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

Matthew Swain

20200873

Contents

[Abstract: 1](#_Toc173104198)

[Introduction: 2](#_Toc173104199)

[Literature Review: 2](#_Toc173104200)

[Understanding File Systems: 2](#_Toc173104201)

[The Need for ZFS: 2](#_Toc173104202)

[The Creation and Philosophy of ZFS: 3](#_Toc173104203)

[ZFS Configurations and Features: 6](#_Toc173104204)

[Stripe: 7](#_Toc173104205)

[Mirror: 7](#_Toc173104206)

[Striped Mirror: 7](#_Toc173104207)

[Raidz: 7](#_Toc173104208)

[Raidz2: 7](#_Toc173104209)

[Raidz3: 7](#_Toc173104210)

[Proposed Experiment: 10](#_Toc173104211)

[Methodology: 10](#_Toc173104212)

[Environment: 10](#_Toc173104213)

[Benchmark Software: 10](#_Toc173104214)

[Issues: 12](#_Toc173104215)

[Analysis: 13](#_Toc173104216)

[Preface: 13](#_Toc173104217)

[Filebench: 13](#_Toc173104218)

[Flexible I/O Tester: 16](#_Toc173104219)

[Filesystem Tuning and Optimisation: 25](#_Toc173104220)

[Conclusion: 26](#_Toc173104221)

# Abstract:

# Introduction:

Since its release to the public in 2006, ZFS, formerly known as the Zettabyte File System, has become a popular filesystem choice among system administrators. Oracle’s proprietary implementation is currently the 8th most used storage infrastructure (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 use, it may not be as performative or reliable as expected and so this paper aims to explore the different ZFS configurations and which scenario they may be most suited to.

# Literature Review:

## Understanding File Systems:

With Random Access Memory being volatile, devices such as hard disk drives (HDDs) or solid-state drives (SSDs) are needed to keep the data permanently stored after power has been turned off. However, when a user wants to save a file to long term storage, a program needs to 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). They are however, 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, a feat that a single drive cannot perform. For this reason, many disks are 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. Whenever a meta-data operation, an operation that modifies the structure of the file system such as creating, deleting or renaming a file or directory, was performed, the system would record the change it was making into a log (Seltzer et al., 2000, p3). This theoretically ensured that any file or directory would still be accessible in the event of a crash or power loss event while the operation was being performed, however Bonwick et al argued “consider the common case in which a bootloader reads the root file system in order to find the files it needs to boot the kernel. If log replay is needed in order to make the file system consistent enough to find those files, then all the recovery code must also be in the bootloader” (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 of the read and writes independent from the filesystem. The file system would then trust that any data read from the array would always be the same as the data written to the array, but this 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 (Bonwick et al., 2003, p. 5) leading to issues with bit rot. This is a phenomenon where data slowly becomes partially or completely corrupted and can be caused by drive deterioration/failure, bit flips, administrative errors or bugs in the disk 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). Not to mention if the RAID controller fails, the entire storage array is lost, making it a single point of failure and the weakest link in the chain. Data loss would be is extremely undesirable for a consumer, but in business where data is the most valuable asset (Veldkamp, 2023), it is particularly unacceptable as 93% of business that experience data loss for more than 10 days will file for bankruptcy within one year (www.workspace.co.uk, n.d.).

Finally, storage capacity was 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 completed 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. Instead of 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. As a consequence, 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, has 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 the mirrored disk which the user will likely have if data integrity is a priority.

Moving up a layer, the Data Management Unit handles the conversion between the blocks presented by the SPA and the files, otherwise known as objects, that live within the dataset, 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 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 limiting management complexity. 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. 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.

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 209, 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, however for random reads, ZFS achieved the highest at 8.1 MB/s, Btrfs in second with 6.1 MB/s and ext4 with 6.1 MB/s (Heger, 2009).

However, in a similar paper in 2016 written by Anders Lundholm, when using another RAID10 comparable configuration, this time using 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 for sequential reads the results were largely the same. Btrfs and XFS consistently achieved the highest speeds at just below 270 MB/s, ext4 followed with 215 MB/s and ZFS in last with 160 MB/s. The sequential read speeds are much different however, with ZFS having the highest at an average of 310 MB/s, Btrfs and XFS next at around 270 MB/s and ext4 last at around 240 MB/s (Lundholm, 2015).

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. It does however show that different file systems have their own strengths and weaknesses and that configuring an array appropriately for its workload is critical. For instance, Btrfs often performed ZFS in Lundholm’s benchmarks, such as the RAID5 random reads where ZFS configuration used is unable to keep up reading 4kB files from larger recordsizes. Conversely, Lundholm’s RAID10 sequential read tests the ZFS configuration is much more suited to, outperforming all other file systems.

## ZFS Configurations and Features:

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. In this way, both random writes, small operations that are not in the same location and therefore cannot be grouped together for greater speeds and efficiency, are considerably faster. Sequential writes, large operations that tend to be for the same file and can therefore be grouped together for much faster speeds and greater efficiency, are also increased. In the same way, random and sequential reads are also considerably increased as each disk only needs to retrieve their respective blocks.

The downside to this arrangement is that data stored in this way is extremely vulnerable to data loss through disk failure. If one disk in an array fails, the data it stores has not 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 every 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. 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, those blocks would be striped across the 3 vdevs and the 2 blocks that each

(iXsystems, 2020, p5-6)

These vdevs can then be put into a stripe array, allowing for the disk failure resiliency mirror arrays offer, along with the performance benefits of a stripe array.

### Raidz:

Beginning

### Raidz2:

Beginning

### Raidz3:

Beginning

ZFS does also have the ability to treat multiple drives as one device, called a virtual device or vdev. For instance, a user with 4 drives could make 2 vdevs each with 2 disks in a mirrored configuration and while they would only have the storage capacity of 2 drives, data could be striped across the two vdevs, getting the two times performance increase of a striped array with a 2 drive redundancy (although the two failed drives would have to come from separate vdevs, if 2 drives in the same vdev failed, all data would be lost)[[1]](#footnote-2).

This feature richness of ZFS was not there from the start however and needed to be built up over time

/

Discuss configuration types

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

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

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

Another trend is that the extra data integrity measures ZFS implements do not always result in lower performance and it could be argued that the data integrity, features and matureness of ZFS could outweigh the performance negatives as, while Btrfs has made great improvements in its performance, its RAID 5 and 6 equivalents are infamously unreliable and incomplete. It also shows that file systems are constantly evolving, with performance changing over time and new features being added that may sway the file system choice.

This quirk of copy-on-write is then used to add a useful feature to ZFS called snapshots. As no blocks are deleted, a user can take a snapshot of the current überblock which will point to all the blocks

Old work below

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

ZFS Storage Pool Layout, written by iXsystems, provides a higher-level explanation of the various pool layouts and uses example drive speeds to show how the layouts affect aspects such as read/write speeds and IOPS, however these speeds are only theoretical, therefore they don’t give a completely accurate picture of how each pool configuration would perform. While the paper does comment on how the pool configuration would affect fault tolerance, it does not comment on how likely the pool is to fail and how resilvering speeds would be affected. For instance, two of the example configurations given are a 6 2-way mirror and 2 6-wide RAIDZ2, and while it does mention that only 1 drive per vdev before the pool is lost for the mirror and 2 per vdev can be lost for the RAIDZ2, it does not mention that as time goes on, the probability of the mirror pool being lost is significantly higher than the RAIDZ2 pool being lost as can be seen below.[[2]](#footnote-3)

A graph of a function

Description automatically generated

The most common way ZFS has been benchmarked is in comparison to another filesystem, usually BTRFS, such as Dominque A. Heger[[3]](#footnote-4) who compared the sequential and random read/write speeds of ZFS, BTRFS and ext4 using 4KB files/blocks. He found that when using 4 vdevs of 2 mirrored disks, ZFS consistently performed better than BTRFS with sequential read/write speeds of 227.74MB/sec and 193.74MB/sec and random read/writes of 8.1MB/sec and 32.9MB/sec. By comparison, BTRFS using a similar RAID 10 configuration, achieved sequential read/write speeds of 162.52 MB/sec and 109.12MB/sec and random read/write speeds of 7.5MB/sec and 12.48MB/sec. While all tests were repeated 10 times and the mean results taken to ensure accuracy, it only tests one ZFS configuration, a pool of devs with 2 mirrored drives in each vdev. This is a configuration that specifically optimises speed above all else and the paper does not take into account factors such as drive failure tolerance and storage space efficiency which, in this configuration, are sacrificed for speed. The paper overall shows ZFS to be the higher performing filesystem, however the results may no longer be accurate as it was published in 2009 and a paper from 2015 by Anders Lundholm shows ZFS to be one of the slower filesystems.

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

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

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

NEED TO FOCUS ON PATTERNS RATHER THAN NUMBERS

Struggling to find anything that shows patterns

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

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

The paper also touches on resilvering time, testing how long the 2 disk mirror would take to resilver with 40GB, 80GB and 120GB of the capacity used, finding it took 10.7 minutes, 22.7 minutes and 36.7 minutes respectively. Lou Wrentius[[4]](#footnote-5) also looked at resilvering time in ZFS: Resilver Performance of Various RAID Schemas Not a scientific paper. , where he tested up to 11 disks in various mirror, RAID-Z, RAID-Z1 and RAID-Z2 configurations and found that his 2 disk mirror with around 256GB of data on the array, it took 37 minutes for it to resilver and with 512GB of data it took 75 minutes. Interestingly he also found that regardless of how many vdevs were added to the pool, resilvering times did not seem to increase dramatically with pools ranging from vdev sizes of 2-5 taking between 43-35 minutes. He also found that a RAID-Z configuration with 5 or fewer drives seemed to have comparable resilvering times to mirrored arrays, however beyond that times did increase substantially. Finally, he demonstrated that, while resilvering times for RAID-Z2 and RAID-Z3 did increase as more drives were added to the vdev, as more drives have been allocated for parity, the longer times may not be as much of an issue as there would still be extra drives in the event of a failure and so less risk of data loss. However, it does not state whether the tests have been repeated to verify results, nor does it give the entire range disks for the RAID arrays, making it more difficult to draw patterns. Finally, it would also have been more helpful to see how a RAID-Z2 array with 2 vdevs of 5 disks performed, as this may be a more commonly used configuration than 1 vdev with 10 disks.

