Which ZFS pool configuration is most appropriate for a given requirement and number of disks?

Matthew Swain

20200873

Contents

[Abstract 2](#_Toc170402292)

[Introduction: 2](#_Toc170402293)

[Literature Review: 2](#_Toc170402294)

[Understanding File Systems: 2](#_Toc170402295)

[The Need for ZFS: 3](#_Toc170402296)

[Creation of ZFS: 4](#_Toc170402297)

[Evolution of ZFS: 7](#_Toc170402298)

[Test: 11](#_Toc170402299)

[Disclaimer: 11](#_Toc170402300)

[Methodology: 11](#_Toc170402301)

[Issues: 12](#_Toc170402302)

Abstract:

# Introduction:

Since its release to the public in 2006, ZFS, formerly known as the Zettabyte File System, has become a popular filesystem choice among system administrators. Oracle’s proprietary implementation is currently the 8th most used storage infrastructure[[1]](#footnote-2) and it is also becoming popular choice among home server enthusiasts. When it released, it was seen as a revolutionary new file system for its dedication to ease of use, data integrity and long term viability at a time when alternative file systems struggled with these aspects and since then it has continued to add functionality that has only made it even more competitive. Despite this, if the storage solution is not properly configured for its use case, it may not be as performative or reliable as expected and so this paper aims to explore the different ZFS configurations and which scenario they may be most suited to.

# Literature Review:

### Understanding File Systems:

With Random Access Memory being volatile, devices such as hard disk drives (HDDs) or solid state drives (SSDs) are needed to keep the data permanently stored after power has been turned off. However, when a user wants to save a file to long term storage, a program needs to handle storing that data to the drive in a safe and organised way so that it can be retrieved again. This is the role of a file system and examples of common file systems include New Technology File System (NTFS), the default file system for Windows, fourth extended filesystem (ext4), the default file system for popular Linux distros such as Ubuntu, and Apple File System (APFS), the default file system for Apple devices. When a user wants to create a file system, they must select the drive they wish to use, partition the drink if required and then format the drive with the desired file system to create a volume. When the drive is being formatted, the file system will divide all of the available space into blocks of a size chosen by the user, with a common default being 4096 bytes or 4 kibibytes (a kibibyte being 1024 bytes, as oppose to a kilobyte being 1000 bytes). A block is an atomic unit of damage storage, and so any stored file needs to be broken down into as many blocks as necessary, with a 1 gibibyte file needing to be split into 262,208 blocks rounded up from 262,207.03125. As can be seen in this example, a 1GiB file does not break down into a whole number of 4KiB blocks and a large part of the block is wasted holding a small amount of data, which is known as slack space, however if a user is aware of the average size of the data they are storing, they may be able to adjust block size accordingly to improve performance by increasing block size and reducing the number of blocks needed to store data or limit wasted space by reducing block size.

Mention inodes

These traditional file systems are relatively easy to setup, performant and fairly reliable and therefore used by billions of devices[[2]](#footnote-3), however they have been designed mainly for use in desktops and laptops and therefore not suited for all situations. The large amounts of critical data that companies handle requires it to be stored on many disks in a RAID (Redundant Array of Independent/Inexpensive Disks) like configuration with extreme robustness to minimise the risk of data loss, high performance to ensure no long delays between read and writes and an ease of management. The aforementioned file systems would not be able to fulfil all of these requirements as they have simply not been designed or built to do so.

### The Need for ZFS:

And so it was, during the late 1990s, after a typo in a command during a routine maintenance operation on Sun Microsystem’s Jurassic server left a building full of a thousand employees with nothing to do until the data restoration was complete, Jeff Bonwick and Tim Marsland began lamenting Sun’s inability to create a new filesystem. There had been several attempts to create a new one which may have helped resolved the difficulty involved with performing basic administration. As it stood, if a system administrator wanted to perform basic operations such as growing or shrinking a file system or adding storage to a partition, it was either not possible or overly convoluted.[[3]](#footnote-4) Adding storage required the user to take the whole file system offline, rendering it inaccessible and any disks to be added needed to be partitioned meaning if multiple file systems needed to be set up with a limited number of disks, the partition sizes would need to be guessed and it would not always be possible to change these sizes.

