.33 s → 2.82 s**
- Same AGC benchmark, 9 files, 8 RDF threads: execute **6.3× faster**, 210.8 → 33.7 s
  - **and now comparable with going direct**
- An install at the site is all that is needed

`?xrdcl.intent=` — per file, or per open

- Forces XrdCl onto a separate socket instead of pooling — **works today**, nothing to merge
- **9 files**, one intent each — a full AGC run: **5.2× faster**, 249.6 → 47.8 s
- **One file, 16 opens, 16 intents — 9.9 → 3.5 s** against FNAL dCache, the same floor as the server fix
- Requires control over **every URL**, and each extra open is another login and another socket — and, if multiplexing on one file, another TTreeCache

THE CONCURRENCY WAS ALREADY BEING ASKED FOR

A bare, unmodified XrdCl asked for **16 vector reads at once** and took **9.33 s** — against **9.69 s** for the same reads issued one at a time. Nothing overlapped: the server answered them in sequence.

Partial responses and per-chunk error recovery are a separate question — real client work.

XCache is a server and a client — they upgrade differently

	job → cache	cache → origin
endpoints	users' ROOT / XrdCl — LCG, containers, pinned images	a cache and an origin, both site-run
upgrade horizon	years — HEP software deployments are often kept over full experimental runs	the next release

- Job-facing, anything needing a new client is **dead on arrival** — which is why the fix there is server-side `do_ReadV`
- Origin-facing, a protocol change is viable: **`kXR_pgreadv` 's XrdCl work ships with XCache**, not with anybody's ROOT
  - it does still need the origin server upgraded
  - the outgoing leg pools the same way — **Andy has work started there**

WHICH IS WHY THE WINDOW IS NOW

Sites may begin deploying **6.x within months**. Anything that lands before that arrives on an upgrade they are already doing.

AND ONE SETTING THAT BELONGS TO NEITHER COLUMN

Concurrency is free against a WAN origin, can hurt a saturated spinning disk, and on NVMe does not matter at all. So it is a property of the **deployment**, not of the protocol — a setting at the cache, where such settings already live.

The final request multiplier: `cschk net` — the unfortunate default

client readV into cache	origin requests	mean in flight	wall
`cschk net`, sync	4 096	58	8.82 s
`cschk net`, async	4 096	**1078**	2.70 s
`cschk off`, async	**32**	13.9	2.80 s

One client, one file, **San Diego → FNAL, ~60 ms RTT**: **32 readvs of 128 chunks × 16 kB**, 16 in flight, 67.1 MB in all, at `iosize 64k`. With `net` that is one request per chunk — 4 096 of them, and in 2.70 s that is ~1500/s. The remote storage would notice.

- `pgRead` has no vector form, so **every block run becomes its own request**
- Async does not change the count — it removes the throttle that was hiding it
- With `cschk off` they batch back into readvs, **one per client readv: 32**
- Same wall-clock **at one client**
  - a facility scales its cache for load — it should not have to scale it for a multiplier

> **THE REQUEST AMPLIFICATION HAS TO STOP**
> A vector `pgRead`, or checksums carried in `readv`. **Minimal change in XCache either way** — we do not need page granularity for anything. Choose it on protocol grounds.

What it would cost

Change	Where	Lines
Block runs, `pfc.iosize`, `pfc.iogap` one `readv` in, one `readv` out — with `cschk off`	`XrdPfc` only	**+846 / −282**
Async `do_ReadV` experimental branch off the block-run branch — measures the effect, not a finished design	`XrdXrootd`	+268 / −6
`kXR_pgreadv` a new request, or an extension of `readv`	six modules	**+1496 / −65**
the protocol point	`XProtocol`	+33 / −3
server side	`XrdXrootd`	+117 / −4
client side — **the cache's own** XrdCl, on its outgoing leg, including bad-block re-fetch	`XrdCl`	+1035 / −34
cache plumbing	`XrdPosix`, `XrdOuc`	+250 / −4
the cache turns its block runs into one — the job still sends plain `readv`	`XrdPfc`	+61 / −20

Each title links to a **frozen branch** on github.com/osschar/xrootd, so these counts stay true whatever happens next.

