Z.I.A.: An Interface to Accelerate ZFS Features

Jason Lee

Oct 16, 2023

LA-UR-23-31173
Why Does LANL Care about ZFS?

- One of two available backings for Lustre
- Open source
- High integrity
  - 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

Lustre OSS

LDiskFS/ZFS-based OST

~100 of PiBs of HDD

Lustre MDS

LDiskFS/ZFS-based MDT

<5 PiBs of DRAM

Current Generation PFS

Compute/ Clients

BB/PFS Routers

I/O Backbone (Infiniband)

Compute/ Clients

3.2 TB/s

1.2 TB/s

<10 of PiBs of Flash

~100 of PiBs of HDD

<5 PiBs of DRAM

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

Next Generation PFS

Compute/ Clients

I/O Gateways (optional)

I/O Network (RoCE/IB)

[PiBs of Advanced Memory]

NVMe Fabrics Enclosures

NVMe Devices

[10-50s PBs of Flash]

Campaign Storage [100s of PB of HDD]

Tape Archive [100s of PB]

[400G Eth]

PFS Servers

Lustre Servers

Specialized S/W Stacks

misc

FS Gateways

Hot

Warm

Cold
ZFS gen3 NVMe Zpools Sequential Write Performance
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)
Doesn’t ZFS already support offloading?

- Intel® QuickAssist Technology (Intel® QAT)
  - Requires extra hardware adapter on AMD machines
  - Requires ZFS to be reconfigured and link with Intel® QAT libraries
  - Each offload operation is done independently of each other
    ▪ Encryption – AES-GCM
    ▪ Compression – GZIP
    ▪ Checksum – SHA256
  - API updates need to be merged upstream
    ▪ Chaining

- No generic way of offloading to other accelerators
ZFS Interface for Accelerators

+ Data Processing Unit Services Module

(and Providers)
Accelerated ZFS with Converged Storage

User Space

- ZFS + Z.I.A.
- Another Kernel Process

Linux Kernel Space

- DPUSM
- Software Provider
- BlueField 2 Provider
- GPU Provider

Non-Restrictive License

GPLv2

PCIe Bus

NVMe CSP

NVMe

NVMe

NVMe

NVMe

NVMe

NVMe

NVMe
ZFS Write Pipeline

- **ZPL**: ZFS POSIX layer
- **DMU**: Data Management Unit
- **ARC**: Adjustable Replacement Cache
- **ZIO Pipeline**: ZFS I/O Pipeline
- **VDEVs**: Virtual Devices
- **Drives**: Backing Storage Devices
ZFS Write Pipeline

Host (CPU)

ARC Buffer → ZIO Compress → ZIO Checksum → VDEV RAIDZ → VDEV Disk File → NVME I/O
Z.I.A. Write Pipeline

- ZPL
- DMU
- ARC

ZIO Pipeline

- VDEVs
- Drives

DPUSM

Provider

Accelerator

User API

Provider API

Accelerator API
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
- Declares what the accelerator provides
- Usually implemented by accelerator vendor
- Does not know anything about user
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 opaque handles that reference 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
    - Also offloaded decompression (read)
  - Checksum
  - RAIDZ
    - Generation
    - Reconstruction
  - I/O

- User data access not affected
  - During write
  - Afterwards
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
ZFS Write Pipeline

Host (CPU)

ARC Buffer → ZIO Compress → ZIO Checksum → VDEV RAIDZ → VDEV Disk File → NVME I/O
Z.I.A. Write Pipeline

Host (CPU)

Accelerator

ARC Buffer

ZIO Compress

abd_zia_handle

ZIO Checksum

VDEV RAIDZ

rr_zia_handle

VDEV Disk File

NVME I/O

Compress (GZIP)

Checksum (Fletcher)

EC (RAIDZ2)

Issue I/O
General Description of ZIO Pipeline Modifications

• If data is not offloaded at start of stage, offload it
• Run the operation
• Return status code (not payload)
  – Checksum offload returns checksum since it is only a few bytes, and is needed in ZFS
• 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
Accelerated ZFS with Disaggregated Storage

**Host System**
- CPU
- User App
- VFS
- ZFS + Z.I.A.
- DPUSM
- Provider

**NVMe-oF CSS (ZFS Offloads)**

**Software Component**

**Accelerated NVMe-oF Enclosure (ABOF)**
- CPU/DPU
- NVMe Controller
- Accelerator
- NVMe I/O

**NVMe**

**Figure:** Accelerated ZFS with Disaggregated Storage
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
- vio_vdev_io_done
- vdev_raidxio_done
- Verify Parity
- Issue I/O

Accelerator

- RAIDZ Reconstruct Missing Data
- Error
- Checksum
- Success
- RAIDZ Reconstruct
- Checksum
- RAIDZ Generate
- Issue I/O

NVME I/O
Important

• Z.I.A. does not provide new algorithms
  - GZIP is GZIP, not a custom compression algorithm
  - If hardware not available, still able to read/write data using software

• Software functions used in existing ZIO pipeline stages should not be moved out of ZFS into a provider
  - Need to be able to fallback to default implementation
  - Having provider is optional
zia-software-provider

- Exemplar implementation of provider (not operations)

- Links back into ZFS for functionality
  - Real providers should not do this

- “Offloads” data to memory outside of ZFS
  - Still RAM

- Stand-in when actual hardware is not available
  - Allows for testing
Intel® QAT Provider

• [https://github.com/hpc/zfs/tree/qat-provider](https://github.com/hpc/zfs/tree/qat-provider)

• Moved compression and checksum code into provider
  – No more reconfiguring ZFS
  – No more linking Intel® QAT libraries into ZFS
  – Encryption dropped (for now)
    ▪ Also need Z.I.A. + DPUSM hooks

• Work in progress
  – Untested
Future Directions

• More Offloads
  – Other ZIO stages/operations
  – Read Pipeline
  – DRAID Resilver

• Asynchronous Offloads
  – Analogous to Intel® QAT Chaining
Links

• Z.I.A. Pull Request
  − [https://github.com/openzfs/zfs/pull/13628](https://github.com/openzfs/zfs/pull/13628)
    ▪ Need more comments/reviews
    ▪ Need help with build system cleanup

• Data Processing Unit Services Module
  − [https://github.com/hpc/dpusm](https://github.com/hpc/dpusm)

• Direct I/O Pull Request
  − [https://github.com/openzfs/zfs/pull/10018](https://github.com/openzfs/zfs/pull/10018)
Questions?
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`
- `sudo scripts/zfs.sh`
Build and Load Provider

• Should not link with ZFS, so normally can build and load before 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

• 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

```plaintext
[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!