Include that FS and volume manager were seperate

There were also issues with data integrity as most file systems used a write method called journalling. Whenever a meta-data operation, an operation that modifies the structure of the file system such as creating, deleting or renaming a file or directory[[4]](#footnote-5), was performed, the system would record the change it was making into a log. This theoretically ensured that any file or directory would still be accessible in the event of a crash or power loss event while the operation was being performed, however Bonwick et al argued “consider the common case in which a bootloader reads the root file system in order to find the files it needs to boot the kernel. If log replay is needed in order to make the file system consistent enough to find those files, then all the recovery code must also be in the bootloader”[[5]](#footnote-6).

Data corruption could also be caused by the drives, drivers and controller as, during this time, RAID was achieved using a RAID controller, a device that all of the drives would be connected to and would then create the desired RAID configuration and present the array to the operating system as one device, handling all of the read and writes independent from the filesystem. The file system would then trust that any data read from the array would always be the same as the data written to the array, but this led to issues as the controllers would not always have extensive error handling or any error handling at all[[6]](#footnote-7), leading to issues with bit rot. This is a phenomenon where data slowly becomes partially or completely corrupted and can be caused by drive deterioration/failure, bit flips, administrative errors or bugs in the disk or controller drivers that can cause misdirected reads (data read from the wrong block), misdirected writes (data written to the wrong block) and phantom writes (the device reporting the data has been written when it hasn’t). This also doesn’t mention that if the RAID controller fails, the entire storage array is lost, making it a single point of failure and the weakest link in the chain. Data loss would be is extremely undesirable for a consumer, but in business where data is their most valuable asset[[7]](#footnote-8), it is particularly unacceptable as 93% of business that experience data loss for more than 10 days will file for bankruptcy within one year. (need better source, possible remove as correlation causation etc)

Finally, far exceeding Moore’s law, storage capacity had been doubling and the cost halving every 9-12 months[[8]](#footnote-9) and the 32-bit block addresses being used by file systems limited the maximum file system size to a few terabytes, something that could feasibly be achieved at the time, and could easily be surpassed not long after. A simple switch to a greater amount of block addresses would have not solved the issue either, as algorithms for operations such as directory lookup and block allocation would need to scale well with larger amounts of storage, something that the algorithms in existing filsystems would not have been designed to do.

Bonwick and Marsland envisioned a file system that resolved these issues by treating disks like memory. When a server’s memory needed to be expanded, “you just open it up, plug in some more DIMMS, reboot and the OS starts using the added memory, all the apps go faster and there’s no real administration. There’s no DIMM config that you have to run, you don’t have to create virtual DIMMs, there’s no DIMM management software, there’s no web GUI for it, it’s just easy”[[9]](#footnote-10). Unfortunately, the conversation ended soon after as the data restoration completed and both went back to their respective projects.

### Creation of ZFS:

Trying to order this the same way that the previous section was, i.e simplicity, data integrity and speed and capacity, not sure if it flows very well

Around 2000, a few years after Bonwick and Marsland had pictured this method of storage management and another of Sun’s file system projects had been scrapped, Bonwick finally decided to try and put his plan into action[[10]](#footnote-11). After an attempt at the idea was made with too many engineers divided between too many locations failed, in July of 2001, Bonwick took another attempt at the project, this time with a team just consisting of him and an engineer named Matthew Ahrens and by October they had a working prototype. After bringing in more engineers to work on the project, including Mark Maybee and Mark Shellenbaum, the team grew to twelve people and on October 31 2002, they achieved their first kernel mount. Progress continued with the team starting to use ZFS to store their own files in 2004, an integration into Sun’s internal version of/proprietary version Solaris on the 31st October 2005 and finally releasing to the public in May of 2006 with Sun’s free and open-source Solaris 10 operating system.

In an interview with Jeff Bonwick and Bill Moore in 2007, they talked about how they took the ideas that had been thrown around in the office during the server down time and used them to create a new file system to solve the issues with existing file systems. Rather than there being a tight coupling between the logical namespace and the physical devicie, there is “a pooled storage model. The disks are like DIMMs, and the file systems are like applications. You add devices into the storage pool, and now the file system is no longer tied to the concept of a physical disk. It grabs data from the pool as it needs to store your files, and as you remove or delete your files, it releases that storage back to the pool for other file systems to use.”[[11]](#footnote-12) They accomplished this by removing the separation between the volume manager and the file system, instead replacing it with a Storage Pool Allocator (SPA) at the bottom layer next to the drive, a Data Management Unit (DMU) in the middle and the ZFS POSIX Layer (ZPL) at the top. Rather than needing to apply a logical volume over a single or multiple disks, the SPA acts as an interface through which virtually addressed blocks can be allocated or freed and nothing above this layer knows where the blocks are physically located.