THIS IS WHERE WE ARE

All three are **written and measured**. Block runs touch nothing outside `XrdPfc` — merge that one now. The other two need a decision **soon, to catch the actual 6.x deployment**.

Resource monitor

With `pfc.dirstats` on, the cache tracks the state and the traffic of every directory it holds. Turning that into something an operator can use is the unfinished half.

RESOURCE MONITOR

What the cache already knows

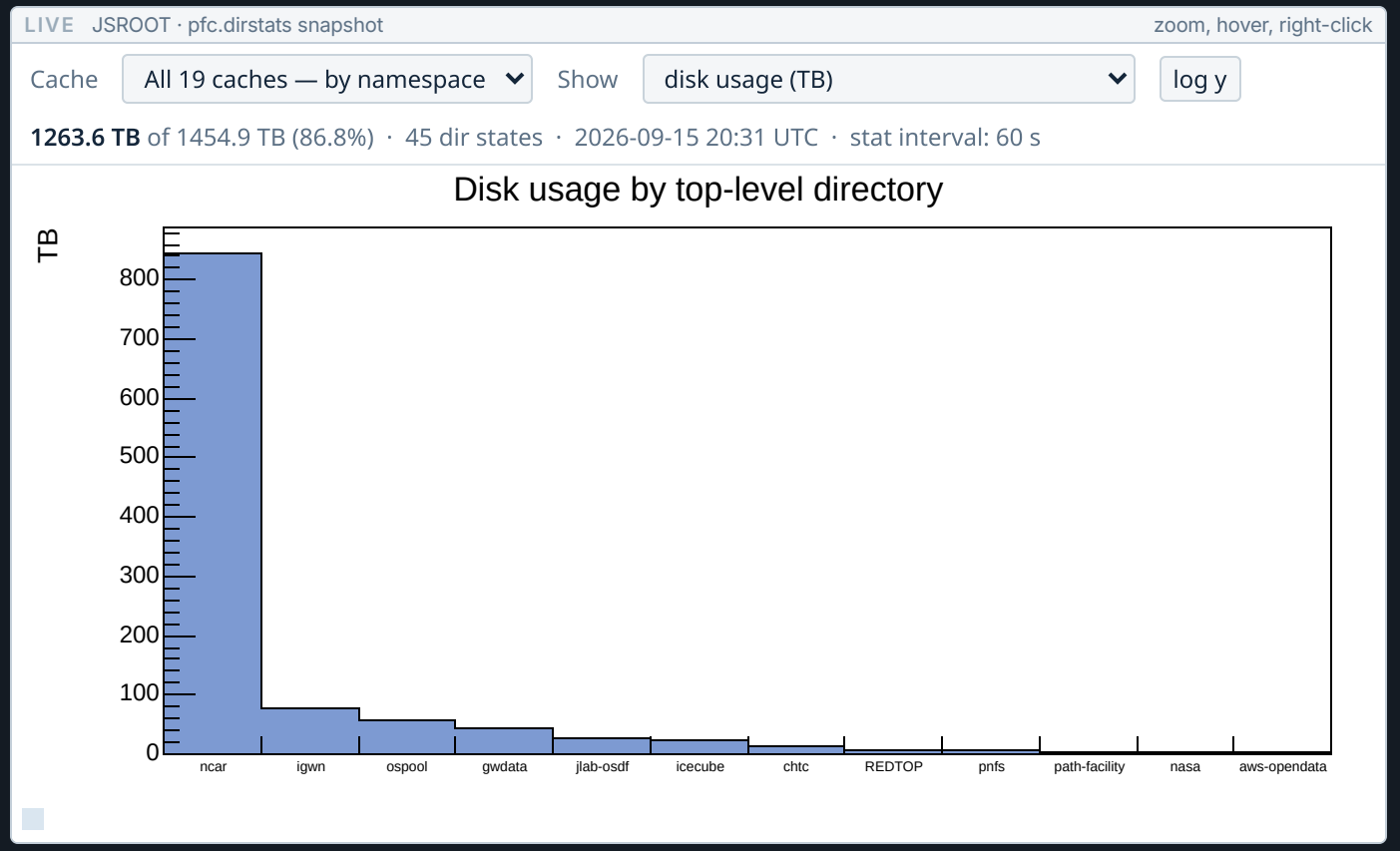
SHIPPED IN 5.8

- Every directory, state **and** traffic
- Updated from event queues **every 10 s**
- Exported as JSON on a quantised interval
  - one of eight values, 1 min to 1 h — **default 15 min**
- Drives the purge plugin

LIVE, REAL DATA — 19 OSDF CACHES, 1.26 PB

Read off the running pods on **Tuesday**. Every snapshot falls inside the same minute, because OSDF exports every 60 s.

For html slides, the plot and its menus are active: it is drawn in the slide, and **the aggregation is done in the browser**. The merger does not exist yet.



The problem: snapshots are not history

- A snapshot is two things: **state** as of the dump, and **traffic** since the previous one
  - which is exactly what makes them mergeable — state takes the newest, deltas add
- What is missing is that nobody keeps them. You cannot
  - look back past the last interval — the counters are cleared
  - average over a window you choose
- Keeping every JSON is not the answer — you parse a whole snapshot to read one directory's counter

What it needs

- A binary snapshot format alongside the JSON — the payload is already POD; the **names are the only thing needing work**
- An external merger: roll up to 15 min, 30 min, an hour, a day — and across a cluster
- **Publish them through the cache itself** — it is already a file server, with the site's own auth
  - the merger is a regular client

Latest fixes and upcoming changes in Resource Monitor

Three memory-safety bugs, found and fixed FIXED

#2808	hit while writing a LotMan purge plugin, on 5.9.5 — `std::length_error` in `perform_purge_check()`
