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

What workloads are each of the ZFS configurations most appropriate for?

Matthew Swain

20200873

Contents

[Abstract: 2](#_Toc173592534)

[Introduction: 2](#_Toc173592535)

[Literature Review: 2](#_Toc173592536)

[Understanding File Systems: 2](#_Toc173592537)

[The Need for ZFS: 2](#_Toc173592538)

[The Creation and Philosophy of ZFS: 4](#_Toc173592539)

[ZFS Configurations: 7](#_Toc173592540)

[Stripe: 7](#_Toc173592541)

[Mirror: 7](#_Toc173592542)

[Vdevs and Striped Mirror: 7](#_Toc173592543)

[RAIDZ: 8](#_Toc173592544)

[Investigation: 9](#_Toc173592545)

[Methodology: 9](#_Toc173592546)

[Environment: 9](#_Toc173592547)

[Benchmark Software: 9](#_Toc173592548)

[Issues: 11](#_Toc173592549)

[Analysis: 12](#_Toc173592550)

[Preface: 12](#_Toc173592551)

[Filebench: 12](#_Toc173592552)

[Flexible I/O Tester: 15](#_Toc173592553)

[Filesystem Tuning and Optimisation: 24](#_Toc173592554)

[Conclusion: 25](#_Toc173592555)

# Abstract:

ZFS is an extremely in-depth file system and the ideal configuration for a given workload can be endlessly debated, tuned and optimised. There is no one size fits all solution and each configuration sacrifices one aspect in exchange for another. However, some of these aspects can be offset by resources outside of the disks, such as memory that can be used as a cache to greatly improve performance. Because of this, it may be worth placing disk failure tolerance at a higher priority, therefore making a single vdev RAIDZ2 a good general performer at 6 disks. At higher disk levels, multiple 6 disk RAIDZ2 vdevs make a good option as the storage and disk failure balance is maintained and higher vdev numbers resolve the suboptimal random performance numbers. Alternatively, if data integrity was of the utmost importance, RAIDZ3 vdevs could be used. At lower disk levels, single or multi vdev mirror arrays make reasonable options dependant on whether performance and data integrity or storage capacity is of higher priority. Single vdev RAIDZ1 arrays also offer more capacity in exchange for performance. Pure stripe arrays are rarely a good option unless the maximum performance is required and there are other, easily accessible copies of the data.

# 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 (6sense, n.d.) 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. Since then, it has continued to add functionality that has only made it even more competitive. Despite this, if the storage array chosen is inappropriate for its workload, it may not be as performative or reliable as expected. Therefore, 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 store the data 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), Window’s default file system, fourth extended filesystem (ext4), the default file system for many popular Linux distros, and Apple File System (APFS), Apple’s default file system. When a user wants to create a file system, they must select the drive they wish to use, partition it 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 available space into blocks of a size chosen by the user, with a common default being 4096 bytes or 4 kibibytes (1024 bytes). A block is an atomic unit of data storage, and so any stored file needs to be broken down into as many blocks as necessary. For instance, a 1 gibibyte file would need to be split into 262,144 blocks of 4KiB. Once this is done, the file system will then send requests to the drives to write these blocks and retrieve them again when the user requests.

These traditional file systems are relatively easy to setup, performant and reliable enough for most users and are therefore used by billions of devices (Endicott, 2022). However, they been designed mainly for use in desktops and laptops and therefore not always suited to situations outside this. Whether it be an individual or a large company, there will always be data that should not be lost, and this is a task that a single drive cannot perform. For this reason, many disks can be grouped together in a RAID (Redundant Array of Independent/Inexpensive Disks) configuration, allowing resilience against data loss, high performance and an ease of management. The aforementioned file systems are not always capable of achieving these goals in complex scenarios.

## The Need for ZFS:

And so it was, during the late 1990s, a typo in a command during routine maintenance on Sun Microsystem’s Jurassic server left a building full of a thousand employees with nothing to do until the data restoration was complete. During this wait, Jeff Bonwick and Tim Marsland began lamenting Sun’s inability to create a new filesystem. There had been several attempts to create a new solution which may have helped resolved the difficulty involved with performing basic administration (Klara Inc, 2021). As it stood, if a system administrator wanted to perform basic operations such as growing and shrinking a file system or adding storage to a partition, it was either not possible or overly convoluted. 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. 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 later (Bonwick et al., 2003, p. 2).

There were also issues with data integrity as most file systems used a write method called journalling. A commonly performed operation is a meta-data operation that modifies the structure of the file system such as creating, deleting or renaming a file or directory. Whenever one of these operations was performed, before it was started, the system would record the change it was making into a log (Seltzer et al., 2000, p3). In theory, this 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. In practice however, Bonwick 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” (Bonwick et al., 2003, p. 4).

Data corruption could also be caused by the drives, drivers and controller as, during this time, RAID was achieved using a RAID controller. This device, that all drives would be connected to, would create the desired RAID configuration and present the array to the operating system as one device, handling all 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 would not always be the case. It led to issues as the controllers would not always have extensive error handling or any error handling at all leading to issues with bit rot (Bonwick et al., 2003, p. 5). 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 and controller drivers. These bugs can then 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). Critically, 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 extremely undesirable for a consumer, but in business where data is the most valuable asset (Veldkamp, 2023), it is particularly unacceptable. It is estimated that 93% of businesses that experience data loss for more than 10 days will file for bankruptcy within one year (Workspace, n.d.).

Finally, storage capacity had far outstripped Moore’s law and had been doubling and the cost halving every 9-12 months (William Toigo, 2000, p. 59). Because of this, 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. 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 file systems 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” (OpenZFS, 2015). Unfortunately, the conversation ended soon after as the data restoration was complete and both went back to their respective projects.

## The Creation and Philosophy of ZFS:

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 (Klara Inc, 2021). After an attempt with too many engineers divided between too many locations failed, in July of 2001, Bonwick took another attempt at the project. This time, the team consisted of just him and an engineer named Matthew Ahrens and by October they had a working prototype (Klara Inc, 2021). 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 (Klara Inc, 2021). Progress continued with the team starting to use ZFS to store their own files in 2004 and an integration into Sun’s internal version of Solaris on the 31st of October 2005. Finally, it was released to the public in May of 2006 with Sun’s free and open-source Solaris 10 operating system (Klara Inc, 2021).

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 and physical device, 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” (Stanik, 2007). They accomplished this by removing the separation between the volume manager and the file system. Instead, they replaced 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.

A screenshot of a computer

Description automatically generated

*Traditional file system block diagram (left) vs the ZFS block diagram (right)* (Bonwick et al., 2003, p. 6)

Beginning with the bottom layer, the SPA congregates all the physical devices together to form one large pool of blocks. It then 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. Rather than applying a logical volume over a single or multiple disks, this method of disk management allows loose coupling between the storage and disks. If disks are removed or added, the layers above the SPA do not know, instead seeing that the pool of available blocks will have grown or shrank. Consequently, this also allows arrays to be grown or shrunk much more easily as the number of blocks available to an array can be adjusted without having to change what parts of the disks the array can use.

Another reason for taking this approach was because, during the same interview, Moore also stated that one of the key design principles for ZFS was “never, ever trust the underlying hardware” (Stanik, 2007). This was because, 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. This is a calculation that is performed on the data and should always yield the same result. If the result is different, then the data will have changed, possibly due to corruption. In the case of ZFS, the result of the checksum is stored in the parent block, going 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, have already been retrieved to have retrieved the child block in the first place. 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 then repair the data based on the correct data from other disks (Bonwick et al., 2003, p. 6-8).

Moving up a layer, the Data Management Unit handles the conversion between the blocks presented by the SPA and the files that live within the dataset, the rough equivalent of a volume (Salter, 2020). 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.

Rather than the edited blocks being overwritten, they are copied to a new block with the changes applied. The parent block is then also copied to a new block with an updated pointer and the process repeats up to the überblock. Once the überblock is reached, it is rewritten in place in a single transaction, instantly switching from one tree of blocks to another (Bonwick et al., 2003, p. 8). The überblock also contains its own checksum, enabling the filesystem to detect if any corruption has occurred and restore from a backup überblock. This, along with the change not being committed until all blocks along the tree have been updated, ensures that a power loss event can never result in data loss. It also does not take up any extra space as any blocks, even if they contain data, can be overwritten if they are not pointed to by a parent block (Bonwick et al., 2003, p. 8-9).

The lack of extra space required for old blocks also gives rise to another of ZFS’ main features, called snapshots. When a snapshot is taken, a copy of the current überblock is taken and stored separately. This separate überblock will not be changed and, as any changes to the files do not result in the old data being overwritten, the blocks that the old überblock point to still exist. In this way, the state of the filesystem can be rolled back to a previous snapshot simply by restoring an older version of the überblock, which will point to the older versions of the blocks. This also uses a very small amount space as the only data required to be stored is the überblock (Salter, 2020). However, as the filesystem starts to fill up, ZFS will not allow blocks pointed to by any snapshots to be overwritten and so the amount of available space will decrease (Oracle, n.d.). For this reason, snapshots are usually only taken and kept on a fixed schedule, for instance, snapshots may be taken every day and kept for a month before being deleted. It is also for this reason that snapshots are not a form of backup, as, if the array fails, all data will still be lost. Instead, they can be seen as a file system wide undo, useful for situations such as reverting an array infected with ransomware to a previous state where it did not. This can, however, be used in conjunction with replication, a feature that allows the überblock from a snapshot and all its referenced blocks to be copied to a separate device, creating a full backup (Salter, 2020). Using snapshots, replication and schedules, allows for several layers of redundancy to be created, greatly minimising the risk of data loss.

This system also allows for another of ZFS’ main features, which is deduplication. If an array is storing large amounts of duplicate data, this feature allows for all the duplicate blocks to be erased. That duplicate data is instead written to a separate block and the deduplication table, or DDT, keeps track of what data contains duplicate blocks and where the block containing the deduplicated data is stored. This can save large amounts of space if there are large amounts of duplicate data, however the overhead for handling this data is significant. For best performance, the DDT should be stored in RAM, and so large amounts of memory are required, taking away from the amount the main system can use. Additionally, finding the on-disk location of the deduplicated data will mean having to frequently seek small, random blocks, a task that HDDs, with heads that must physically move, are poorly suited to. It is therefore recommended that deduplication be used with SSDs, which do not have any moving parts and are therefore give much better performance for these random operations. Finally, reading a data stream that contains duplicate data, finding the location of the data in the DDT and then retrieving the data from its centralised location is a very CPU intensive operation and so should not be used with weak processors (TrueNAS, n.d.).

The final ZFS POSIX layers main responsibility is to handle the conversion between ZFS operations and the Portable Operating System Interface, or POSIX. This ensures compatibility with all Unix-like systems by using a file system management standard containing a rich number of features and simplifying management. It also helps ensure data integrity and performance by grouping related changes together into one atomic operation, ensuring there will never be any data inconsistencies (Bonwick et al., 2003, p. 9-10). All individual changes are also written into the ZFS Intent Log, or ZIL on disk, so if power is lost while changes are being grouped, there will be a non-volatile list of the changes ZFS needs to commit (Kennedy, 2017).

It may seem that ZFS sacrifices performance with the number of data integrity operations included, however it does not. The exponential increase in processor speed renders any performance impact negligible, demonstrated by researchers in 2016. They found that, using the Bonnie++ benchmarking software, a system with checksum disabled only had read speeds increased by 2.6 MB/S and write speeds by 0.1 MB/s (Widianto, Prasetijo and Ghufroni, 2016).

