

■ ZFS BEST PRACTICES ■

BY ALLAN JUDE

ZFS is known for its reliability and for protecting data from the dreaded bitrot. However, a concerning number of users have run into trouble because they did not understand just how different ZFS is from every previous file system they have ever used. The goal of this article is to set out a number of the best practices for using ZFS to help you avoid running into these common misconceptions. This article covers hardware, configuration, tuning, and features, as well as a few other tips.

Beware Hardware RAID


The most costly mistake people make is not giving ZFS control over their redundancy. It is preferable for ZFS to provide the redundancy (using ZFS mirroring or RAID-Z), rather than using a hardware RAID controller. Important data in ZFS is stored multiple times, in what are called ditto blocks. Pool-wide data has three ditto blocks (so is stored three times), and file system metadata has two ditto blocks. In addition, ZFS purposely stores the ditto blocks on different disks, so that the failure of any one disk cannot lead to the loss of all copies of the metadata. This gives ZFS better visibility into what's actually going on with the hardware, allowing more types of errors to be corrected.

When a ZFS pool is created on top of hardware RAID, the RAID controller presents a single logical volume to the operating system (and therefore to ZFS). When ZFS has only one disk to work with, it doesn't have anywhere to store the parity information required to rebuild the array in the event of a failure, and no way to ensure the ditto blocks end up on separate disks. While this doesn't seem like a very large issue because the underlying RAID controller can rebuild the missing disk from parity that exists at the hardware RAID level, using hardware RAID

means you do not get the benefit of ZFS' block level resilvering process. The hardware has to resilver every byte on the entire disk, rather than only the blocks that contain active data. Hardware RAID also doesn't help in the situations where ZFS excels: detecting and repairing flipped bits and other types of otherwise undetectable errors in your data or the file system itself. ZFS' checksumming is one of its most powerful features, but if there is no ZFS redundancy, it can only detect, not repair, the errors.

Hardware RAID controllers can also cause countless other issues, and you are much better off with a less expensive simple HBA controller instead. Hardware RAID controllers will sometimes mask certain errors and silently retry commands, hiding this information from ZFS. This is mostly a holdover from the era where the operating system could not be trusted to gracefully deal with the error. Without this information, ZFS cannot make informed decisions, nor can it self-heal a failing drive by reading the data from the parity location(s).

Even when trying to use a RAID controller as a simple HBA (in "IT", or "JBOD" mode), many controllers will require that you create a single disk RAID 0 (or a JBOD) out of each disk before you can use it. In these cases,



when it comes time to replace a failed disk, one cannot simply just swap the disk out of the chassis, but an operator must use software provided by the adapter manufacturer to create the new volume with the replaced disk. Such a tool may not exist, or may not be supported on FreeBSD, requiring a reboot to access the BIOS level tools provided by the adapter. Suddenly your hot-swap drives still require a reboot and are not providing the high availability you expect. One of the other advantages to ZFS is pool portability. All of the disks in a pool can be moved to another server, easily imported, and brought back into operation in short order. However, if the disks are labeled by manufacturer A's RAID utility, they won't be accessible by manufacturer B's RAID controller. This might make it impossible to read the disks if the controller fails, and cannot be replaced with an identical controller.

If some outside factor absolutely requires that you use hardware RAID, always present a pair of logical volumes to be mirrored in ZFS, or 3 or more volumes for RAID-Z. The added redundancy provided by the hardware RAID will cost capacity and performance, but not taking advantage of ZFS' better redundancy will eventually make you wish you had. Also be sure to disable "write back" mode on the controller. If write back is not disabled, the cache flush commands ZFS relies upon to ensure all the pending data is actually written to the disk are ignored, possibly causing data loss. Ultimately, ZFS on top of hardware RAID decreases performance while adding complexity, and that increases risk.

ECC Ram

Many articles and posts will emphatically state that in order to use ZFS, you must have ECC RAM. To quote Matt Ahrens, cocreator of ZFS: "ZFS on a system without ECC is no more dangerous than any other file system on a system with ECC." The features of ZFS are extremely compelling and extraordinarily useful, no matter what role the machine is playing. The data protections provided by ZFS outclass that of every other file system, even without the benefit of ECC RAM. Don't let the FUD scare you away from running ZFS, even on a single disk on your laptop. Just be sure to take careful backups.