#2934	found here while investigating it — use-after-free on forced eviction
no issue	found in the accounting work — `AccessToken&` left dangling when the token store reallocated

- **PR #2935** open; every commit builds alone, 19/19 CI green
- Ten further accounting fixes behind it, going up as **one second PR**
  - `maxdepth` honoured, `st_blocks` no longer lost on reap, wider IO counters

> THIS IS THE LOOP WORKING
>
> The first was filed by the **Pelican team**, writing their purge plugin for LotMan. Chasing it turned up the second; the accounting work turned up the third. All fixed, with regression tests.

Cache control & consistency

HEP namespaces are effectively immutable. Almost nobody else's are — including, it turns out, our own user directories.

The problem: is the cached copy still the same file?

- XCache assumed LFNs are unique and content never changes. `/store/user/` breaks both, and so does most non-HEP data
- We are also **federated**: many replicas, one namespace
  - the cache keys on the LFN and records **no origin URL**, so it revalidates against whichever replica it is sent to
- HTTP gives the transport, `ETag` and `Cache-Control`. The hard part is **what goes into the ETag** — it has to identify the **file**, not the copy in front of you

What shipped in 6.0 — for HTTP origins

Origin side: `XrdHttp` behaves like an httpd

- Returns `Etag` on HEAD, like any httpd
- Every OSDF / Pelican origin is an HTTP origin, so this is the deployed case

Cache side

- `pfc.httpcc` — ETag and max-age as cinfo xattrs; **no origin contact while `max-age` holds**
- `pfc.qfsredir` — when the origin is a redirector, ask the **data server** it points at, not the redirector (default **on**)
- Two new `kXR_query` subcodes: **`kXR_QFinfo`** on a file, **`kXR_QFSinfo`** on a path

Origin-agnostic — **the cache asks the same question whatever the origin speaks**. An xroot origin could answer it too, once there is a validator worth returning.