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

For filebench stuff, followed directions in

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

Personally found that results did not stabilise enough after 3 mins

Now that all required information has been explained, onto methodology

# Proposed Experiment:

## Methodology:

### Environment:

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

The second workload, a videoserver, takes a different approach, containing two filesets of 2GB files. 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, the 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. Like the fileserver workload, this scenario contains a large amount of small, random read and write operations, however it does only have one instance compared to the fileservers 48 so its unclear how similar the two results are likely to be.

All workloads have been tuned to fit the creators’ recommendations[[5]](#footnote-6), as the default tests may not give useful or accurate results without editing. The default data-set size of all workloads is not large, with the web servers only being 16MB[[6]](#footnote-7), easily capable of being cached in memory and skewing the results. For this reason, the 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 it’s 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[[7]](#footnote-8).

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, letting them see how each engine’s method of file handling affects performance, however 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 data-set is recieving small 4KiB operations and the number of jobs and iodepth are 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 of 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 using more memory than was available, resulting in the Out of Memory Killer shutting down processes in order of least importance. The amount of memory dedicated to the VM was cautiously increased to 4GB, resolving the issue of out of memory scenarios, however this did bring the concern of the memory being tested, rather than the storage.

When benchmarking, it is critical to ensure that the size of the test file being used in the benchmark is larger than the amount of memory, as ZFS can utilise a feature named Adaptive Replacement Cache, or ARC. The filesystem will claim a certain amount of memory for its own use and use it to cache any writes or commonly requested reads, for instance, if a system has 8GB of memory and a 2GB file is written to the pool, rather than writing directly to the disks, ZFS will instead write into memory and then write to the slower disks, giving the user the impression of a quick transfer. Whilst this is extremely useful during normal operation, when trying to benchmark storage, it can lead to inaccurate results where the only metric being tested is the amount of and speed of the memory. For this reason, the maximum size of the ARC was set to 128MiB, half the smallest file size used in any of the tests, negating any risk files being stored in memory. To test this, the same 4GB benchmark was run in the VM with both 4GiB and 16GiB of memory, both achieving around 250MiB/s, with the throughput spiking to 6000MiB/s after the ARC size limit had been removed, clearly indicating that the file had been stored in memory. As a result, all 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 in order to manage it, however doing this would violate ZFS’s principle of interacting directly with the disk. Proxmox sitting between ZFS and the disks would be similar to a RAID controller sitting between the two layers, with ZFS not being able to guarantee that the data read is the same as what was written. Furthermore, Proxmox having access to the disks, may create scenarios where the benchmark results are impacted due to Proxmox performing taxing operations of its own, such as a scrub. Ideally, the HBA that the drives are attached to should be passed through as a PCIE device, ensuring Proxmox does not have any direct access to the disks. This has not been possible in this instance as the device installed in the Dell R520 is a PERC H310 RAID Controller. While this could be overcome by installing a new HBA, the current device was instead flashed to Initiate Target or IT mode, where the controller does not perform any RAID operations on the disks and acts as a HBA.

This unfortunately led to another issue where the boot drives and drives to pass through were connected to the PERC H310 in IT mode, thus when the HBA is passed through, Proxmox loses access to its own drive and the system crashes. This was remedied through Proxmox’s ability to pass individual devices through based on their ID, and so the drives can be used by ZFS without any obstruction to the bare disks. Whilst there may be some performance impact due to the

ZFS disks having to share bandwidth with the boot drives, it is unlikely to be a significant amount, the ZFS disks will not be used by any other services that would cause large drops in performance and all configurations share the same environment, so all results will contain the small potential bottleneck.

Need to mention having to manually set ashift

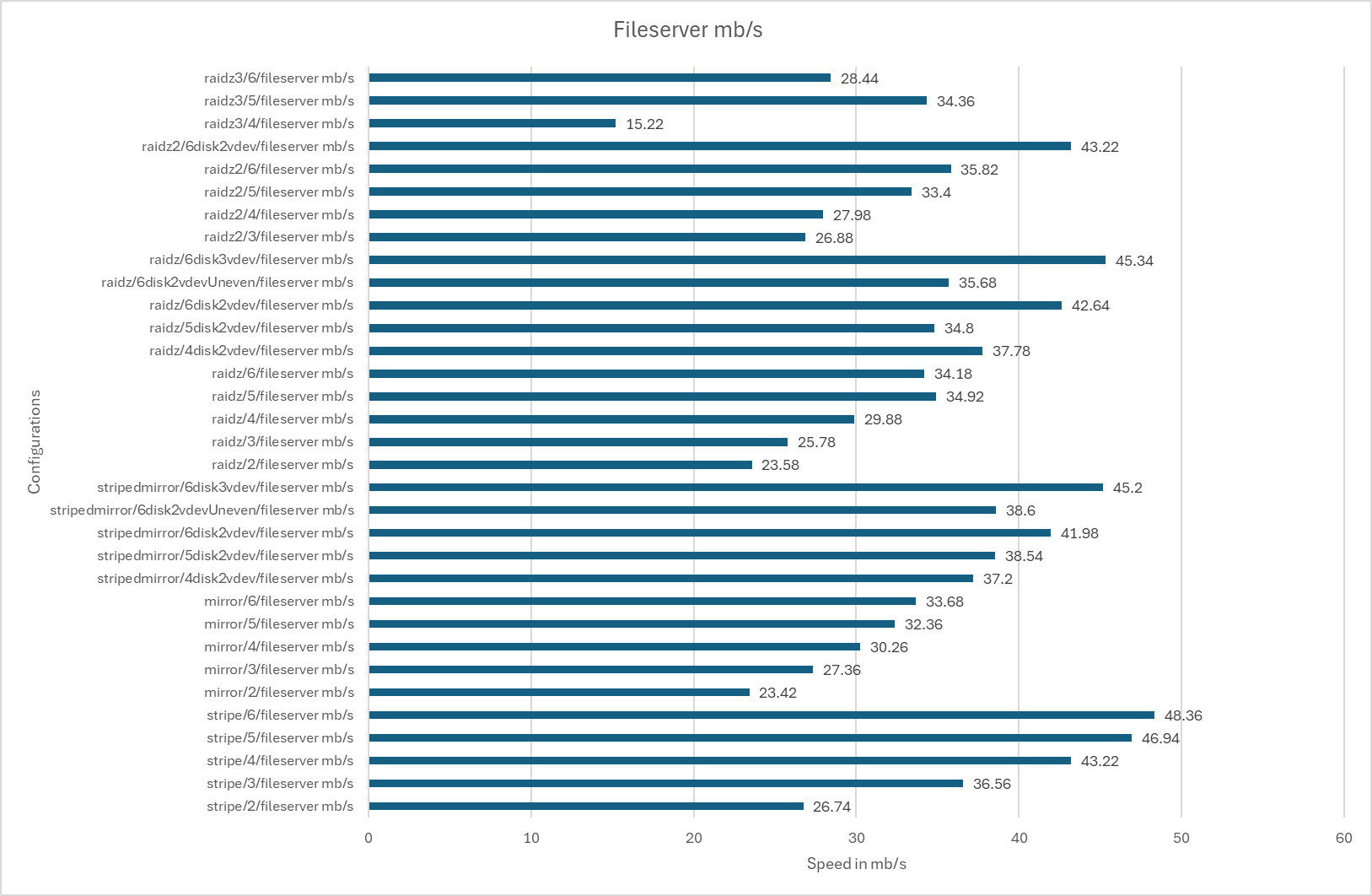
## Analysis:

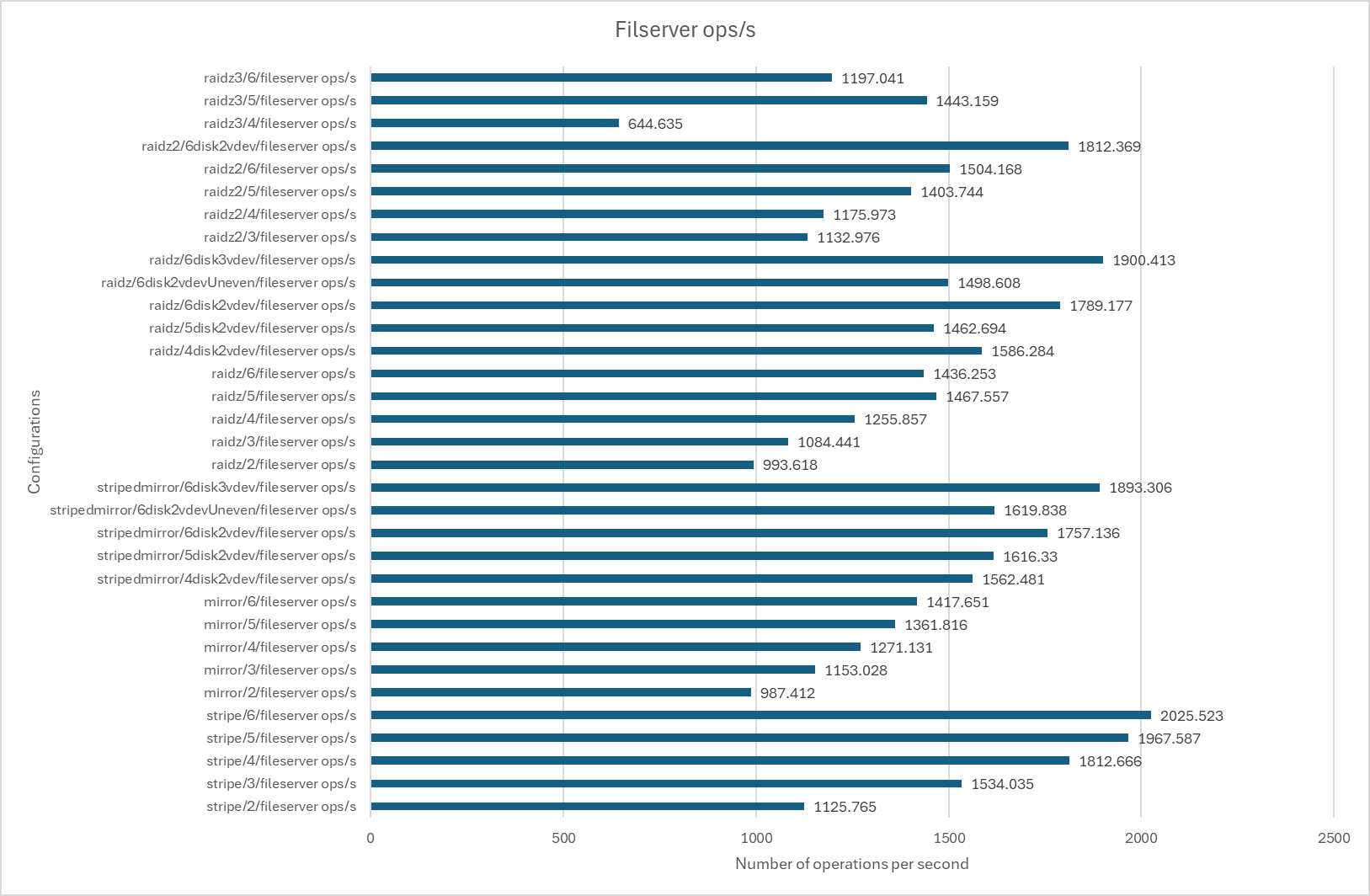
### Preface:

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, performance will be left on the table. Therefore, before any layout designs are decided, the type of work that the pool will be subjected to should be carefully studied and understood, making it clear what “performance” is and how it is measured in that scenario. Furthermore, the performance of the array is not solely dictated by the arrangement of disks, and factors such as available memory, caches and other pieces of fine tweaking can drastically affect performance. Disk configuration is still a good place to start however, as certain factors such as disk failure resiliency, ashift, available storage and resilvering times are less easy to make up for once the configuration has been decided. For 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 short of a second array to backup 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 raidz 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 raidz arrays performance of 45.34mb/s and 1900.413 ops/s. This large performance delta is because arrays with multiple vdevs can stripe the data across those vdevs, allowing some configurations to achieve speeds close to striped arrays, without the high risks of data loss.

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 are able to fail before data is lost, whereas if 2 drives in the same vdev fail in the striped mirror array, all data will be lost. As can be seen in the below figure from Reliable RAID Configuration Calculator[[8]](#footnote-9), 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.

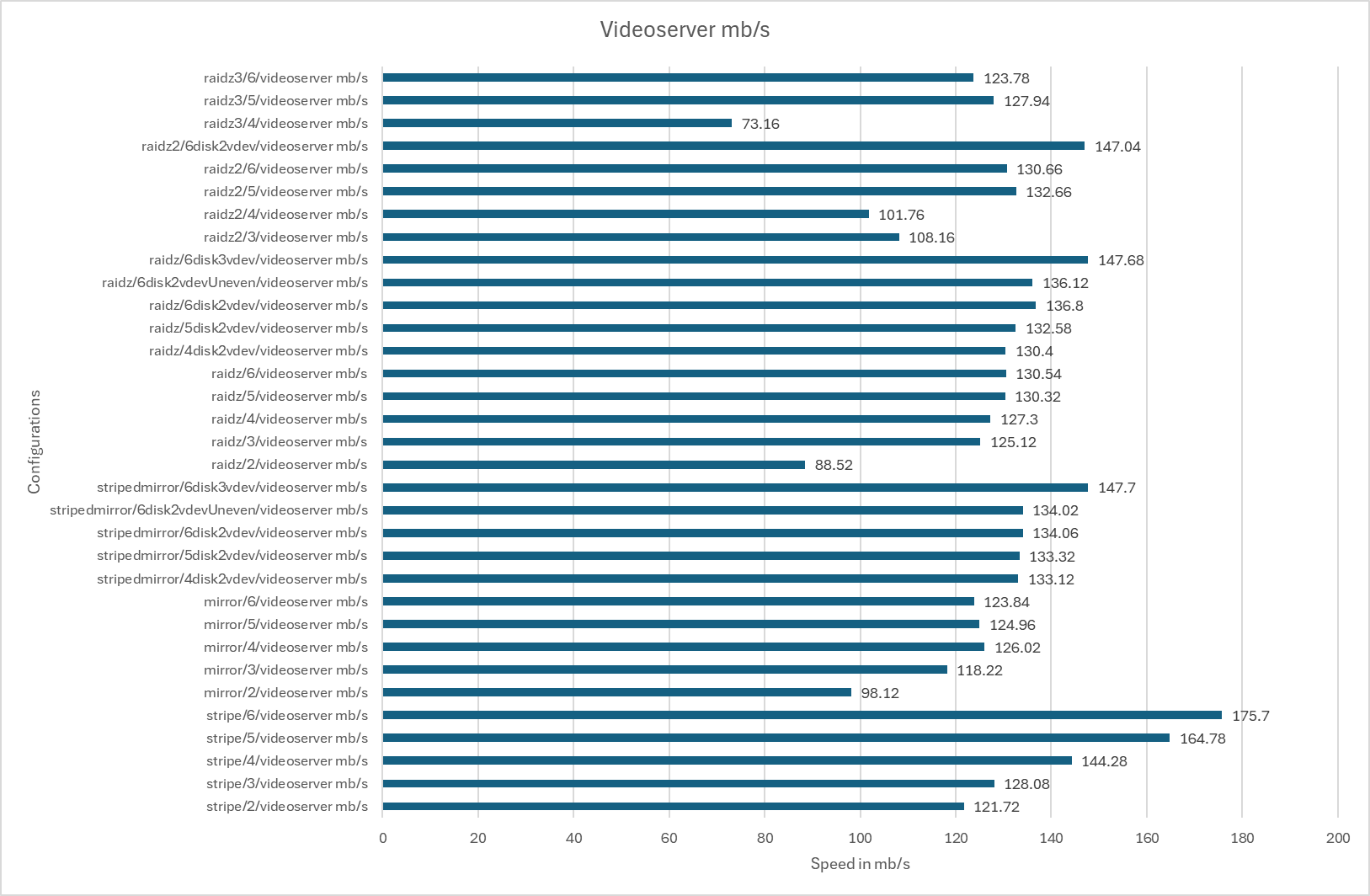
A graph with a line

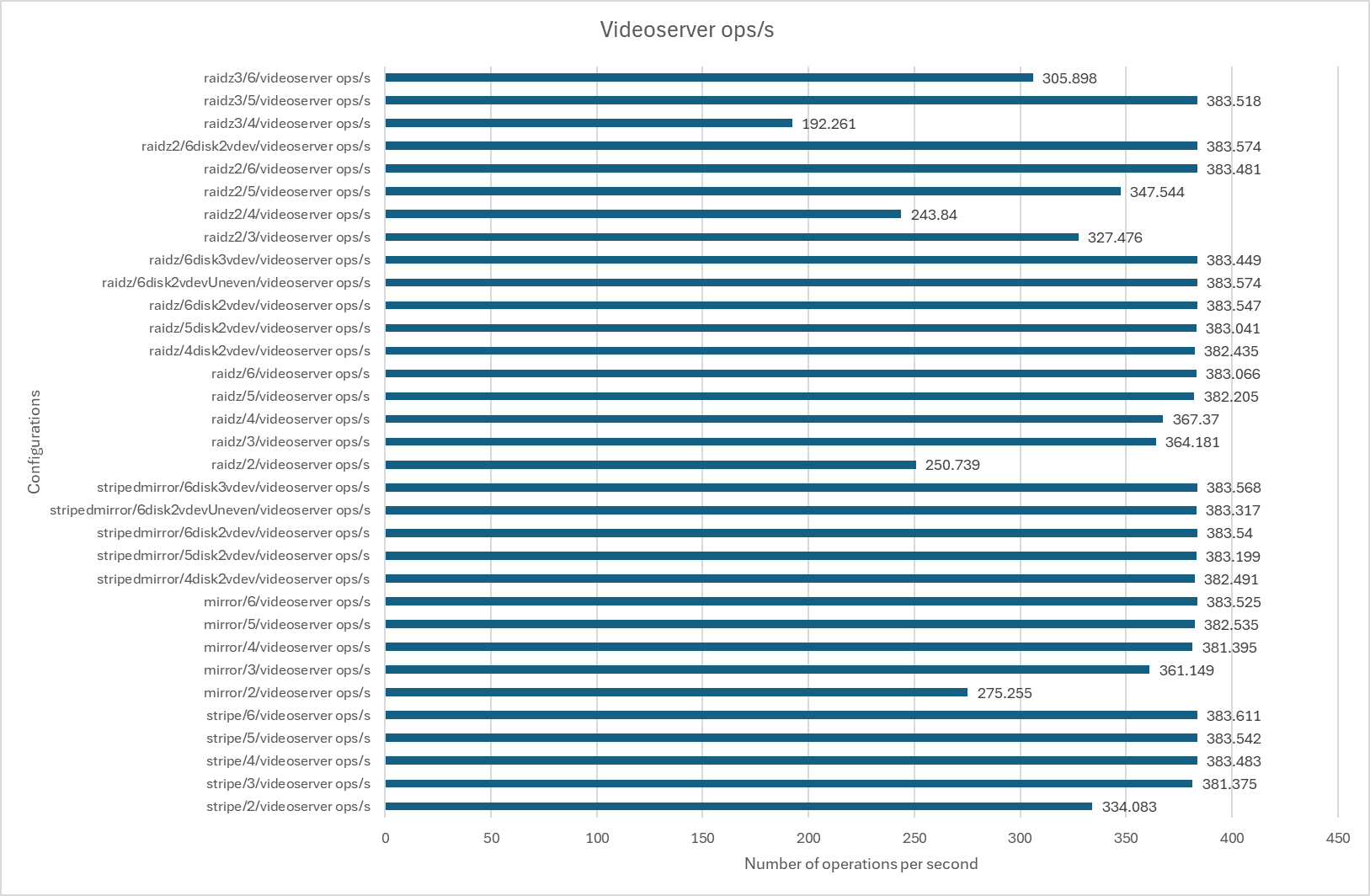
Description automatically generated

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 amount 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 disk numbers increase 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. This, however, is likely caused by a factor that will be addressed later.