ZFS has also been benchmarked against the more traditional ext4 filesystem as well as B-Tree File System, or Btrfs. Pronounced “butter F S” and released in 2009, it also emphasises data integrity 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, 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, but in 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 (Heger, 2009).

However, in a similar paper in 2016 written by Anders Lundholm, when using another RAID10 comparable configuration, with the IOzone benchmarking tool, he achieved different results. Rather than testing with a fixed block size, or recordsize as it is known in ZFS, he tested from 64 KB blocks to 16,384 KB blocks to see how the file systems might differently react. He found that 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. This is unexpected as, despite the default recordsize being 128KiB, ZFS uses variable block sizes, meaning a 4KiB file will be stored in a 4KiB block. This should resolve any potential performance issues with smaller files as the file system will not need to read and write out 124KiB worth of blank data, however this does not seem to be the case here. Lundholm theorises that, despite specifying smaller block sizes, e.g. 64kB, the software still writes out larger files and only includes 64kB worth of data in them. This would hugely disadvantage ZFS as it would see the larger file sizes and therefore use the maximum block size, but only a small part of the block is useful information. This means that ZFS is having to read and write out a large amount of blank information, leading to much slower random performance (Lundholm, 2015, p. 27). This, however, would only be true at smaller recordsizes, and ZFS continues to perform worse at higher recordsizes. This suggests that the extra data integrity features could have some impact on performance.

This pattern does not continue with sequential performance however, with ZFS having the highest at an average of 310 MB/s. Btrfs and XFS follow at around 270 MB/s and ext4 is last at around 240 MB/s (Lundholm, 2015, p.61-81). This could suggest that in the smaller, less efficient operations, the extra data integrity features may cause performance loss, but in the larger, more efficient operations, they may not.

It is important to remember that the results of these two papers cannot be directly compared. The arrays used by the two authors are different configurations with a differing number of disk and so direct numbers cannot be meaningfully compared. The studies were also conducted 7 years apart and so it is extremely likely that perform improvements to all file systems tested have been made in that time. However, this does show that while the extra data integrity features of ZFS may have some effect on performance, the largest impact is caused by the choice of configuration. Above all else, the choice of disk layout must be appropriate for the given workload, otherwise performance will be lacking.

## ZFS Configurations:

As choosing the right configuration is critical, it is important to understand what options are available, along with their strengths and weaknesses.

### Stripe:

The first, most basic type of array configuration is the stripe. In this arrangement, whenever data is written to the array, each block is only written to one disk, cutting the number of actions a single disk performs before the operation is complete. For instance, if a file consists of 6 blocks, in a 6 disk array, each disk must only write one block of data, rather than 6. Because of this, both random reads and writes, that is, small operations that are not in the same location and therefore cannot be grouped together for greater speeds and efficiency, are considerably faster. Sequential reads and writes, large operations that tend to be for the same file and can therefore be grouped together for greater speeds and efficiency, are also increased (iXsystems, 2020, p. 2-3).

The biggest weakness for this arrangement is that the array is extremely vulnerable to data loss through disk failure. This is because, if one disk in an array fails, the data it stores has not been stored on any other disk and has therefore been irretrievably lost (iXsystems, 2020, p. 2-3). This configuration type is therefore only suitable to scenarios where performance is of the utmost importance at the expense of all else. Alternatively, this configuration could be suitable in a scenario where the stored data could easily be retrieved again a disk failure causes the array to cease functioning.

### Mirror:

Mirror arrays take the opposite approach, writing every block to every disk in the array. This means that almost every single disk in the array could fail, but as long as one disk remains, no data loss will occur. This does mean that, as all data is mirrored across all drives, the amount of available storage is very low, only ever being that of a single drive. Similarly, write speeds are also severely hampered, never being able to increase beyond the speed of a single drive. For instance, if 6 blocks were being written to a 6 disk array, the operation would not be complete until those 6 blocks had been written to every single disk in the array.

This is not reflected in random and sequential read speeds however, as, referring to the previous example, if 6 blocks were being read from the array, one block could be retrieved by each disk. This reduces the number of actions a single disk must perform, therefore increasing the speeds substantially (iXsystems, 2020, p3-4). Overall, the mirror configuration prioritises data integrity above all else and therefore isn’t always suitable for workloads that require fast write speeds and large amounts of storage. If only 2-3 disks were available for the array, a mirror may be a suitable choice as there would be a lower percentage of performance and storage space sacrificed.

### Vdevs and Striped Mirror:

To receive the advantages of a 2-3 disk mirror and those of a striped array, multiple vdevs may be utilised to create a striped mirror. A vdev, or virtual device, is a feature of ZFS that allows multiple drives to be grouped together and treated as one logical drive (Bonwick et al., 2003, p. 7). A popular use of this feature is to create multiple vdevs of 2-3 mirrored drives and put them into a striped array. For example, there may be an array that contains 3 vdevs in a stripe configuration, with each vdev containing 2 mirrored drives. If 6 blocks were to be written to this array, each vdev would receive 2 blocks which would then be written to both disks. This provides a good mix of the performance of a stripe and the resiliency of a mirror. Predictably, read speeds are also very fast as, like a pure mirror array, disks can be coordinated to retrieve different blocks, reducing the number of actions an individual disk must do.

This arrangement does give up the amount of storage space available in exchange for the increased performance. As one vdev will always contain at least 2 drives in a mirrored configuration, the amount of storage space will never be greater than 50% of the total amount of storage. Additionally, due to the mirroring, up to 3 drives could fail before there is any data loss, however these 3 drives must all be from different vdevs. If 2 drives in the same vdev fail, a section of the striped data would be irretrievably lost and the array would fail (iXsystems, 2020, p5-6). This configuration is therefore suitable for workloads such as a storage area network, or SAN, where small amounts of storage may be used as boot devices for work machines. Here, performance is essential as operating systems perform many random read and writes, and a certain amount of drive failure resiliency is required to prevent disruption. If critical work files were then stored on a more resilient network area storage, or NAS, in the event that a vdev were to entirely fail, no essential data would be lost. Instead, there would simply be a small amount of down time while the work machine operating systems were restored.

### RAIDZ:

The final configuration option aims to provide a better balance performance, fault tolerance and storage availability at the expense of a higher level of complexity. The RAIDZ configuration is comparable to traditional RAID 5 and 6 arrays, where a certain number of disks worth of space are yielded to be used for parity data. This is extra information added when data is written to the array that ZFS can use to reconstruct lost data, allowing a certain number of disks to be lost without array failure. The number following RAIDZ refers to how many disks worth of space is used for parity data. For example, in a 6 disk RAIDZ2 array, up to 2 disks could be lost without any data loss as two disks worth of space are used for parity data. ZFS can then perform calculations on the remaining normal and parity data to work out what the lost disk would contain and then write it to a replacement disk in a process known as resilvering.

Predictably, reading and writing to a RAIDZ pool is more complicated than a stripe or mirror. Normally, when reading or writing a file to an array, the file is split into blocks, the size of which is determined by recordsize. If the file size is smaller than the recordsize, it will be stored in a single undersized block. If it is greater than the recordsize, it will be split into multiple blocks which will be written to the array. However, once the block is handed to the vdev, it is then split down further into sectors. These are the units of storage a device stores data in and is set by the ashift property. For example, most modern disks have a sector size of 4KiB and so a 128KiB block will be split needs to be split into 32 sectors and distributed among the disks in the vdev.

For a stripe vdev, this is a simple process as the sectors are striped sequentially across the disks. This means that if a random read or write is performed, only one disk needs to be utilised, leaving the other disks free to perform their own operations. Similarly, if a random read is performed on a mirror array, as all drives contain the same data, one drive can retrieve all relevant sectors for the block while the other drives can perform their own reads. For RAIDZ however, the file system must work out the parity sectors and then stripe the data and parity sectors across every disk. As a result, when a random operation occurs, all disks must be accessed to retrieve their sectors to form the block for ZFS to operate on. Thus, the file system must wait for the last drive to finish retrieving its sectors before a full block can be formed and the operation completed. For this reason, the random performance of a RAIDZ vdev is only as fast as its slowest drive.

Sequential operations on the other hand scale well with adding more drives, as, despite the file system still needing to wait for each disk to finish its sector operations, other disks can start on the next operation in the group of operations while the last drive finishes. As a result, unlike the random operations which are not grouped together for greater efficiency, there is never any down time while the file system waits for the last disk to finish before starting another, different operation (iXsystems, 2020, p. 6-7).

If multiple RAIDZ vdevs are used, random performance can increase by the number of vdevs. For instance, in an array containing 3 vdevs with 2 disks in a RAIDZ1 configuration, if 3 128KiB files were stored, each vdev would receive one 128KiB block. The file system can then perform random operations on each of those 3 files as one vdev does not have to wait for any of the other vdevs to finish retrieving sectors before the operation is completed. Therefore, random performance would increase to three times that of a single disk. For this reason, RAIDZ arrays may only be appropriate where a large number of disks are available to create multiple vdevs. Alternatively, they may also be appropriate if random performance is of secondary importance.

# Investigation:

## Methodology:

### Environment:

To test what situation these configurations are most appropriate for, a test platform is needed to compare 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 Proxmox hypervisor is installed in a ZFS mirror and running an Ubuntu Server 22.04.4 virtual machine with 4 CPU cores, either 2GB or 4GB of memory and the 6 4TB hard drives to be used by ZFS. Inside this VM contains a script automating the creation of all possible ZFS configurations capable with 6 disks and after each one is created, a benchmark is run on it.

### Benchmark Software:

#### Filebench:

The first set of tests are run with Filebench, a piece of storage benchmarking software created by Sun Microsystems in 2002 and open-sourced in 2005. It allows users to create micro or macro workloads to test their storage arrays, however, it also includes a series of pre-made workloads such as a mail, web or OLTP server. This gives users a starting point for the kinds of behaviours that tend to be seen in those systems while still allowing them to further tailor them to their needs. For a wide range of use cases, the file, video and mongo servers have been chosen.

The first workload, the fileserver, consists of a mix of different operations including creating a new file, writing 1MB to a file, closing a file, opening a file, appending 16kB to a file and deleting a file. The operation has a total of 35,000 files, creating a dataset that is larger than the 2GB of memory the system has during the Filebench tests. The number of instances is also set to 50, which with the large number of files, creates a workload very similar to that of a busy fileserver. As a result of the large amount of random read and write operations, this scenario is likely to favour configurations that can perform lots of mixed, medium sized operations.

The second workload, a video server, contains two filesets of 2GB files. In the first fileset, the active videos fileset, there are 16 videos and 48 instances of the files receiving continuous reads of 256kb, possibly representing a new video being streamed by many people. The second passive videos fileset, contains 48 videos but only 1 instance, performing deletions, creations and writes every 10 seconds. This could represent videos that are no longer as popular being moved off the main, high speed streaming server to a slower secondary server. Once these have been removed, they are then replaced by new videos that are more in demand. Although there are some operations aside from read, the majority are and so this workload is likely to favour those configurations that can provide high sequential read speeds.

The final Filebench workload represents the type of behaviour that would be seen in a Mongo database. As a result, the fileset contains 300,000 16kB files and has one instance performing various operations including opening a file, appending 16kB to a file, closing a file and deleting a file. This scenario does not contain any larger continuous operations, only smaller random ones and so will be testing how a configuration performs in a more challenging environment.

