LLNL-TR-669183

Lawrence Livermore National Laboratory

ZFS on RBODs – Leveraging RAID Controllers for Metrics and Enclosure Management

Marc Stearman

Approved for public release

International Workshop on The Lustre Ecosystem: Challenges and Opportunities
Annapolis, MD
March 3-4, 2015

Disclaimer

ZFS on RBODs – Leveraging RAID Controllers for Metrics and Enclosure Management

Marc Stearman
Livermore Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA
stearman2@llnl.gov

ABSTRACT: *Traditionally, the Lustre file system has relied on the ldiskfs file system with reliable RAID (Redundant Array of Independent Disks) storage underneath. As of Lustre 2.4, ZFS was added as a backend file system, with built-in software RAID, thereby removing the need of expensive RAID controllers. ZFS was designed to work with JBOD (Just a Bunch Of Disks) storage enclosures under the Solaris Operating System, which provided a rich device management system. Long time users of the Lustre file system have relied on the RAID controllers to provide metrics and enclosure monitoring and management services, with rich APIs and command line interfaces. This paper will study a hybrid approach using an advanced full featured RAID enclosure which is presented to the host as a JBOD, This RBOD (RAIDed Bunch Of Disks) allows ZFS to do the RAID protection and error correction, while the RAID controller handles management of the disks and monitors the enclosure. It was hoped that the value of the RAID controller features would offset the additional cost, and that performance would not suffer in this mode. The test results revealed that the hybrid RBOD approach did suffer reduced performance.*

KEYWORDS: ***Lustre, ZFS, RAID, monitoring, JBOD, RBOD, metrics***

INTRODUCTION

Prior to Lustre 2.4, Lawrence Livermore National Laboratory (LLNL) operated numerous Lustre file system clusters with ldiskfs as the backend file system. Ldiskfs is based off of ext3, a journaled file system, and was the default file system for most Linux distributions. The ext3 file system dates back to the early 1990s and was originally a desktop file system. Ldiskfs, when used on the Lustre MetaData Server (MDS), would encur long file system check (fsck) times when problems arose. This led to the exploration of ZFS as an alternate backend file system for Lustre. ZFS, developed by Sun Microsystems, was designed to have an online consistency check (scrub) along with many other features that would be useful as a backend file system for Lustre. ZFS has been ported to Linux, and was integrated as a backend file system for Lustre 2.4 and beyond. Much of this work was performed by software developers at LLNL, and the ZFS on linux project can be found at http://zfsonlinux.org/.

Lustre has historically depended on reliable hardware, and the most commonly used method of protection has been to use advanced, high-speed RAID controllers to provide redundancy beneath the backend file systems. ZFS was designed with RAID style protection integrated into the file system from the beginning, and therefore prefers to manage drives directly. If ZFS manages the storage directly, it can detect and repair inconsistencies. When LLNL first started using ZFS with Lustre, ZFS was layered on top of RAID controllers to mitigate risk. LLNL had grown used to the RAID controllers managing hard drives and disk enclosures, and there was concern that using a JBOD approach with ZFS we would be failing drives over aggressively and that we would not be able to detect and troubleshoot problems as they arose. Our preferred path forward was to allow ZFS to do the RAID, and not rely on RAID controllers for data protection.

Over the past decade LLNL has developed a sizeable infrastructure for monitoring our numerous Lustre file systems' underlying hardware, and utilized the APIs and command line interfaces provided by advanced RAID controllers to profile the I/O traffic. For example, RAID controllers keep track of SCSI sense codes, key errors, individual disk drive read and write times, fan speeds, cache performance, I/O request sizes, queue depth, etc. Moving to a pure JBOD solution with ZFS would require all new methods of gathering this data. Many development hours would need to be spent developing tools to manage the systems, and new procedures would have to be written to change the way LLNL operates their multiple Lustre file systems. This paper proposes a hybrid approach that would allow ZFS to manage the RAID portion and consistency checking, while continuing to use the RAID controller for metric gathering and monitoring.

