SDC 18
September 24-27, 2018
Santa Clara, CA
www.storagedeveloper.org

Using persistent memory and RDMA for Ceph client write-back caching

Scott Peterson, Senior Software Engineer
Intel

Ceph Concepts

Ceph from an application's POV

How Ceph places and replicates objects

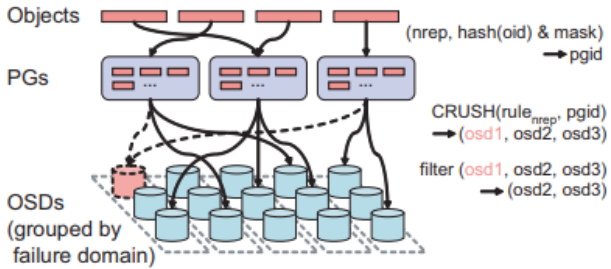


Figure 6.2: Objects are grouped into *placement groups* (PGs), and distributed to OSDs via CRUSH, a specialized replica placement function. Failed OSDs (*e. g.* `osd1`) are filtered out of the final mapping.

https://ceph.com/wp-content/uploads/2016/08/weil-thesis.pdf

Block storage in Ceph

RADOS Block Device (RBD)

- RADOS is the underlying Ceph object store
  - Scales to 100's of nodes, and 1000's of OSDs
  - Per-pool (OSD grouping) replication/EC policy
  - Objects and replicas relocated or replaced on OSD add/fail/remove
- RBD volumes are a series of objects
  - Typically 4M
  - Replication/EC determined by the OSD pool that contains the volume
  - Thin provisioning (discard), snapshots, and cloning at object granularity
  - RBD clients map LBAs to objects, and perform remote object IO
- Performance implications
  - Lots of devices usually improves throughput
  - Meeting volume latency targets requires all OSDs to meet that target
  - Average e2e latency and variability can be a problem

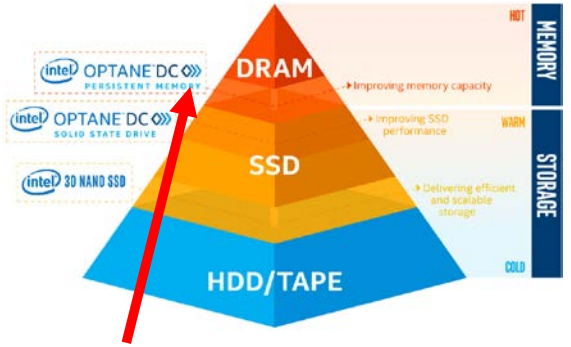
Intel RBD client caching work

Improving RBD performance with fast client-local storage

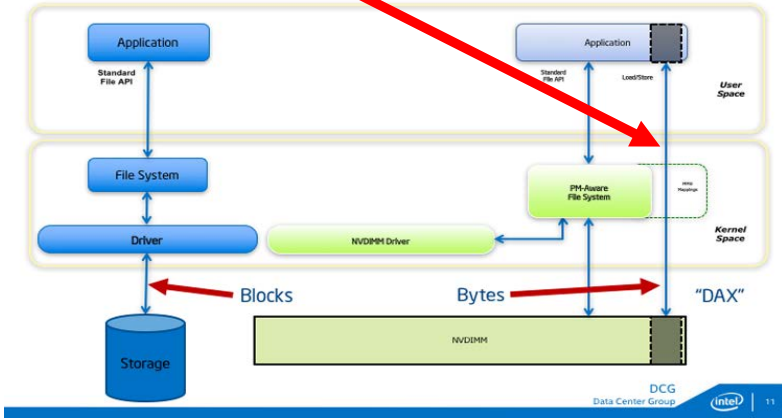
- Shared read cache (https://github.com/ceph/ceph/pull/22573)
  - Objects in snapshot parents are immutable
  - Cache these in local SSD, share with all open child images
  - Uses object dispatch interface (bottom edge of librbd)
- Write back cache (https://github.com/ceph/ceph/pull/24066)
  - Block writes complete when persisted to local Optane DC
  - Guarantee writes flush to RADOS eventually
  - Uses ImageCache interface (top edge of librbd)
- Replicated write back cache
  - Mirror Optane DC persistent memory to another node over RDMA
  - Block writes complete when persisted in both pmem devices

