Note on Benchmarks

► Benchmarks from several sources, primary system:
   DDN AI 400
   1 x Client(1 x XeonGold 6338, 512GB DDR4 3200MHz, 1 x IB-HDR200, CentOS8.3, MOFED-5.2)

► Best case numbers: IOR with 256 MB I/O size, 4M stripe size
Single Stream Performance: Definitions

▶ “single stream”: The I/O output of a single userspace process using standard POSIX interfaces
▶ “How fast can dd go?”

Interesting because:
- Foundation of other performance behavior
- Behavior of one stream creates (or prevents) scalability across many streams
- Many activities have single stream portions
Single Stream Performance: Where We Started

Pre Lustre 2.15 (2.12, 2.14...)

4 GiB/s
Single Stream: Current Performance

- This is **best case**, any I/O size, any stripe/RPC size, etc.
- Limited to ~1.5-2.0 GiB/s for buffered or direct I/O (except for buffered reads)
- Has only increased with CPU speed since ~2012
- NB: Buffered reads 3.5-4.0 GiB/s with parallel readahead

- Not that fast – GPFS is faster, and some object stores much faster
- Why don’t we do better? Buffered I/O is hard, but what about direct I/O...?
Direct I/O: Simple

- User provides aligned memory
- No need for memcopy() or allocation of pages in the kernel
- No page cache – don’t have to insert and manage pages
- Much simpler than buffered I/O, much more scalable with multiple processes
- Expected to be synced to disk after write call completes \( \Box \) sync is costly, but makes for simple I/O lifecycle
Direct I/O: Simple, not fast(?)

► Small direct I/O performs badly
  • cost of sync() is painful for writes
  • no readahead possible for reads (because no cache)
► But ... what about large Direct I/O?
► If a user provides (or asks for) a large amount of data, why can’t we write or read that data quickly?
► There’s no cache to fill, so we should be able to process more rapidly than for buffered I/O
► But Lustre direct I/O doesn’t scale with size.
Direct I/O is Serial(!)

► It turns out Direct I/O RPC issuance is serialized with each RPC sync()'ed before sending others(!)
► Example: User does 16 MiB I/O, Lustre using 4 MiB RPCs:
  Prepare 4 MiB RPC  wait for sync()  prepare 4 MiB RPC  wait for sync() ... etc. (Read is similar)
► Time to write (or read) data:
  Prep RPC + sync + prep RPC + sync ... = n*(prep RPC + sync), where n is # of RPCs
► Zero parallelism(!)
Parallel Direct I/O

- Prepare RPC send Prepare RPC send ... sync() after all data is sent.
- Send all RPCs and *then* wait. For the 16 MiB I/O and 4 MiB RPCs, we send 4 RPCs.
- Time is:
  rpc ... rpc ... rpc ... rpc
  sync ...........
  sync ...........
  sync ...........
  sync ...........
- Time = n*(create RPC) + sync*1 (all sync()s are in parallel)
Performance with Parallel DIO

Parallel DIO

- **Buffered Write**
- **Buffered Read**
- **DIO Write**
- **DIO Read**

- **2.14**
- **2.14 + PDIO**

<table>
<thead>
<tr>
<th>Operation</th>
<th>2.14</th>
<th>2.14 + PDIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffered Write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffered Read</td>
<td>4 GiB/s</td>
<td></td>
</tr>
<tr>
<td>DIO Write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIO Read</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 GiB/s
Parallel Direct I/O: First results

- Results: 4.5-5.0 GiB/s best case (compare to previous 1.5-2 GiB/s)
- This is great! But ... can we do better?
- The answer is yes – very much so.
Direct I/O Code Efficiency

- Direct I/O code was never made efficient - not visible because all time spent waiting for `sync()` (so more efficient direct I/O code just spent more time waiting for `sync()`)
- Much code shared with buffered (i.e., page cache) path – careful page management for caching/concurrent access
- Every page in the page cache has an independent life – can be accessed, updated, or removed by itself, at any time
- Managing page state is expensive – set up, refcounting, locking...
- None of this is required for direct I/O
Direct I/O Code Efficiency

- Direct I/O pages are not accessible to other threads – they only exist during the I/O, and are not in cache.
- No independent life cycle for each page, so (almost) no per-page:
  - Locking
  - Refcounting
  - State management
- There is not *zero* management required for direct I/O pages – but it’s close.
- Many small changes to take advantage of this in 2.15... Cuts per page time by ~70%.
Changes (Examples from LU-13799)