A HYBRID APPROACH

Imagine if you could keep the RAID controller, but rather than a typical 8+2 RAID6 layout of the drives, you presented each drive as its own single disk in either a passthrough mode or as a single drive RAID0. LLNL currently uses NetApp E5460 RAID enclosures. The E5460 is a 4U enclosure with 60 3TB nearline SAS drives. There are two RAID controllers with a shared cache and four QDR Infiniband ports on the back, which we have connected to two Lustre OSS nodes. The current configuration in production creates six 8+2 RAID6 volumes on the NetApp array. Three of the volumes are

presented to each host, which are used to create a zpool. The hosts are then configured as a Lustre OSS high availability (HA) failover pair. A hybrid configuration would present sixty individual drives as single drive RAID0 (striping) volumes, which will then be used to create the zpool using 10 drives via RAIDZ2, providing equivalent protection compared to RAID6. The RAID controller would still be able to monitor the drives, providing hueristics and performance data, while ZFS would just see a large number of disk devices. Figure 1 below depicts the difference between LLNL's current ZFS configuration and a potential hybrid apporach.

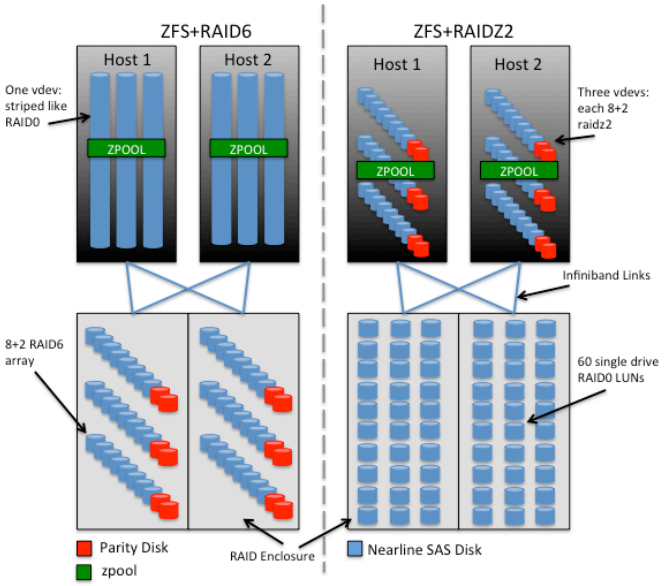

Figure 1. Current LLNL Production Model on the left, Hybrid Model on the right.

In the ZFS+RAID6 configuration, each host sees three 72TB volumes (LUNS), which are then striped with ZFS. ZFS will be able to detect if there is any data corruption in the pool, but due to the nature of striping, ZFS cannot correct any data errors. That job is left to the NetApp RAID array. However in the ZFS+RAIDZ2 configuration, the NetApp is just doing striping and ZFS now controls the double parity RAID, meaning it can fix any problems, and rebuild failed drives.

Test Configurations

To compare the performance and operational management of RBOD vs. JBOD, two different configurations were used. The testing was performed in the Hyperion testbed at LLNL using three NetApp E5500 units with dual enclosures and 120 3TB disk drives of identical make and model. The hosts and NetApp systems used configuration tunings consistent with LLNL production systems. Configuration 1, LLNL Production Model, builds on what LLNL runs in production, using the high performance RAID controller in the E5500 disk enclosure. With 120 drives, twelve 8+2 RAID6 LUNs were created, with six LUNs fronted to each host. ZFS is then used to create a single vdev, striping over the LUNs in a RAID0 fashion. In this configuration, ZFS can detect data miscompares, but cannot fix any problems. The RAID controller in the NetApp is relied up for data integrity. Figure 2 below depicts Configuration 1, modeled after LLNL Production systems.