#### Video Server:





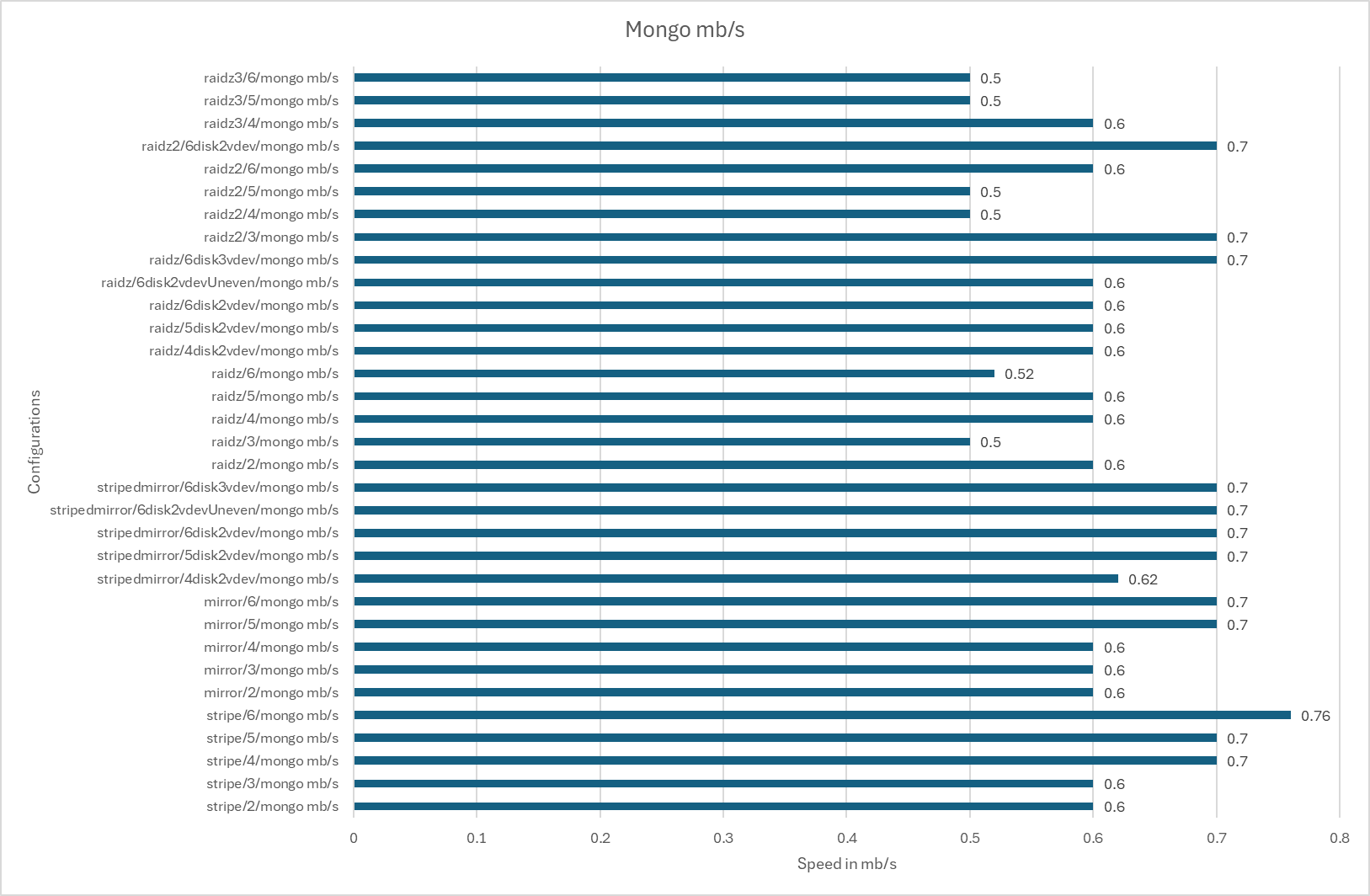
The results from the videoserver workload become slightly more difficult to interpret as, while the results in mb/s show a wide variation in performance, many configurations show near identical ops/s performance. In this case, the number of operations the file system can perform in a second is less important a large amount of data manipulation is not happening. Instead, the majority of the workload is reading large video files and so the most important metric is how quickly the system can do that.

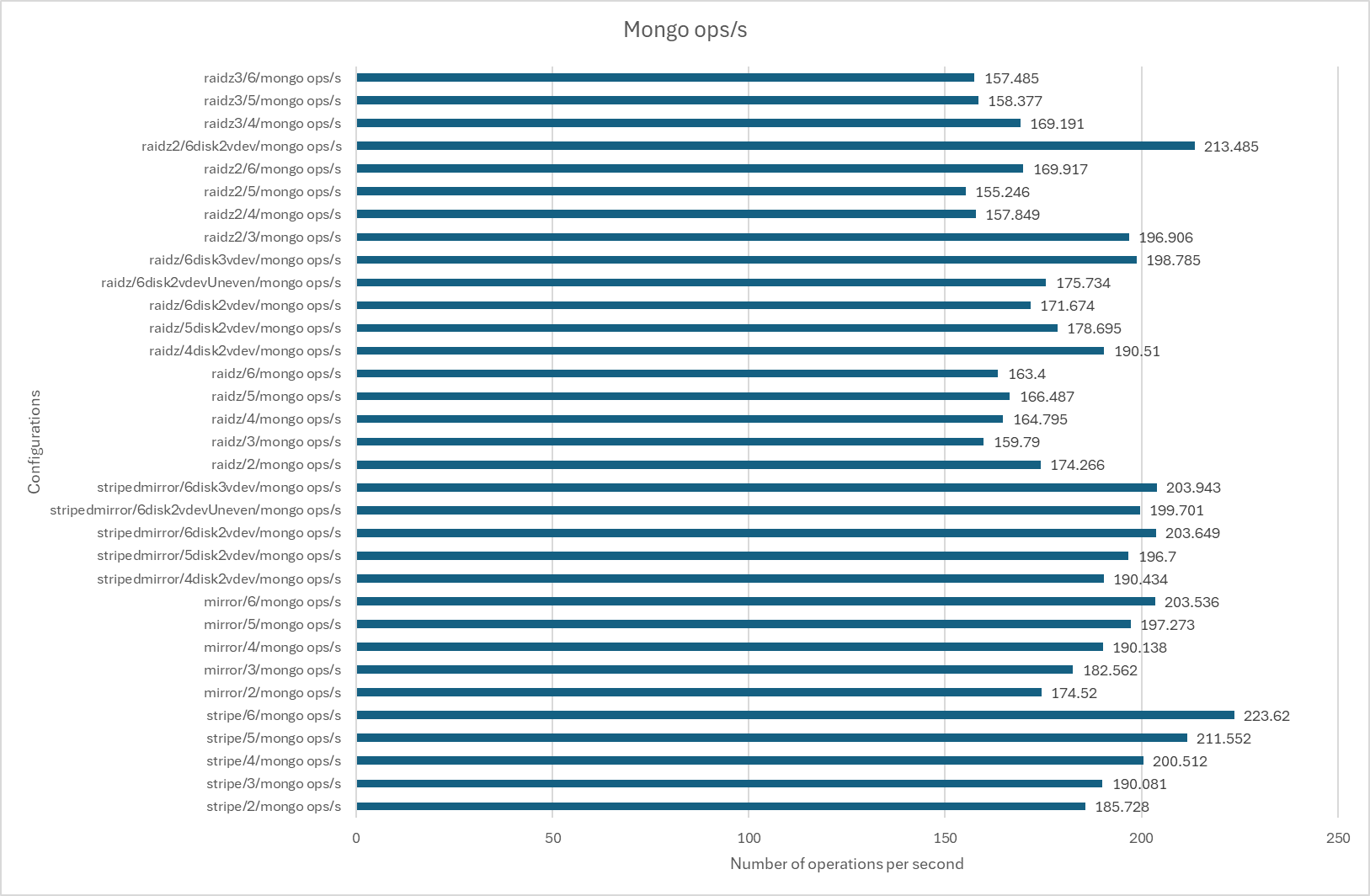
Subsequently, the 6 disk stripe array predictably becomes the most performant, however, in this scenario, it may be a viable configuration. If the filesystem’s only purpose was to stream video to clients and the server was in a high availability cluster, the high risk of array failure could be worth 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 2vdev raidz2 and 3vdev striped mirror and raidz arrays all achieving around 147mb/s. A single vdev raidz3 array offers the same amount of disk failure tolerance, however due to only having a single vdev, performance lags behind achieving only 123.78mb/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. A single vdev 6 disk raidz2 array in particular offers 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 raidz 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 raidz’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, as the MongoDB scenario heavily utilises one operation type, in this case random read and writes, one measurement is not as helpful as the other. Unlike the previous workloads where high throughput levels were critical, the data is much smaller, therefore being able to read or write large amounts of data is not as important as being able to perform as many small operations per second as possible.

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, does not offer a great 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.

The exception to this is minimum disk size raid 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 raidz 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. The reason for this high performance is unclear but could be caused by a factor mentioned next.

### 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 file, such as emails, text files or slideshows, which would drastically slow speeds and result in very different behaviour from the various array.

This becomes immediately apparent, as a 6-disk stripe did not score highest, reporting only 59.2MiB/s, instead being outperformed by a 6 disk array with 2 raidz2 vdevs, which achieved 76.633MiB/s. An array with multiple vdevs performing the best is predictable, as files being striped across multiple vdevs allows each vdev to read separate parts of the files asynchronously, improving performance. Because of this, the expected result would be that arrays with more vdevs would perform better as the file is split into even smaller parts that can all be read independently, however a 6 disk array with 3 mirror arrays is still outperformed by the raidz2 array.

The reason for this is that the default recordsize of 128KiB happens to benefit the 2 vdev raidz2 array greatly. The general use test creates operations of 64KiB and file sizes of 256MiB, which, when split in half to be striped across the array, means each vdev

Might be to do with the Block size being 64kib?

must store 128KiB files, fitting perfectly into the default block size. In random read tests where the array does not perform large amounts of contiguous operations and cannot get up to higher speeds, the advantage of files that fit perfectly into a single block is significant. The array does not have to spend time seeking the multiple blocks that file has been split into, nor does it have to read a single block that has been largely unused, wasting space and time reading empty data.

The other configurations that do not happen to have perfectly sized blocks follow a more predictable pattern, with the 6 disk striped array performing the second best. The two raidz 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 due to no raid calculation overhead. The 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.

The decrease in performance for the raidz3 array as the number of disks increases from 5 to 6 also makes sense, as the in the 5 disk array, the 256KiB file could be split into two chunks of 128KiB, perfectly fitting the block size. Once the number of disks was increased to 6, the file had to be split into blocks of 85.3, increasing the number of blocks needed to be found per file.