All workloads have been tuned to fit the creators’ recommendations, as the default tests may not give useful or accurate results without editing (Tarasov, Zadok and Shepler, 2016, p. 11-12). The default data-set size of all workloads is not large, with the web servers only being 16MB, easily capable of being cached in memory and skewing the results. For this reason, all data set sizes have been increased to 4GB, double the 2GB of memory allocated to the machine. Similarly, the default run time of all jobs is 60 seconds, however this is usually not long enough to allow results to stabilise and provide an accurate result and so the run time has been increased to 10 minutes. Finally, the commands “create fileset” “system ‘sync’” and “system ‘echo 3 > /proc/sys/vm/drop\_caches’” have been added, forcing the OS to erase its caches before the test starts and preventing any files that may have been cached in memory from affecting the results. Each test was then run 5 times for each configuration and the average calculated, ensuring results are consistent and any potential outliers would have minimal effect on the overall performance.

#### Flexible I/O Tester:

The simplified and easy to understand results from Filebench are helpful in getting a broad understanding of how each configuration may perform, however, to analyse more granular statistics, Flexible I/O Tester, or fio, has also been used. Fio allows control over factors such as the operation type, file size, block size, number of files and iodepth, giving users much more granular control over the tests and allowing them to more precisely tailor the benchmark to their desired workload. For this paper, three different fio tests have been used, representing three different datasets and have been adapted from recommended tests (Salter, 2020a).

The first test represents a general use dataset, designed to show how the configuration would perform handling a wide range of files, anywhere from a 4kB database entry, a 10MB slideshow or a 50GB video. It is not possible to give a truly average file size for all users, as the average file size for a user differs, however for this test, a file size of 256MiB is used as a rough average. The block size, representing the size of the operations being performed on the dataset, has been set to 64KiB, meaning small but significant read and writes are being made to the files. The number of jobs has been set to 16, making the array a busy system receiving multiple requests, and the iodepth also being set to 16 allows the OS to group multiple requests together and run them asynchronously, improving efficiency. Fio also allows the user to set which ioengine is used, letting them see how each engine’s method of file handling affects performance. For this test, the default libaio or the Linux native asynchronous I/O engine is used to limit the number of independent variables. Each test then runs for 60 seconds, with a 60 second pause beforehand to allow readings to stabilise before the results are recorded after the last operation has finished.

The second test is a best-case scenario where the array is storing large files that are receiving large operations, possibly representing large video files. To this extent, the data-set size has been set to 16GiB the block size has been set to 1MiB. While the number of jobs and iodepth have only been set to 1, the large block and file size should allow the test to reach high speeds.

The final test is a worst-case scenario for the array, where the 4GiB dataset is receiving small 4KiB operations. The number of jobs and iodepth are also set to 1, disallowing the OS to improve efficiency by running multiple operations asynchronously. These kinds of files may often be dealt with by an array running an operating system, which would be performing smaller, random tasks, such as writing to a log file, interacting with a web browser or sending an email.

As fio allows the user to test random and sequential read, write and mixed read/write speeds separately, the previously mentioned tests have been tested with all these operations to get as wide a spectrum of information as possible. Additionally, like the Filebench tests, the default ZFS block size of 128KiB was used and the fio tests were repeated multiple times, in this case 6, with the results averaged. However, unlike the Filebench tests, these tests were run with 4GiB of memory.

### Issues:

Whilst trying out various types of fio tests to find the most suitable, 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, more memory was required 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 utilise a certain amount of memory 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 4GiB benchmark was run in the VM with both 4GiB and 16GiB of memory, both achieving around 250MiB/s. The ARC size limit was then removed and throughput spiked to 6000MiB/s, clearly indicating that operations were happening in memory. As a result, all FIO tests thereafter were performed with 4GiB of memory, 4GiB benchmark file sizes and the ARC size set to 128MiB.

However, before these tests could even be run, issues were encountered during the creation of the VM that would run the tests. Firstly, while creating the VM, for Proxmox to pass a disk through to a VM, it must first mount and create a filesystem to manage it. However, doing so 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 then 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. Additionally, 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.

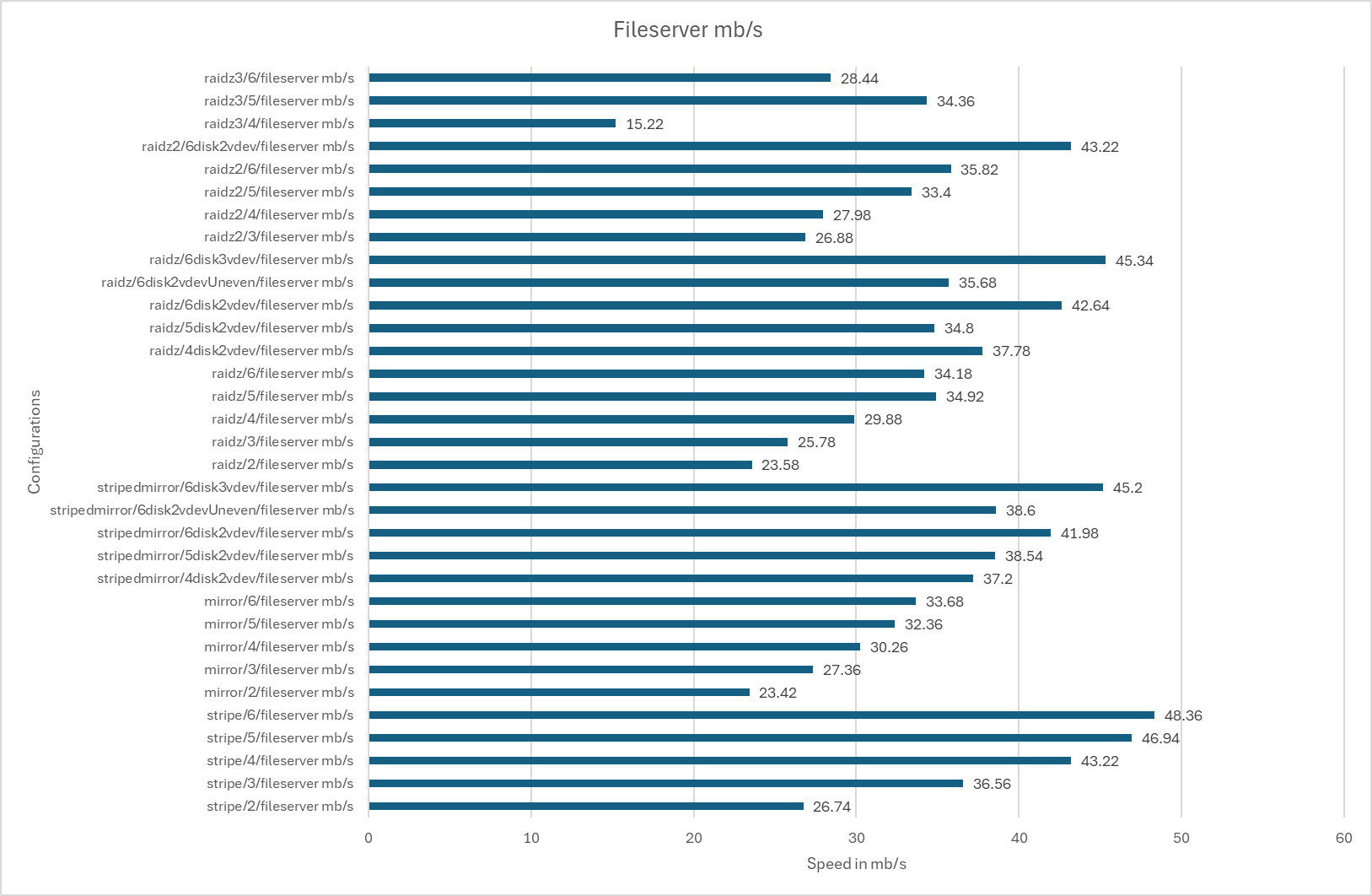
## Analysis:

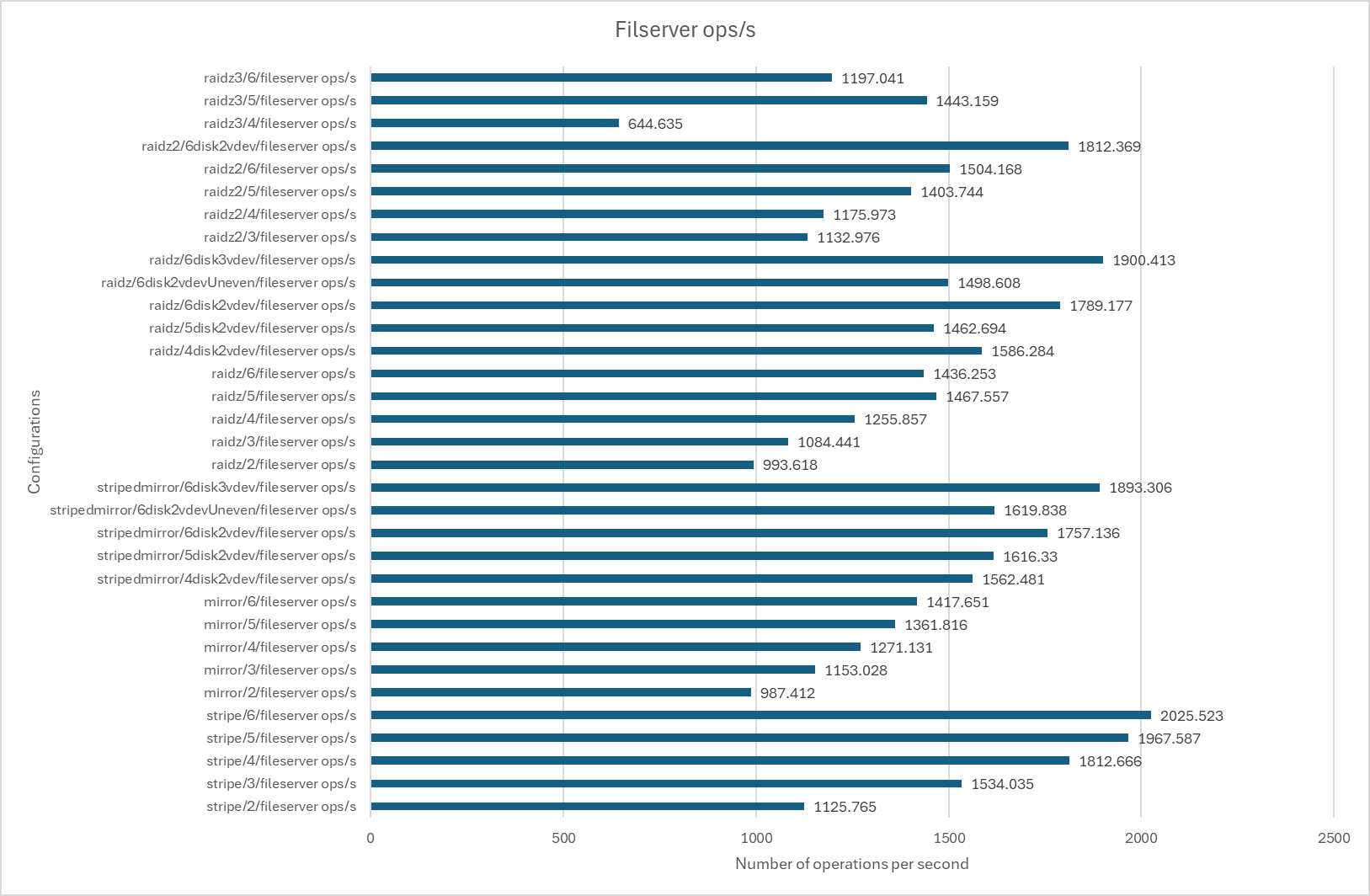
### Preface:

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, some performance will be lost. 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 tuning 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 instance, a pattern that will likely be seen throughout the results is that an array with 6 disks in a stripe will be the best performing, and would be able to perform even better with the addition of more memory, caching disks, tuning etc. This, however, ignores the critical issue that the data contained on that array will be at an extremely high loss risk, something that nothing other than a second backup device can resolve. 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.

