The History and Future of OpenZFS

Allan Jude, *Klara Inc.*

Abstract

The ZFS project started at Sun Microsystems in the summer of 2001. It has since grown to be one of the most robust, relied upon filesystems in use today. Over the course of its nearly 20-year history, it has continued to expand with additional features and to support additional platforms. Despite its age, ZFS remains on the leading edge of reliability and flexibility while maintaining performance. After a brief review of the history of ZFS, this paper will explore the future of today's OpenZFS project and FreeBSD's role in it.

1. History

The ZFS project started after several failed attempts to create a new filesystem at Sun. These previous attempts seemed to be brought down by too large a team in the beginning and too much or too little direction. ZFS started from scratch with Sun veteran Jeff Bonwick and a newly minted undergraduate of Brown University, Matt Ahrens, who today has become one of ZFS's most well-known luminaries. The guiding philosophy of ZFS is simple: adding additional storage should be as easy as adding additional RAM is. Just connect the additional hardware to the system, and you can use it. There is no "ramctl" command where you need to configure the additional sticks of RAM before they can be used. By combining the previously separated roles of volume manager and filesystem, ZFS simplifies administration and improves flexibility.

1.1. Initial Development

Traditional filesystems typically work on a single disk. Thus, when a filesystem larger or more redundant than a single disk is desired, a volume manager combines multiple physical disks into a single logical volume on which a filesystem can be placed. This has several disadvantages. Such a filesystem does not understand the layout of the blocks on the physical disks, thus, it cannot optimize the layout for better redundancy or performance. Most filesystems are still designed to use a contiguous range of block addresses on a storage device, so it is not possible to reclaim unused space and make it available to another filesystem. ZFS works around this issue by creating a single large *pool* of space, typically made up of many disks, from which thinly provisioned filesystems are created. As each filesystem allocates new blocks, it takes space from the pool; when those blocks are freed, they are returned to the pool. This simple concept allows many filesystems to share available blocks. The *copy-on-write* (CoW) nature of ZFS also allows the same blocks to be shared by multiple filesystems, in particular through the mechanism of ZFS snapshots and clones.

ZFS also incorporates the powerful ability to serialize a filesystem into a *replication stream*. This allows a filesystem to be stored on a different filesystem, other media (tape), or sent over the network where it may be stored as generic data, or reconstructed into a remote ZFS filesystem on another machine. The internal format of ZFS, where each block has a birth time, also means it is computationally inexpensive and does not require excessive amounts of IO to determine which blocks have changed between two snapshots. The resulting ZFS *incremental replication* thus has far lower overhead than similar methods, such as rsync.

In a blog post, Matt Ahrens said[1]: "*ZFS send and receive wasn't considered until late in the development cycle. The idea came to me in 2005. ZFS was nearing integration, and I was spending a few months working in Sun's new office in Beijing, China. The network link between Beijing and Menlo Park was low-bandwidth and high-latency, and our NFS-based source code manager was painful to use. I needed a way to quickly ship incremental changes to a workspace across the Pacific. A POSIX-based utility (like rsync) would at best have to traverse all the files and directories to find the few that were modified since a specific date, and at worst it would compare the files on each side, incurring many high-latency round trips. I realized that the block pointers in ZFS already have all the information we need: the birth time allows us to quickly and precisely find the blocks that are changed since a given snapshot. It was easiest to implement ZFS send at the DMU layer, just below the ZPL. This allows the semantically-important changes to be transferred exactly, without any special code to handle features like NFSv4 style ACLs, case-insensitivity, and extended attributes. Storage-specific settings, like compression and RAID type, can be different on the sending and receiving sides. What began as a workaround for a crappy network link has become one of the pillars of ZFS, and the foundation of several remote replication products.*"

1.2. OpenSolaris

In 2005, Sun released ZFS as open source software under the CDDL as part of OpenSolaris. This made ZFS available to everyone and stimulated great interest in ZFS among a number of other open source projects. Open source updates

[1] https://www.delphix.com/blog/delphix-engineering/zfs-10-year-anniversary

to ZFS continued to be released until 2010 when Oracle bought Sun and discontinued the open source development of ZFS. This meant any new features developed by Oracle would be proprietary.