This pattern does not seem to repeat in single vdev raidz2 arrays however, where the expected behaviour would be for a 4 disk array to perform extremely well, as the 256MiB file could

##### 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, 2vdev 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 provide an excellent amount of resiliency, even if the amount of usable storage space is lacking.

Configurations otherwise perform very similar, averaging around 58MiB/s, with some notable low performers such as a 2 disk raidz at 53.7MiB/s.

##### Worst Case:

A graph of a number of text

Description automatically generated with medium confidence

#### Random Write:

##### General use:

A graph of numbers and letters

Description automatically generated

Continuing to sequential write speeds

##### Best case:

A graph of a number of writing

Description automatically generated with medium confidence

sdf

##### Worst Case:

A graph of random writing

Description automatically generated

#### Random Read/Write:

##### General use:

A graph of numbers and text

Description automatically generated with medium confidence

Continuing to sequential write speeds

##### Best case:

A graph of numbers and text

Description automatically generated with medium confidence

sdf

##### Worst Case:

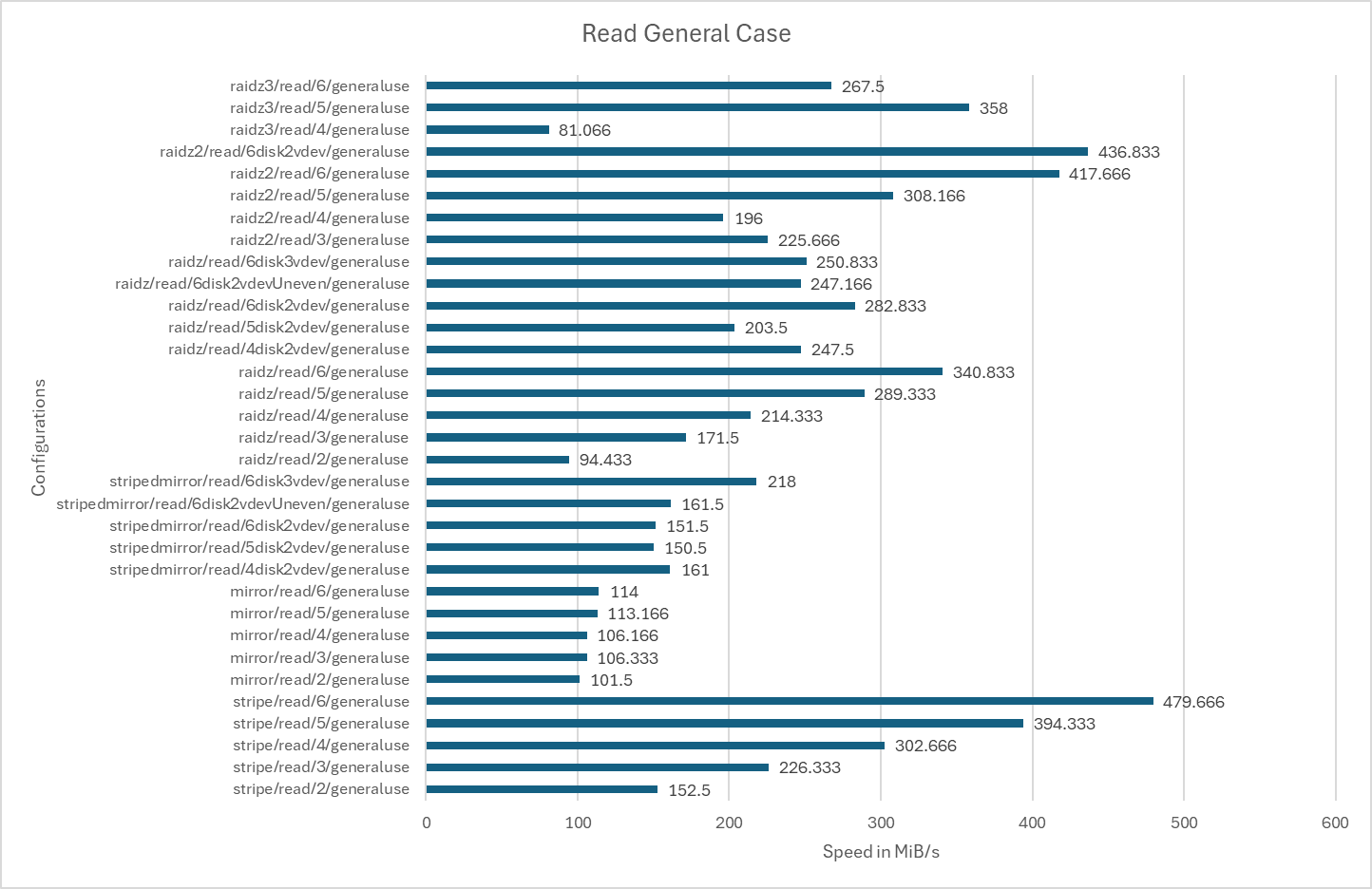
A graph of numbers and text

Description automatically generated with medium confidence

gj

#### Sequential Read:

##### General use:



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, however may appear in something 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. This configuration may be useful for something such as a game server, where loading speeds and large amounts of storage are most important, and the data can easily be replaced if the array goes down due to a drive failure.

A more practical choice may be a raidz2 configuration, with the second highest performance of 436.833MiB/s coming from an array with 2 raidz2 vdevs each containing 3 disks. This would however, lead to only 2 of the 6 disks being used as storage disks, even if the array were extremely resilient to failure with each vdev being able to sustain 2 drive failures, although all data would be lost if a vdev were to sustain 3 drive failures.

Therefore, using 6 disks in a single raidz2 vdev may be the most practical option of all, with a small a small downgrade in performance to 417.666MiB/s, but an increase to 4 of the 6 disks being used as storage disks with the array still being able to sustain 2 drive failures without data loss.

Raidz2 continues to remain a reasonable option as the number of disks decrease with the caveat that, as the number of disks decreases, the percentage of the array that can be used for storage decreases significantly. For instance, with 3 disks a raidz2 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. Upon inspecting the un-averaged results[[9]](#footnote-10), this does not appear to have been caused by any outlying results.

This is also seen in the raidz3 configurations, where a 6 disk raidz3 array achieves 267.5MiB/s, markedly worse than a 5 disk array achieving 358MiB/s. A 4 disk raidz3 array achieving the worst result of all at 81.066MiB/s is expected as the data is effectively only being read from a single disk, however a larger number of disks performing worse than a smaller number is unexpected.

Could be caused by aforementioned zfs default block size reason.

This could potentially be caused by the file being split into smaller pieces to be striped across the increased amount of storage disks resulting in a larger number of smaller files needing to be read. For example, a 6 disk raidz3 vdev storing a 64KiB file would split it into pieces of around 21.3KiB, with 3 21.3KiB parity files being stored in the parity disks. By contrast, a 5 disk would split the file into 2 32KiB pieces, resulting in a smaller number of larger reads compared to a larger number of smaller reads, which could be less efficient. This behaviour, however, is only seen in raid arrays, suggesting it may have more to do with the added raid overhead. Even then, this behaviour is not seen in any raidz configurations, where a larger number of disks does result in an expected linear increase in performance, so more research would have to be conducted on this.

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 raidz configuration may be more appropriate, depending on whether performance or disk failure resiliency are more desirable.

Finally, single vdevs of mirrors do show strong performance numbers, being able to coordinate the retrieval of different blocks on each disk. However, this does come with the obvious issue of only one drive ever being utilised as available storage, even if the array is very resilient to data loss.

##### Best case:

A graph with numbers and letters

Description automatically generated

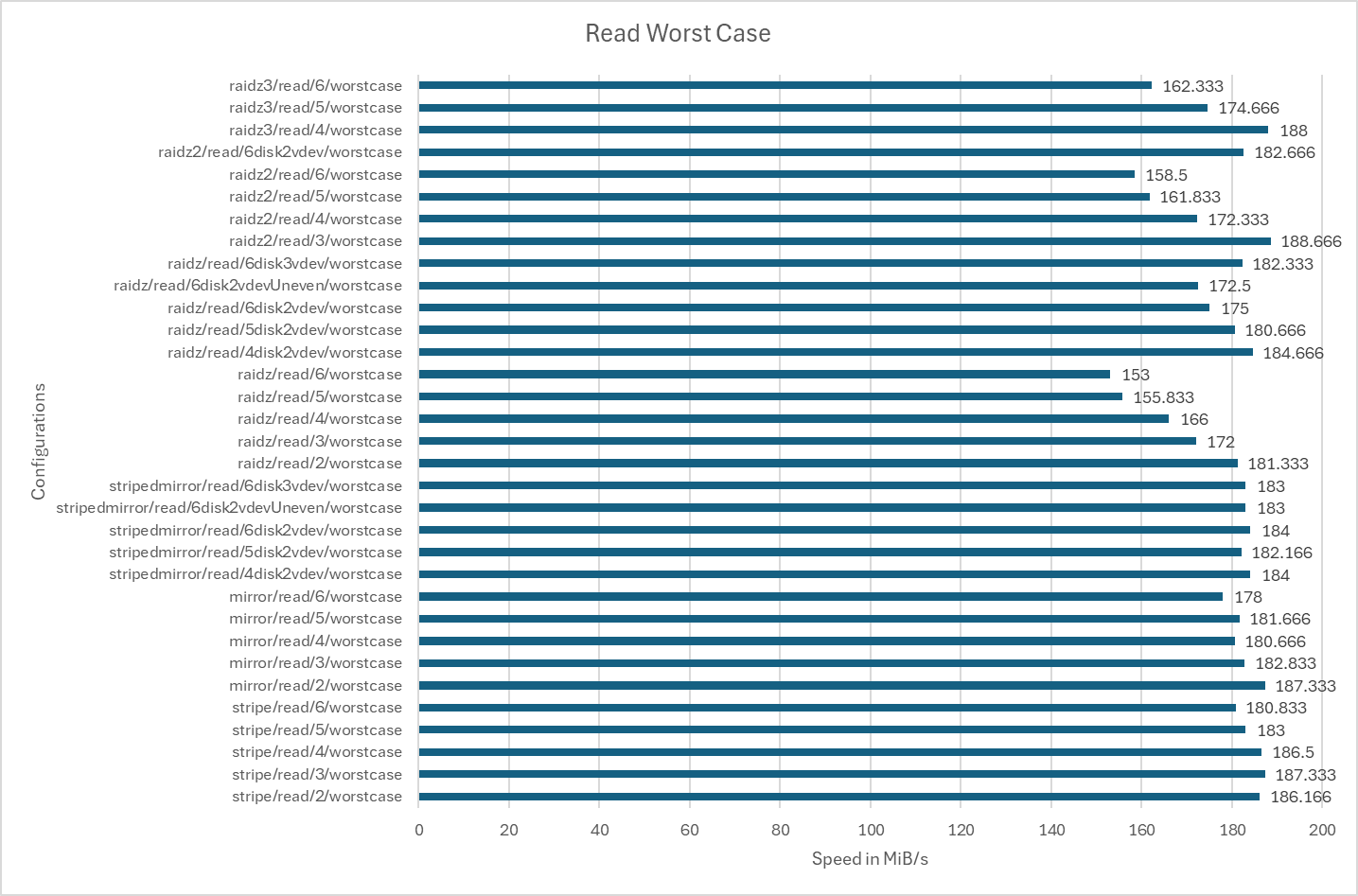
Sequential reads speeds in a best-case scenario data-set would likely reflect those seen in a video streaming server, where large files, continuous video files are being read in a steady and predictable way. The conclusions drawn from this scenario, don’t differ too much from the previous scenario, where the best performing array is a 6 disk stripe at 700.666MiB/s, still with the caveat of no disk failure resiliency. 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 raidz, reporting worse read speeds than a 5 disk raidz 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. This is likely due to the smaller amount of vdevs allowing the file to be split into larger pieces, allowing more efficient reads for large, contiguous files.