May need to move the following section to somewhere that flows more nicely

This means that the user can easily manipulate the number of disks in the configurations that ZFS released with, such as the striped and mirrored configurations. A striped disk array allows any data that is written to be spread out across as many disks that are in the pool, leading to very fast read/write streaming and IOPS speeds, the number of disks multiplied by the drive speed, but no redundancy; if one drive fails, all data on the pool is lost[[12]](#footnote-13). The user could also use a mirrored array, in which data is written to as many disks in the array, vastly improving redundancy, as at least one drive can fail and the data still survive, but does reduce performance, with the read streaming and IOPS speeds being the number of disks in the mirror multiplied by the read speed of a single drive, but the write streaming and IOPS speed only being the write speed of a single drive[[13]](#footnote-14). ZFS does also have the ability to treat multiple drives as one device, called a virtual device or vdev. For instance, a user with 4 drives could make 2 vdevs each with 2 disks in a mirrored configuration and while they would only have the storage capacity of 2 drives, data could be striped across the two vdevs, getting the two times performance increase of a striped array with a 2 drive redundancy (although the two failed drives would have to come from separate vdevs, if 2 drives in the same vdev failed, all data would be lost)[[14]](#footnote-15). The SPA would allow the user to easily add a new device or virtual device to expand a pool by simply allocating new blocks to the new device, or shrink the pool by copying blocks off the device to be removed if there is space to do so.

Ease of administration section may need work

During the same interview, Moore also stated that one of the key design principles for ZFS was “never, ever trust the underlying hardware”[[15]](#footnote-16) as, despite drive sizes increasing exponentially, the bit-error rate had remained constant, meaning it had become increasingly likely a user would experience an uncorrectable error. For this reason, before the data is written to the disk, each block is checksummed, a calculation that validates all the data equals a specific value and is therefore valid, with each checksum being stored in its parent’s block. This goes all the way up to the root of the block tree with the überblock, the only block in the system that stores its own checksum. As the checksum is stored in the parent block, the likelihood of the both the data block and parent block being corrupted is unlikely, reducing the risk of data loss. It also improves performance as the checksum does not need to be read in from another block, the parent block and therefore the checksum, has already had to be read to get to the data block. The checksum also allows the data to be self-repairing, as, when the SPA reads data from a block, the checksum will show whether the data is valid and if not, the SPA can repair the data based on the correct data from the mirrored disk which the user will likely have if data integrity is a priority.

Moving up a layer, the Data Management Unit handles the conversion between the blocks presented by the SPA and the files, otherwise known as objects, that live within the dataset, i.e. the ZFS equivalent of a volume or the folder that all the data stored with the ZFS filesystem is located. When the user makes a change to a file, the DMU ensures that data integrity is maintained by replacing the journalling approach with a transactional copy-on-write system. The edited block or blocks are not overwritten but copied to a new block with any changes implemented, then, as there is no parent block currently pointing to it, the previous parent block is copied with its pointer pointing to the new block. This process is repeated up the block tree until it reaches the überblock, at which point, when the new überblock is created, in a single transaction the change is committed as before this, the original überblock would simply point down to the original file. In this way, a power loss event can never result in inconsistent data as the change can only ever be completed during the single überblock rewrite transaction.

The final ZFS POSIX layer handles converting the ZFS objects into the form set by the Portable Operating System Interface, or POSIX, ensuring compatibility with all Unix-like systems and feature rich file management. It also helps ensure data integrity by grouping

Whilst all the checksums and data validation that ZFS implements, it may seem like it sacrifices performance in exchange for data integrity, however this is not the case. The exponential increase in processor speed renders any performance impact negligible, demonstrated by Widianto et al in 2016, finding that, using the Bonnie++ benchmarking software, a system with checksum disabled only had a sequential read speed of 2.6 MB/S and a write speed of 0.1 MB/s faster than that of a system with it enabled[[16]](#footnote-17) Need to explain what read/write sequential, random and IOPS are here?.