Persistent memory attributes

- Mapped into application's address space
  - Direct load/store -
- Byte addressable
- Can DMA/RDMA to/from pmem
  - Remote persistence still evolving
  - Platform variation hidden by PMDK
  - FSDAX support not here yet
    - RDMA to fsdax a few kernel revs away
    - Devdax has fewer issues, but is less convenient for user mode apps to use



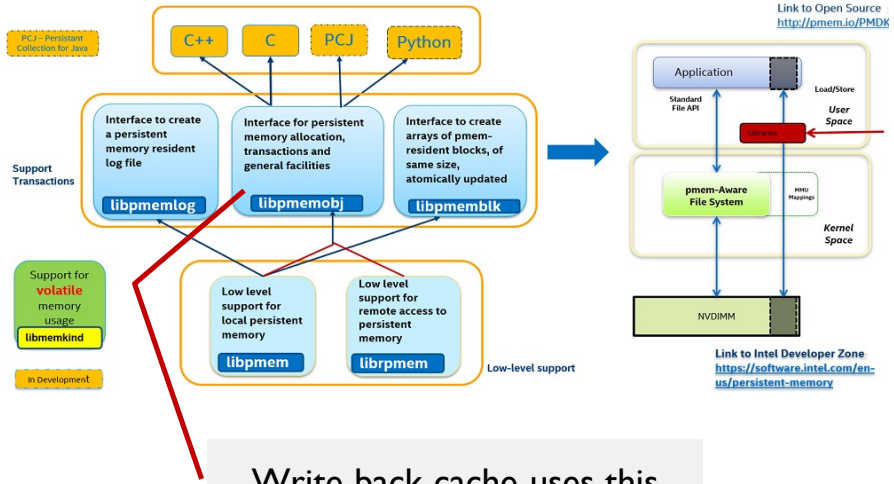
This is what we're using.
This is how we're using it.



Persistent memory programming

- Almost like DRAM
  - But apps need to think about failure
- Manage change visibility
  - New object and pointer to it can't appear without the other
    - Libpmemobj reserve/publish helps here
  - Use PMDK transactions
  - Visibility includes persistence
- Still need runtime locking
  - Transactions aren't locks

PMDK overview
(https://pmem.io/)

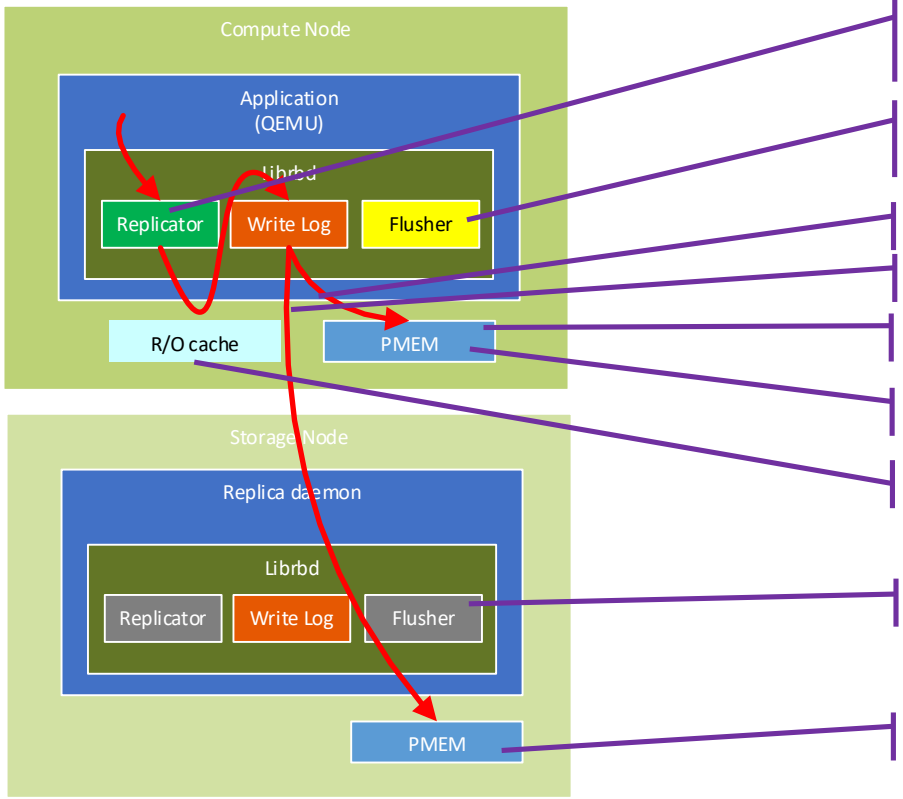


Write back cache uses this

Why use persistent memory for cache?