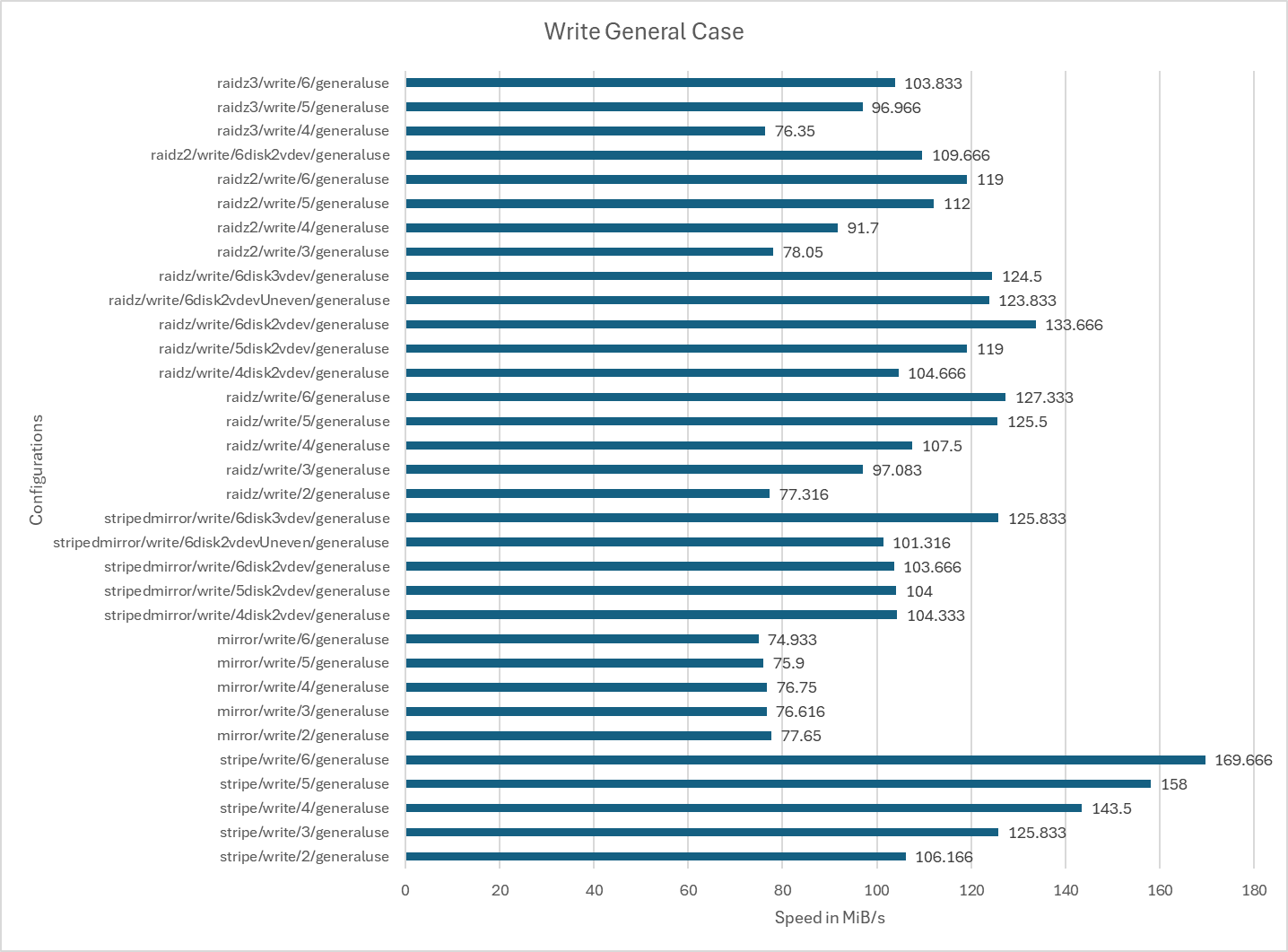
##### Worst Case:



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 raid 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. This could, once again, be caused by blocks needing to be split into increasingly small pieces, to the point where large arrays are having to perform numerous small and inefficient read operations. This would also cause a huge amount of slack space if the default ZFS recordsize were left to 128KiB.

#### Sequential Write:

##### General use:



Continuing on to the write operations, data seen in the general use scenario may be see sequential writes such as these in the of a download, possibly of a large application. For instance, a video game would have many files of different sizes such as dlls, logs, audio files and textures which may all range from many gigabytes to a few kilobytes, all downloaded sequentially.

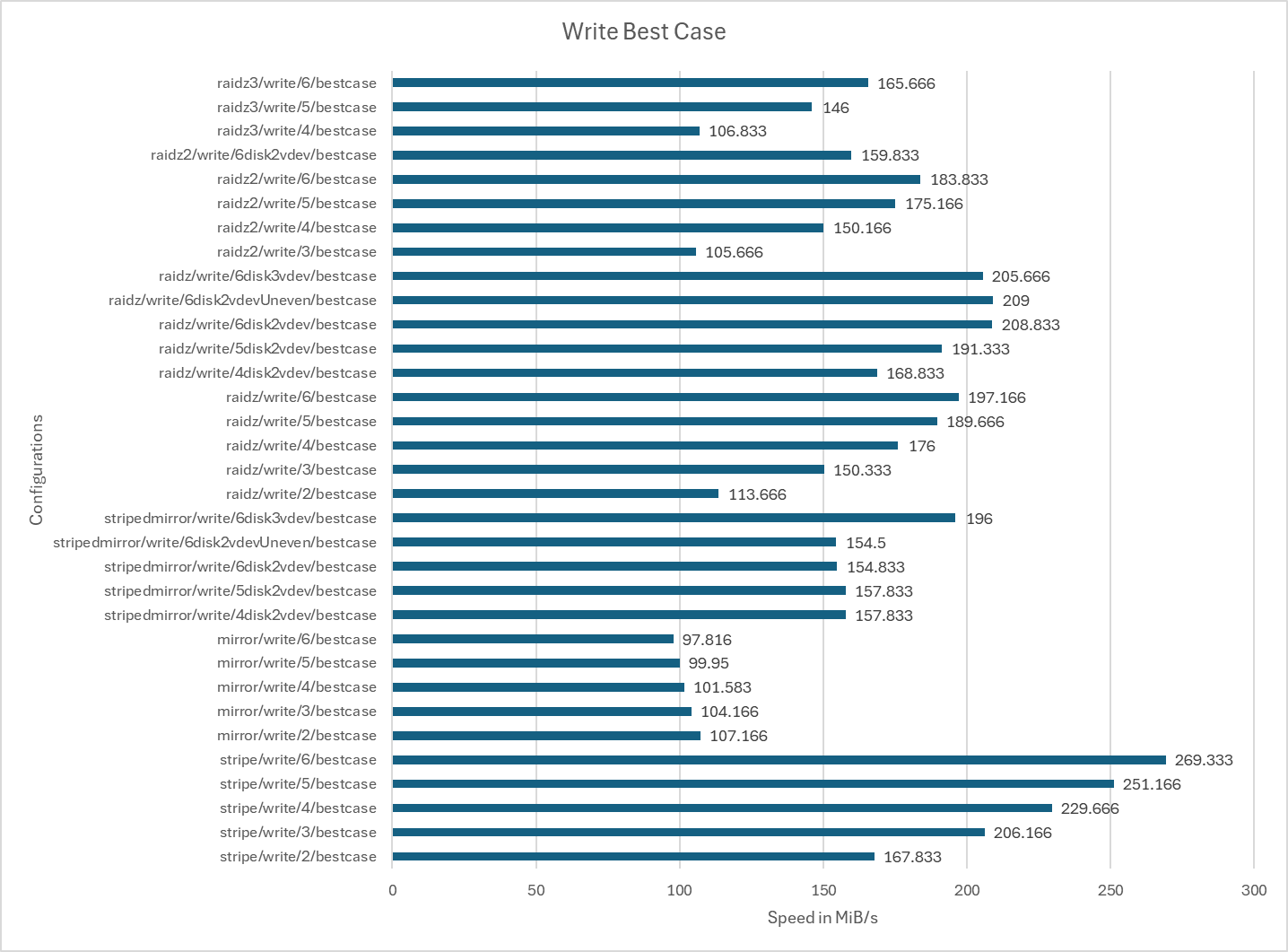
A 6 disk stripe continues to be the most performant at 169.666MiB/s, scaling in an expected linear manner as the number of disks decreases. This time however, 2 raidz vdevs, a single raidz vdev, and 3 mirror vdevs all appear to perform very well at 133.666MiB/s, 127.333MiB/s and 125.833MiB/s respectively. The single 6 disk raidz2 vdev that been a strong performer in the read tests, this time only gives a result of 119MiB/s, lagging behind a large number of other 6 disk configurations. The configurations which contain multiple vdevs seem to favour the smaller more numerous and random files of the general use data-set and the 6 disk raidz benefits from 5 disks being used as storage and minimal overhead from the raid parity.