### Filebench:

#### File Server:





Beginning with the file server workload from Filebench, the reported mb/s and ops/s numbers show the same pattern, with a 6 disk stripe array unsurprisingly achieving the highest performance of 48.36mb/s and 2025.523 ops/s.

It is closely followed by more practical arrays containing multiple vdevs, with the top 3 being a RAIDZ1 array with 3 vdevs of 2 disks, a striped mirror array also with 3 vdevs containing 2 disks each and a RAIDZ2 array with 2 vdevs of 3 disks.

Single vdev arrays lag significantly behind with the best performing array, a 6 disk RAIDZ2, only achieving 35.82mb/s and 1504.168 ops/s compared to the previously mentioned RAIDZ1 arrays performance of 45.34mb/s and 1900.413 ops/s. This large performance delta is due to the array assigning blocks to different vdevs, allowing each vdev to complete its block operation asynchronously.

Configurations otherwise scale in an expected manner, with performance increasing as more disks are added to the array. A 6 disk RAIDZ2 array offers the best single vdev performance at 35.82mb/s and 1504.168 ops/s while offering a good balance of storage space and resilience. It allocates less space for parity than a 3 vdev mirror, however as the RAIDZ2 array is only a single vdev, 2 drives can fail before data is lost. In the striped mirror arrays, if 2 drives in the same vdev fail, all data will be lost. As can be seen in the below figure from Reliable RAID Configuration Calculator (Rose, 2014), while the probabilities of 2 drives in a single vdev failing are low, the striped mirror array does have a higher chance of complete array failure. At disk numbers greater than 6, this could become less of an issue, however at the number of disks tested, it should be considered whether the better performance is worth the array resiliency and storage space trade-offs.

A graph with a line

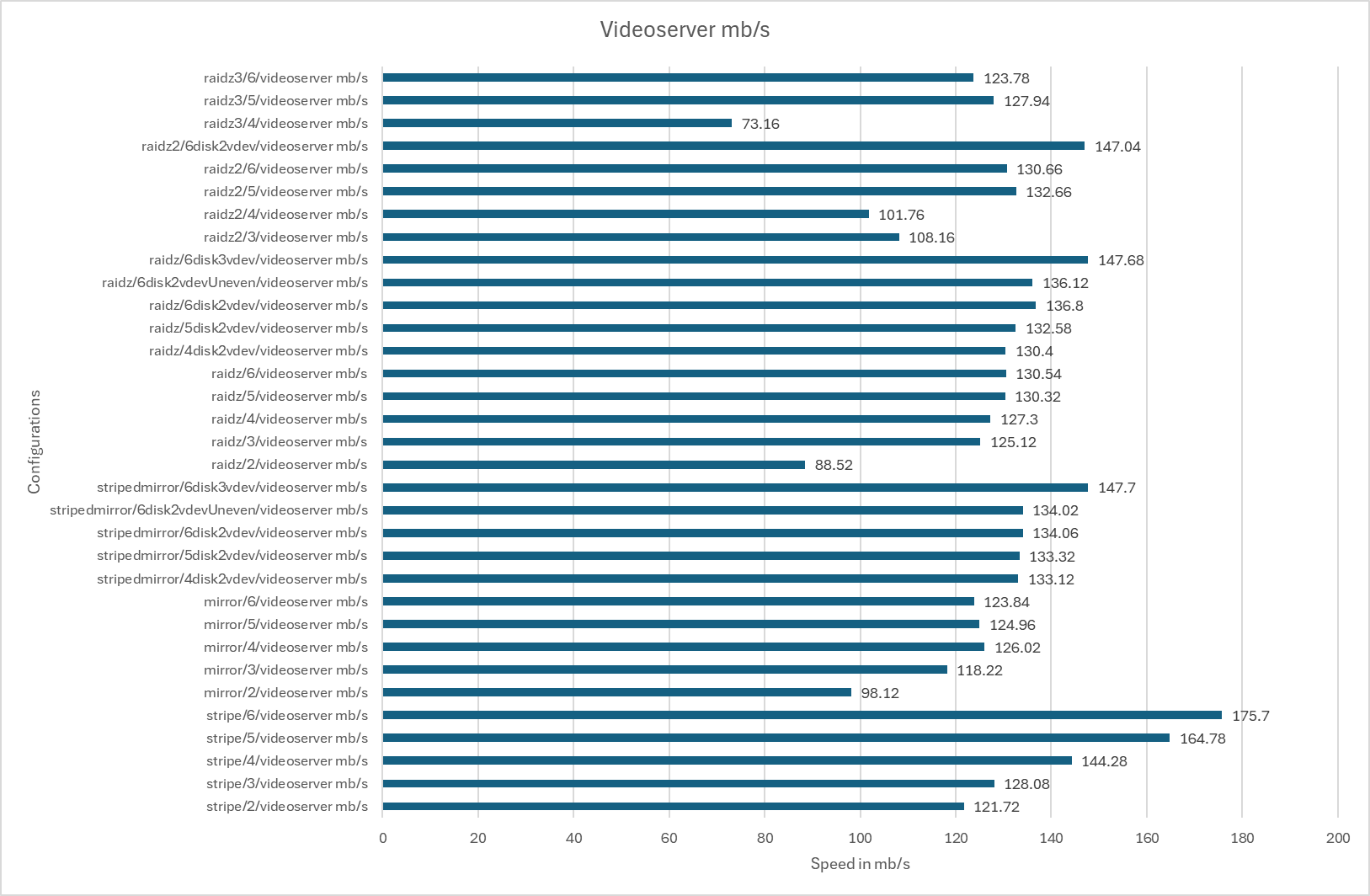
Description automatically generated

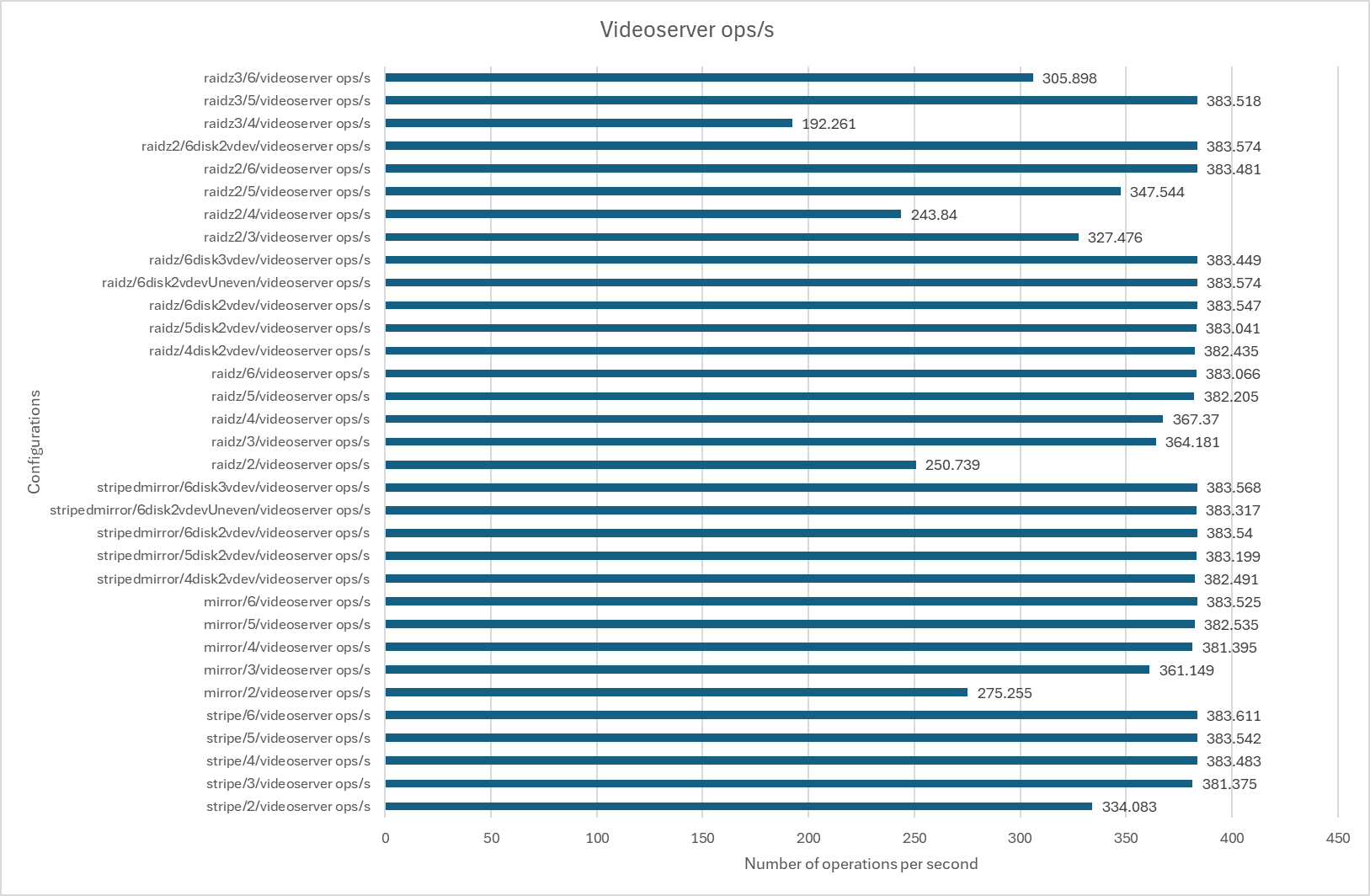
The mirror arrays also perform slightly unexpectedly as performance does increase when drives are added despite there being a significant amount of write operations in the workload. An increase in the number of disks should not result in a performance increase as all blocks need to be written to every disk in the mirror array. The workload does, however, feature a number of read operations, something that mirror arrays do see a performance increase from as drives can asynchronously retrieve separate blocks. As the workload features a mix of read and writes, the increased read speeds in the single vdev mirror arrays likely make up for the lack of improvement in writes, leading to an increased average performance.

One of the major outliers is the 6 disk RAIDZ3 array which reported only 28.44mb/s and 1197.041 ops/s, a significant decrease in performance compared to a 5 disk RAIDZ3 which achieved 34.36mb/s and 1443.159 ops/s. As the 5 disk array performed better than the 4 disk array, which scored only 15.22mb/s and 644.635 ops/s, the expected outcome would be for the 6 disk array to perform even better.

The reason for this could perhaps be that, with 6 disks, ZFS must wait for a greater number of disks to finish their sector operations, thus reducing IOPS. While the continuous operations would have improved with more disks, as this is a mixed workload, the random performance may have dragged the average score down. Comparatively, the 5 disk array may have outperformed the 4 disk array as the second data disk would have significantly improve continuous performance. While the random performance would have remained the same or potentially worsened, the better continuous performance may have dragged the average score up.