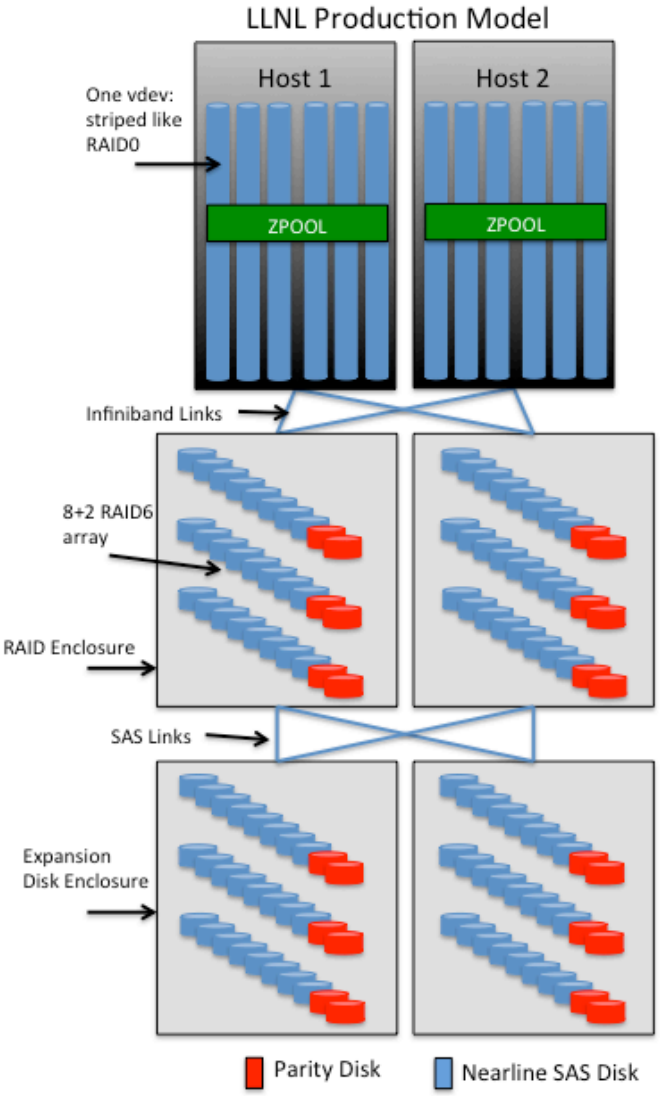

Figure 2. Configuration 1, LLNL Production Model

Configuration 2, ZFS Data Protection, is designed to allow ZFS to not only detect mismatched data, but also to correct any data miscompares. Rather than create RAID6 volumes on the NetApp array, 120 single drive RAID0 volumes were created to present the appearance of a JBOD to the hosts. From the host perspective, this looked like 120 single disk drives. These drives were then arranged into zpools creating six vdevs per host, comprised of ten-drive RAIDZ2 virtual devices (vdevs). In this configuration, the NetApp RAID controller was not doing any amount of integrity checking, but it would manage the health status of the drive, and continue to monitor for errors. ZFS would be handling the RAID aspect for the devices and provide data integrity. Configuration 2 is shown in Figure 3 below.