If your goal is high availability, the extra cost of ECC is easily justifiable. Server grade hardware uses ECC RAM, because it mitigates the

risk of memory errors, and the application crashes those errors can cause. Using ECC RAM reduces (but does not entirely eliminate) the risk of corruption on in-flight data, after it leaves the application, but before ZFS has calculated the checksum and written it to stable storage. If you care about the reliability and availability of your system, you should use ECC RAM, regardless of what file system you use.

Backups Are Still Required

While the reliability and durability of ZFS make us feel much safer about our data, too few people still maintain proper backups of the data they store on their ZFS pools. No amount of RAID redundancy is as good as a backup. Accidentally destroy the wrong pool? Where is your backup? Pull the wrong disk while replacing a failed one? Where is your backup? Add an extra disk as a new vdev instead of attaching it to the existing mirror? Where is your backup?

ZFS provides a number of features that make taking backups much easier and safer. The first is instantaneous snapshots. Instead of backing up a live system, where files can be changing constantly, take a backup of a recursive snapshot of the system. This way, everything contained in the backup is saved as it was at an exact moment in time, even if the backup takes multiple days to complete. The other powerful backup feature is the built-in incremental replication system. The first advantage to this method is that, compared to walking the file system with Bacula or rsync, ZFS walks its internal representation of the data, resulting in a fast contiguous read of the blocks. The output of "zfs send" can be incremental or full. In incremental mode, the output maintains the copy-on-write aspects of the data stream, including all the snapshots. This output can either be stored as massive binary block (a ZFS data stream), or piped to "zfs receive", where a replica of the dataset can be recreated on another machine, effectively creating a "warm spare" of the dataset. ZFS also has another feature that is great for taking live backups: pool splitting. When your pool consists of mirror vdevs, with ideally at least 3 devices in each mirror set so that the pool is not at risk from a single device failure during this operation, one device from each mirror vdev can be detached by the "zpool split" command, creating a new pool. This new pool contains one drive from



each vdev and all of the data from the current pool. These disks can then be ejected and moved to an off-site location, similar to a tape backup. New disks are then swapped in, and rejoin the mirror set and resilver. Repeated on an ongoing basis, this process provides a complete offsite backup that takes only a few seconds to create. Drives can be reused once they have exceeded your backup retention threshold. Compared to a regular backup, the pool split method can be less performance impacting because the backup time is amortized over the period after the new drives are added to the pool. As new data is written to the pool, it is mirrored to the regular mirror devices, but also to the device that will be split off when the next backup is taken. In this setup, it may be wise to adjust the resilver throttling (`vfs.zfs.resilver_delay`), to avoid unduly impacting performance when the new drives are added after the split.

Disk Labeling

There are a number of ways to handle labeling disks and creating the logical connection between the device exposed by the operating system and the physical disk in its physical location in the chassis. The one that has worked best for me in production has been the physical slot number followed by the disk's serial number. For example, our one chassis has 24 drives in the front, and 12 more in the rear. The drive in slot 6 in the front is f06-WMC1F125320, and a drive in the rear might be r02-9WM7HATN. There are a number of reasons to use the serial number, mainly inventory and warranty management; the serial number is required to check if the drive is still under warranty and to get it replaced. It is a convenient unique identifier, although depending on the model and drive manufacturer, it can be a bit unwieldy. The serial number for one of my Intel SSDs is CVDA333604282403GN, which one would struggle to fit on a label on the front of a 2.5 inch drive carrier, and doesn't fit within the 15 character GPT label. It is best to truncate and keep the most significant digits, so if the serial numbers share a common beginning, use the last characters of the serial number. Adding the slot number as a prefix helps operators and data center technicians more quickly identify the correct physical drive, without having to compare the entire serial number, especially when a series

of drives may have a common prefix in the serial number. The next step is to change the operating system's representation of the drive to match. The recommended approach is to use GPT partition labels, like so:

```
gpart modify -i 2 -l f01-9WM6T60L da0
```

This will create an alias for da0 called `/dev/gpt/f01-9WM6T60L`

Note: The maximum length of the label is 15 characters.