#### Video Server:





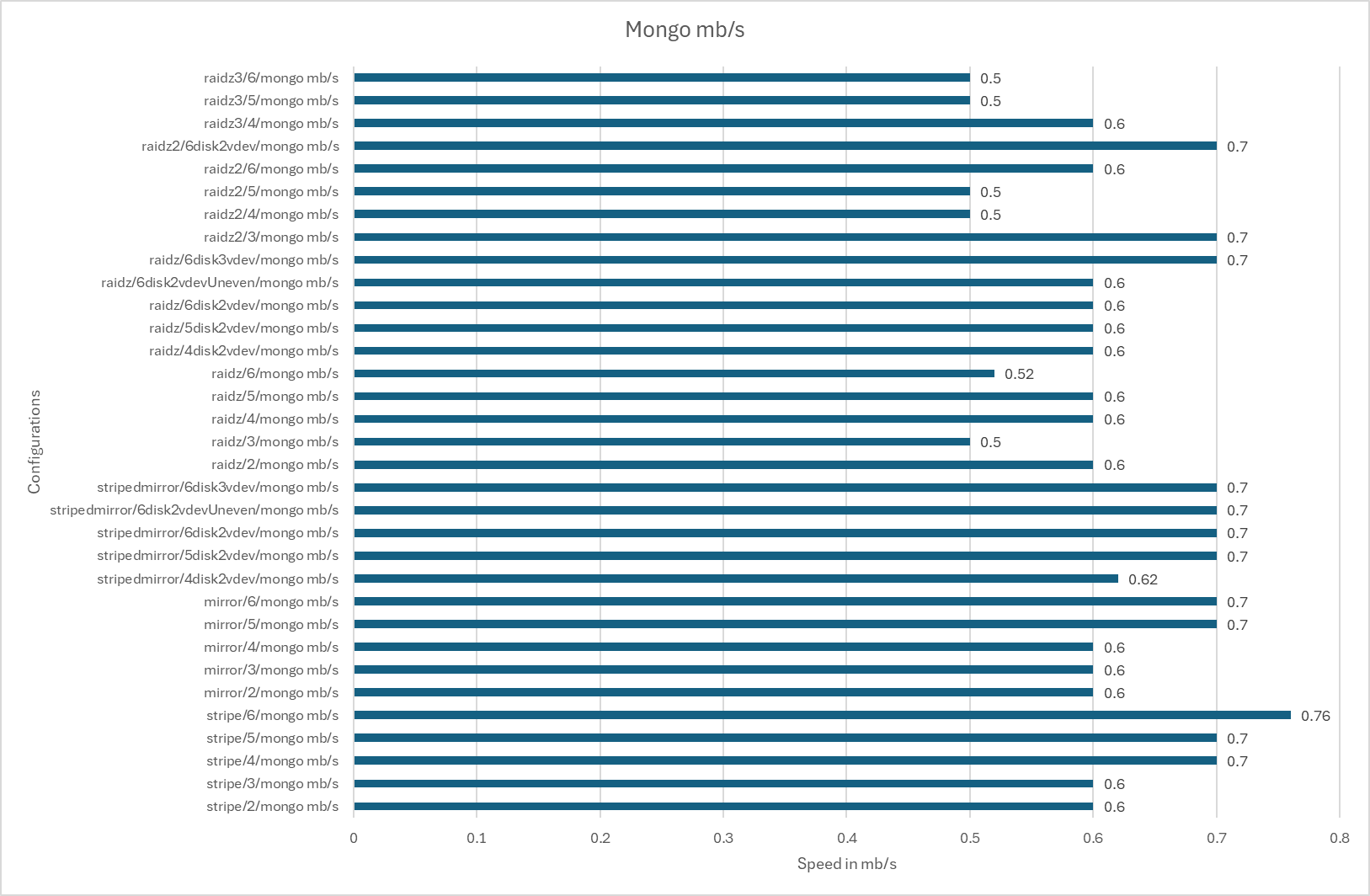
As can be seen in the op/s chart, many configurations perform very similarly and so it is difficult to draw many conclusions from the results. This is because IOPS are a measurement of random performance, something that the videoserver workload does not contain. Comparatively, the mb/s chart gives much more informative results as this measures the continuous performance, the main operation type seen in this workload. Because of this, analysis will mainly focus on these results.

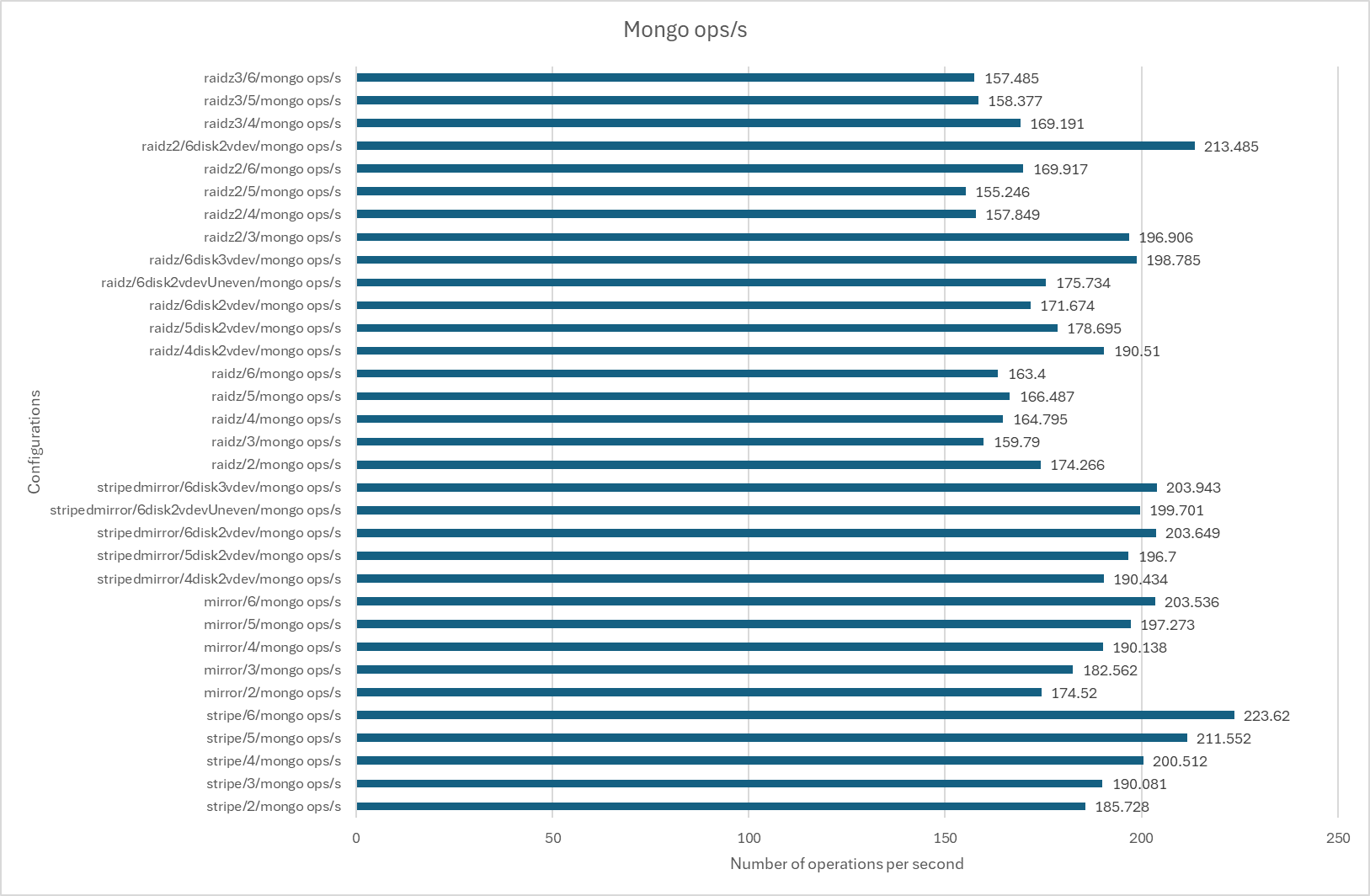
Predictably, the 6 disk stripe array provides the highest continuous operation performance numbers, however, in this scenario, it may be a viable configuration. If the filesystem’s only purpose was to stream video to clients and the server was in a high availability cluster, the high risk of array failure could be worth the 175.7mb/s the array offers. If the array did then fail, other machines running the arrays could then take over while the failed array is repaired.

Alternatively, if redundancy was required, multiple vdev arrays also perform relatively well with a 2 vdev RAIDZ2, a 3 vdev striped mirror and a 3 vdev RAIDZ1 array, all achieving around 147mb/s. These arrays only provide a maximum of 50% storage disk availability however, so if more disks are needed for storage, single vdev arrays are only slightly less performant. For example, a single vdev 6 disk RAIDZ2 array appears to offer a good balance, allowing 4 disks of storage availability and 2 disks of redundancy while still achieving speeds of 130.66mb/s. Alternatively, most single vdev 6 disk arrays perform at a similar level, so configurations can be chosen based on the need for storage space or array resiliency.

Finally, at lower disk numbers, the various configurations also offer similar levels of performance, with a 3 disk RAIDZ1 array achieving 125.12mb/s against the 118.22mb/s of a 3 disk mirror array which offers a higher level of array resiliency. Similarly, a 5 disk RAIDZ3 offers 127.94mb/s compared to a 5 disk RAIDZ1’s 130.32mb/s, with the latter offering a much higher level of storage disk availability. The only outlier to this is a 4 disk RAIDZ3 array, where 3 disks worth of space is dedicated to parity information and so the array can only perform at the speed of one drive.

#### MongoDB Server:





Like the previous workload, the MongoDB workload heavily utilises one operation type, in this case random operations. As such, the continuous measurements do not offer enough detail to draw any meaningful conclusions and so analysis will focus on the random performance.

Once again, a 6 disk stripe is the most performant at 223.62 ops/s and only recommended if the stored data is easily recoverable, something that may not be the case for a database. The second most performant array, a 2 vdev RAIDZ2, achieves a competitive 213.485 ops/s, however, once again, offers a poor balance of disk failure resiliency and storage availability. A 3 vdev striped mirror array improves this balance by reducing the number of parity drives to 3 and still offering good performance at 203.943 ops/s, with the caveat that only 1 drive per vdev can fail. Alternative configurations containing multiple vdevs also score well, offering different balances of disk failure resiliency and storage space.

Surprisingly, single vdev mirror arrays offer strong performance across all disk numbers, outperforming many configurations that allow for greater than single disk write speeds. While the Mongo workload does contain writes, they are only 16kB appends, whereas the reads are 1MB, so the ability for the arrays to split the reads across all disks must make up for the comparatively slow write speeds. These high performance numbers do of course mean there can only ever be one drives worth of storage space, something that may be impractical at higher disk numbers.

Predictably, most single vdev RAIDZ arrays do not perform well as the operations cannot complete until every drive in the array has retrieved its sector. Because of this, they are not well suited to the random operation heavy workload of a database.

The exception to this is minimum disk size RAIDZ arrays, with a 3 disk RAIDZ2 achieving 196.906 ops/s, nearly as many as a 6 disk 3 vdev RAIDZ array at 198.785 ops/s. This is also seen to a lesser extent with a 2 disk RAIDZ1 array at 174.266 ops/s and a 4 disk RAIDZ3 at 169.191 ops/s, all achieving higher numbers than the same configurations with more disks. This could possibly be because arrays with a single data disk do not have to stripe the data sectors across multiple disks. Therefore, when reading and writing these sectors, the single drive can improve efficiency by grouping all operations together. Additionally, once the drive has finished its operations, it does not have to wait for any other drive. When a second data drive is introduced, the data sectors must then be striped across two disks. This could reduce the number of operations each drive can group together to improve efficiency and would cause one drive to have to wait for another to finish. Therefore, the reason that a 2 vdev RAIDZ2 array performs so well is because it has two vdevs to split block operations between and each vdev only has one drive retrieving sectors. However, as previously mentioned, this results in only two drives being used for data and therefore a 3 vdev RAIDZ1 or mirror array may be more appropriate.

