THE FUTURE OF STORAGE

BY ALLAN JUDE

THE Z FILE SYSTEM (ZFS)

created by Jeff Bonwick and Matthew Ahrens at Sun Microsystems is fundamentally different from previous file systems.

THE KEY difference is that ZFS is, in fact, more than just a file system, as it combines the roles of RAID controller, Volume Manager, and File System.

Most previous file systems were designed to be used on a single device. To overcome this, RAID controllers and volume managers would combine a number of disks into a single logical volume that would then be presented to the file system. A good deal of the power of ZFS comes from the fact that the file system is intimately aware of the physical layout of the underlying storage devices and, as such, is able to make more informed decisions about how to reliably store data and manage I/O operations.

Originally released as part of OpenSolaris, when Sun Microsystems was later acquired by Oracle, it was decided that continued development of ZFS would happen under a closed license. This left the community with ZFS v28 under the original Common Development and Distribution License (CDDL). In order to continue developing and improving this open source fork of ZFS, the OpenZFS project was created—a joint effort between FreeBSD, IllumOS, the ZFS-On-Linux project, and many other developers and vendors. This new OpenZFS (included in FreeBSD 8.4 and 9.2 or later) changed the version number to "v5000 - Feature Flags", to avoid confusion with the continued proprietary development of ZFS at Oracle (currently at v34), and to ensure compatibility and clarity between the various open source versions of ZFS. Rather than continuing to increment the version number, OpenZFS has switched to "Feature Flags" as new features are added. The pools are marked with a property, `feature@featurename`, so that only compatible versions of ZFS will import the pool. Some of these newer properties are read-only backwards compatible, meaning that an older implementation can import the pool and read, but not write to it, because they lack support for the newer features.

What Makes ZFS Different?

The most important feature sets in ZFS are those designed to ensure the integrity of your data. ZFS is a copy-on-write (COW) file system, which means that data is never overwritten in place, but rather the changed blocks are written to a new location on the disk and then the metadata is updated to point to that new location. This ensures that in the case of a shorn write (where a block was being written and was interrupted before it could finish) the original version of the data is not lost or corrupted, as it would be in a traditional file system. In the case of a power failure or system crash, the file is left in an inconsistent state in which it contains a mix of new and old data. Copy-on-write also enables another powerful feature—snapshots. ZFS allows you to instantly create a consistent point-in-time snapshot of a dataset (and optionally of all its child datasets). The new snapshot takes no additional space (aside from a miniscule amount of metadata) and is read-only. Later, when a block is changed, the older block becomes part of the snapshot, rather than being reclaimed as free space. There are now two distinct versions of the file system, the snapshot (what the file system looked like at the time the snapshot was taken) and the live file system (what it looks like now). The only additional space consumed are those blocks that have been changed; the unchanged blocks are shared between the snapshot and the live file system until they are modified. These snapshots can be mounted to recover the older versions of the files that they contain, or the live file system can be rolled back to the time of the snapshot, discarding all modifications since the snapshot was taken. Snapshots are read-only, but they can be used to create a clone of a file system. A clone is a new live file system that contains all the data from its parent while consuming no additional space until it is written to.

These features protect your data from the usual problems: crashes, power failures, accidental deletion/overwriting, etc. However, what about the cases where the problem is less obvious? Disks can suffer from silent corruption, flipped bits, bad cables, and malfunctioning controllers. To solve these problems, ZFS calculates a checksum for every block it writes and stores that along with the metadata. When a block is read, the checksum is again calculated and then compared to the stored checksum; if the two values do not match, something has gone wrong. A traditional file system would have no way of knowing there was a problem, and would happily return the corrupted data. ZFS, on the other hand, will attempt to recover the data from the various forms of redundancy supported by ZFS. When an error is encountered, ZFS increments the relevant counters displayed by the zpool status command. If redundancy is available, ZFS will attempt to correct the problem and continue normally; otherwise, it will return an error instead of corrupted data. The checksum algorithm defaults to fletcher, but the SHA256 cryptographic hashing algorithm is also available, offering a much smaller chance of a hash collision in exchange for a performance penalty.

