OpenZFS New Features Including Direct I/O and Compression/Erasure Offloads

Brian Atkinson, Jason Lee - LANL
Kelly Ursenbach - Eideticom

May 22, 2023

LA-UR-23-25110
Agenda

• Discuss OpenZFS developments to improve performance with NVMe Devices

• There are two parts to this BoF
  – Part 1: Addition of O_DIRECT to OpenZFS
  – Part 2: Computational Storage Offloads to OpenZFS
Why Does LANL Care about ZFS?

• One of two available backing FS’s for Lustre
• Open sourced
• High integrity (Flag in the ground)
  − Erasure coding (raidz)
  − Mirrors
  − Checksums
  − Snapshots
• Feature rich
  − Compression
  − Dedup
  − Encryption
• ZFS traditionally has good performance with HDD
Why are we focused on NVMe Device Performance with ZFS?

<10 of PiBs of Flash

Compute/Clients
BB/PFS Routers
I/O Backbone (Infiniband)

<5 PiBs of DRAM
Current Generation PFS

~100 of PiBs of HDD

Lustre OSS
LDiskFS/ZFS-based OST
1.2 TB/s

Lustre MDS
LDiskFS/ZFS-based MDT

Current Generation PFS
Why are we focused on NVMe Device Performance with ZFS?

Next Generation PFS

[PiBs of Advanced Memory]

[10-50s PBs of Flash]

[100s of PB of HDD]

[100s of PB]

Hot

Warm

Cold

Compute/Clients
I/O Gateways (optional)
I/O Network (RoCE/InfiniBand)

Lustre Servers
Specialized S/W Stacks
FS Gateways

misc

NVMe Fabrics Enclosures

Campaign Storage

Tape Archive

400G Eth
ZFS gen3 NVMe Zpools Sequential Read Performance

Throughputs of 1MB Reads For Multiple Files
Using Disk and Raidz VDEV's with 12 Samsung PM1725a NVMe's

48%-57% Missing  70% Missing
Where did all that read performance go?

Memory Copies
ZFS gen3 NVMe Zpools Sequential Write Performance
Where did all that write performance go?
Motivation: Why would you want to bypass the ARC? Have we lost our minds?!

- Pass O_DIRECT flag in open() call
- From the Linux man page for open():
  - Try to minimize cache effects of I/O to and from this file... File I/O is done directly to/from user-space buffers.
- Loose rules around exact semantics
- Currently ZFS happily accepts the O_DIRECT flag but silently ignores it
- Direct I/O is not the solution to all problems (The ARC still important)
- Databases – Own caching mechanisms
- Writing large amounts of data not intending read (checkpoint)
- Reading large amounts of data once (restart)
- **Using low latency, high bandwidth devices (NVMe) as VDEVs**
  - Average 1.5 – 3x Speedup with Direct IO support added to ZFS
Implementation: Direct I/O
Read in ZFS (Big Picture)

Buffered
- Cached? → Copy from ARC
- Issue to ZFS pipeline
  ▪ Copy to ARC
  ▪ Copy to user buffer

Direct I/O
- Bypass ARC
- User pages are directly mapped
Implementation: Direct I/O Write in ZFS (Big Picture)

Buffered I/O Write Path

Direct I/O Write Path

Transaction Group (TXG) Lifecycle

Sync Phase

Memcpy
Userspace -> Kernel space

Return Back To Write Call

Map
Userspace -> Kernel space

User Buffer

ARC Buffer

Open Context

Quiesce

Sync

User Buffer

ZIO Pipeline

Issue Write to VDEVs

User Buffer

ZIO Pipeline

Issue Write to VDEVs

Return Back To Write Call

Sync
Implementation: General Details

• O_DIRECT does not imply O_SYNC
• Must be PAGE_SIZE aligned if using O_DIRECT flag
  − Returns EINVAL
• Mmap’ed files will **silently** ignore direct I/O and used buffered
• Direct I/O accounting:
  − Linux: /proc/spl/kstat/zfs/poolname/iostats
  − FreeBSD: sysctl kstat.zfs.poolname.misc.iostats
• Direct I/O requests are issued with sync priority
  − ZIO_PRIORITY_SYNC_{READ/WRITE}
• ARC is coherent with O_DIRECT
Caveat 1: Direct I/O Write

Stale data in ARC is removed or synced out (ARC Coherency)