### Flexible I/O Tester:

#### Random Read:

##### General use:

A graph of data with numbers and letters

Description automatically generated with medium confidence

Moving to the second piece of benchmarking software, the first scenario tests a configuration’s random read speeds. Random speeds are arguably the most important metric as, while sequential speeds are important to be aware of, these speeds can be misleading. The general use scenario is designed to emulate an array that stores a wide range of file types and sizes, being utilised by multiple users or processes. Because of this, the array is unlikely to be dealing with the large contiguous files that allows high read speeds to be achieved. Instead, the array is likely to be dealing with a large amount of smaller, non-contiguous files, such as emails, text files or slideshows, resulting in slower speeds.

The results immediately start with an unusual result as a 6 disk stripe did not score highest, reporting only 59.2MiB/s. Instead, it was outperformed by a 6 disk array with 2 RAIDZ2 vdevs, which achieved 76.633MiB/s.

This could partially be due to the same, previously explained reason, where a single data disk in a vdev performs better as all sector retrievals take place on the same disk. This theory is also supported by the fact that both single vdev RAIDZ1 and RAIDZ2 arrays see performance decrease as more disks are added. Additionally, the extra vdev allows multiple blocks to be operated on asynchronously. However, if this were the only reason, then a 3 vdev mirror or RAIDZ1 array would perform even better.

Could have something to do with more parity disks?

Another interesting anomaly is that the RAIDZ3 arrays do not follow the expected pattern. As this workload tests random reads, the expected behaviour would be for the performance to decrease however this is not reflected in the results. The minimum size 4 disk array reports the lowest performance at 11.183MiB/s, the 5 disk array increases 27.8MiB/s and the 6 disk array decreases again to 19.316MiB/s.

Possible that all the RAIDZ configs have different performance patterns and it isn’t as simple as “theyre all the same but just have different parity numbers”? Perhaps more parity drives helps somehow? May have to say more testing required.

Other configurations follow a more predictable pattern, with the 6 disk striped array performing the second best. The two RAIDZ1 and mirror arrays with 3 vdevs predictably also perform very well and similarly, with the mirror array performing slightly better at 52.883MiB/s against 50.933MiB/s, possibly due to the slight RAIDZ overhead. Performance of the single vdev mirror arrays also improves in a predictably linear manner as the number of disks increases, with the read operation being split across more disks.

Single vdev RAIDZ arrays also perform predictably, with performance decreasing as the number of disks is added. This is likely caused by the blocks being split into a smaller number of sectors for each disk to store. Thus, when the sectors are being retrieved, the disks are performing a smaller number of sectors, leading to less efficiency. The file system must then wait for all disks to finish their inefficient sector retrievals before the operation can be complete. In comparison, when there are a smaller number of disks, each disk has a higher number of sectors and so the retrieval of them is a more efficient operation.

##### Best case:

A graph of a number of text

Description automatically generated with medium confidence

The best case dataset doesn’t necessarily fit well with the random read tests as it contains large files and performs large operations. This isn’t a scenario that wouldn’t often appear in real world usage as random reads are usually smaller operation sizes and larger operation sizes tend to be sequential.

In the uncommon event that there were large random reads, the 6 disk, 2 vdev RAIDZ2 array reports the best results at 67.583MiB/s, closely followed by the 5 and 6 disk stripes at roughly 64MiB/s. Once again, the 2 vdev RAIDZ2 array is likely outperforming the stripe array due to the aforementioned reasons but does not provide a large amount of storage capacity.

Configurations otherwise perform very similar, averaging around 58MiB/s, and so the best array for this workload can be chosen based on the preferred storage and parity disk balance. Some notable exceptions to this are the 2 disk RAID1 array at 53.7MiB/s compared to the 2 disk mirror at 56.616MiB/s. While these are both under the average result, only 2 disks are available which limits performance and so the mirror array is the best performing array with redundancy at 2 disks. Another notably poor performer is the 6 disk RAIDZ3 array once again, achieving only 55.466MiB/s. This result is lower than the average performance and worse than a single vdev 6 disk RAIDZ2 array, which achieves 60.566MiB/s and offers more storage capacity.

##### Worst Case:

A graph of a number of text

Description automatically generated with medium confidence

For the final random read test, the worst case workload produces slightly more predictable results. The highest performing arrays are the stripe arrays, with multiple vdev arrays then being the next best performers. Like in the previous benchmarks, the 2 vdev RAIDZ2 array does perform notably well, achieving 0.479MiB/s. This time however, it performs in a more expected manner, achieving roughly the same performance as other two vdev arrays. Indeed, it is outperformed by the 3 vdev striped mirror array which achieved 0.482MiB/s as it does not have any of the random performance penalties of RAIDZ arrays. The RAIDZ2 array does, however, still outperform a 3 vdev RAIDZ1 array, further suggesting possible behaviour differences between the levels of RAIDZ.

Single vdev RAIDZ2 performance is similarly anomalous, with single vdev arrays achieving 0.36MiB/s at 4 disks, decreasing to 0.341MiB/s at 5 disks and increasing again to 0.361MiB/s at 6 disks.

RAIDZ3 arrays also perform unexpectedly, with a 4 disk array achieving 0.391MiB/s and the result not decreasing when an extra data disk is added at 5 disks, achieving 0.394MiB/s. Performance then does decrease at 6 disks, dropping to 0.344MiB/s. A pattern does appear to emerge with random reads operations in RAIDZ3 configurations where a 5 disk configuration will consistently outperform all other disk numbers. In the worst and best case scenarios, a 4 disk configuration performs second best, while in the general use workload, a 6 disk performs significantly better. This unexpected behaviour further evidences the idea that RAIDZ1, RAIDZ2 and RAIDZ3 have differences beyond the number of assigned parity disks.

~~It is possible that the smaller 4 KiB operation sizes lead to better performance as each operation is the size of a single sector, therefore fitting on one data and parity disk and not needing to be striped. This would subsequently allow other disks to perform other asynchronous operations, however if this were the case, it should also be reflected in the RAIDZ1 results which it is not. Indeed, for single vdev RAIDZ1 results, performance decreases as more disks are added. Normally, this would be expected as, for every disk added, the file system must stripe the block across an increasing number of disks and a smaller number of sectors. Thus when the block is read, each disk must retrieve the appropriate sectors and the file system must wait~~

~~In this scenario, as operation sizes are only 4 KiB, the expected outcome would be for RAIDZ arrays to not necessarily lose performance as disk numbers increase. As sector sizes are also 4 KiB, an operation could theoretically be performed on a single data disk~~

#### Random Write:

##### General use:

A graph of numbers and letters

Description automatically generated

Random write performance numbers in the general use scenario follow a relatively predictable pattern. Stripe arrays outperform all other configurations with the same disk numbers but offer no disk failure protection. Mirror arrays see very little performance improvement as more disks are added due to every piece of data needing to be written to every disk. Arrays with multiple vdevs achieve the best performance numbers for arrays with redundancy, as each vdev can handle a separate block operation.

RAIDZ1 and RAIDZ2 random write speeds increase as more disks are added, contrary to the expected behaviour of maintaining the same random performance (iXsystems, 2020, p. 6-7). RAIDZ3 also follows the same pattern seen in random writes, with the 5 disk array achieving 37.116MiB/s, the 6 disk array achieving 34.583MiB/s and the 4 disk array achieving 24.5MiB/s.

##### Best case:

A graph of a number of writing

Description automatically generated with medium confidence

In the best case scenario, stripe arrays once again achieve the best results, however RAIDZ1 arrays appear to outperform all other arrays. The highest performing array is 6 disk array with only 2 vdevs, achieving 195MiB/s. This is followed by the 6 disk 2 vdev uneven array, where one vdev contains 4 disks and the other contains 2, which achieved 190.666MiB/s. The 3 vdev array then comes in third, only achieving 188MiB/s. A 6 disk 2 vdev RAIDZ1 array performing the best is extremely unexpected and has not been seen in any other results. A 6 disk 2 vdev RAIDZ2 array has previously outperformed 3 vdev arrays, however this has been an occurring pattern. Random operations are usually small and so a workload of random 1MiB write operations would be unusual. It is possible that this larger than normal operation size is causing arrays to perform closer to their sequential speeds, thus leading to increased and unexpected performance.

Aside from the unusual performance of the RAIDZ1 arrays, mirror arrays perform decrease in performance as disk numbers increase, presumably due to the increased wait for all disks to write the large 1MiB operations. Performance then also increases in a linear manner as more mirror vdevs are added, although decreasing when more disks are in each vdev.

Single vdev RAIDZ, RAIDZ2 and RAIDZ3 arrays all increase in performance as more disks are added, in opposition to what would be expected, for it to remain the same or decrease. This could, however, also be explained by the unusually high random write operation size causing them to perform closer to their sequential performance.

##### Worst Case:

A graph of random writing

Description automatically generated

The size of the random writes in a worst case scenario are much closer to a real world scenario, only being 4KiB in size. For this reason, results follow a much more predictable pattern, with stripe arrays performing best. A 3 vdev striped mirror array follows at 1.553MiB/s, beating a 3 vdev RAIDZ1 array at 1.475MiB/s with its lack of random performance overhead.

Mirror arrays slightly increase in performance as more disks are added, a surprising result as all sectors would still need to be written to all disks. Write speeds also increase in a linear manner as disks are separated into more vdevs meaning either two or three block operations can be performed asynchronously.

RAIDZ1 and RAIDZ2 arrays also display this pattern, with write speeds increasing as more disks are added to the array. For these configurations however, this is expected as the writes are only 4KiB, the same size as a disk sector. As the operation sizes are so small, each disk is able to handle its own operation and so multiple asynchronous operations can occur. For example, in a single vdev 6 disk RAIDZ2 array, there are 4 data disks, each able to handle its own 4KiB write operation. There will be some added overhead as a 4KiB parity sector must also be written and there are only 2 parity disks, however this will still lead to greater performance than if an operation spanned multiple disks. Because of this, for random writes of 4KiB or below, single vdev RAIDZ arrays do make a reasonable option, with random performance that is not significantly worse and a good storage space and redundancy balance. The exception to this is the RAIDZ3 arrays which have reverted back to the previously seen patterns. The 5 disk array was the best performing, this time better than most other 5 disk configurations, while the 4 and 6 disk arrays perform notably worse.

#### Random Read/Write:

##### General use:

A graph of numbers and text

Description automatically generated with medium confidence

A mix of random read and write operations

##### Best case:

A graph of numbers and text

Description automatically generated with medium confidence

sdf

##### Worst Case:

A graph of numbers and text

Description automatically generated with medium confidence

gj

#### Sequential Read:

##### General use:

A graph with numbers and text

Description automatically generated

Continuing to the sequential read results for the general use case, this scenario would be unusual as the smaller average file size would be less likely to be read from continuously. This may, however, appear in a workload such as a game drive, where lots of medium sized textures, all in the same location, may be loaded from storage.

As expected, the best performing array is a stripe with 6 disks reaching 479.666MiB/s, with performance predictably decreasing by the speed of each drive, roughly 150MiB/s, as the number of disks in the array shrinks.

A configuration with more data security would be a 2 vdev RAIDZ2 array, with the second highest performance of 436.833MiB/s. This would, however, lead to only 2 of the 6 disks being used as storage disks. Additionally, while 4 disks are able to fail, they must be from separate vdevs. If 3 drives were to fail from the same vdev, the array would be lost.

Therefore, using 6 disks in a single RAIDZ2 vdev may be a more practical option, increasing the number of storage drives to 4. This does, however, result in a small downgrade in performance to 417.666MiB/s.