1.3. FreeBSD

A port of ZFS to FreeBSD was started by Pawel Dawidek sometime in 2006 with elements first committed to FreeBSD's head branch in April of 2007[2]. Full ZFS support (including booting from ZFS) was included in the next major release, FreeBSD 7.0, in February 2008[3]. Over time, additional updates were continuously pulled in from OpenSolaris, upgrading support from zpool version 6, to 13, 14, 15, and finally 28. After ZFS development was no longer open source, FreeBSD switched upstreams to illumos (see 1.5), an open source fork of the last version of OpenSolaris available under the CDDL.

1.4. Linux

Due to the perceived incompatibility between the CDDL and GPL, the first attempt to port ZFS to Linux used the FUSE system to run the code in userspace. While this approach was generally functional, userspace implementations of filesystems suffer from well-known performance and stability issues.

A native port of ZFS to Linux was started by developers at Lawrence Livermore National Laboratory in 2008. This port was specifically to act as a backend for the Lustre distributed filesystem. Because of this focus, the early versions of this port only supported ZVOLs for Lustre and did not implement the POSIX filesystem layer, which was added in 2011. With its arrival on Github, the project attracted a community of early adopters and by 2012 had its first pools in production. Now known as ZFS-on-Linux (ZoL), the project delivered its first general availability release in 2013 with version 0.6.1. Today, the latest stable branch (0.8.x), which came out in mid-2019, serves as the de facto ZFS reference implementation for countless Linux operating systems.

1.5. Oracle Solaris and Illumos

Oracle continues to release new versions of Solaris, including Oracle Solaris 11.4 SRU 12 in August of 2019 which offered zpool version 46, under a now proprietary umbrella. Worse, a previous release (version 29) fundamentally broke compatibility with previous ZFS on-disk and replication stream formats. This prevents replication from newer versions of Oracle ZFS to OpenZFS pools, despite OpenZFS maintaining full backwards compatibility.

When Oracle announced the cessation of development for OpenSolaris, the last open source release was forked to create the illumos project, which serves as the base for a number of Solaris-like open source operating systems today. FreeBSD and ZFS-on-Linux switched to the illumos repository as an upstream at approximately this time.[4]

1.6. OS X

There were a number of ports of ZFS to Mac OS X, including an internal one at Apple that was never released, an open source MacZFS that supported zpool version 8, and ZEVO that was freeware that supporting up to version 28.

After the end of the MacZFS project, work was restarted as OpenZFS on OS X (known as O3X), based on the latest version of ZFS-on-Linux.

When asked about the history, lead developer Jörgen Lundman said: "*I started on OSX back in Feb 2013, because the ZEVO version didn't have lz4 compression or the support for zvols that I wanted, so figured I'd help with that. But, ZEVO is closed-source. I did look at macZFS initially, but that port has all the changes in one big file, and I liked the separation of the SPL (Solaris Porting Layer) more, so to learn I figured I would pick up more understanding by doing it again. I started with the ZFS-on-Linux repo for its separation of SPL, ZFS and OS, as well as the autoconf work (with illumos and FreeBSD, you build the whole kernel, so I would have to figure out the cut myself). It Took quite a while to reach that first "import", around May or so, but it feels longer, since it was working with three unknowns.*"

By 2015 O3X was considered stable and supported all features that were supported on Linux. Today, this implementation is generally kept up-to-date, and is the standard ZFS implementation for Mac OS X.

1.7. Windows

In early 2017, Jörgen Lundman branched out from OS X and started a native port for Windows.

"*As the number of supported platforms grew, we always joked about 'doing the Windows' port next, as I'm sure everyone did. But over time, after a few years, it seemed less and less insane. To me, since ext3, ufs, and others have Windows ports, which use the same vnops calls as zfs, I think everyone knew it 'could' be done, it was more a question of effort and ability. Amusingly, I started looking into it in earnest, also in Feb (2017). As in, downloading a Windows VM and tried to compile/deploy a "hello world" kernel module. That alone took 3 weeks, and I was close to quitting then. After some persistence, the first "import -N" was in April 2017, so a little faster than with OS X, as there was only one "unknown" this time around. I was chatting with the guys in the OS X IRC channel about the work and progress, but in general, I kept it a secret for the presentation at the 2017 OpenZFS Developers Summit. Matt Ahrens found out when I sent in my talk proposal, but otherwise I think we managed to keep it a surprise.*"

The Windows port, which is a native implementation and does not reply upon WSL (Windows Subsystem for Linux), now supports most ZFS features. That being said, the

[2] https://lists.freebsd.org/pipermail/freebsd-current/2007-April/070544.html

[3] https://www.freebsd.org/releases/7.0R/announce.html

[4] https://www.youtube.com/watch?v=qrC63B_bGRI

Windows implementation of ZFS remains mostly a curiosity at this time.