- Caching is metadata intensive
  - Write, Write Same, or discard? Length? Offset?
  - Pmem is happy with small writes. SSDs, not so much.
- PMDK enables transactional changes and allocation
  - Multiple changes to persistent structures all persist, or none do
  - Variable sized write buffers, can be freed in any order
- We can replicate persistent memory via RDMA
  - PMDK applies all writes and transactions to local and remote
    - This “pool replication” feature is still experimental
  - All-hardware replica write path possible

Replicated Write Log components



Source of writes for RBD volume (e.g. "image1"). Only one at any time.

Active flusher (flush RBD writes through cache below to RADOS)

Buffer & log entry writes (multiple steps)

RDMA to replica an each persist step

DAX capable filesystem (e.g. /pmem0)

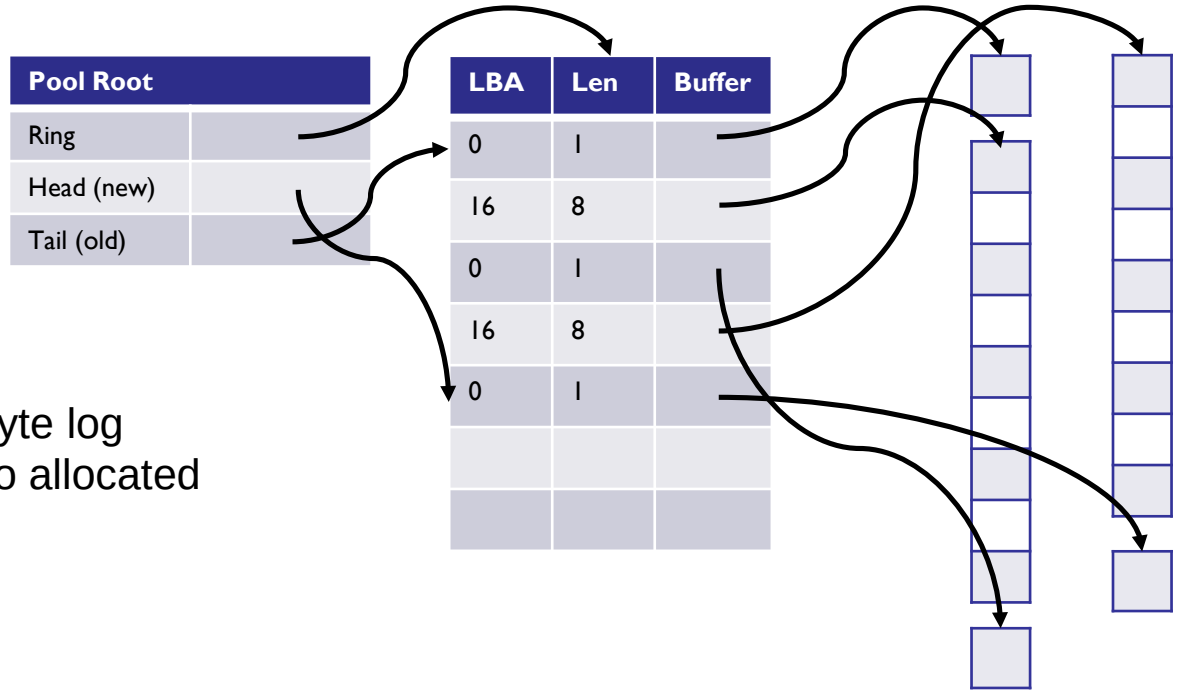
Pool file (e.g. /pmem0/rwl-image1.pool)