Future-proof Storage

ZFS is designed to overcome the arbitrary limits placed on previous file systems. For example, the maximum size of a single file on an EXT3 file system is 2^31 (2 TiB), while on EXT4 the limit is 2^44 (16 TiB), compared to 2^55 (32 PiB) on UFS2, and 2^64 (16 EiB) on ZFS. EXT3 is limited to 32,000 subdirectories, with EXT4 limited to 64,000, while ZFS can contain up to 2^48 entries (files and subdirectories) in each directory. The limits in ZFS are designed to be so large that they will never be encountered, rather than just being good enough for the next few years.

Owing to the fact that ZFS is both the volume manager and the file system, it is possible to add additional storage devices to a live system and have the new space available on all the existing file systems in that pool immediately. Each top level device in a zpool is called a vdev, which can be a simple disk or a RAID transform, such as a mirror or RAID-Z array. ZFS file systems (called datasets) each have access to the combined free space of the entire pool. As blocks are allocated, the free space available to the pool (and file system) is decreased. This approach avoids the common pitfall with extensive partitioning where free space becomes fragmented across the partitions.

Doing It in Software Is Better?

Best practices dictate that ZFS be given unencumbered access to the raw disk drives, rather than a single logical volume created by a hardware RAID controller. RAID controllers will generally mask errors and attempt to solve them rather than reporting them to ZFS, leaving ZFS unaware that there is a problem. If a hardware RAID controller is used, it is recommended it be set to IT "Target" or JBOD mode, rather than providing RAID functionality. ZFS includes its own RAID functionality that is superior.

When creating a ZFS Pool (zpool) there are a number of different redundancy levels to choose from. Striping (RAID0, no redundancy), Mirroring (RAID1 or better with n-way mirrors), and RAID-Z. ZFS mirrors work very much the same as traditional RAID1 (except you can place 3 or more drives into a single mirror set for additional redundancy). However, RAID-Z has some important differences compared to the analogous traditional RAID configurations (RAID5/6/50/60). Compared to RAID5, RAID-Z offers better distribution of parity and eliminates the "RAID5 write hole" in which the data and parity information become inconsistent after an unexpected restart. When data is written to a traditional RAID5 array, the parity information is not updated atomically, meaning that the parity must be written separately after the data has been updated. If something (like a power failure) interrupts this process, then the parity data is actually incorrect, and if the drive containing the data fails, the parity will restore incorrect data. ZFS provides 3 levels of RAID-Z (Z1 through Z3) which provide increasing levels of redundancy in exchange for decreasing levels of usable storage. The number of drive failures the array can withstand corresponds to the name, so a RAID-Z2 array can withstand two drives failing concurrently.

If you create multiple vdevs, for example, two separate mirror sets, ZFS will stripe the data across the two mirrors, providing increased performance and IOPS. Creating a zpool of two or more RAID-Z2 vdevs will effectively create a RAID60 array, striping the data across the redundant vdevs.

ZFS also supports the dataset property `copies`, which controls the number of copies of each block that is stored. The default is 1, but by increasing this value, ZFS will store each block multiple times, increasing the likelihood it can be recovered in the event of a failure or data corruption.

WHEN INITIALIZING NEW **POOLS** and adding a device to an existing pool, ZFS will perform a whole-device TRIM, erasing all blocks on the device to ensure optimum starting performance.

Faster Is Always Better!

In addition to providing very effective data integrity checks, ZFS is also designed with performance in mind. The first layer of performance is provided by the Adaptive Replacement Cache (ARC), which is resident entirely in RAM. Traditional file systems use a Least Recently Used (LRU) cache, which is simply a list of items in the cache sorted by when each object was most recently used. New items are added to the top of the list, and once the cache is full, items from the bottom of the list are evicted to make room for more active objects. An ARC consists of four lists—the Most Recently Used (MRU) and Most Frequently Used (MFU) objects, plus a ghost list for each. These ghost lists track recently evicted objects to prevent them from being added back to the cache. This increases the cache hit ratio by avoiding objects that have a history of only being used occasionally. Another advantage of using both an MRU and MFU is that scanning an entire file system would normally evict all data from an MRU or LRU cache in favor of this freshly accessed content. In the case of ZFS, since there is also an MFU that only tracks the most frequently used objects, the cache of the most commonly accessed blocks remains. The ARC can detect memory pressure (when another application needs memory)
and will free some of the memory reserved for the ARC. On FreeBSD, the ARC defaults to a maximum of all RAM less 1 GB, but can be restricted using the `vfs.zfs.arc_max` loader tunable.