1.8. Others

ZFS has been ported to other platforms, though most of these are relatively unknown or incomplete.

2. OpenZFS

In the wake of the end of OpenSolaris, partly driven by the desire to see ZFS development continue without diverging on different operating systems, Matt Ahrens announced the start of the OpenZFS project in September 2013[5]. The OpenZFS community's goals are:

- To raise awareness of the quality, utility, and availability of OpenZFS by consolidating documentation for developers and sysadmins alike, and by engaging with the larger tech community through conferences, meetups, and online interactions.
- To encourage open communication about ongoing efforts to improve OpenZFS, by creating a collaborative website and mailing list to discuss OpenZFS code changes.
- To ensure consistent reliability, functionality, and performance of all distributions of OpenZFS by making it easier to share code between platforms, by creating cross-platform tests, and by encouraging cross-platform code reviews.

OpenZFS's fundamental strategy allows focus to be on unified concepts and codebases across relevant platforms, allowing user experience--and developer expertise--free passage among them. If you know how to use ZFS on FreeBSD, those skills should translate to OS X, Linux, or even Windows.

To avoid confusion between version numbers of proprietary ZFS and OpenZFS, and to better support the fact that OpenZFS development was now happening on a number of separate operating systems and distributions, the zpool version was arbitrarily set to 5000 and the concept of *feature flags* was introduced to track which newly-introduced ZFS features were required on a particular pool. This allows features to be developed and added to platforms as they become ready, rather than in a particular sequence. Some features are *read-only compatible*, meaning that you can import the pool, but not write to it, if your system does not support the new feature. Many features are only activated when used. For example when support for the SHA512 and Skein hashing algorithms was added, a pool did not become incompatible until you started using one of the new hashes on a dataset. New features, once activated in a pool (thus affecting compatibility with earlier versions of ZFS), are often reversible, in that when any new datasets using these features are destroyed, the general compatibility of the ZFS pool is restored.

One of the original goals of the OpenZFS project was to move the common (OS agnostic) code to a unified repository, so that it could be compiled and tested in userspace. This would require each distribution to maintain their own "glue" to integrate the common ZFS code into their OS. The main issue with this approach was funding the additional work of keeping the common code up to date, and integrating and testing patches from the various distributions. There was little commercial value in maintaining the common code by itself, so the idea never gained traction.

With a lack of volunteers to maintain the "one true repo", the OpenZFS repo was created as a fork of the illumos repo on GitHub. This allowed developers from the other platforms to open Pull Requests to integrate patches without needing to follow the more involved illumos RTI (Request to Integrate) process. It also provided a central place to discuss and review changes, and to post issues.

2.1. OpenZFS Developers Summit

In November 2013, the first annual OpenZFS developers summit was hosted in San Francisco with 30 developers in attendance. By 2016, the summit had grown to over 100 developers and added a second day hackathon where new features and enhancements are prototyped. Having so many ZFS developers in one place leads to many productive conversations and brainstorming that would be impossible remotely. These conferences provide significant value by keeping the developer community up to date with changes that are happening across platforms, and allowing hyper-focused and in-person collaboration.

2.2. OpenZFS Users Conference

Datto, one of the companies that makes heavy use of ZFS-on-Linux and developed its native encryption feature, hosted the ZFS Users Conference in 2017 and 2018, although attendance was less than 30 people, possibly due to the venue being hard to reach (Norwalk, CT, USA). Low registration resulted in the cancellation of the 2019 conference. Despite these setbacks, the usefulness of a user conference is obvious: ZFS is often one of the most well attended birds-of-a-feather (BoF) sessions at conferences such as BSDCan, vBSDCon, and MeetBSD. There have also been ZFS BoFs at USENIX Lisa, Linux Fest North West, and many other conferences. A key to success may be attaching to an existing event that already draws a large number of ZFS users. The FreeBSD project has had success by attaching to conferences like USENIX FAST, FOSDEM, and Linux.conf.au as a "miniconf" either preceding or as a breakout session of the main conference.