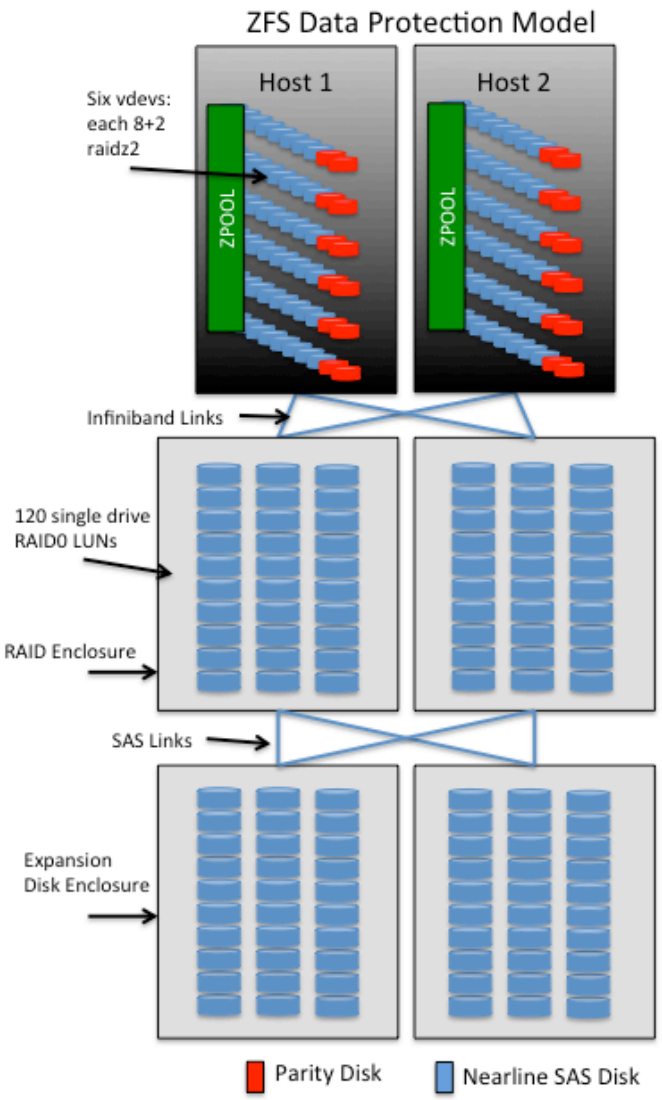

Figure 3. Configuration 2, ZFS Data Protection

Configuration 2, the ZFS Data Protection model was also tested without the RAID controller, using the same enclosures as JBOD storage. The host was connected with SAS adapters rather than Infiniband adapters.

Test Approach

ZFS is a complicated file system with a large number of features and enchancements that increase the difficulty of running benchmarks. Many benchmarks are designed to pre-create a set number of files, and then run a variety of read and write operations upon those files. This is often useful to see how fast your hardware can go, but is typically not representative of a real user-driven workload. In addition, ZFS has some intelligence built in where it tries to understand and adapt to a workload and improve I/O by grouping transactions into groups. ZFS will examine the list of transactions within a group and if there are multiple write operations to the same file region, it will throw out the unneeded writes and just do the last transaction. In this instance, your benchmark may say it wrote 2GB to the file, but in fact only 1GB was sent to the storage devices due to writes hitting the same regions of the file. Also, due to the copy-on-write nature of ZFS, the random I/O write tests come out similar to the sequential I/O write tests. On the read side, ZFS has a rather advanced cache, a varient of the ARC (Adaptive Replacement Cache) algorithm. This uses RAM for level 1 caching and can use SSDs for a level 2 cache. Because the ARC aids write performance disabling it for benchmarking was not considered a reasonable option. One can see that it is difficult to find ways to defeat the intelligence of ZFS. For this testing two approaches were chosen. The main interest was to test a realistic user workload, in order to simulate what users may expect to get from their applications. Before that testing could be done, the limits of the hardware and the ZFS software on top of it needed to be understood.

The first set of tests run used XDD, reading and writing to 10 files through the posix layer of ZFS. The command used for the write test is:

```
/usr/bin/xdd -targets <DEV_LIST> -op write -blocksize 4096 -reqsize 2048 -runtime 60 -numreqs 1000000 -queuedepth 16 -seek range 13771092032 -seek random -output ./all-ran-write-2048-4096
```

and for the reads:

```
/usr/bin/xdd -targets <DEV_LIST> -op read -blocksize 4096 -reqsize 2048 -runtime 60 -numreqs 1000000 -queuedepth 16 -seek range 13771092032 -seek random -output ./all-ran-write-2048-4096
```

This is a straighforward test using an 8MB buffersize to see how fast the file system can write and read with a sequential I/O pattern.