This way, even if the order in which the operating system recognizes the devices changes, the name of the device stays the same. It also means that the output of "zpool status" will show each disk, with its location and serial number. With this technique, a missing disk will be obvious, and it will be easy to communicate to the operator or data center technician which disk needs to be replaced.

FreeBSD offers a number of different ways to label disks, and it may be helpful to disable the unused ones, to avoid ZFS picking up those device names instead:

```
/boot/loader.conf:
```

```
kern.geom.label.disk_ident.enable=1
kern.geom.label.gptid.enable=0
```

Disk Ident example:

```
/dev/diskid/DISK-07013121E6B2FA14
/dev/diskid/DISK-%20%20%20%20%20WD-WCC131365642
/dev/diskid/diskid/DISK-%20%20%20%20%20%20%20%20%20%20%20Z300HTCE
```

GPT ID example:

```
/dev/gptid/b829bf8c-46ad-11e3-ae0f-002590721162
/dev/gptid/b88eeff5-46ad-11e3-ae0f-002590721162
```

Notice how at first glance, the two appear to be identical. The difference is at the beginning of the string, rather than the end.

Another disadvantage to Disk Ident and GPTID is that the partition identifiers get tacked on the end, so the 2nd partition on the disk is

```
/dev/diskid/DISK-07013121E6B2FA14p2,
```

which blends into the unique ID of the disk

In more advanced setups with SAS expanders, dual ported disks, and multiple controllers, each disk may be presented to the operating system multiple times, once for each unique path.

FreeBSD's GEOM storage management layer has a system for this, `gmultipath`. This writes a unique label to the disk, and then when 2 or more devices appear with the same label, they are classified as multiple paths to the same physical disk.

In the end, it looks something like this:

```
# gmultipath status
multipath/f01-WMC1F125320  OPTIMAL      da0 (ACTIVE)
                               da36 (PASSIVE)
multipath/f02-WMC1F125298  OPTIMAL      da1 (ACTIVE)
                               da37 (PASSIVE)
multipath/f03-WMC1F125506  OPTIMAL      da2 (ACTIVE)
                               da38 (PASSIVE)
```

Setting Up the Disks

Before creating the pool, the disks need to be prepared. There are a number of guides, blogs, and other resources that state that ZFS should always be used on an entire disk, not a partition. While this is true in Solaris, because of the way the disk cache works, it is not true under FreeBSD. There are a few considerations at this point. If the system will boot from the pool, then all of the disks should contain the ZFS boot code. In case of a failure, it may not be possible to predict which disk the system will try to boot from. The `freebsd-boot` partition should be 512kb; this is just shy of the maximum imposed by the FreeBSD ZFS boot blocks. The purpose behind using the maximum size is to ensure that there will be enough room for the boot blocks to grow over time. All of the partitions created on the disk should be aligned to 4k boundaries, to keep the partition layout consistent across all the disks. This can be accomplished by using `-a 4k` with each `gpart` command when creating the partitions.

```
gpart create -s gpt ada0
gpart add -t freebsd-boot -l bootfs0 -s 512k -a 4k ada0
gpart add -t freebsd-swap -l swap0 -s 2g -a 4k ada0
gpart add -t freebsd-zfs -l f01-9WM6T60L -a 4k ada0
gpart show -l ada0
=>  34 7814037100  ada0  GPT (3.7T)
    34         6          - free - (3.0k)
    40        1024        1      bootfs0 (512k)
    1064       984        - free - (492k)
    2048      4194304     2      swap0 (2.0G)
    4196352   7809839104  3      f01-9WM6T60L (3.7T)
    7814035456 1678          - free - (839k)
```