2.3. OpenZFS Leadership Meeting

[5] http://open-zfs.org/wiki/Announcement

There are important advantages in collecting a large number of OpenZFS developers at a yearly meeting. At the 2018 summit, it was decided that having a 30 minute monthly call to follow up on the progress of decisions made at the annual conference and to further coordinate development and feature parity across all of the platforms would be extremely useful. After the first few meetings, it was decided to expand to a full hour. The meetings are open to the public, streamed live, and the recordings are posted[6] to YouTube for all to see. Topics include the latest developments and bug reports, questions and assistance with porting features to different platforms, but also promotes earlier design review and considerations, establishing policies (deprecation of ZFS features, deprecation of supported platforms), and ensuring better compatibility across platforms.

The leadership meetings have resulted in more discussions about improving the naming conventions for tunables, and projects such as implementing a system to be used during zpool creation to allow the user to specify a pool compatible with all OpenZFS implementations as of a specific date. There has also been an effort to make the NFS properties either translate to each platform's native implementation, or to split the property per platform, so there are, for example, no errors when importing a pool with Linux NFS settings on FreeBSD.

It wasn't until 18 years into the existence of ZFS that the question of a deprecation policy was addressed. How much notice needs to be given to remove a feature? How do we make sure the removal aligns with long-term support releases of different platforms?

The first feature being considered for deprecation is the ZFS *send deduplication* feature. This feature often causes confusion among users who assume it is somehow related to the regular pool-wide online deduplication feature. It applies deduplication to the ZFS replication stream to avoid sending duplicate blocks over the network. The dedup table is only created in memory on the sending system, and is not transmitted to the receiving system. Any block that appears more than once in the stream is sent as a back-reference to its first occurrence. On the receiving system, blocks are written normally. Deduplication is only applied if it is enabled on the receiving side, where it works the same whether the replication stream used the dedup feature or not.

However, OpenZFS strives to maintain full backward compatibility of the replication stream format, so that older streams can always be received in the future, and so that the replication feature can still be used to send data to an older pool. Accordingly, it was decided that a standalone tool should be created to "rehydrate" deduplicated streams. That is, read in a deduplicated stream, and write out a version of the stream without the deduplication feature, replacing each back-reference with the full object. This would allow old send streams to still be received even after support for the deduplication feature has been removed from OpenZFS.

The second feature slated to be removed is *dedupditto*. The idea behind this feature is that if a single block has been deduplicated many times, there is a risk of leverage. ZFS should store an additional copy of it to avoid the chance that the only copy will be corrupted, impacting numerous references to that block. However, it was discovered in 2019 that, for a number of years, these additional copies of the block (*ditto blocks*) were neither being checked nor repaired as part of scrub or resilver. If they were corrupted, that would still be detected by the checksum, but this oversight drastically reduces the efficacy of this feature.

Lastly, the question of platform deprecation has recently been raised by the ZFS-on-Linux project. As of January 2020, support for Redhat Enterprise Linux / CentOS 6 has been removed from the master branch. Support is retained in the 0.8.x branch, but older Linux kernels will not be supported in the next release.

3. The Rise of Linux

Over the course of 2016 and 2017, it became clear that the proportion of development of ZFS was rapidly growing on ZFS-on-Linux (ZoL). At the same time, illumos was starting to suffer from a lack of development in areas that had less direct commercial backing. In particular, areas such as drivers and infrastructure were suffering in the illumos ecosystem.

In early 2018, Delphix, a database virtualization appliance vendor (and one of the largest contributors to OpenZFS) announced that they would be changing their platform from illumos to Linux. They had previously considered switching to FreeBSD to get support for Microsoft Azure, but managed to port platform support from FreeBSD to illumos to forestall that requirement. This enabled Delphix, ultimately, to choose Linux.