Diagram:
- ZPL
- DMU
- ARC
- ZIO Pipeline
- VDEV’s
- Open Context (OC)
- Data buffer

Flowchart:
1. Dirty data in ARC?
   - Yes: Previous Dirty Records?
     - Yes: Wait for Dirty Records To Sync
     - No: Update BP
   - No: Update BP

Sync Phase (S)
Caveat 2: Direct I/O Write Continued…

- Must be recordsize aligned to avoid Read/Write/Modify
  - If not O_DIRECT flag silently ignored
ZFS Seq. Read gen3 Performance Results

Throughputs of 1MB Reads For Multiple Files Direct IO vs Buffered with Striping and Raidz Using 12 Samsung PM1725a NVMe SSD's

Percentage Total Device Bandwidth with O_DIRECT
Seq. Write gen3 Performance Results

Throughputs of 1MB Writes For Multiple Files Direct IO vs Buffered with Striping and Raidz Using 12 Samsung PM1725a NVMe SSD's

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Stripe: 99%</th>
<th>Raidz1: 93%</th>
<th>Raidz2: 92%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage Total Device Bandwidth with O_DIRECT</td>
<td>1.5x</td>
<td>1.75x</td>
<td>1.66x</td>
</tr>
</tbody>
</table>

Stripe: 99% Raidz1: 93% Raidz2: 92%
Percentage Total Device Bandwidth with O_DIRECT
ZFS Seq. Read gen4 Performance Results

Throughputs of 1MB Direct IO and Buffered Reads For Multiple Targets Using 2 Zpools each with Raidz VDEV and Stipling across 10 Disk VDEVs using 20 Samsung PM1733's NVMe SSD's

Stripe: 99%  Raidz1: 97%  Raidz2: 97%
Percentage Total Device Bandwidth with O_DIRECT
ZFS Seq. Write gen4 Performance Results

Throughputs of 1MB Direct IO and Buffered Writes For Multiple Targets Using 2 Zpools each with Raidz VDEV and Striping across 10 Disk VDEVs using 20 Samsung PM1733's NVMe SSD's

- Stripe: 2.84x
- Raidz1: 3.52x
- Raidz2: 3.0x

Percentage Total Device Bandwidth with O_DIRECT

- Stripe: 97%
- Raidz1: 92%
- Raidz2: 91%
<table>
<thead>
<tr>
<th>arcstat 1</th>
<th>zpool iostat –qv 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>zpool status and FIO runs</td>
<td>zpool iostats</td>
</tr>
</tbody>
</table>
Computational Storage Offloads for OpenZFS
ZFS gen3 NVMe Zpools Sequential Write Performance

Throughputs of 1MB Writes For Multiple Files
Using Disk and Raidz VDEV's with 12 Samsung PM1725a NVMe's

34% Missing
47% Missing
44% Missing
Stock ZFS Performance with Various Features Enabled

Throughputs of 1MB Writes For Single File Using ZFS Raidz2 (10+2) Using NVMe-oF from Host to Target

- EC Only
- EC + CKSM
- EC + CKSM + LZ4
- EC + CKSM + GZip

- < 1% Perf. Loss
- 17% Perf. Loss
- 94% Perf. Loss!
What can we do to improve performance?

• Use computational storage to offload operations
  − Perform operations that are CPU/memory bandwidth intensive when run on host
  − Can be implemented with FPGAs
  − Data Processing Unit (DPU)

Eideticom Noload
Computational Storage Processor (CSP)

NVIDIA BlueField2 DPU
Doesn’t ZFS already support offloading?

- Intel® QuickAssist Technology (Intel® QAT)
  - Doesn’t work on AMD machines

- Requires ZFS to be reconfigured

- Each offload operation is done independently of each other
  - Encryption – AES-GCM
  - Compression – GZIP
  - Checksum – SHA256

- Not extensible
  - API updates need to be merged upstream
Accelerated ZFS with Disaggregated Storage

Host System

CPU
User App
VFS
ZFS + Z.I.A.
DPUSM
Provider

Accelerated NVMe-oF Enclosure (ABOF)

CPU/DPU
Accelerator
NVMe Controller
NVMe
Accelerator

CPU/DPU
Accelerator
NVMe Controller
NVMe
Accelerator

Software Component

NVMe-oF CSS (ZFS Offloads)