ZFS has also been benchmarked against the more traditional ext4 filesystem as well as Btrfs, a file system released integrated into the Linux kernel in 2009 that also emphasises data integrity by using copy-on-write but differs from ZFS by still using a traditional volume manager. In 2009, Dominique A Heger found that, when using a RAID-10 equivalent array, that is a stripe of 4 vdevs each containing 2 mirrored drives and the Flexible Filesystem Benchmark, ext4 was the most performant file system for sequential read/writes at an average of 248.1 MB/s and 208.6 MB/s respectively. ZFS was second with sequential read speeds of 227.74 MB/s and sequential writes of 193.74 and Btrfs with reads of 162.52 MB/s and 109.12 MB/s. The results are similar for random writes, however for random reads, ZFS achieved the highest at 8.1 MB/s, Btrfs in second with 6.1 MB/s and ext4 with 6.1 MB/s[[17]](#footnote-18). Mention file sizes and request sizes. Also talk about how this paper shows the extra stuff doesn’t impact performance that much.

However, in a similar paper in 2016 written by Anders Lundholm, when using another RAID10 comparable configuration, this time with a stripe of 2 vdevs each with two mirrored disks and using the IOzone benchmarking tool, he achieved different results. Rather than testing with a fixed block size, he tested from 64 KB blocks to 16,384 KB blocks to see how the file systems might differently react and found that for sequential reads the results were largely the same. Btrfs and XFS consistently achieved the highest speeds at just below 270 MB/s, ext4 followed with 215 MB/s and ZFS in last with 160 MB/s. Btrfs appears to have made a significant performance improvement, going from the worst performing to the most performant and ZFS has become the worst performing, possibly bogged down by its extra data integrity features. The sequential read speeds are much different however, with ZFS having the highest at an average of 310 MB/s, Btrfs and XFS next at around 270 MB/s and ext4 last at around 240 MB/s.

Whilst it is important not to directly compare the numbers between the two papers due to vastly different hardware, the trends can be analysed to show that the extra data integrity measures ZFS implements do not always result in lower performance and it could be argued that the data integrity, features and matureness of ZFS could outweigh the performance negatives as, while Btrfs has made great improvements in its performance, its RAID 5 and 6 equivalents are infamously unreliable and incomplete. It also shows that file systems are constantly evolving, with performance changing over time and new features being added that may sway the file system choice.

Whilst it may seem that btrfs better, raid5 still needs work, something about both having 16 exbibyte max file, zfs has max voume size of 256 trillion yobibytes, btrfs only 16EiB, somewhat irrelevant though

Mirror and raid 10 may also not be directly comparable due to differences in the way they work

### Evolution of ZFS:

Cant find any release dates for solaris versions

ZFS has also continued to evolve after its release as part of OpenSolaris Nevada b36 in 2006, with one of the biggest changes being the addition of RAIDZ2 in OpenSolaris Nevada b42 and RAIDZ3 in Nevada b120, storage configuration layouts like mirroring and striping, that offer a middle ground between the two. Rather than writing the same block onto as many disks as are in the vdev, RAIDZ dedicates at least one disks worth of space as parity data

RAIDZ dedicates as many drives as selected in the RAIDZ level to dedicate to parity information. For instance, in a 6 drive vdev at RAIDZ2, ZFS will add parity blocks to the data and write the data to 4 of the disks, with the other 2 holding the parity information so that up to 2 disks can be lost without any information being unrecoverable.

In 2010 Sun, a company that was very supportive of open-source software, was bought by Oracle, a company who was extremely protective of their software, and so OpenSolaris was discontinued and any further development for ZFS no longer being freely available. The last version of OpenSolaris was forked into the illumos project later that year and work continued on both it and ZFS, however development became uncoordinated and fragmented. A lot of the work being done was being duplicated by multiple groups and so in an effort to stop this, one of the original developers for ZFS, Matthew Ahrens, created the OpenZFS group in 2013, aiming to coordinate development of the open-source and freely available version of ZFS.

Include zfs having default block size of 128 KiB somewhere

Bring up raidz (not sure if appropriate as it could have been in from the beginning, could bring it up when raidz3 was brought in), deduplication, snapshots, resilvering etc

If the pool contains any raidz vdevs, the vdev cannot be removed and the pool shrink could happen with mirrors

Old work below

--------------------------------------------------------------------------------------------------------------------------------------

ZFS Storage Pool Layout, written by iXsystems, provides a higher-level explanation of the various pool layouts and uses example drive speeds to show how the layouts affect aspects such as read/write speeds and IOPS, however these speeds are only theoretical, therefore they don’t give a completely accurate picture of how each pool configuration would perform. While the paper does comment on how the pool configuration would affect fault tolerance, it does not comment on how likely the pool is to fail and how resilvering speeds would be affected. For instance, two of the example configurations given are a 6 2-way mirror and 2 6-wide RAIDZ2, and while it does mention that only 1 drive per vdev before the pool is lost for the mirror and 2 per vdev can be lost for the RAIDZ2, it does not mention that as time goes on, the probability of the mirror pool being lost is significantly higher than the RAIDZ2 pool being lost as can be seen below.[[18]](#footnote-19)