"*At the core of the Delphix virtualization product is an application with tight ties to the OpenZFS filesystem. Ten years ago, we chose the Illumos operating system as the delivery vehicle for that combination, owing to the fact that it was a great OS with strong ZFS support, and our employees had deep familiarity with it. Illumos took us from VMware to AWS and Azure, but each new platform comes with a set of challenges—hypervisor and instance type support, extended security reviews, expanded QA, and vendor support to name a few. OpenZFS has grown over the last decade, and delivering our application on Linux provides great OpenZFS support while enabling higher velocity adoption of new environments.*"[7]

[6] https://www.youtube.com/playlist?list=PLaUVvul17xSeHoMLiE_cp68mPvowtYteN

[7] https://www.delphix.com/blog/kickoff-future-eko-2018

By the fall of 2018 it was obvious that most new development for ZFS was happening in the ZoL repo, rather than the illumos repo. This presented a problem as FreeBSD was using illumos as its upstream. Changes were taking longer and longer to be ported from the ZoL repo and land in illumos, before they could be imported into FreeBSD.

In an attempt to accelerate this process, developers from iXsystems attempted to port the ZFS Native Encryption feature from ZoL directly to FreeBSD. This proved very difficult, as the two repositories had long since diverged.

This led to a new project, ZFS-on-FreeBSD (ZoF), aiming to port the ZFS-on-Linux repo to FreeBSD. Keeping parts of the existing OpenSolaris compatibility code for FreeBSD in place of the Linux SPL (Solaris Porting Layer) shortened the porting time. Further, after coordination with the leadership of the ZoL project, it was decided to move any Linux-specific implementation code into an os/linux directory, and allow the FreeBSD-specific implementations to be added to the ZoL repo under os/freebsd. This ongoing work will soon result in a unified repository that can run on either Linux or FreeBSD. Once this work is complete, the CI run against each proposed patch will need to pass the ZFS test suite on both operating systems before it will be merged. The O3X project plans to adopt this same system and ultimately join this single repository.

At the end of 2019 it was announced that the former OpenZFS repository (the fork of illumos) will be archived, and the current ZoL repo, which will soon contain the compatibility code for FreeBSD, will be moved to the OpenZFS organization on GitHub. The first release of OpenZFS with consolidated support for both Linux and FreeBSD will be called OpenZFS 2.0, with a goal to release a new major version once per calendar year. This will make it easier to compare and coordinate the feature sets of OpenZFS across supported platforms[8].

4. Recent Features

This section provides a list of features not yet in FreeBSD's in-kernel version of OpenZFS. Most (if not all) of these features are expected to be imported in time for FreeBSD version 13.0 as part of the ZFS-on-FreeBSD project.

3.1. ZoL 0.7

- **Latency and Request Size Histograms** - Use the `zpool iostat -l` option to show on-the-fly latency stats and `zpool iostat -w` to generate a histogram showing the total latency of each IO. The `zpool iostat -r` option can be used to show the size of each IO. These statistics are available per-disk to aid in finding misbehaving devices.
- **Large Dnodes** - This feature improves metadata performance allowing extended attributes, ACLs, and symbolic links with long target names to be stored in the dnode. This benefits workloads such as SELinux, distributed filesystems like Lustre and Ceph, and any application which makes use of extended attributes.
- **Multiple Import Protection** - Prevents a shared pool in a failover configuration from being imported on different hosts at the same time. When the *multihost* pool property is **on**, perform an activity check prior to importing the pool to verify it is not in use.
- **User/group object accounting and quota** - This feature adds per-object user/group accounting and quota limits to existing space accounting and quota functionality. The `zfs userspace` and `zfs groupspace` subcommands have been extended to set quota limits and report on object usage.
- **Vectorized RAIDZ** - Hardware optimized RAIDZ which reduces CPU usage. Supported SIMD instructions: sse2, ssse3, avx2, avx512f, and avx512bw, neon, neonx2
- **Vectorized Checksums** - Hardware-optimized Fletcher-4 checksums which reduce CPU usage. Supported SIMD instructions: sse2, ssse3, avx2, avx512f, neon