The ARC can optionally be augmented by a Level 2 ARC (L2ARC). This is one or more SSDs that are used as a read cache. When the ARC is full, other commonly used objects are written to the L2ARC, where they can be more quickly read back than from the main storage pool. The rate at which data is added to the cache devices is limited to prevent prematurely wearing out the SSD with too many writes. Writing to the L2ARC is limited by `vfs.zfs.l2arc_write_max`, except for during the "Turbo Warmup Phase"; until the L2ARC is full (the first block has been evicted to make room for something new), the write limit is increased by the value of vfs.zfs.l2arc_write_boost. OpenZFS also features L2ARC compression controlled by the secondarycachecompress dataset property. This increases the effective size of the L2ARC by the compression ratio, but also increases read performance as data is read as quickly as possible but then decompressed, resulting in an even higher effective read speed. L2ARC compression only uses the LZ4 algorithm because of its extremely high decompression performance.

Fine-Grained Control

A great deal of the power of ZFS comes from the fact that each dataset has a set of properties that control how it behaves, and are inherited by its children. A common best practice is to set the atime property (which tracks the last access time for each file) to "off". This prevents having to write an update to the metadata of a file each time it is accessed. Another powerful feature of ZFS is transparent compression. It can be enabled and tuned per dataset, so one can compress /usr/src and /usr/ports but disable compression for /usr/ports/distfiles. OpenZFS includes a selection of different compression algorithms including: LZJB (modest compression, modest CPU usage), GZIP1-9 (better compression, but more CPU usage, adjustable), ZLE (compresses runs of 0s, useful in specific cases), and LZ4 (added in v5000, greater compression and less CPU usage than LZJB). LZ4 is a new BSD-licensed high-performance, multi-core scalable compression algorithm. In addition to better compression in less time, it also features extremely fast decompression rates. Compared to the default LZJB compression algorithm used by ZFS, LZ4 is 50% faster when compressing compressible data and over three times faster when attempting to compress incompressible data. The performance on incompressible data is a large improvement; this comes from an "early abort" feature. If ZFS detects that the compression savings is less than 12.5%, then compression is aborted and the block is written uncompressed data, but once decompressed, provides a much higher effective throughput. In addition, decompression is approximately 80% faster; on a modern CPU, LZ4 is capable of compression at 500 MB/s and decompression at 1500 MB/s per CPU core. These numbers mean that for some workloads, compression will actually give increased performance—even with the CPU usage penalty—because data can be read from the disks at the same speed as uncompressed data, but then once decompressed, provides a much higher effective throughput. This also means it is now possible to use dataset compression on file systems that are storing databases, without a heavy latency penalty. LZ4 decompression at 1.5 GB/s on 8k blocks means the additional latency is only 5 microseconds, which is an order of magnitude faster than even the fastest SSDs currently available.

ZFS also provides very fast and accurate dataset, user and group space accounting in addition to quotas and space-reservations. This gives the administrator fine grained control over how space is allocated and allows critical file systems to reserve space to ensure other file systems do not take all of the free space.

On top of all of this, ZFS also features a full suite of delegation features. Delegating various administrative functions such as quota control, snapshotting, replication, ACL management, and control over a dataset's ZFS properties can increase security and flexibility and decrease an administrator's workload. Using these features, it is possible to take consistent backups based on snapshots without root privileges. An administrator could also choose to use a separate dataset for each user's home directory, and delegate control over snapshot creation and compression settings to that user.