- lov: Cache stripe offset calculation
- llite: Move free user pages
- llite: Implement lower/upper aio
- osc: Always set aio in anchor
- llite: Simplify cda_no_aio_complete use
- osc: Improve osc_queue_sync_pages
- clio: Skip prep for transients
- llite: Adjust dio refcounting
- lov: Improve DIO submit
- llite: Remove transient page counting
- llite: Modify AIO/DIO reference counting
- osc: Simplify clipping for transient pages
- clio: Implement real list splice
- osc: Don't get time for each page
Where We Are

Lustre 2.14, Lustre 2.15

- 2.14
- 2.15

- **Buffered Write**
  - Lustre 2.14: 4 GiB/s
  - Lustre 2.15: 40 GiB/s (10x)

- **Buffered Read**
  - Lustre 2.14: 4000 MiB/s
  - Lustre 2.15: 40000 MiB/s (10x)

- **DIO Write**
  - Lustre 2.14: 1000 MiB/s
  - Lustre 2.15: 10000 MiB/s (10x)

- **DIO Read**
  - Lustre 2.14: 1000 MiB/s
  - Lustre 2.15: 10000 MiB/s (10x)
What’s left?

► Prototype changes to increase batching (many things only need to be done per I/O)
► Changes to remove more page state tracking
► Various small simplifications and code removals
► Some big stuff left, some element of diminishing returns...
2.15, 2.16+: Direct I/O

- Currently: ~18 GiB/s (slightly better than graph shows)
- Existing prototype changes to ~25+ GiB/s (2.16?)
- Hard to say final limit. Would prefer not to speculate. Still some headroom.
- Other benefits:
  - Direct I/O is lockless(!)
  - Improves shared file writes
Buffered I/O: Lagging behind

Lustre 2.15: Buffered vs Direct

<table>
<thead>
<tr>
<th></th>
<th>2.14</th>
<th>2.15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffered Write</td>
<td>0x</td>
<td>0x</td>
</tr>
<tr>
<td>Buffered Read</td>
<td>0x</td>
<td>0x</td>
</tr>
<tr>
<td>DIO Write</td>
<td>10x</td>
<td>10x</td>
</tr>
<tr>
<td>DIO Read</td>
<td>10x</td>
<td>10x</td>
</tr>
</tbody>
</table>
Performance with I/O Size

Performance with I/O Size (Write)

- 2.14 Buffered
- 2.14 DIO
Buffered I/O vs Direct I/O

► Buffered I/O is good at small sizes (aggregation, readahead)
► But doesn’t scale – Direct I/O dominates at larger sizes and in shared files
► Buffered I/O doesn’t scale because of costs of caching (interestingly, not memcopy – caching)
► Direct I/O must be aligned – buffered I/O handles alignment inside the kernel (data is still aligned before going on the wire – just done by the kernel)
► But most data is used once: Read once, or written but not read (at least not in the same job)...
Uncached Buffered I/O

► An ‘uncached’ variant of buffered I/O, where data is copied to a buffer (but not placed in cache), would be much faster at larger sizes
► Switch to this at larger sizes
► Essentially create a buffer and do direct I/O from that buffer
► Saves expense of placing data in the page cache
► Not as fast as Direct I/O, but much faster than regular buffered I/O (50+% of direct I/O?)
► We can do it – prototyped successfully. (2.16+?)
Wrap Up

- Direct I/O is serialized at the RPC level, and (it turns out) very inefficient
- 2.15: Direct I/O single stream performance from ~2.0 GiB/s  18 GiB/s
- Future expectations: 20+ GiB/s
- Because Direct I/O is lockless, reduces/removes shared file contention(!)
- Requires using Direct I/O, requiring alignment, has poor performance at smaller sizes

Future (2.16+):
- Buffered I/O: Cost is mostly in **caching**, not memcopy
- Possible to make a buffered/direct hybrid path: Use buffered at smaller sizes, use new **uncached** buffered I/O at larger sizes
- Much more scalable than existing buffered I/O
- Early prototype
Thank you

► Thank you for listening.
► See LU-13798, LU-13799 and linked tickets for further details.
► Questions to pfarrell@whamcloud.com

► Quick thanks to Nathan Rutman, Shilong Wang, and Andreas Dilger for assistance and support on this