3.2. ZoL 0.8

- **Native Encryption** - The *encryption* property enables the creation of encrypted filesystems and volumes. The aes-256-ccm algorithm is used by default. Per-dataset keys are managed with `zfs load-key` and associated subcommands.
- **Raw Encrypted 'zfs send/receive'** - The `zfs send -w` option allows an encrypted dataset to be sent and received to another ZFS pool without decryption. The received dataset is protected by the original user key from the sending side. This allows datasets to be efficiently backed up (and even take advantage of incremental replication) to an untrusted system without requiring keys or decryption at rest or in motion.
- **Pool TRIM** - The `zpool trim` subcommand provides a way to notify underlying devices which sectors are no longer allocated. This allows an SSD to more efficiently manage itself and helps to maintain performance. Continuous background trimming can be enabled via the new `autotrim` pool property.
- **Project Accounting and Quotas** - This features adds project based usage accounting and quota enforcement to the existing space accounting and quota functionality. Project quotas add an additional dimension to traditional

[8] https://openzfs.topicbox.com/groups/developer/T6aa3c033248cef9c/zol-repo-move-to-openzfs

user/group quotas. The `zfs project` and `zfs projectspace` subcommands have been added to manage projects, set quota limits, and report on usage.

- **Allocation Classes** - Allows a pool to include a small number of high-performance SSD devices dedicated to storing specific types of frequently accessed blocks (e.g. metadata, DDT data, or small file blocks). A pool can opt-in to this feature by incorporating a *special* or *dedup* top-level device.
- **Parallel Allocation** - The allocation process has been parallelized by creating multiple "allocators" per-metaslab group. This results in improved allocation performance on high-end systems.
- **Deferred Resilvers** - This feature allows new resilvers to be postponed if an existing one is already in progress. By waiting for the running resilver to complete, redundancy is restored as quickly as possible.
- **ZFS Intent Log (ZIL)** - New log blocks are created and issued while there are still outstanding blocks being serviced by the storage, effectively reducing the overall latency observed by the application.

5. OpenZFS 2.0

5.1. Why Jump to 2.0?

It was decided by OpenZFS leadership that a unified repository supporting multiple operating systems constituted a major milestone in ZFS development. It was also decided that having regular annual major releases would make discussions about OpenZFS easier, especially in the case of long-term support distributions that might still be shipping OpenZFS from 2019 in 2021. It also makes it easier to create pools that are compatible across distros by targeting a common OpenZFS version. In fact, this new OpenZFS repository will go even further, as the platform-specific code will also reside in the common repository, separated into platform-specific directories. This improves the utility of the common repo, as it allows full platform integration testing of proposed patches, rather than just ZFS userspace tests. This will ensure that proposed new features work on all platforms supported by the common repo before being officially integrated. Of course, this will require platform maintainers to help review patches and implement missing platform-specific functionality or face the prospect of their platform losing the ability to block feature additions.

6. FreeBSD Features

There are also ZFS features that only exist in the FreeBSD implementation. These should be upstreamed to OpenZFS to make them more widely available and to reduce FreeBSD's maintenance burden.

- **Throttle Scan I/O** – When other IOs are in progress, reduce the rate that scrub/resilver IOs are issued to avoid impacting workload performance during a scrub/resilver.
- **Vdev Ashift Optimization** – Deals with situations where devices with different sectors sizes are mixed together, such as when replacement disks only support 4k sectors.
- **zfs rename -u** – FreeBSD has the ability to rename a dataset without unmounting it. This is extremely useful when a dataset is in use and cannot be easily unmounted.
- **NFSv4 ACL support** – ZFS uses NFSv4 ACLs, which FreeBSD and illumos implementations support, but Linux does not.
- **ACLMODE** – FreeBSD's version of ZFS has an additional pool property, *aclmode*, that controls how ACLs are impacted by chmod(2).

As well, there are some FreeBSD specific interactions with GEOM and the virtual memory subsystem (*ARC backpressure*). As of the end of 2019, some additional work still needs to be done to ensure nothing is lost in the switch from the current in-kernel ZFS to OpenZFS 2.0.

7. Features Expected in 2020

7.1. Async CoW[9]

This feature is designed to avoid copy-on-write faults, which occur primarily when overwriting ZFS data blocks in less-than-*recordsize* chunks. Additionally, when faults are necessary, they are resolved asynchronously, significantly reducing their impact on throughput and latency. Applications that fill ZFS data blocks in such chunks now enjoy performance approaching what they would if they were writing out in larger chunks.