Other caches layered below

Standby flusher

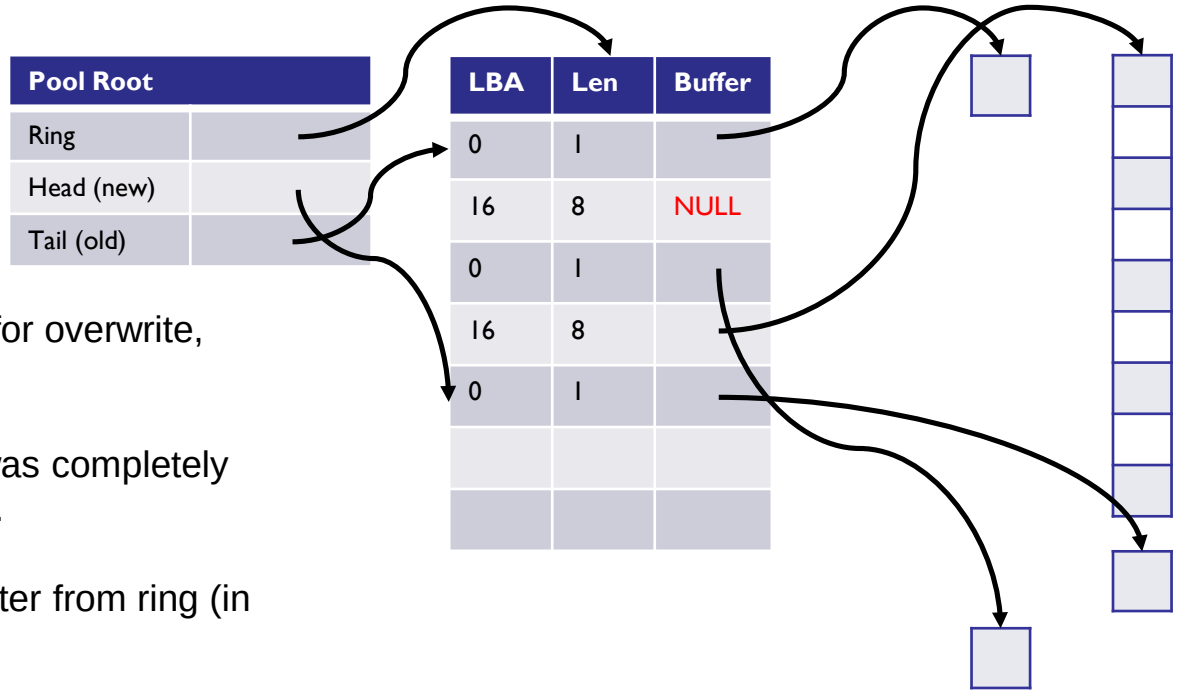
Replica pool file (e.g. /pmem0/rwl-image1.pool)

RWL pmem structures



Fixed size ring of 64-byte log entries, with pointers to allocated data buffers

Separate buffers enable out or order retire



Out of order free possible for overwrite, discard, etc.

Here the write to LBA 16 was completely overwritten by a later write.

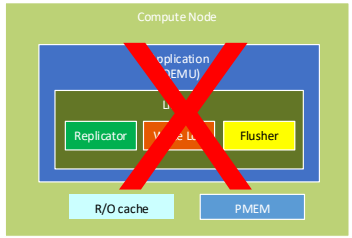
Free buffer and NULL pointer from ring (in a transaction)

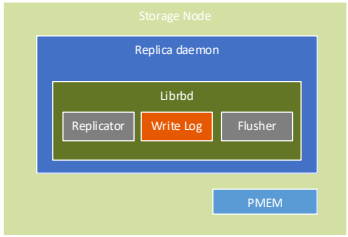
Basic steps in an RWL write

- Application calls rbd_aio_write()
- Writes that overlap with in-flight writes are deferred
- Reserve data buffer (non-persistent local operation)
- Copy and persist/replicate payload data
- Schedule log entry append, which determines the write order
- Copy, persist/replicate a group of ready log entries
- In a transaction
  - Advance ring head to include appended group
  - Publish payload buffer allocations for all appended entries
- Schedule completion callbacks for appended entries

Failure scenario: Writing node dies

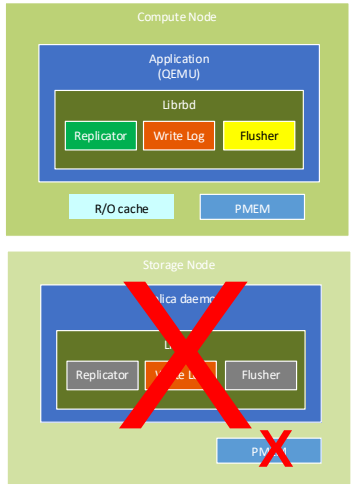
- Previously appended writes will be readable and flushed
  - Flushed either on failed node reboot, or by the replica node
- Writes that didn't finish appending will not be readable
  - If the append transaction doesn't complete, those writes didn't happen
  - Entries may have been persisted, but the head pointer won't include them
  - Payload buffers may have been persisted, but no valid entry points to them and their allocation was never published