Other configurations otherwise perform fairly 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 raid 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:



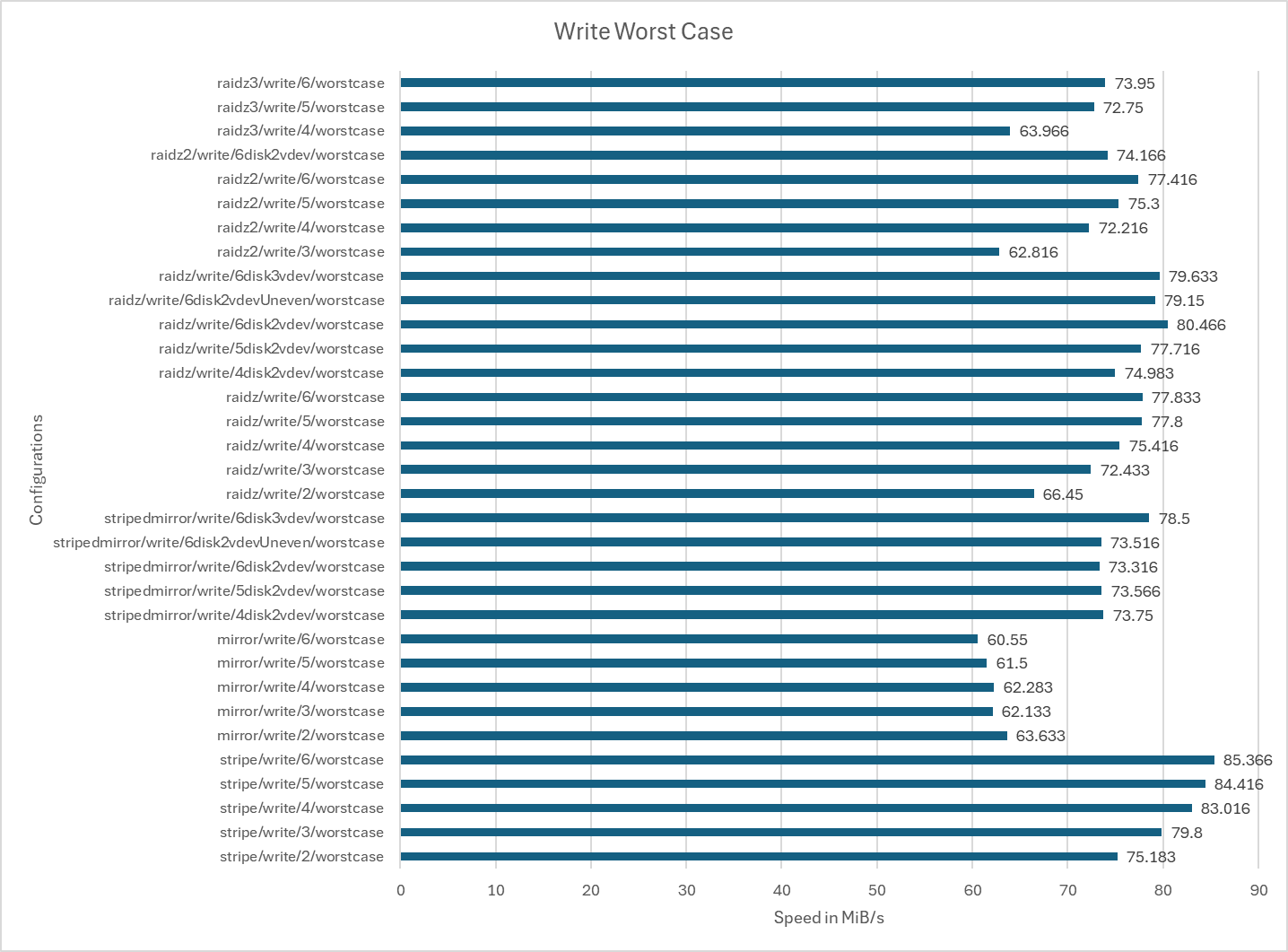
The figures seen in the best case sequential write scenario would likely be reflected in an environment such as a video ingest station, where large contiguous files are being written to the array.

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 raid 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:



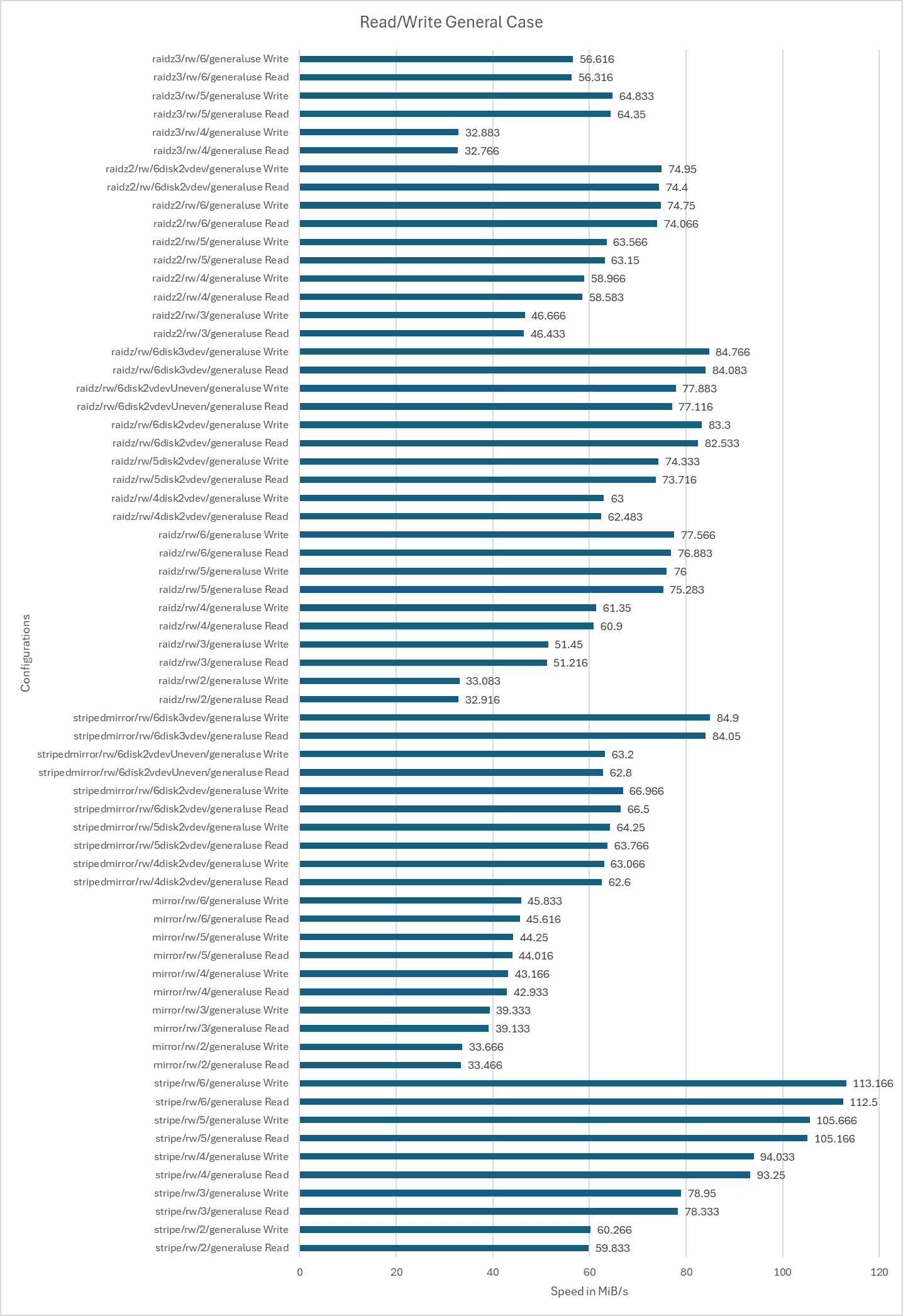
Similar to the worst case read speeds, the worst case write speeds mostly appear to perform similarly, with a 6 disk array of 2 raidz vdevs performing the best at 80.466MiB/s, closely followed by a 6 disk array of 3 raidz vdevs and a 6 disk array of 3 mirror vdevs at 79.633MiB/s and 78.5MiB/s.

Single vdev raid 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:



Sequential read/write speeds demonstrate 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.