NVMe Fabric

CPU/DPU
Accelerator
NVMe Controller
NVMe
Accelerator

NVMe I/O

NVMe I/O
Target side kernel module

- Tie into NVMe-oF module
- Use a set of vendor op codes to create/free buffer
  load/store buffer from disk
  read/write buffer
  Perform operation on buffer
    (compress/decompress/checksum/EC)
Z.I.A. Performance with Eideticom NoLoad CSP

Throughputs of 1MB Writes For Single File Using ZFS Raidz2 (10+2) with Z.I.A. Using NVMe-oF from Host to Target

16x Speedup
ZFS Interface for Accelerators

+ Data Processing Unit Services Module
ZFS Write Pipeline

- ZPL: ZFS POSIX layer
- DMU: Data Management Unit
- ARC: Adjustable Replacement Cache
- ZIO Pipeline: ZFS I/O Pipeline
- VDEVs: Virtual Devices
Z.I.A. Write Pipeline

ZPL

DMU

ARC

ZIO Pipeline

Z.I.A.

VDEVs

User API

Provider API

Accelerator API

DPUSM

Provider

Accelerator
Data Processing Unit Services Module (DPUSM)

• Kernel module

• Standardized APIs for leveraging computational storage
  – Provider API
  – User API

• Acts as registry for providers
Providers

- Kernel module
- DPUSM wrapper for accelerator specific code
- Does not know anything about user
- Declares what the accelerator provides
- Usually implemented by accelerator vendor
Provider Implementation Basics

• `#include <accelerator_header.h>`
• `#include <dpusm/provider_api.h>`
• Fill in DPUSM provider functions struct
  – Analogous to VFS function pointers
• Register provider with DPUSM on module initialization

1. Give user handle that references accelerator memory
2. Get user (in-memory) data into accelerator (copy, dma, etc.) via handles
3. Accept handles for operations

• Communication with accelerator is connection protocol agnostic
Using a Provider

• `#include <dpusm/user_api.h>`

• Find provider

• Use provider functions in DPUSM user functions struct

1. Get opaque handle (`void *`) to accelerator memory (wrapped by provider)
2. Get in-memory data to accelerator via handle
3. Pass handle(s) to provider functions to operate on data
ZFS Interface for Accelerators (Z.I.A.)

- Modifications to the ZFS **write** pipeline
- Transparent acceleration of CPU and memory intensive ZFS write operations with accelerators
  - Compression
  - Checksum
  - RAIDZ (Generation and Reconstruction)
  - I/O
- User data access not affected
  - During write
  - Afterwards

---

**Diagram Description:**

- **ZPL**
- **DMU**
- **ARC**
- **ZIO Pipeline**
- **VDEVs**
- **Z.I.A.**
- **DPUSM**
  - Provider API
  - Provider
  - Accelerator API
  - Accelerator
ZFS Write Pipeline

Host (CPU)

ARC Buffer → ZIO Compress → ZIO Checksum → VDEV RAIDZ → VDEV Disk File → NVME I/O
General Description of Changes

• If data is not offloaded at start of stage, offload it
• Run the operation
• Return status code (not data)

• If Z.I.A. fails, bring data back to memory, fall back to running operation in software
• If offloaded data cannot be returned to memory, restart write pipeline
  – A copy of the original data is still available in ZFS
Z.I.A. Write Pipeline

Host (CPU)

- ARC Buffer
- ZIO Compress
  - abd_zia_handle
- ZIO Checksum
- VDEV RAIDZ
  - rr_zia_handle
- VDEV Disk File
- NVME I/O

Accelerator

- Compress (Gzip)
- Checksum (Fletcher)
- EC (RAIDZ2)
- Issue I/O
Z.I.A. Performance with Eideticom NoLoad CSP

Throughputs of 1MB Writes For Single File Using ZFS Raidz2 (10+2) with Z.I.A. Using NVMe-oF from Host to Target

16x Speedup
Z.I.A. (RAIDZ) Resilver

Host (CPU)
- io_start
- Reconstruct Missing Data
- abd_zia_handle
- rr_zia_handle
- Success
- Verify Checksum
- Error
- zio_vdev_io_done
- vdev_raidedz_io_done
- Verify Parity
- Combinatorial Reconstruct Data
- Verify Checksum
- Issue I/O
- NVME I/O

Accelerator
- RAIDZ Reconstruct Missing Data
- Error
- Checksum
- Success
- RAIDZ Reconstruct
- Checksum
- RAIDZ Generate
- Issue I/O
Get Z.I.A.