A graph of a function

Description automatically generated

The most common way ZFS has been benchmarked is in comparison to another filesystem, usually BTRFS, such as Dominque A. Heger[[19]](#footnote-20) who compared the sequential and random read/write speeds of ZFS, BTRFS and ext4 using 4KB files/blocks. He found that when using 4 vdevs of 2 mirrored disks, ZFS consistently performed better than BTRFS with sequential read/write speeds of 227.74MB/sec and 193.74MB/sec and random read/writes of 8.1MB/sec and 32.9MB/sec. By comparison, BTRFS using a similar RAID 10 configuration, achieved sequential read/write speeds of 162.52 MB/sec and 109.12MB/sec and random read/write speeds of 7.5MB/sec and 12.48MB/sec. While all tests were repeated 10 times and the mean results taken to ensure accuracy, it only tests one ZFS configuration, a pool of devs with 2 mirrored drives in each vdev. This is a configuration that specifically optimises speed above all else and the paper does not take into account factors such as drive failure tolerance and storage space efficiency which, in this configuration, are sacrificed for speed. The paper overall shows ZFS to be the higher performing filesystem, however the results may no longer be accurate as it was published in 2009 and a paper from 2015 by Anders Lundholm shows ZFS to be one of the slower filesystems.

While the two testing environments are not exacty similar (Heger uses 8 7,200 RPM drives while Lundholm uses 4 10,000 RPM), the same configurations are used (a pool of vdevs with 2 mirrored drives), so Heger and Lundholms results should still be comparable, with ZFS still being the highest performer. This, however, is not the case. Lundholm found that ZFS achieved an average of 312.6MB/sec sequential reads and 156MB/sec sequential writes, making the reads significantly faster than Heger’s findings and the writes slower. He also found that the random read/write speeds were 89MB/sec and 167MB/sec, making them significantly faster than Heger’s results. On top of this, he found that in almost all instances, BTRFS was faster than ZFS, achieving sequential read/write speeds of 264.6MB/sec and 263.9MB/sec, with the random read/writes reaching 169.2MB/sec and 261.2MB/sec respectively. These speeds are vastly different from Hegers’s and this performance delta (not sure if correct word) could come from the differences in testing environments; the higher number of drives in Heger’s environment should lead his results to be higher while Lundholm’s 10,000RPM drives may lend themselves to better sequential read/write performance. Despite this, BTRFS overtaking ZFS in the 6 year gap between papers does show that filesystems are not static, with performance and features improving over time, meaning that after 9 years, Lundholm’s performance figures may no longer be accurate either, making it interesting to see what ZFS’ performance is like in the current day (need a better way to link this back to the main subject).

Think need to rewrite as have gotten file, block and record mixed up.

May also need to rewrite as I think I am getting too mixed up in comparing results and comparing my would be results to theirs when so many other factors may cause differences and results aren’t really comparable. Probably need to refocus and talk about how they compare their results against the different configurations. Issue is nobody really does that.

NEED TO FOCUS ON PATTERNS RATHER THAN NUMBERS

Struggling to find anything that shows patterns

Link to a paper that doesn’t benchmark against other fileystsmes <https://ieeexplore.ieee.org/document/7881982>

In 2016, Eko D. Widianto et al also conducted benchmarks on ZFS, this time using one vdev with 2 mirrored disks to compare the different kinds of compression algorithms that ZFS supports