Even if the current disks use 512 byte sectors, in the future it may not be easy to obtain 512 byte sector disks to replace these, so setting up the entire pool based on 4k sectors ensures that complications will not arise when a disk needs to be replaced in the future. With that same goal in mind, the partition that will hold the ZFS data should be created slightly smaller than the available size of the disk. This extra space can be used as a swap partition. This slack will offer some wiggle room in the case where the replacement disk does not have the exact same sector count as the original disk. Lastly, the ZFS pool itself should use 4k sectors. Again, even if your drives are 512 byte sectors, their future replacements may not be, and it is not possible to mix sector sizes, nor to change the ZFS sector size after the pool is created. To force ZFS to use a 4k sector size, set the `sysctl vfs.zfs.min_auto_ashift=12` ($2^{12} = 4k$) before creating the pool. The downside to 4k sectors is slightly worse space efficiency and possibly worse performance for very small reads or writes (less than 4k). In the case of a database type workload and sub 1 TB disks, 512 byte sectors may be desirable.

Some disk models include an "XP Jumper," which offsets each LBA address by 1, so that the default starting location of the first MBR partition (63rd sector) becomes the 64th 512 byte sector and is therefore 4k aligned. However, if this jumper is set and we ask the partitioning tool to align the partitions to 4k, they will all be off by 1 sector, and will cause the performance penalties we were trying to avoid in the first place.

Pool Layout

Determining the best way to lay out the disks and `vdevs` in your pool is one of the hardest questions facing a user creating a new pool.

There are many factors to consider, and once the decision is made, it generally cannot be changed. The biggest factors are random I/O performance, streaming performance, space efficiency, and fault tolerance. Each different configuration provides different benefits. For the best IOPS performance for random reads, the best solution is always more `vdevs`. Sets of mirrors (equivalent to RAID 10) provide the best performance because the IOPS of each `vdev` is effectively limited to that of the slowest device, so 12 disks in 6 mirror sets provides 6x the IOPS of a single disk, whereas all 12 disks in a single RAID-Z (1, 2, or 3) provides only 1x the IOPS of a single disk.



More performance can be gained by running the 12 disks as 2 RAID-Z2 vdevs, or even 3 or 4 RAID-Z1 vdevs, at the cost of less usable space, but the performance never reaches that of the mirror sets. In the case of streaming performance, where IOPS make much less of a difference, spindle count is all that matters.

Assume a modest set of commodity spinning disks, 1 TB in size and capable of 250 IOPS and streaming read/writes at 100 MB/s:

Avoiding Single Points of Failure

Avoiding single points of failure will increase the availability of your pool. With some planning and informed design decisions, the same hardware can be organized in a more fault-tolerant configuration. In larger installations, where all the disks may not reside in the same physical chassis, consideration should be given to which disks belong head with 3 external JBOD chassis with 36 disks


| Disks | Configuration | Read IOPS | Write IOPS | Read MB/s | Write MB/s | Usable Space | Fault Tolerance |
|-------|--------------------|-----------|------------|-----------|------------|--------------|-----------------|
| 2 | 1x 2 disk Mirror | 500 | 250 | 200 | 100 | 1 TB | 1 |
| 3 | 1x 3 disk Mirror | 750 | 250 | 300 | 100 | 1 TB | 2 |
| | 1x 3 disk RAID-Z1 | 250 | 250 | 200 | 200 | 2 TB | 1 |
| 4 | 2x 2 disk Mirror | 1000 | 500 | 400 | 200 | 2 TB | 1 (2*) |
| | 1x 4 disk RAID-Z1 | 250 | 250 | 300 | 300 | 3 TB | 1 |
| 5 | 1x 5 disk RAID-Z1 | 250 | 250 | 400 | 400 | 4 TB | 1 |
| | 1x 5 disk RAID-Z2 | 250 | 250 | 300 | 300 | 3 TB | 2 |
| 6 | 3x 2 disk Mirror | 1500 | 750 | 600 | 300 | 3 TB | 1 (3*) |
| | 2x 3 disk Mirror | 1500 | 500 | 600 | 200 | 2 TB | 2 (4**) |
| | 1x 6 disk RAID-Z1 | 250 | 250 | 500 | 500 | 5 TB | 1 |
| | 1x 6 disk RAID-Z2 | 250 | 250 | 400 | 400 | 4 TB | 2 |
| 12 | 6x 2 disk Mirror | 3000 | 1500 | 1200 | 600 | 6 TB | 1 (6*) |
| | 4x 3 disk Mirror | 3000 | 1000 | 1200 | 400 | 4 TB | 2 (8**) |
| | 2x 6 disk RAID-Z1 | 500 | 500 | 1000 | 1000 | 10 TB | 1 (2*) |
| | 2x 6 disk RAID-Z2 | 500 | 500 | 800 | 800 | 8 TB | 2 (4**) |
| 36 | 18x 2 disk Mirror | 9000 | 4500 | 3600 | 1800 | 18 TB | 1 (18*) |
| | 12x 3 disk Mirror | 9000 | 3000 | 3600 | 1200 | 12 TB | 2 (24**) |
| | 1x 36 disk RAID-Z2 | 250 | 250 | 3400 | 3400 | 34 TB | 2 |
| | 2x 18 disk RAID-Z2 | 500 | 500 | 3200 | 3200 | 32 TB | 2 (4**) |
| | 4x 9 disk RAID-Z2 | 1000 | 1000 | 2800 | 2800 | 28 TB | 2 (8**) |
| | 6x 6 disk RAID-Z2 | 1500 | 1500 | 2400 | 2400 | 24 TB | 2 (12**) |