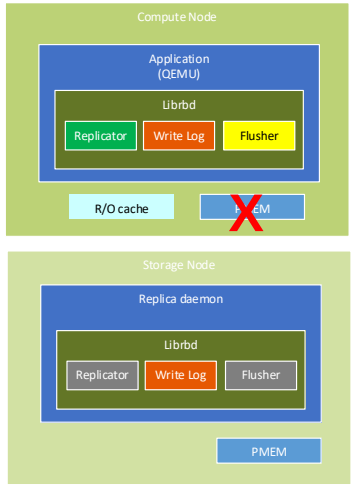
Failure scenario: Replica node or replica NVDIMM dies

- Writing node proceeds without it
- New replica chosen and resilvered
- Failover not implemented
  - PMDK pool replication can't currently be changed on the fly
  - Pool replication is still experimental



Failure scenario: Writing node NVDIMM dies

- If that means you get a machine check …
  - Game over for writing node - same as node failure
  - If we know this is true, we can exploit it
    - In persist-on-flush mode writes can complete before persist & replicate
- If the writing node survives …
  - If there's another local NVDIMM …
    - Resilver replacement local replica there
    - Restart in-flight writes with new poolset (must still have caller's buffer)
  - If there's no local NVDIMM …
    - Flush from replica, disable RWL, restart in-flight writes, and continue w/o cache
  - We could also use a local pmem replica, but haven't tested that yet



Performance goals and measurements

- RWL target use case is a typical VM volume
  - VM throttled at 2000 4K writes/sec per volume
  - QD8, barriers every 128 writes
- Goal: complete 99.99% of writes in <3mS

Test setup

- 3 Ceph nodes, 12 NVMe OSDs
  - 4 NVMe SSDs each
  - Single 25G Ethernet
  - 2 sockets, 96 cores, 377G
- Separate client node, with pmem
  - 2 sockets, 112 cores, 188G, same 25G Ethernet
  - EXT4 on a 3DXpoint DIMM in fsdax mode
- A version of fio that supports RBD barriers (calls rbd_aio_flush())
- Pmem pool for each RBD volume is 1G
  - PMEM_IS_PMEM_FORCE=1 (not safe for production)
  - That's 2x or 4x what these need, but RWL can't go any smaller yet

Measurements

Volumes	1 (100G)		16 (10G each)	
With RWL	No	Yes	No	Yes
Average	2.2mS	165uS	4.3mS	127uS
90%	2.4mS	163uS	7mS	126uS
99%	3.8mS	326uS	21mS	289uS
99.9%	5.9mS	1.06mS	40mS	783uS
99.99%	22.4mS	2.3mS	53mS	2.2mS

Fio args: --rw=randwrite --bs=4k --direct=1 --iodepth=8 --rate_iops=2K --rate_cycle=5000 --time_based=1 --runtime=600 --ramp_time=20 --write_barrier=128 --rate_process=poisson

Summary and Call To Action

- RWL improves latency for writes. 99.99% <3mS, vs. 20mS without.
- Persistent Memory (e.g., OptaneDC) coupled with high speed fabric gives best possible RWL latency
  - Essentially zero CPU overhead on the replica side
  - Lab experiments show promising results with RDMA replication
- RWL patch is in review now
  - Replication possible with that code, but not feature complete
    - Manual setup, custom kernel, no failover
  - Can be used now without replication.
    - Lower performance until fsdax is fully baked
  - Try it. Review it. Let us know how you'd use it.

Thank you

- Shared persistent read-only cache
  - https://github.com/ceph/ceph/pull/22573
- Replicated Write Log
  - https://github.com/ceph/ceph/pull/24066
- FIO with RBD barrier support (rbd_aio_flush() on barrier)
  - https://github.com/sdpeters/fio/tree/rbd-barriers
- PMDK
  - https://pmem.io/
  - Pool replication (experimental): https://goo.gl/VHz7Wi