7.2. Enable Compression by Default

LZ4 compression is extremely fast and performant. There is little downside to having it enabled, even for entirely incompressible data. Its speed means that it often reduces latency over writing uncompressed data, since there is less data to write. Making it the default will generally make efficacious use of storage without degrading the user's experience.

7.3. Improved AES-GCM Performance

Improvements to the SIMD implementation of AES-GCM have been made to perform encryption in larger blocks and

[9] https://www.bsdcan.org/2012/schedule/events/316.en.html

consequently avoid paying the overhead of save/restoring the FPU as frequently.

7.4. ZSTD Compression

While LZ4 is extremely fast, it generally provides only modest compression ratios. ZSTD is a newer compression algorithm from the same author that aims to provide gzip (or better) levels of compression while retaining LZ4-like speed. ZSTD further offers an extensive variety of compression levels, allowing it to range from faster than LZ4 to reaching almost LZMA (XZ) compression ratios. Importantly, ZSTD decompression maintains high performance (i.e., it is for all intents and purposes transparent) independent of the compression level.

7.5. Per Vdev Properties

The primary administrative policy interfaces for ZFS have traditionally been pool and dataset properties. These allow an administrator to control compression, checksumming, record size, sharing, and more, at the granularity of datasets. This new work increases the granularity of this interface to allow controlling the properties of individual vdevs below the datasets. With features like *allocation control* and *mirror read bias* (prefer to read from a specific member of a mirror), and read-only statistical properties, it becomes possible to more finely control and monitor the performance of a pool.

7.5. Persistent L2ARC[10]

Currently, each time a system is rebooted the contents of the *L2ARC* are discarded. The L2ARC works by storing headers in the ARC that point to the L2ARC device. When the system is rebooted, these headers are lost, and there is no way to recreate them, despite the fact of L2ARC data itself persisting. This new feature adds a lightweight metadata structure to the L2ARC that can be read at boot, enabling the recreation of ARC headers. This approach means that the cache persists across a reboot and also allows the L2ARC entries to be reloaded asynchronously after the pool is imported, rather than blocking the import processing and increasing booting time.

7.6. Redacted Send and Receive[11]

Redacted send/receive allows users to send subsets of data to a target system. One possible use case for this feature is to not transmit sensitive information to a data warehousing, test/dev, or analytics environment. Another is to save space by not replicating unimportant data within a given dataset.

Redaction is accomplished by cloning a snapshot, removing or overwriting the data to be redacted, and then creating a second snapshot. That snapshot is then processed by the `zfs redact` command to create the redaction bookmark, which is finally passed as a parameter to `zfs send`. When sending to the redacted snapshots, the redaction bookmark is used to filter out blocks that contain sensitive or unwanted information, and thus those blocks are not included in the send stream.

7.7. Log Space Maps[12]

This new feature changes the way space is allocated and freed in ZFS. Rather than updating every modified spacemap on each vdev during each transaction, a single *log spacemap* is created for each transaction group. A small number of spacemaps are then updated with each transaction group. This makes transactions take significantly less time to flush and reduces the number of overhead IOPS for internal ZFS space accounting.

7.8. Fast Clone Deletion[13]

Deleting a cloned filesystem or volume requires finding all blocks that belong only to the clone and are not shared with the origin snapshot. This is done by traversing the entire block tree which can take a long time and can be very inefficient for clones that have not deviated very much from the origin. This feature adds an option during clone creation to track the blocks with a *livelist*. All blocks that are modified or created by the clone are kept on this list, allowing the clone to be removed quickly by simply traversing and freeing the blocks on the livelist.

7.9. Zpool Wait[14]

This new features introduces a command that does not return until the indicated operation is complete. This allows an administrator to trigger other actions only after a scrub, resilver, device initialize, device removal, device replacement, or background freeing, is complete. It also supports waiting until a *zpool checkpoint* is discarded. Future work will also allow waiting until manually triggered TRIM operations have completed.