The paper also touches on resilvering time, testing how long the 2 disk mirror would take to resilver with 40GB, 80GB and 120GB of the capacity used, finding it took 10.7 minutes, 22.7 minutes and 36.7 minutes respectively. Lou Wrentius[[20]](#footnote-21) also looked at resilvering time in ZFS: Resilver Performance of Various RAID Schemas Not a scientific paper. , where he tested up to 11 disks in various mirror, RAID-Z, RAID-Z1 and RAID-Z2 configurations and found that his 2 disk mirror with around 256GB of data on the array, it took 37 minutes for it to resilver and with 512GB of data it took 75 minutes. Interestingly he also found that regardless of how many vdevs were added to the pool, resilvering times did not seem to increase dramatically with pools ranging from vdev sizes of 2-5 taking between 43-35 minutes. He also found that a RAID-Z configuration with 5 or fewer drives seemed to have comparable resilvering times to mirrored arrays, however beyond that times did increase substantially. Finally, he demonstrated that, while resilvering times for RAID-Z2 and RAID-Z3 did increase as more drives were added to the vdev, as more drives have been allocated for parity, the longer times may not be as much of an issue as there would still be extra drives in the event of a failure and so less risk of data loss. However, it does not state whether the tests have been repeated to verify results, nor does it give the entire range disks for the RAID arrays, making it more difficult to draw patterns. Finally, it would also have been more helpful to see how a RAID-Z2 array with 2 vdevs of 5 disks performed, as this may be a more commonly used configuration than 1 vdev with 10 disks.

With this section, I need to focus on comparing the overall conclusions such as “mirrors resilver faster even if the number of drives involved is increased” rather than focus too much on the actual statistics as any results I may have, wont actually be comparable to his due to hardware differences

For filebench stuff, followed directions in

https://www.usenix.org/system/files/login/articles/login\_spring16\_02\_tarasov.pdf

Personally found that results did not stabilise enough after 3 mins

Now that all required information has been explained, onto methodology

# Test:

### Disclaimer:

Before any conclusions can be drawn it must first be clearly understood that there is no best, “one size fits all” configuration. Different workloads measure performance by different metrics, for instance, a SQL database will likely be making large amounts of small, non-sequential read and writes, whereas a video streaming server will be mostly reading large sequential files. What may be seen as well performing for one scenario may be considered badly performing for another. Certain configurations may work sufficiently well for all scenarios, however if a pool will only be used for one type of workload, performance will be left on the table. Therefore, before any layout designs are decided, the type of work that the pool will be subjected to should be carefully studied and understood, making it clear what “performance” is and how it is measured in that scenario. Furthermore, the performance of the array is not solely dictated by the arrangement of disks, and factors such as available memory, caches and other pieces of fine tweaking can drastically affect performance. Disk configuration is still a good place to start however, as certain factors such as disk failure resiliency, ashift, available storage and resilvering times are less easy to make up for once the configuration has been decided. For the purposes of this paper, no single use case will be decided upon, however certain configurations will be suggested as more appropriate for a given scenario.

### Methodology:

For the different ZFS configurations to be analysed, a test platform is needed to try out the various layouts and perform various benchmarks. The machine being used for these tests is a Dell R520, fitted with two Intel Xeon E5-2407 CPUs, 88GB of DDR3 1066 MHZ RAM, a PERC H310 controller, 6 Ironwolf 4TB ST4000VN006 hard drives and 2 WD Green 240GB SSDs. On these SSDs, the virtualisation hypervisor Proxmox is installed in a ZFS mirror, and Proxmox is running an Ubuntu Server 22.04.4 virtual machine with 4 CPU cores, 4GB of memory and the 6 4TB hard drives to be used by ZFS.

The VM contains a bash script, automating the creation of all possible ZFS configurations with 6 disk and ashift set to 12 before benchmarking them using Flexible Io Tester, also known as fio. Fio allows users to simulate any kind of workload by letting them control factors such as the file size, block size, number of files and iodepth. For this paper, three different fio tests[[21]](#footnote-22) have been adapted from recommended tests and used to roughly emulate three types of filesystem usage. These scenarios have all been tested with the read, write, rw (read/write), randread (random read), randwrite and randrw operations to collect as much data as possible, however some combinations of scenarios and operations are more commonly applicable than others and will be highlighted as such.

fio --name=randrw --rw=randrw --size=256m --bs=64k --numjobs=16 --iodepth=16 --ioengine=libaio --runtime=60 --startdelay=60 time\_based --end\_fsync=1 --directory=/zfstest --output=/home/zfstest/results/zfsresultsfio/raidz/randrw/4/generaluse