- Z.I.A. Pull Request
  - https://github.com/openzfs/zfs/pull/13628
  - Comes with software provider that links back into ZFS

- Data Processing Unit Services Module
  - https://github.com/hpc/dpusm
Z.I.A. Demo
Backing Drives

• Direct attached

• NVMe-oF
DPUSM Setup

- `git clone https://github.com/hpc/dpusm.git`
- `cd dpusm`
- `make`
- `sudo insmod dpusm.ko`
Compile ZFS + Z.I.A.

- git clone -b zia https://github.com/hpc/zfs.git
- cd zfs
- ./autogen.sh
- ./configure --with-zia=${HOME}/dpusm
- make -j
Build and Load Provider

• Should not link with ZFS, so can build and load after ZFS is loaded

• Z.I.A. software provider is a special case
  – Depends on ZFS
  – Builds with Z.I.A.
  – Load after loading ZFS
    ▪ module/zia-software-provider.ko
Set up zpool

• Load ZFS
  - `scripts/zfs.sh`

• Limit ARC size
  - `echo 17179869184 > /sys/module/zfs/parameters/zfs_arc_max`

• Create zpool
  - `zpool create -o ashift=12 local_zpool raidz2 /dev/nvme0n{1..12}`

• Set up zpool properties
  - `zfs set recordsize=1M local_zpool`
  - `zfs set compression=gzip local_zpool`
  - `zfs set checksum=fletcher4 local_zpool`
Write Enough Data to Force Memory Pressure on ARC

- `fio zia_demo.fio`

- 16 target files x 4GB
  - 4x ARC size to force eviction

- ~1GB/s
  - Bottleneck is memory bandwidth due to compression, not I/O

```ini
[global]
name=zia_demo
direct=0
ioengine=psync
bs=1m
size=4g
fallocate=0
rw=write
buffer_compress_percentage=25

[job0]
filename=/local_zpool/file0

[job1]
...
```
Enable Offloading with Z.I.A.

• `zpool set zia_provider="athena_example_provider"` local_zpool

• `zpool set zia_compress="on"` local_zpool
• `zpool set zia_checksum="on"` local_zpool
• `zpool set zia_raidz2_gen="on"` local_zpool
• `zpool set zia_disk_write="on"` local_zpool

zia_compress
zia_decompress
zia_checksum
zia_raidz1_gen
zia_raidz2_gen
zia_raidz3_gen
zia_raidz1_rec
zia_raidz2_rec
zia_raidz3_rec
zia_file_write
zia_disk_write
Write Enough Data to Force ARC Flush (Again)

- `fio zia_demo.fio`

- Much Faster!
Come See Our Technical Talks!

- **Tuesday, May 23, 1:00 – 3:00**
  - Accelerated Disks and Flashes: LANL's early experience in Speeding Up Analytics Workloads Using Smart Devices
    - Qing Zheng
  - Computational Storage Solutions Over Fabrics for ZFS
    - Kelly Ursenbach

- **Wednesday, May 24**
  - 1:00 – 3:00
    - MarFS as a Multi-Level Erasure Archive
      - Garrett Ransom
  - 3:30 – 4:30
    - DNA Storage Erasure Encoding
      - Dominic Manno
• Questions?

• Comments?

• Test our pull requests
Addendum Slides
Implementation: Direct I/O Write Continued…

• Guard against users manipulating page data during write
Implementation: Direct I/O Write Continued…

- Added new module parameter zfs_vdev_direct_write_verify_pct
  - Required because anonymous pages on Linux can not be placed under write protection
  - Percentage of O_DIRECT writes to checksum verify before updating BP
  - Default set to 2%
  - zpool status –d can show if any O_DIRECT write checksum verify failures exist
  - Also logs ZED event dio_verify

```
pool: local_zpool
state: ONLINE
config:

  NAME   STATE  READ WRITE CKSUM DIO
local_zpool ONLINE 0 0 0 0
raidz1-0 ONLINE 0 0 0 0
nvme0n1 ONLINE 0 0 0 0
nvme1n1 ONLINE 0 0 0 0
nvme2n1 ONLINE 0 0 0 0

errors: No known data errors
```
Implementation: One last Note on Direct I/O in ZFS