* Provided that the failures are limited to 1 per vdev

** Provided that the failures are limited to 2 per vdev

Using a larger number of smaller groups of disks increases performance at the cost of reduced usable space (more parity). Streaming read and write performance is constrained by the number of non-parity spindles. This leads to another consideration for both random and streaming performance: spindle count. An array of 12x 1 TB drives will usually outperform 6x 2 TB drives, because the greater spindle count increases both IOPS and streaming performance.

each should be configured such that each RAID-Z2 vdev consists of 2 disks from each JBOD (18 vdevs of 6 disks each). In this configuration, even if one JBOD's power supply, HBA, or cabling fails, each vdev is still functional. Whereas if the configuration consisted of vdevs made up of disks all in the same JBOD, a number of vdevs would be faulted, and the system would not be able to continue. The same approach can be taken in smaller systems that might not have multipath to tolerate an HBA failure. If constructing mirror



pairs, ensure that each disk is paired with another disk that is not on the same controller, so the failure of one controller only degrades, rather than faults, each affected vdev.

Compression

ZFS supports transparent compression where data is compressed as it is written to the disk and decompressed as it is read back, without the user or application needing to be aware. In addition to the obvious decrease in storage utilization this provides, it also increases performance. Even with the small CPU usage penalty, compressed data can be read from the disks at the same speed as uncompressed data, but once decompressed, provides a much higher effective throughput. If a disk can read 100 MB/s, and data is compressed 50%, then that data can now be read at effectively 150 MB/s. The same applies to writes, where there is increased throughput and decreased latency because a smaller amount of data takes less time to write. This makes compressed datasets very useful for databases, which often contain highly compressible text and always benefit from higher throughput and lower latency. The newer LZ4 compression algorithm used in ZFS also has an “early abort” feature, which will store a block uncompressed if the compression ratio on the first bit of the block is less than 12.5%. This further reduces the performance impact of using compression, since incompressible files are quickly skipped. With this in mind, you can consider using LZ4 compression on the entire pool.

Deduplication

ZFS supports online deduplication, meaning data is deduplicated as it is written. While this feature appears attractive, you will see it is quite expensive. Here is why. In order to deduplicate the data as it is written, ZFS builds a hash table of the SHA256 checksum of each block (called the deduplication table, or DDT). The DDT is stored in main memory as part of the ARC (Adaptive Replacement Cache). As new blocks are queued to be written, they are first compared to the DDT, and if a match is found, the ref count of the existing block just needs to be increased, instead of writing out the block. If no match is found, the new data is written and the new checksum is added to the DDT. Each entry in the DDT takes 320 bytes of memory or more. If there is not enough room in the metadata portion of the ARC, the DDT entries are written to the L2ARC (usually an SSD or other fast storage device used as a second level cache) where they take even