The first scenario is designed to emulate the kind of operations seen on a file server, with files of varying sizes. These can be anywhere from 4kB database entries to 10MB slideshows to 50GB videos, therefore a file size of 256MiB has been used to represent an average file size. The block size of 64KiB has also been used to best work with an average file size, with larger files not being overly fragmented and smaller files not creating large amounts of slack space, however the default ZFS recordsize of 128KiB has been kept, a factor that will be touched on later. The number of jobs has been set to 16, meaning multiple requests are being made at the same time, representing a user who is manipulating multiple files or a number of users all working off the same array. The iodepth has also been set to 16, allowing multiple operations to be grouped together to improve efficiency. It is important to note that this will not be a perfect representation of a general use file server, as they will, by their nature, contain different types of files of different sizes and so it will be impossible to provide a truly accurate representation of a file system as that does not exist. It will however, provide a good overview, from which, the group can be narrowed down and more extensively tested.

fio --name=read --rw=read --size=16g --bs=1m --numjobs=1 --iodepth=1 --ioengine=libaio --runtime=60 --startdelay=60 --time\_based --end\_fsync=1 --directory=/zfstest --output=/home/zfstest/results/zfsresultsfio/raidz3/read/6/bestcase

The second scenario is designed to emulate a favourable scenario for the array where a single, large 16GB file is split into large blocks of 1MB that, in this example, is being read from the array.

fio --name=write –rw=write --size=4g --bs=4k --numjobs=1 --iodepth=1 --ioengine=libaio --runtime=60 --startdelay=60 --time\_based --end\_fsync=1 --directory=/zfstest --output=/home/zfstest/results/zfsresultsfio/mirror/read/2/worstcase

The final scenario is designed to simulate a worst case scenario, where a single 4GB file that is split into blocks of 4kB is being written to the ZFS array. The iodepth is set to 1, disallowing the OS to group multiple writes together to improve efficiency, instead having to individually acknowledge, write and report the success of each block. The test uses the Linux native asynchronous I/O engine, delays the start of results recording for 60 seconds to allow numbers to stabilise, records the results for another 60 seconds, forces the final write to complete before the test is considered finished and then records the result to the specified directory.

### Issues:

Before the tests could be run however, a number of issues to needed to be overcome. Firstly, while creating the VM, for Proxmox to pass a disk through to a VM, it must first mount and create a filesystem in order to manage it, however doing this would violate ZFS’s principle of interacting directly with the disk. Proxmox sitting between ZFS and the disks would be similar to a RAID controller sitting between the two layers, with ZFS not being able to guarantee that the data read is the same as what was written. Furthermore, Proxmox having access to the disks, may create scenarios where the benchmark results are impacted due to Proxmox performing taxing operations of its own, such as a scrub. Ideally, the HBA that the drives are attached to should be passed through as a PCIE device, ensuring Proxmox does not have any direct access to the disks. This has not been possible in this instance as the device installed in the Dell R520 is a PERC H310 RAID Controller. While this could be overcome by installing a new HBA, the current device was instead flashed to Initiate Target or IT mode, where the controller does not perform any RAID operations on the disks and acts as a HBA. This unfortunately led to another issue where the boot drives and drives to pass through were connected to the PERC H310 in IT mode, thus when the HBA is passed through, Proxmox loses access to its own drive and the system crashes. This was remedied through Proxmox’s ability to pass individual devices through based on their ID, and so the drives can be used by ZFS without any obstruction to the bare disks. Whilst there may be some performance impact due to the ZFS disks having to share bandwidth with the boot drives, it is unlikely to be a significant amount, the ZFS disks will not be used by any other services that would cause large drops in performance and all configurations share the same environment, so all results will contain the small potential bottleneck.

After all disks had been passed through and various benchmarks were being tested, the VM began running out of memory while the tests were running, throwing the error “Out of memory: Killed process 2864539 (fio) total-vm:260900kB, anon-rss:20kB, file-rss:2476kB, shmem-rss:24kB, UID:0 pgtables:188kB oom\_score\_adj:0“. However, this only seemed to occur with specific configurations, namely a raidz2 with 4 disks, a raidz3 with 4 disks and a raidz3 with 6 disks. The VM being used to test ZFS had 2GB of memory and the tests used file sizes of 4GB, however it seemed that when performing benchmarks on configurations with a greater number of parity drives, the amount of memory required exceeded the available memory. This could perhaps have been caused by the configurations with a larger number of parity drives requiring more parity data to be calculated, thus using more memory than was available, resulting in the Out of Memory Killer shutting down processes in order of least importance. The amount of memory dedicated to the VM was cautiously increased to 4GB, resolving the issue of out of memory scenarios, however this did bring the concern of the memory being tested, rather than the storage.