• New dataset property added
  – direct=standard (default)
    ▪ Follows semantics outlined so far
  – direct=always
    ▪ Treat every read/write I/O request as Direct I/O (best effort)
    ▪ However if not PAGE_SIZE aligned fall back to buffered
  – direct=disabled
    ▪ Silently ignore O_DIRECT
Implementation: O_DIRECT vs O_SYNC

- O_DIRECT does not imply O_SYNC
  - Direct IO writes are written immediately to VDEVs
- O_SYNC will commit Direct IO block pointers (BP) to ZIL
Z.I.A. Usage (Admins)

• Currently need to reconfigure ZFS with --with-zia=<DPUSM Root>
  – Expect that ZFS will always compile Z.I.A. once merged
  – Z.I.A. will not cause issues if DPUSM is not found at load time

• Select a provider
  – `zpool set zia_provider=<provider name> <zpool>`

• Enable offloading
  – `zpool set zia_<property>=on <zpool>`
  – Offloading only occurs if the ZFS stage is enabled
Accelerated ZFS with Converged Storage

Linux Kernel Space

- ZFS + Z.I.A.
- DPUSM
- Software Provider
- BlueField 2 Provider
- GPU Provider
- libnoload

User Space

- Another Kernel Process

Non-Restrictive License

GPLv2
Status of O_DIRECT on OpenZFS master

- Open PR on OpenZFS master (#10018)
  - https://github.com/openzfs/zfs/pull/10018
- Needs further code reviews
- FreeBSD performance testing
- Direct I/O for ZVOLS
- Aiming for inclusion in major point release (December 2023)
Performance Results: NVMe Zpool Test Configuration

- **Server**
  - Gigabyte R272-Z32-00
  - Single Socket: AMD EPYC Rome 7502 32c (Hyperthreading 64c)
  - 128 GB RAM, 8x 16 GB 2933 MT/s DDR4
  - CentOS 8.2, Kernel: 4.18.0-193.el8.x86_64
  - 12 Samsung PM1725a 1.6 TB

- **IO Workload**
  - Request block size 1M
  - 2 TB of total data read/written to Zpool
  - Sequential Read/Write, 1 – 1024 Files

- **ZFS Settings/Configuration**
  - ZFS recordsize=1M
  - zfs_vdev_sync_{read/write}_max_active=64
  - zfs_vdev_async_write_max_active=64
  - 1 Zpool 12 NVMe Disk VDEVs (Striping)
  - 1 Zpool 1x Raidz1 (11+1) VDEV
  - 1 Zpool 1x Raidz2 (10+2) VDEV
  - 1 Zpool 1x dRAID1 (11d:12c) VDEV
  - 1 Zpool 1x dRAID2 (10d:12c) VDEV
Gen3 Seq. Read Performance Results: Raidz

Throughputs of 1MB Reads For Multiple Files
Direct IO vs Buffered with Striping and Raidz
Using 12 Samsung PM1725a NVMe's

Log Scale
Gen3 Seq. Write Performance Results: Raidz

Throughputs of 1MB Writes For Multiple Files
Direct IO vs Buffered with Striping and Raidz
Using 12 Samsung PM1725a NVMe’s

Log Scale
Gen3 Seq. Read Performance Results: dRAID
Gen3 Seq. Write Performance Results: dRAID

Throughputs of 1MB Writes For Multiple Files
Direct IO vs Buffered with Striping and dRAID
Using 12 Samsung PM1725a NVMe's

Log Scale
ZFS Seq. Read gen4 Performance Results

**Raidz**

- Stripe: 2.39x
- Raidz1: 2.51x
- Raidz2: 2.58x

*Percentage Total Device Bandwidth with O_DIRECT*

- Stripe: 99%
- Raidz1: 97%
- Raidz2: 97%

**dRAID**

- dRAID1: 2.42x
- dRAID2: 2.46x

*Percentage Total Device Bandwidth with O_DIRECT*

- dRAID1: 76%
- dRAID2: 78%
ZFS Seq. Write gen4 Performance Results

**Raidz**

- Percentage Total Device Bandwidth with O_DIRECT:
  - Stripe: 2.69x
  - Raidz1: 3.09x
  - Raidz2: 3.0x

- Stripe: 97%
  - Raidz1: 92%
  - Raidz2: 91%

**dRAID**

- Percentage Total Device Bandwidth with O_DIRECT:
  - dRAID1: 3.13x
  - dRAID2: 2.93x

- dRAID1: 73%
  - dRAID2: 72%