RAIDZ2 continues to remain a reasonable option at lower disk numbers, with the caveat that, as the number of disks decreases, the percentage of available storage space decrease significantly. For instance, a 3 RAIDZ2 array does give better performance than a RAIDZ1, but only allows 33% of the potential space allocated to storage compared to the 66% of a RAIDZ1. Interestingly, a 4 disk RAIDZ2 appears to perform notably worse than a 3 disk RAIDZ2 array, achieving 196MiB/s and 225.666MiB/s respectively.

Unsurprisingly, a 4 disk RAIDZ3 array achieves the worst result of all at 81.066MiB/s as only a single drive is being used for storage. However, once again, a smaller array outperforms a larger one, with a 6 disk RAIDZ3 array achieving 267.5MiB/s, compared to a 5 disk array achieving 358MiB/s.

Unclear why this is happening

Multiple RAIDZ1 vdevs do not show particularly strong performance numbers, with the best performer being 2 devs, one with 2 disks and one with 4 disks, at 547MiB/s. This, however, does not match the performance results of a 6 disk RAIDZ2 and has a much higher risk of data loss, with only one drive being allowed to fail in the second vdev before data loss occurs.

Similarly, multiple mirror vdevs also don’t appear to be a strong contender with all configurations showing worse performance than a 6 disk RAIDZ2 with a lower amount of available storage. At lower numbers of drives, such as 2, a mirror vdev does appear to be the better option, however at 3 drives, a RAIDZ1 configuration may be more appropriate, depending on whether performance or disk failure resiliency are more desirable.

Finally, single vdev mirror arrays do show strong performance numbers, however, this result in poor storage availability, even if the data is very resilient to drive loss.

##### Best case:

A graph with numbers and letters

Description automatically generated

Sequential reads speeds in a best-case scenario dataset would likely reflect those seen in a video streaming server, where large files, continuous video files are being read in a steady and predictable way. The conclusions drawn from this scenario don’t differ too much from the previous scenario, where the best performing array is a 6 disk stripe at 700.666MiB/s. The second-best performing array this time is a 6 disk mirror at 570.5MiB/s, but once again, equates to a very low disk usage percentage.

Therefore, a single 6 disk RAIDZ2 array also appears to be the best option, providing the best third performance at 553.833MiB/s and a good balance of disk failure resiliency and available storage.

Other arrays appear to show the same degree of viability as seen in the previous scenario with a few exceptions. A 6 disk RAIDZ3 does not appear to show a particularly large increase in performance compared to a 5 disk RAIDZ3, at 404.333MiB/s from 389.5MiB/s. Worse than this is a 6 disk RAIDZ1, reporting worse read speeds than a 5 disk RAIDZ1 vdev at 487.666MiB/s and 533.5MiB/s respectively.

Unlike last time, where smaller, non-contiguous files were being read, a 6 disk 2 vdev RAIDZ array appears to perform notably better than a 6 disk 3 vdev RAIDZ array, achieving 547.166MiB/s compared to 463.5MiB/s.

Unclear why this is happening

##### Worst Case:

A screen shot of a graph

Description automatically generated

For the final set of read results, most configurations seem to perform very similarly, with the delta only being a few MiB/s. An interesting pattern does appear to emerge however, with an increased number of disks appearing to result in a decrease in performance. The most pronounced examples of this are the RAIDZ1 arrays, with the most performant of all configurations being a 3 disk RAIDZ2 at 188.666miB/s, but a 6 disk RAIDZ2 dropping to one of the worst performing at 158.5MiB/s. This is also seen in other configurations such as a 6 disk stripe, the most performant configuration in the previous tests, achieving an unremarkable 180.833MiB/s compared to a 3 disk stripe achieving 187.333MiB/s.

Unclear why this is happening

#### Sequential Write:

##### General use:

A graph of data with numbers and letters

Description automatically generated with medium confidence

Continuing to the sequential write operations, a 6 disk stripe continues to be the most performant at 169.666MiB/s, scaling in an expected linear manner as the number of disks decreases. This time however, 2 RAIDZ1 vdevs, a single RAIDZ1 vdev, and 3 mirror vdevs all appear to perform very well at 133.666MiB/s, 127.333MiB/s and 125.833MiB/s respectively. The previously well performing single vdev 6 disk RAIDZ2 array this time only produces a result of 119MiB/s, lagging behind a large number of other 6 disk configurations.

Other configurations otherwise perform expectedly, with single vdev mirrors predictably not seeing any performance improvement with an increase in disks. Unlike read operations where different blocks can be read from different disks, all blocks must be written to all disks and so write speeds never increase beyond that of a single disk, making single vdev arrays unsuitable to write heavy environments. Performance does improve once multiple vdevs are introduced, as can be seen in the results for 2 vdev and 3 vdev striped mirror arrays, but come with the usual downsides of never better than a 50% storage disk usage and only 1-2 drive failures of resiliency per vdev.

RAIDZ3 continues to report relatively competitive speeds but continues to offer a low amount of storage space for the number of disks despite offering excellent disk failure resiliency.

Notably, the performance increase for all RAIDZ configurations begins tapering off as more disks are introduced, perhaps suggesting that the performance benefits of splitting the file into smaller pieces and writing them across the array stripe begin to taper off.

##### Best case:

A graph of numbers and letters

Description automatically generated

The figures seen in the best case sequential write scenario would likely be reflected in an environment such as a video ingest station, where large contiguous files are being written to the array.

In this scenario, aside from a 6 disk stripe, a RAIDZ array with 1 or 2 vdevs appears to perform the best, achieving 209MiB/s and 205.666MiB/s respectively. A single RAIDZ vdev does not lag too far behind, reporting 197.166MiB/s and predictably, as the number of parity disks increases in RAIDZ2 and RAIDZ3 arrays, the overhead to calculate the parity information increases and they begin to perform worse at 183.833MiB/s and 165.666MiB/s.

Striped mirror arrays perform reasonably well, but not as well as any of the RAIDZ arrays and only offer up to 50% of usable storage from the raw storage. Interestingly, in single mirror vdevs, the write speeds decrease with an increase in drive numbers, presumably due to the increasing overhead of needing to make sure all drives data exactly matches.

If large sequential writes were the main workload, both an array with 6 disks arranged in 2 RAIDZ vdevs or a single RAIDZ2 vdev are strong performers with the RAIDZ2 array offering slightly more redundancy with its 2 drive failure tolerance compared to 1 drive per vdev.

##### Worst Case:

A graph of data with numbers and text

Description automatically generated with medium confidence

Similar to the worst case read speeds, the worst case write speeds mostly appear to perform similarly, with a 6 disk array of 2 RAIDZ vdevs performing the best at 80.466MiB/s, closely followed by a 6 disk array of 3 RAIDZ vdevs and a 6 disk array of 3 mirror vdevs at 79.633MiB/s and 78.5MiB/s.

Single vdev RAIDZ array also appear to perform reasonably well, with a 6 disk RAIDZ achieving 77.833MiB/s, a RAIDZ2 achieving 77.416MiB/s and a RAIDZ3 achieving 73.95MiB/s. Both performance results and performance gains from added disks predictably decrease as the parity calculation overhead increases. This is also reflected the single vdev mirror arrays.

With fewer numbers of disks, a RAIDZ array still appears to outperform a mirror vdev with a 2 disk RAIDZ array achieving 66.45MiB/s compared to a 2 disk mirror array achieving 63.633MiB/s. This result is repeated as the number of disks increases, with a RAIDZ array repeatedly outperforming any variation of mirrored vdevs, suggesting a single RAIDZ array or multiple RAIDZ vdevs may be the best option at any disk number for this kind of workload.

#### Sequential Read/Write:

##### General use:

A graph of numbers and text

Description automatically generated with medium confidence

Moving to the second piece of benchmarking software, the first scenario tests a configuration’s sequential read/write speeds. This demonstrates how an array would handle performing a mix of sequential read and write operations, making both the read and write operations slower and less efficient.

In the general use dataset, aside from the striped arrays, configurations that make use of multiple vdevs show the best results, with a 6 disk 3 vdev mirror performing the best at read speeds of 84.05MiB/s and write speeds of 84.9MiB/s. It is then closely followed by a 6 disk array with 3 RAIDZ vdevs at read speeds of 84.083MiB/s and write speeds 84.766MiB/s. Presumably, the added overhead of calculating parity information rather than simply writing the same data twice allowed the striped mirror array to perform slightly better than the RAIDZ vdev.

Interestingly, despite single vdev mirrors consistently performing worse the more disks were added in sequential write tests, this is not reflected in this test. A 2 disk mirror reports a read speed of 33.466Mib/s and a write speed of 33.666MiB/s, whereas a 6 disk mirror reports a read speed of 45.616MiB/s and a write speed of 25.833. This could suggest that the performance penalty when adding more disks to a mirror vdev may only occur during extended sequential writes and mixed usage workloads benefit from more drives.

Other configurations perform unsurprisingly, with performance scaling with the number of disks. The exception to this, once again, is a 6 disk RAIDZ3 array, which seems a dip to read speeds of 56.316MiB/s and write speeds of 56.616MiB/s from reads of 64.35MiB/s and writes of 64.833MiB/s in a 5 disk array. Again, this does not seem to be caused by any outlying results and could be caused by a unfavourable file sizes being split across the array.

Across the range of disk numbers, arrays containing either multiple mirror or RAIDZ vdevs appear to be the most logical choices, with the choice depending on whether performance, available storage space or disk failure resiliency is least important to the situation.

##### Best case:

A graph of numbers and text

Description automatically generated with medium confidence

On the best case scenario data-set, read/write operations may be seen in environments such as video editing servers, where users would be intermittently reading large sequential video files and writing edits and cuts to them.

In this scenario, an array with multiple vdevs may once again prove to be the wisest option, with RAIDZ vdevs in particular outperforming all others. Interestingly, a 6 disk array with 2 RAIDZ vdevs outperforms a 6 disk array with 3 mirror vdevs at reads and writes of 149.666MiB/s compared to reads and writes of 142.333MiB/s. This could be due to the RAIDZ array having a greater number of storage disks to utilise compared to the the mirror array, allowing it to pull ahead.

The observed pattern of single vdev mirror arrays increasing in performance as more disks are added is not seen in this dataset. Here, configurations appear to perform roughly the across all disk numbers with some variation. Results dip at 3 disks, spike at 4 disks then drop again at 5 and 6. It is difficult to tell whether this is a genuine pattern or margin of error, therefore more tests would need to be conducted.

Even in arrays with lower disk numbers, RAIDZ configurations outperform all others, making single or multi vdev arrays an optimal choice across all disk numbers. This decision would however, have to be made with the knowledge that a vdev can only sustain a single drive loss before data loss occurs.

##### Worst Case:

A graph of numbers and text

Description automatically generated with medium confidence

In the worst case scenario, multi vdev arrays once again prove to be the strongest performers with 6 disk 3 vdev RAIDZ and mirror arrays performing near identically. RAIDZ achieves reads of 50.866MiB/s and writes speeds of 50.783MiB/s compared to mirror’s reads of 50.383MiB/s and writes of 50.3MiB/s. 2 RAIDZ vdevs also trail extremely closely, achieving reads of 50.366MiB/s and writes 50.283MiB/s, with the added bonus of allowing an extra disk to be used for storage and still maintaining the 1 disk per vdev failure tolerance.

A single vdev RAIDZ2 array also does not lag behind the others significantly, with results across the range of disk numbers only slightly lagging behind the results of single vdev RAIDZ arrays, with the slight trade-off in performance perhaps being worth the increase in disk failure tolerance.

Results otherwise remain as expected with striped arrays offering the best performance with the high risk of data loss and single vdev mirror arrays seeing slight performance drops as more disks are added to the array. RAIDZ3 arrays also continue perform reasonably and offer excellent data safety but still offers poor amounts of storage availability for the number of disks being tested. This arguably leaves RAIDZ arrays as the best option for heavy read/write based scenarios. The choice between a single or multiple vdev would depend on whether performance, volume of storage or disk failure resiliency were more considered more important.