The second set of tests run used FIO to test the usefulness of the ZFS cache, and how well it can improve application performance. This test used the following configuration parameters:

```
bssplit=2m/100
rw=randrw
rwmixread=50
direct=0
size=64g
nrfiles=16
ioengine=posixaio
fallocate=none
random_distribution=zipf:1.2
iodepth=32
runtime=600
```

This job mix uses a 2MB I/O size for a 50/50 mixed workload of reads and writes. The I/O used 16 files that were each 4GB in size, totalling 64GB, which is the size of RAM on the nodes. The important parameter in this test which shows the benefits of the ZFS ARC is the `random_distribution` parameter. The zipf:1.2 uses the zipf power distribution to restrict the I/O to a small portion of the files. In this instance approximately 95% of the I/O will happen in only 4GB out of the 64GB. This is intended to reflect hot data, and commonly accessed regions of files. In this instance we expect ZFS to keep a majority of the data in RAM for the reads, and to discard a large number of I/O transactions that hit the same region of the file.

As stated before, the prior two tests were meant to show the capabilities of the hardware and software, but do not reflect a real user workload. With ZFS as the backend file system for Lustre, we do see a mixed workload of reads and writes, but we typically do not see the case of users rewriting over existing files. The two main classes of our user jobs are simulation and analytics. In the simulation case, users will read an initial input deck and periodically write out a set of restart checkpoints, or results data. This I/O is always written to new files, and it is rare that files are read and overwritten. For data analytics, the users typically have a starting set of data that their application reads in, and after processing some of the data, a reduced set of data is written to new files. Again there is very little rewrite happening. The FIO benchmark was used to simulate this type of behavior, but with a different workload meant to simulate what users may actually be doing on the system.

Designing a workload that simulates user behavior requires knowing what the users are doing. LLNL uses the RobinHood package developed by CEA. RobinHood processes the changelogs on a Lustre file system and stores that information in a database allowing for fast simple queries of file systems characteristics. Using RobinHood, a histogram of our file sizes was generated, showing that 90% of the users' files were smaller than 32KB. The average file size, looking simply at space used divided by total number of files, was approximately 4MB. This indicates that there are a very small number of large files, and a large number of small files. This information facilitated the creation of an input file for FIO that could simulate a large number of small files with a random workload. The job file was comprised with four write oriented jobs, and two read oriented jobs. This also appeared to fit with observed behavoir by watching the output from ltop, an application packaged with the LMT (Lustre Monitoring Tool). The six jobs would run in parallel based on the following input file:

```
[global]
description=Emulation of User Workloads
direct=0
ioengine=posixaio
fallocate=none
iodepth=32
create_on_open=1

# 256K BS, Write, 10000*2m = 20GB
[userLoad1]
directory=/mnt/testfs/1
bssplit=256k/100
rw=write
filesize=2m
nrfiles=10000
size=10000m

# 1M BS, Read, 10000*3m = 30GB
[userLoad2]
directory=/mnt/testfs/2
bssplit=1m/100
rw=read
nrfiles=10000
filesize=3m
create_on_open=0
size=10000m

# 8K BS, Write, 100000*32k = 3.2GB
[userLoad3]
directory=/mnt/testfs/3
bssplit=8k/100
rw=write
filesize=32k
nrfiles=100000

# 16K BS, Write, 100000*64k = 6.4GB
[userLoad4]
directory=/mnt/testfs/4
bssplit=16k/100
filesize=64k
nrfiles=100000
rw=write

# 32K BS, Read, 100000*64k = 6.4GB
[userLoad5]
directory=/mnt/testfs/5
bssplit=32k/100
filesize=64k
nrfiles=100000
rw=read
create_on_open=0

# 1M BS, RandWrite, 10000*2m = 20GB
[userLoad6]
directory=/mnt/testfs/6
bssplit=1m/100
filesize=2m
rw=randwrite
nrfiles=10000
size=10000m
```

The global section specifies that we do not want to use directio and fallocate, which are not supported for ZFS, and to use the posixaio engine for the testing. We also want an I/O queue depth of 32 and when doing the writes, we should create them upon open. Real user applications are not often overwriting existing files, rather they create the file upon opening for writes. For the read tests we have to precreate the files such that there is something to read from. The six parallel jobs are as such:

1. Create 10,000 2MB files writing with a 256KB blocksize, but only write 10GB worth of data.
2. Read a total of 10GB of data from 10,000 3MB files (30GB) using a blocksize of 1MB.
3. Create 100,000 32KB files using an 8KB blocksize.
4. Create 100,000 64KB files using a 16KB blocksize.
5. Read 100,000 64KB files using a blocksize of 32KB.
6. Create 10,000 2MB files using a 1MB blocksize, but only write 10GB worth of data. Uses the random write FIO algorithm.