When benchmarking, it is critical to ensure that the size of the test file being used in the benchmark is larger than the amount of memory, as ZFS can utilise a feature named Adaptive Replacement Cache, or ARC. The filesystem will claim a certain amount of memory for its own use and use it to cache any writes or commonly requested reads, for instance, if a system has 8GB of memory and a 2GB file is written to the pool, rather than writing directly to the disks, ZFS will instead write into memory and then write to the slower disks, giving the user the impression of a quick transfer. Whilst this is extremely useful during normal operation, when trying to benchmark storage, it can lead to inaccurate results where the only metric being tested is the amount of and speed of the memory. For this reason, the maximum size of the ARC was set to 128MiB, half the smallest file size used in any of the tests, negating any risk files being stored in memory. To test this, the same 4GB benchmark was run in the VM with both 4GiB and 16GiB of memory, both achieving around 250MiB/s, with the throughput spiking to 6000MiB/s after the ARC size limit had been removed, clearly indicating that the file had been stored in memory. As a result, all tests thereafter were performed with 4GiB of memory, 4GiB benchmark file sizes and the ARC size set to 128MiB.

As previously alluded to , memory more important in performance

Mention how dataset block size can be altered to improve performance

Plan:

Lit review

Test:

* Disclaimers
* Method ology
* Issues
* Results analysis

Conclusion

1. https://6sense.com/tech/storage-infrastructure/oracle-zfs-market-share [↑](#footnote-ref-2)
2. https://www.windowscentral.com/more-14-billion-devices-now-run-windows-11-or-10 [↑](#footnote-ref-3)
3. P2, 2nd paragraph of 2.1 https://www.cs.hmc.edu/~rhodes/cs134/readings/The%20Zettabyte%20File%20System.pdf [↑](#footnote-ref-4)
4. <https://www.usenix.org/legacy/publications/library/proceedings/usenix2000/general/full_papers/seltzer/seltzer.pdf> section 3 p3 [↑](#footnote-ref-5)
5. <https://www.cs.hmc.edu/~rhodes/cs134/readings/The%20Zettabyte%20File%20System.pdf> section 2.4 p4 [↑](#footnote-ref-6)
6. <https://www.cs.hmc.edu/~rhodes/cs134/readings/The%20Zettabyte%20File%20System.pdf> top of page 5 [↑](#footnote-ref-7)
7. https://www0.gsb.columbia.edu/faculty/lveldkamp/papers/ValueDataArticle\_aug2022\_edited.pdf [↑](#footnote-ref-8)
8. <https://users.ece.cmu.edu/~ganger/ece546.spring01/papers/Toigo00.pdf> from ZFS report p4 reference 21 [↑](#footnote-ref-9)
9. https://www.youtube.com/watch?v=dcV2PaMTAJ4 [↑](#footnote-ref-10)
10. https://klarasystems.com/articles/history-of-zfs-part-1-the-birth-of-zfs/ [↑](#footnote-ref-11)
11. https://queue.acm.org/detail.cfm?id=1317400 [↑](#footnote-ref-12)
12. https://static.ixsystems.co/uploads/2020/09/ZFS\_Storage\_Pool\_Layout\_White\_Paper\_2020\_WEB.pdf [↑](#footnote-ref-13)
13. https://static.ixsystems.co/uploads/2020/09/ZFS\_Storage\_Pool\_Layout\_White\_Paper\_2020\_WEB.pdf [↑](#footnote-ref-14)
14. https://static.ixsystems.co/uploads/2020/09/ZFS\_Storage\_Pool\_Layout\_White\_Paper\_2020\_WEB.pdf [↑](#footnote-ref-15)
15. https://queue.acm.org/detail.cfm?id=1317400 [↑](#footnote-ref-16)
16. https://www.researchgate.net/publication/345489155\_State\_of\_the\_art\_and\_future\_trends\_in\_data\_reduction\_for\_high-performance\_computing [↑](#footnote-ref-17)
17. https://picture.iczhiku.com/resource/paper/shIdkpUhfRUjHbcx.pdf [↑](#footnote-ref-18)
18. https://jro.io/r2c2/ [↑](#footnote-ref-19)
19. https://picture.iczhiku.com/resource/paper/shIdkpUhfRUjHbcx.pdf [↑](#footnote-ref-20)
20. https://louwrentius.com/zfs-resilver-performance-of-various-raid-schemas.html [↑](#footnote-ref-21)
21. https://arstechnica.com/gadgets/2020/02/how-fast-are-your-disks-find-out-the-open-source-way-with-fio/ [↑](#footnote-ref-22)