Replication— Redundancy Beyond the Node

ZFS also features a powerful replication system. Using the zfs send and zfs receive commands it is possible to send a dataset (and optionally its children) to another dataset, another pool, or another system entirely. ZFS replication also supports incremental sends, sending only the blocks that have changed between a pair of snapshots. OpenZFS includes enhancements to this feature that provide an estimate of how much data will need to be sent, as well as feedback while data is being transferred. This is the basis of PCBSD's Life Preserver feature. A planned feature for the future will also allow resumption of interrupted ZFS send/receive operations.

Harnessing the Power of Solid State Drives

In addition to the L2ARC read-cache discussed earlier, ZFS supports optional log devices, also known as ZFS Intent Log (ZIL). Some workloads, especially databases, require an assurance that the data they have written to disk has actually reached "stable storage." These are called synchronous writes, because the system call does not return until the data has been safely written to the disk. This additional safety traditionally comes at the cost of performance, but with ZFS it doesn't have to. The ZIL accelerates synchronous transactions by using storage devices (such as SSDs) that are faster and have less latency compared to those used for the main pool. When data is being written and the application requests a guarantee that the data has been safely stored, the data is written to the faster ZIL storage, and then later flushed out to the regular disks, greatly reducing the latency of synchronous writes. In the event of a system crash or power loss, when the ZFS file system is mounted again, the incomplete transactions from the ZIL are replayed, ensuring all of the data is safely in place in the main storage pool. Log devices can be mirrored, but RAID-Z is not supported. When specifying multiple log devices, writes will be load balanced across all devices, further increasing performance. The ZIL is only used for synchronous writes, so will not increase the performance of (nor be busied by) asynchronous workloads.

Open ZFS project (open-zfs.org) was created with the expressed goals of raising awareness about open source ZFS, encouraging **open communication** between the various implementations and vendors, and ensuring consistent reliability, functionality, and performance among all distributions of ZFS.

OpenZFS has also gained TRIM support. Solid State Disks (SSDs) work a bit differently than traditional spinning disks. Due to the way that flash cells wear out over time, SSD's Flash Translation Layer (FTL)—which makes the SSD appear to the system like a typical spinning disk—often moves data to different physical locations in order to wear the cells evenly, and to work around worn-out cells. In order to do this effectively, the SSD's FTL needs to know when a block has been freed (the data stored on it can be overwritten). Without information as to which blocks are no longer in use, the SSD must assume that any block that has ever been written is still in use, and this leads to fragmentation and greatly diminished performance.

When initializing new pools and adding a device to an existing pool, ZFS will perform a whole-device TRIM, erasing all blocks on the device to ensure optimum starting performance. If the device is brand new or has previously been erased, setting the `vfs.zfs.vdev.trim_on_init` sysctl to 0 will skip this step. Statistics about TRIM operations are exposed by the `kstat.zfs.misc.zio_trim` sysctl. In order to avoid excessive TRIM operations and increasing wear on the SSD, ZFS queues the TRIM command when a block is freed, but waits (by default) 64 transaction groups before sending the command to the drive. If a block is reused within that time, it is removed from the TRIM list. The L2ARC also supports TRIM, but based on a time limit instead of number of transaction groups.

OpenZFS— Where Is It Going Next?

The recently founded OpenZFS project (open-zfs.org) was created with the expressed goals of raising awareness about open source ZFS, encouraging open communication between the various implementations and vendors, and ensuring consistent reliability, functionality, and performance among all distributions of ZFS. The project also has a number of ideas for future improvements to ZFS, including: resumable send/receive, ZFS channel programs to allow multiple operations to be complete atomically, device removal, unified ashift handling (for 4k sector "advanced format" drives), increase maximum record size from 128KB to 1MB (preferably in a way compatible with Oracle ZFS v32), platform agnostic encryption, and improvements to deduplication. •

Allan Jude is VP of operations at ScaleEngine Inc., a global HTTP and Video Streaming Content Distribution Network, where he makes extensive use of ZFS on FreeBSD. He is also the on-air host of the video podcasts "BSD Now" with Kris Moore, and "TechSNAP" on JupiterBroadcasting.com. Previously he taught FreeBSD and NetBSD at Mohawk College in Hamilton, Canada, and has 12 years of BSD unix sysadmin experience.