This set of jobs should defeat the ZFS cache and present a fairly realistic mix of workloads to provide a fair comparison of the different hardware configurations.

TESTING RESULTS

The first set of tests run were the XDD tests which can be seen in Figure 4. All three configurations did relatively well in the write case, with the JBOD mode being the slowest. The RAID controller has a significant amount of write cache in the controller, so as writes occur, the RAID controller can

acknowledge the operation as soon as it hits the cache, whereas in the JBOD case, the host will wait until all 10 drives have completed their I/O before acknowledging the write. The RAID controller yields a 30% performance improvement over the JBOD case for the XDD writes. However the opposite is true for the read case. The JBOD mode is slightly faster than the RAID6 mode, but significantly faster than the RBOD 120 LUN mode. In this case the RAID controller is getting in the way because it has a fixed sector size for the RAID volumes, even for RAID0.

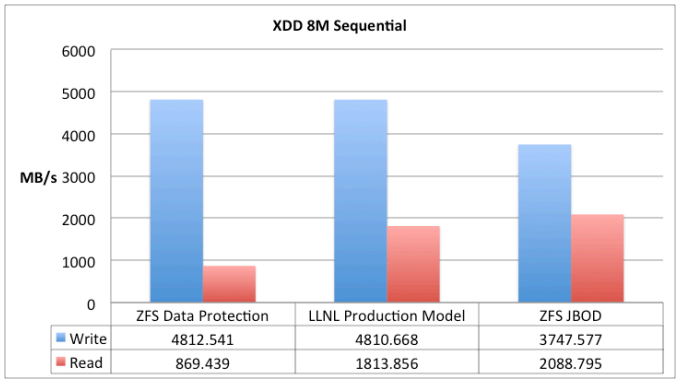

Figure 4. XDD Results

The RAID controller is doing more work to read the data than just accessing the disks directly through the JBOD mode. The variance in read performance in the three RAID modes is due to the number of top level vdevs in play. The fewer top level vdevs you have, the less taxing it is on the queue depth of the RAID controller. It is possible that tuning the queue depths for each case may improve the results, but the hero numbers were not our focus so that exercise was left as potential future work.

The results of the initial FIO workload show how useful the ZFS cache can be to application performance. The graph shown in Figure 5 dipicts what happens as the cache warms up.

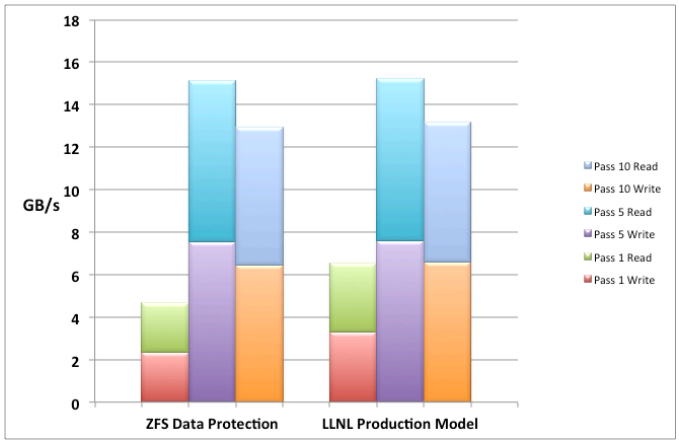

Figure 5. Warming the ZFS cache

This workload yields 5-6 GB/s combined read and writes, but as you run the same job repeatedly on the same files, the cache comes into play and throughput of 15-16GB/s can be achieved.