> **AND NOBODY EMITS `CACHE-CONTROL`**
>
> `XrdHttp` sends `Etag` and `Age`, never `Cache-Control` — so `max-age` and `must-revalidate` have no origin-side source on an XRootD server at all. Real httpds only emit them when an admin says so — `mod_expires`, configured per directory. It is not automatic there either.

CACHE CONTROL

What the ETag actually is, in `XrdHttp`

```
// server, StatGen(): dev+ino
// packed into one long long
Dev.id.lo = buf.st_ino;
Dev.id.hi = buf.st_dev;
snprintf(buff, len, "%lld %lld %d %lld",
         Dev.uuid, fsz, flags, st_mtime);
```

```
// XrdHttp reads that first
// field off the HEAD reply
sscanf(stat, "%lld %lld %ld %ld",
       &etagval,   &filesize,
       &fileflags, &filemodtime);
```

```
// addETagHeader(),
// the whole function
headers += "Etag: \""
        +  std::to_string(etagval)
        +  "\"";
```

- **We meant to add a timestamp and didn't** — found that writing this talk. Size and mtime were already parsed, so inode-size-mtime is a three-line change: **written, not yet a PR**
- What it was: `(st_dev, st_ino)` — **a storage handle**, carrying no property of the file at all
- Written through in update mode, same inode ⇒ **same** ETag ⇒ stale data served — **that one is now closed**
- Identical replica on another server ⇒ **different** ETag ⇒ a needless 10 GB re-fetch

WHAT THAT MEANS FOR A FEDERATION

Size and mtime are still **local**. What a cache needs is a property of the **object**, not of the copy — derived from the content, or assigned by whoever produced it — and wanted only on the paths that need it, `/store/user/*` rather than everywhere.

DEPLOYMENT

The cache side is **6.0-only**: new virtual methods on `XrdOucCache`, so it cannot be backported. The `XrdHttp` ETag is self-contained and could go back on its own.

Testing

The cache had a tokenizer unit test and one ctest. Now it has seven cache configurations and cinfo coverage, running on every pull request.

TESTING

The cache has CI MERGED

- **#2932**, merged 10 Sept: seven ctest cache configurations plus cinfo unit tests
  - 20 files, +2 097 / −5; before it, coverage was **one unit test and one end-to-end test**
- Covers: hit/miss paths, eviction, g-stream records, block sizes, `.cinfo` round-trips
  - `XrdPfc` **line coverage 63%**, against 53% for `src` as a whole
- Every resource-monitor fix since then arrived **with** a test that fails without it

Still missing

- The **clustered** cache — needs a cmsd setup the one-server harness cannot express
- A cache-control scenario: origin + cache + ETag invalidation, end to end
  - which would let the feature be validated without anyone deploying 6.x

CLOSING

Where things stand

Done

- Cache control, `QF[S]Info` — **6.0**
- Per-file blocksize / prefetch — **5.9**
- Cache CI — **merged**
- Resource-monitor fixes — **PR open**

Written, waiting

- Block runs + `pfc.iosize` / `iogap`
- `kXR_pgreadv`
- Async `do_ReadV` — needs a design call

Where somebody could join

Area	What is already there
Resource monitor — aggregation and presentation	snapshot structs split out for a standalone merger; the panel on slide 15 is that view, hand-built in a browser
AF performance — benchmarking and testing	branches written and measured, but on one origin and one client
Prefetching	least clear of the three — see below

On prefetching: we do not think heuristics belong in the cache at all — it cannot see what the job will ask for next. **Pre-read** and **fill-read** are the shapes that might work. Disagreement welcome, with a reason.

With thanks to **Pelican / OSDF**, the **CMS analysis facilities**, **Andy** for the protocol work, and **D. Kovalskyi** for the kick and an independent re-test.

CLOSING

Getting in touch

- Weekly **xcache-devops**, Thursday 11:00 Pacific
  - `xcache@opensciencegrid.org`, Slack `OSG#xcache`
- `xrootd-l@slac.stanford.edu` · `github.com/xrootd/xrootd/issues`