### Filesystem Tuning and Optimisation:

While the correct balance of performance, storage space and resiliency can be difficult to achieve, ZFS does offer a variety of features to counteract some potential disbenefits a configuration poses.

A graph of data on a white background

Description automatically generated

The above graph shows the results of the same sequential read, best case fio benchmark that had previously been conducted, this time with modifications made to increase performance.

Firstly, the recordsize, or block size, has been changed from the default size of 128KiB to 1MiB. For the best-case scenario test which contains a large 16GiB test file and 1MiB operation sizes, this significantly increases performance to 626MiB/s, as the filesystem does not have to retrieve multiple blocks to finish the operation.

Once the recordsize has been tuned appropriately, performance can be greatly enhanced by increasing the amount of RAM. This system memory can be used to create an Adaptive Replacement Cache, or ARC, which can then be used to cache commonly read files. Rather than repeatedly querying slow hard drives, these files can be read from the extremely fast ARC, allowing read speeds to be comparable to those of system memory. As can be seen in the results, the file system must first recognise the commonly accessed files and cache them to memory but after this, speeds can increase dramatically.

An additional level of ARC can be added, known as an L2ARC, typically in the form of an SSD. Once the main ARC is full, less commonly accessed files can be moved onto the L2ARC, still allowing for faster speeds than the slower HDDs, but not as fast as memory. Due to no other type of memory being as fast as system memory, it is almost always better to maximise the amount of L1ARC before adding a L2ARC.

More useful then, is the Separate Intent Log, or SLOG, which allows write operations to be recorded on a secondary device. Rather than waiting for the array to confirm the operation has been safely completed (also known as a synchronous write), it can be written to the SLOG and returned as completed before the array has finished writing. In this way, performance can be increased as the file system does not have to wait for the disks to finish writing the file before it moves onto the next one. There is also minimal risk of data loss as, if power is lost while the array is still catching up to the write operations, there will still be a non-volatile record of what it must complete. With these features, most performance deficits a particular configuration may have can be offset. Indeed, a poorly performing array can often exceed what even the best performing standalone arrays can offer.

To a lesser extent, some of the disk failure resiliency drawbacks posed by certain arrays can also be counterbalanced. As previously mentioned, when a disk does fail, the array needs to undergo resilvering and for some configurations, this process will take longer. While not tested in this paper, a general rule of thumb is that the more disks in a single RAIDZ vdev, the longer the resilvering will take. As any relevant parity information will not be shared outside the vdev, adding more vdevs will not affect other vdevs resilvering times. Additionally, as all data in a mirror vdev is identical, adding more disks does not affect resilvering times (Louwrentius, 2016).

For large vdevs with large amounts of disks and minimal failure tolerance, such as a 6 disk RAIDZ1 vdev, certain disks can be designated as hot spares. These drives spend the majority of their time sitting empty, only being used when a drive in a vdev fails, at which point it will immediately be used for the resilvering process. Once the resilvering has completed, the hot spare will be used by the array as the replacement disk until a new disk is inserted. After this, the contents of the hot spare will be copied to the new disk and the hot spare will once again sit empty. As there will be no downtime between the drive failing and a new drive becoming available for resilvering, hot spares can help reduce the risk of complete array failure. However, the process of resilvering is also one of the most stressful operations an array can undergo as all data on every disk in the vdev must be read to rebuild the data from the lost disk. As the remaining disks can be as old as the drive that failed, it is not uncommon for other disks to fail while this stressful operation occurs. For this reason, hot spares are not a perfect solution and so it is generally not recommended to have overly large vdev sizes or for large vdevs to have more capacity for disk failure.

# Conclusion:

An array that is ideal for every scenario does not exist, however for general use, a 6 disk, single vdev RAIDZ2 array offers a good compromise in all areas. It allows for 66% of the array to be used for storage while still allowing 2 drives to fail, reducing the risk of data loss before and during resilvering. Continuous read and write performance is also good if not outstanding in comparison to other arrays. Random performance is lacking and so an ARC and a SLOG should be used to offset this. IOPS can be improved if more vdevs were added and many vdevs containing 6 disks with 2 parity drives can be added without endangering the array. However, adding 6 drives at a time may not necessarily be an accessible option.

Alternatively, if random performance is critically important, for instance, if the array were being used to run an operating system, a 6 disk 3 vdev mirror array would be more appropriate. This array excels in both random and sequential operations which can be even further increased with the use of an ARC and slog. Additionally, this configuration can be easily expanded, only requiring two disks for create an additional vdev. However, this configuration does offer less storage space and a vdev can only sustain one disk failure before all data in the array is lost. For this reason, a hot spare should be available to remove any delay in resilvering. It may also be worth storing critical files on a separate, more resilient array.

When decreasing to 5 disks, once again, a RAIDZ2 offers good performance however this time only offers 3 disks of space. A 2 vdev mirror array also offers better random performance in exchange for reduced vdev resiliency. At this disk number, a RAIDZ1 may be considered for the extra storage capacity despite only having one disk of failure tolerance.

At 4 disks, single vdev RAIDZ2 arrays and 2 vdev mirror arrays still offer reasonable to good general performance, however the better storage capacity of a RAIDZ1 could be more appropriate. Decreasing to 3 disks, RAIDZ2 arrays should not be considered as the same resiliency is offered by a single vdev mirror array without the performance overhead. Additionally, there are no longer enough disks to create multiple vdevs so the higher performance made possible by this can no longer be achieved. For this reason, either a mirror or RAIDZ1 array should be used, dependant on whether the workload requires more resiliency and random performance or higher storage capacity. At 2 disks, a mirror array should always be used as it will offer the same resiliency and continuous performance of a RAIDZ1 array without the random performance penalties.

Other array configurations tested may be appropriate in certain scenarios, however do not generally have compelling benefits compared to other options. Some configurations, such as unevenly sized multi vdev arrays, largely perform worse than other equivalent sized options. Some, such as minimum size RAIDZ arrays, can perform better, as can be seen in the small random operations, but can be outperformed in other scenarios and so are only sometimes appropriate. Other arrays, such as a 6 disk, 2 vdev RAIDZ2 or a 5 disk RAIDZ3 array, can perform well, but offer very poor levels of storage capacity and so have niche use cases. At disk numbers above 6, these arrays may offer a better balance, however further research would be required. Additionally, the unexpected behaviours, such as a 5 disk RAIDZ3 array often outperforming a 4 and 6 disk array, could be studied to see whether the patterns continue. Finally, while stripe arrays often achieve the highest operation speeds, these should only be used if the stored data is not valuable or can easily be recovered.

# Bibliography:

6sense. (n.d.). *Oracle ZFS - Market Share, Competitor Insights in Storage Infrastructure*. [online] Available at: https://6sense.com/tech/storage-infrastructure/oracle-zfs-market-share [Accessed 18 Dec. 2023].

Bonwick, J., Ahrens, M., Henson, V., Maybee, M. and Shellenbaum, M. (2003). The Zettabyte File System. [online] Available at: https://www.cs.hmc.edu/~rhodes/cs134/readings/The%20Zettabyte%20File%20System.pdf [Accessed 18 Dec. 2023].

Endicott, S. (2022). *Windows 11, Windows 10 are now on more than 1.4 billion devices*. [online] Windows Central. Available at: https://www.windowscentral.com/more-14-billion-devices-now-run-windows-11-or-10 [Accessed 9 Jan. 2024].

Heger, D.A. (2009). Workload Dependent Performance Evaluation of the Btrfs and ZFS Filesystems. *Int. CMG Conference*.

iXsystems (2020). ZFS Storage Pool Layout. [online] iXsystems. Available at: https://static.ixsystems.co/uploads/2020/09/ZFS\_Storage\_Pool\_Layout\_White\_Paper\_2020\_WEB.pdf [Accessed 18 Dec. 2023].

Kennedy, P. (2017). *What is the ZFS ZIL SLOG and what makes a good one*. [online] ServeTheHome. Available at: https://www.servethehome.com/what-is-the-zfs-zil-slog-and-what-makes-a-good-one/ [Accessed 28 Jan. 2024].

Klara Inc. (2021a). *History of ZFS – Part 1: The Birth of ZFS*. [online] Available at: https://klarasystems.com/articles/history-of-zfs-part-1-the-birth-of-zfs/ [Accessed 28 Jan. 2024].

Klara Inc. (2021b). *History of ZFS – Part 2: Exploding in Popularity*. [online] Available at: https://klarasystems.com/articles/history-of-zfs-part-2-exploding-in-popularity/ [Accessed 28 Jan. 2024].

Louwrentius. (2016). *ZFS: resilver performance of various RAID schemas*. [online] Available at: https://louwrentius.com/zfs-resilver-performance-of-various-raid-schemas.html [Accessed 18 Jan. 2024].

Lundholm, A. (2015). A performance comparison of ZFS and Btrfs on Linux.

OpenZFS (2015). *The Birth of ZFS by Jeff Bonwick*. [online] YouTube. Available at: https://www.youtube.com/watch?v=dcV2PaMTAJ4 [Accessed 17 Feb. 2024].

Oracle. (n.d.). *Overview of ZFS Snapshots - Oracle Solaris Administration: ZFS File Systems*. [online] Available at: https://docs.oracle.com/cd/E23824\_01/html/821-1448/gbciq.html [Accessed 25 Jan. 2024].

Rose, J. (2014). *R2-C2*. [online] Jro.io. Available at: https://jro.io/r2c2/ [Accessed 21 Apr. 2024].

Salter, J. (2020a). *How fast are your disks? Find out the open source way, with fio*. [online] Ars Technica. Available at: https://arstechnica.com/gadgets/2020/02/how-fast-are-your-disks-find-out-the-open-source-way-with-fio/ [Accessed 15 Feb. 2024].

Salter, J. (2020b). *ZFS 101—Understanding ZFS storage and performance*. [online] Ars Technica. Available at: https://arstechnica.com/information-technology/2020/05/zfs-101-understanding-zfs-storage-and-performance/ [Accessed 19 Feb. 2024].

Seltzer, M.I., Ganger, G.R., M. Kirk McKusick, Smith, K.A., Craig and Stein, C.A. (2000). Journaling versus soft updates: asynchronous meta-data protection in file systems.

Stanik, J. (2007). A Conversation with Jeff Bonwick and Bill Moore. *Queue*, 5(6), pp.13–19. doi:https://doi.org/10.1145/1317394.1317400.

Tarasov, V., Zadok, E. and Shepler, S. (2016). Filebench: A Flexible Framework for File System Benchmarking. *login Usenix Mag.*, 41.

TrueNAS. (n.d.). *ZFS Deduplication*. [online] Available at: https://www.truenas.com/docs/references/zfsdeduplication/ [Accessed 28 Jan. 2024].

Veldkamp, L. (2023). Valuing Data as an Asset. *Review of Finance*, 27(5).

Widianto, E.D., Prasetijo, A.B. and Ghufroni, A. (2016). On the implementation of ZFS (Zettabyte File System) storage system. doi:https://doi.org/10.1109/icitacee.2016.7892481.

William Toigo, J. (2000). Avoiding a Data Crunch. *Scientific American*, 282(5), pp.58–74. doi:https://doi.org/10.1038/scientificamerican0500-58.

Workspace. (n.d.). *What is the true cost of lost data to business?* [online] Available at: https://www.workspace.co.uk/content-hub/business-insight/opinion-what-is-the-true-cost-of-lost-data-to-bus [Accessed 15 Feb. 2023].