One of the nice things about using FIO to run multiple jobs in parallel, is that it provides detailed information about each job. This includes bandwidth, runtime, and latency statistics. When designing the mixed workload to simulate what user applications may be doing, it was helpful to iterate over a number of different job mixes, and look at the individual results. Figures 6 and 7 show the per job results for a single node, depicting the runtime and bandwidth of each of the six jobs.

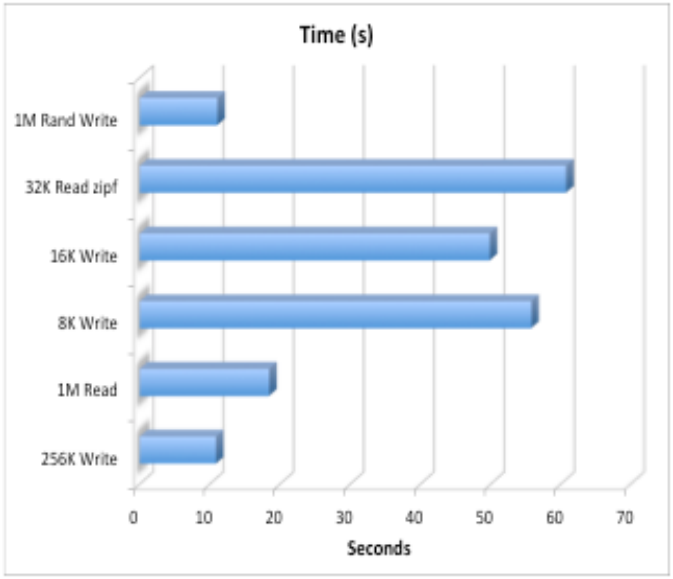

Figure 6. Job Runtimes

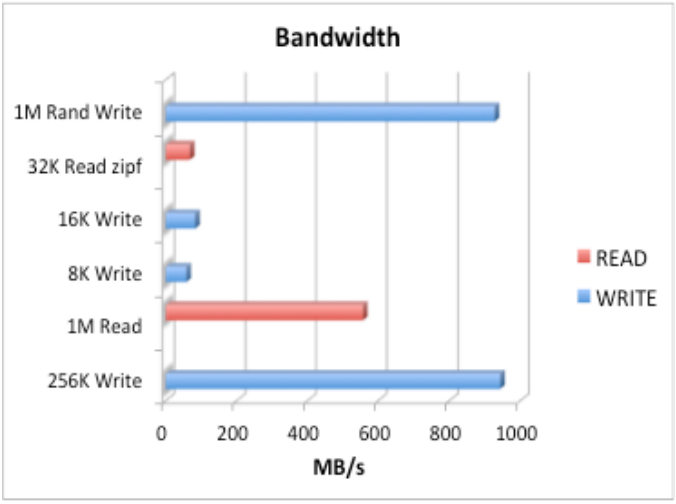

Figure 7. Job Bandwidths

As expected, the jobs with the larger block sizes (256KB and 1MB) completed faster with better bandwidth, while the smaller block size jobs took longer to complete due to smaller bandwidth. Spinning hard drives have a limited amount of I/O operations per second, so the smaller I/O block sizes reduce the apparent bandwidth capabilities of the controller. A file system that is capable of 80GB/s may only yield 2GB/s when user applications read and write with small buffer sizes. ZFS attempts to mitigate this by grouping smaller I/O into larger

streaming I/O, but it is currently limited to 128KB transactions, which will be split up into 16KB I/Os to the individual drives in a RAIDZ2 configuration. A large block patch is ready for testing in the latest branch of ZFS, but was not available during this testing. Had it been ready, we suspect performance gains would have been realized in all configurations, thereby inflating the numbers but not producing a significant difference in the results. In addition, due to the large amount of data being written, the ZFS cache was not able to warm up and come into play. Running this workload multiple times did not yield better performance.

Comparing the results of the FIO mixed workload amongst the three configurations revealed some interesting results as seen in Figure 8.