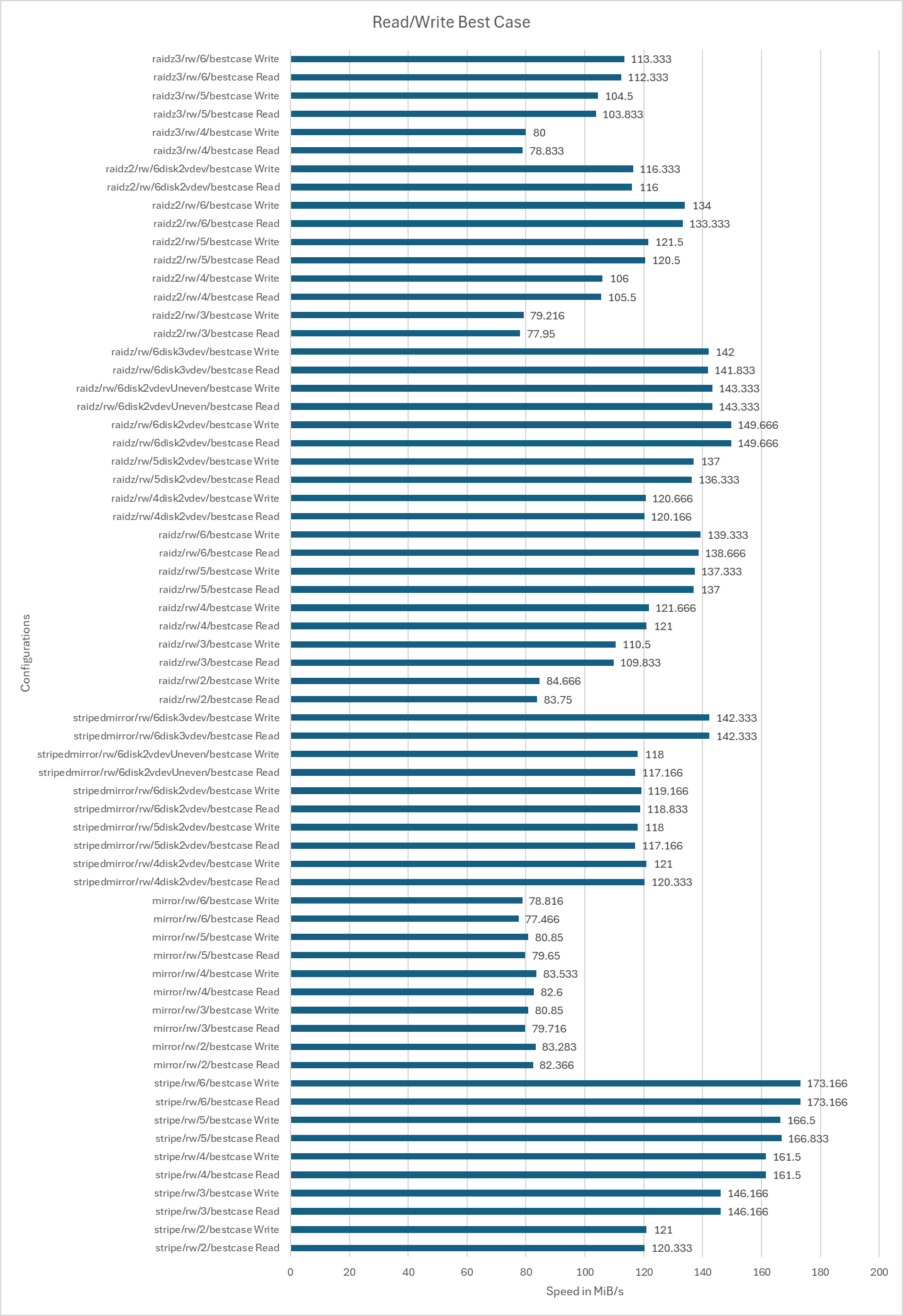
In the general use dataset, aside from the striped arrays, configurations that make use of multiple vdevs show the best results, with a 6 disk array with 3 mirror vdevs performing the best with read speeds of 84.05MiB/s and write speeds of 84.9MiB/s. It is then closely followed by a 6 disk array with 3 raidz vdevs at read speeds of 84.083MiB/s and write speeds 84.766MiB/s. Presumably, the added overhead of working out 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:



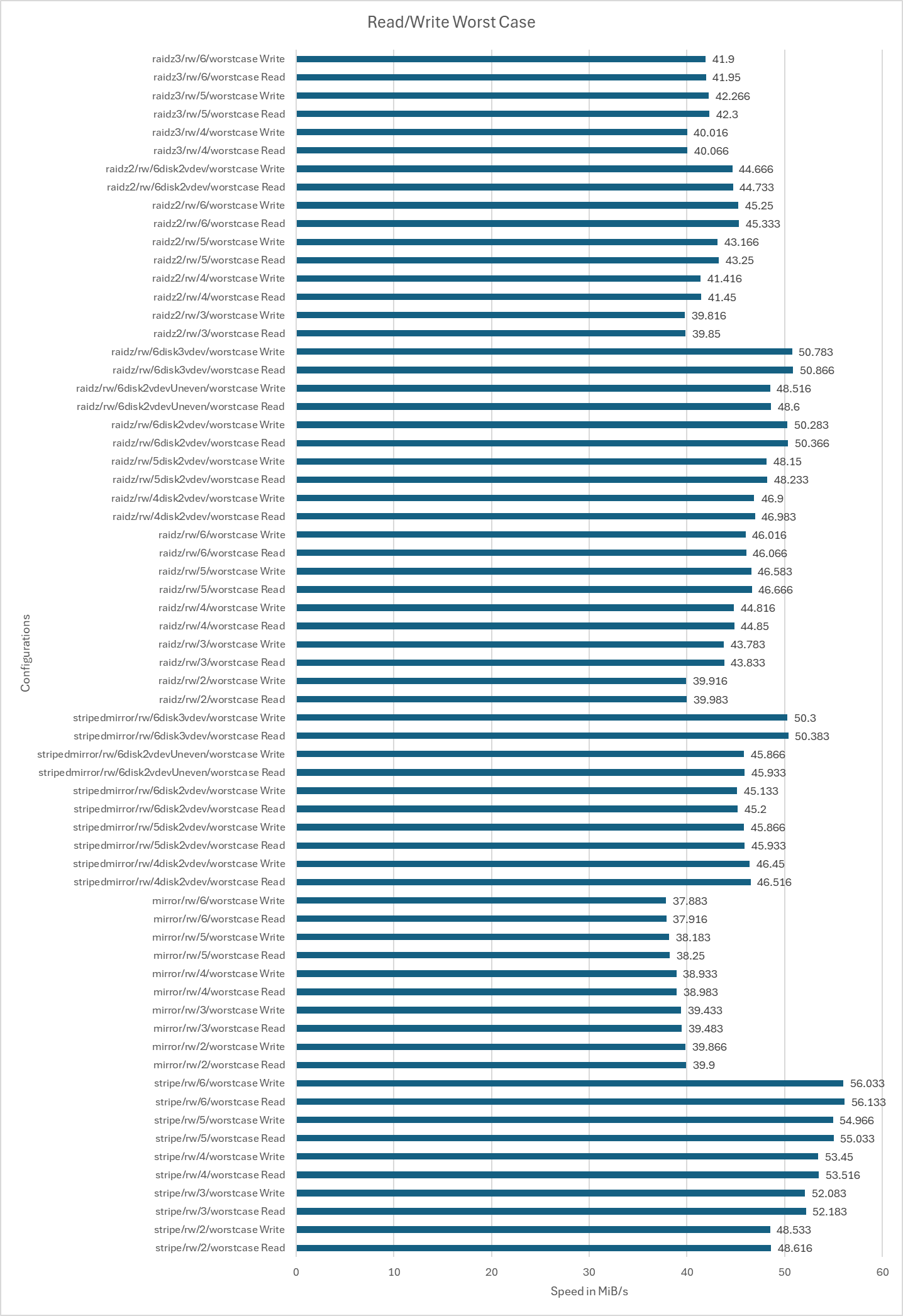
On the best case scenario data-set, read/write operations may be seen in environments such as video editing servers, where users would be intermittently reading large sequential video files and writing edits and cuts to them.

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:



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 dtested. 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:

It can be easy to fixate over which disk configuration achieves the best balance of performance, storage space and resiliency but it is important to remember that there can be ways to make up for a configuration’s shortfalls. For instance, if an array’s main purpose was to be as performant as possible for random operations, an array with multiple 2 disk mirror vdevs may be chosen. One of the disadvantages for this configuration is that only one drive per vdev can fail before all data in an array is completely lost, but extra drives could be set aside as hot spares. Setting an amount of drives this way allows ZFS to hold onto them, not to store any data, but to use as soon as any drive failure is detected. When a drive failure is detected, the hot spare is immediately spun up and begins the resilvering process, eliminating any time between initial failure and human response. This method is not perfect as the other drive in the vdev could fail while the resilvering process is happening. It is in fact, the time when drives are most likely to fail, as the resilvering must rebuild the array from all of the parity data remaining on the existing drive(s) and this process is extremely stressful on drives. It is however, better than nothing and the lack of delay in response could save the array.

Alternatively, if a single vdev, 6 disk raidz2 array was chosen for its balance of storage space and resiliency, steps can be taken to compensate for performance deficiencies.

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 operation sizes of 1MiB are performed, this significantly increases performance to 626MiB/s, as the filesystem does not have to retrieve multiple blocks to commit the operation. On the opposite end of the scale, this tuning can also be repeated on smaller datasets, for instance, the recordsize could be set to 4KiB for a MySQL database, where the default entry sizes are 4KiB which would still require the same amount of block retrievals, but hugely reduce the amount of slack space. Recordsize must be set mindfully however, as setting it small in a scenario where large files are being stored would increase the amount of block retrievals needed, thus decreasing performance. Additionally, if recordsize was set large in a scenario where small files were stored, this would hugely increase the amount of wasted space.

Once the recordsize has been tuned appropriately, performance can be further increased by incorporating caches. In this scenario, a portion of the boot drive has been set aside as a Level 2 Adaptive Replacement Cache, or L2ARC, allowing ZFS to store frequently accessed files in a faster storage device, such as an SSD. This allows ZFS to fetch the data from the faster device, rather than needing to read the data from the slower disk array, improving read speeds. Before the L2ARC can be effective however, ZFS must first observe the read operations to see which files are most commonly read and in need of caching. For this reason, the first run with the L2ARC was slower, but during the second run, when the files had been cached, speeds increased substantially.

For writes, there is also the Separate Intent Log, or SLOG, which allows write operations to be recorded on a secondary device. This increases write speeds as, rather than waiting for the array to commit the operation and confirm it has safely been completed (also known as a synchronous write), this operation can be written to the SLOG and returned as completed before the array has finished writing. In this way, if power is lost, rather than the operation being lost, there will be a non-volatile record of what operations the array was performing and activity can continue from the last completed operation.

# Conclusion:

1. https://static.ixsystems.co/uploads/2020/09/ZFS\_Storage\_Pool\_Layout\_White\_Paper\_2020\_WEB.pdf [↑](#footnote-ref-2)
2. https://jro.io/r2c2/ [↑](#footnote-ref-3)
3. https://picture.iczhiku.com/resource/paper/shIdkpUhfRUjHbcx.pdf [↑](#footnote-ref-4)
4. https://louwrentius.com/zfs-resilver-performance-of-various-raid-schemas.html [↑](#footnote-ref-5)
5. https://www.usenix.org/system/files/login/articles/login\_spring16\_02\_tarasov.pdf [↑](#footnote-ref-6)
6. https://www.usenix.org/system/files/login/articles/login\_spring16\_02\_tarasov.pdf [↑](#footnote-ref-7)
7. https://arstechnica.com/gadgets/2020/02/how-fast-are-your-disks-find-out-the-open-source-way-with-fio/ [↑](#footnote-ref-8)
8. https://jro.io/r2c2/ [↑](#footnote-ref-9)
9. https://github.com/MatthewSwain21101660/ZFSDissertation/blob/main/Results/Transformed%20Results/Results/fioAggregatedResults [↑](#footnote-ref-10)