more space. In the typical case of a zvol backed iSCSI target, the block size is 8 KB, meaning 1 TB of unique data would generate nearly 48 GB of DDT. If that won't fit in ram, it is instead stored in the L2ARC, but that has a memory cost of its own. In addition, DDT entries take more space on disk than they do in memory, and have worse performance. In the event of a problem with the pool, the DDT needs to be able to fit in memory when the pool is imported, and if it cannot, the pool may not be able to be imported. DDT and L2ARC mappings also count as metadata, which by default is restricted to only 1/4 of the available ARC memory. If you are going to use deduplication, the metadata limit will need to be adjusted. The benefit of deduplication has a very high cost—if your dataset is only going to get a moderate deduplication ratio, you are most likely much better off just using LZ4 compression. If I still have not dissuaded you from using deduplication, you should conduct some tests on your data to ensure you are getting a very good deduplication ratio, and calculate how much ram will be required to store the DDT for all of your data. Be sure to leave room in the ARC for actual metadata, in addition to the DDT and L2ARC index, and, of course, the cache of your actual data. You will need a lot of ram.

Reservations

The pooled nature of storage in ZFS means that all the available free space is available to every dataset. Compared to the traditional way of doing things, partitioning a RAID volume or creating separate volumes, the free space does not become fragmented. The downside to this is that one workload or user can consume all the available space. While this can be addressed with quotas, that doesn't always solve the problem. ZFS also offers a reservation system where a specific dataset for a critical database or for security logs can be guaranteed a minimum amount of space unavailable to any other dataset. To ensure that the e-commerce database never runs out of room because of HTTP logs, give its dataset a reservation.

Although recent improvements to ZFS have improved the situation greatly, a ZFS pool that is nearly full will perform badly. If the pool becomes completely full, the administrative commands to resolve the situation can take an exceedingly long time. One way around this is to create a new dataset, called “reserved,” with a reservation of 20% to 25% of the total capacity of the pool. This will prevent that critical last bit of space from being used up. The reservation can be relaxed to

allow the administrator to perform needed operations or to tide the system over until the pool can be expanded.

Tuning

The biggest variable in ZFS is the size of the ARC. The ARC is where ZFS stores recently-used and frequently-used data as well as its metadata. The ARC provides most of the amazing performance that comes with ZFS. The maximum size of the ARC defaults to 1 GB less than all memory in the system (or 1/2 of all memory in machines with little memory). For a dedicated file server, this makes sense; however, if there are going to be other applications that require memory, you might want to tune this to something more modest, leaving room for other applications like a web server or database server. The limit is set with a tunable in `/boot/loader.conf`:

`vfs.zfs.arc_max`. Generally it is best to leave at least a few gigabytes of memory for the OS and for applications, but maximizing the memory available to the ARC will increase performance. The ARC will release memory back to the OS when it detects memory pressure, but this is not instant and may cause heavy swapping. As dis-

cussed earlier, the default limit on metadata is 1/4 of the ARC. If your dataset contains a very large number of small files, it might be advantageous to increase this value.

Hopefully these tips and best practices guide you well and help you avoid some of the common mistakes made by those new to ZFS. We hope to see you join the thriving community of OpenZFS users and developers. Special thanks to Dan Langille, the backup guru, and Michael Dexter, who has dressed many self-inflicted foot wounds for various clients and shared what he learned. •

Allan Jude is VP of operations at ScaleEngine Inc., a global HTTP and Video Streaming CDN (Content Distribution Network), where he makes extensive use of ZFS on FreeBSD. He is also the host of the video podcasts *BSD Now* (with Kris Moore) and *TechSNAP* on *JupiterBroadcasting.com*. Allan is a FreeBSD doc committer, focused on improving the handbook and documenting ZFS. He taught FreeBSD and NetBSD at Mohawk College in Hamilton, Canada, from 2007 to 2010 and has 12 years of BSD UNIX sysadmin experience.

ISILON The industry leader in Scale-Out Network Attached Storage (NAS)

Isilon is deeply invested in advancing FreeBSD performance and scalability. We are looking to hire and develop FreeBSD committers for kernel product development and to improve the Open Source Community.



We're Hiring!

With offices around the world, we likely have a job for you! Please visit our website at <http://www.emc.com/careers> or send direct inquiries to karl.augustine@isilon.com.



EMC²

ISILON