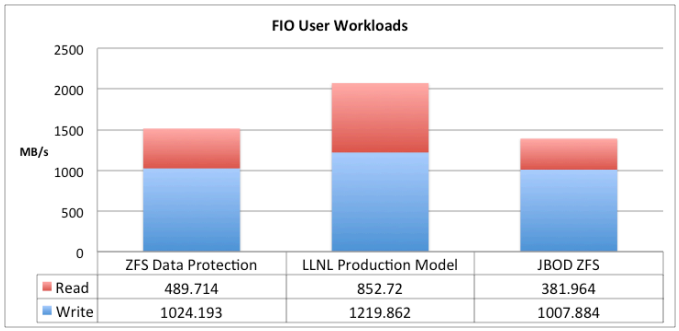

Figure 8 FIO Mixed Workload.

The LLNL Production Model is 20% faster on writes, similar to before, and almost double the speed on reads. The NetApp RAID controller has optimized RAID engines and the advantage of the write cache. For each block, when ZFS is managing the RAID calculations, it sends ten I/Os to the disks, whereas in the LLNL Production RAID6 mode, the host sends one I/O to the RAID controller, which splits up the data for the ten drives involved. Since the RAID controller can acknowledge the I/O once it hits cache, the host can send much more data down the pipe to the enclosure. The ZFS Data Protection model is also very sensitive to the drive path configuration. Each disk has two channels coming out of it, and the RAID controller has multiple I/O controllers to maximize disk bandwith. If the drives are not layed out properly, you can oversubscribe some channels, and not utilize others. With the RAID controller, you have full control over which paths are assigned to drives and controllers. In the ZFS JBOD case, you lose that control and it is likely you may not be utilizing the drive channels properly. This will certainly affect maximum performance of the system.

Analysis

This study was undertaken because of the belief that the hybrid solution would be viable. RAID controllers add approximately 10% to the cost of the system when all other parts are the same. That is a pretty good value, considering the benefits gained:

- Metrics – drive and channel heuristics, performance counters, drive latencies, etc.
- Monitoring – enclosure services, predictive failure analysis, rich API, battery status, event logs, etc.
- Firmware – vendors work closely with drive manufacturers
- Fine tuning of I/O paths

Being able to keep the RAID controllers and let ZFS handle the RAID seems like the best of both worlds, until you examine the performance numbers. Since you have now paid the cost of the RAID controller to reap the benefits, you may as well use it for what it was designed for, and run the enclosure in RAID6 mode. True, ZFS cannot fix problems, but then again, you are relying on the RAID controller to do the job for you. You still get all of the benefits of ZFS, such as compression, online consitency checking, snapshots, etc. Further research could be done to see if it is possible to tune the JBOD mode to improve the performance and bring it equal to the RAID6 mode. Testing at the Lustre layer may also level the playing field.

Summary

LLNL decided to layer ZFS on top of RAID controllers as a risk mitigation strategy when adopting ZFS as the backend file system for Lustre. The intent was to eventually move toward a JBOD style enclosure and save money by not having to purchase RAID controllers. Historically, the RAID controllers were a significant cost component of the storage architecture, and ZFS was seen as an alternative that would save money. In addition, because ZFS stores the checksum for the data block elsewhere on the drive, the RAID implementation appears to be a superior solution to most hardware RAID controllers. However, the question of how to manage drives and enclosures in a JBOD without relying on RAID controllers remained. As the price of RAID controllers dropped over time, a hybrid solution appeared to be an excellent path forward, but the performance of the RBOD mode is lower than expected. Considering that the RAID6 mode performs best, and the RAID controller provides the enclosure management features desired, the risk mitigation strategy of ZFS atop RAID controllers made three years ago continues to be a good decision, and is likely to be our path forward for the forseeable future. Given resources and time, it would be interesting to see if it is possible to tune the system to improve the ZFS Data Protection Model in a way that improves the performance above the LLNL Production Model, but that is outside the scope of this paper.