7.10. Reduced Metaslab Memory Usage[15]

This new feature replaces AVL trees used to implement the range tree structure used to track allocated space with a btree like structure. The new structure is up to two-thirds more space-efficient, requiring less memory for each loaded metaslab. The changes also result in less CPU utilization and also improves metaslab loading logic.

[10] https://github.com/zfsonlinux/zfs/pull/9582

[11] https://github.com/zfsonlinux/zfs/commit/30af21b02569ac192f52ce6e6511015f8a8d5729

[12] https://github.com/zfsonlinux/zfs/commit/93e28d661e1d704a9cada86ef2bc4763a6ef3be7

[13] https://github.com/zfsonlinux/zfs/commit/37f03da8ba6e1ab074b503e1dd63bfa7199d0537

[14] https://github.com/zfsonlinux/zfs/commit/e60e158eff920825311c1e18b3631876eaaacb54

[15] https://github.com/zfsonlinux/zfs/commit/ca5777793ee10b9f7bb57aef00a6c8d57969625e

8. Future Features

8.1. DRAID[16]

Declustered parity RAID (DRAID) is designed to reduce the "risk window" when devices fail. In traditional RAID-Z, a pool is often made up of separate RAID-Z vdevs, particularly in large pools. If a disk fails in a given vdev, only the drives comprising that vdev would be actively involved in resilvering the data; the remaining drives in the vdev would contribute reads, while writes would be bottlenecked to the single replacement disk. With DRAID, all of the drives are combined into a single vdev, but the layout is shuffled so that data and parity are distributed. Additionally, spares for the vdev are not separate physical devices; rather, one or more spares are created by combining segments of space from each physical disk. As a result, when a disk fails, parity is read from every remaining physical disk, and the writes to the designated virtual spare are efficiently distributed across every remaining physical disk. In systems with large numbers of disks, this reduces greatly the time the pool is vulnerable to concurrent failures, safely allowing a lower overall parity ratio. While the act of "resilvering" the actual faulted disk may be expected to take longer, DRAID significantly reduces the "risk window" where an entire pool would be at risk.

8.2. RAID-Z Expansion[17]

This much-anticipated feature will allow an existing RAID-Z vdev to be widened by a single disk. This is accomplished by "reflowing" the existing blocks, similar to widening a column in a word processor. As all of the data on the pool is rewritten to the new layout, the additional free space ends up as a contiguous chunk at the end of the vdev, avoiding any fragmentation.

8.3. Temporal Dedup

In a ZFS leadership meeting in 2019, Josh Paetzel of Panzura committed to open sourcing their alternative implementation of ZFS *deduplication*. This implementation groups together blocks that deduplicated around the same time. Current implementations of ZFS order blocks according to an effectively random hashing function, failing to take advantage of events which affect likely-related groups of deduplicated blocks.

8.4. Adaptive Compression

With the implementation of ZSTD (see section 7.4), it will be possible to extend ZFS to adaptively adjust compression levels used on a dataset based on the amount of *dirty data*. This technique better aligns compression with available system resources on-the-fly for improved performance.

9. Conclusion

9.1. What FreeBSD Gains

For FreeBSD, the switch to the OpenZFS 2.0 upstream will bring a lot of new features and bug fixes and, importantly, restore FreeBSD's position as a leader in ZFS development. It is important that FreeBSD have timely access to new features in ZFS; always having the latest features has been a key driver of the adoption of ZFS on FreeBSD. In addition, switching upstreams will result in more active development and testing of ZFS on FreeBSD, to say nothing of increased visibility of FreeBSD itself.

9.2. The Cost of Switching

The project to merge FreeBSD support into ZFS-on-Linux has been ongoing for over a year, with a number of developers putting significant time and effort into the project. This effort is what has led directly to the advent of OpenZFS 2.0. During the initial import of OpenZFS 2.0 as a replacement for illumos ZFS, there will be some disruption. A number of integrations will need adjustments to properly fit into the new model. The goal is that the upgrade to 13.0-RELEASE (when it comes out) will be smooth, but those using ZFS on -CURRENT builds may be exposed to a few visible issues during the transition.

9.3. What FreeBSD is Committing To

In order for FreeBSD to remain an enduring platform for OpenZFS, members of the FreeBSD community will need to fully engage in this upstream project. This means providing timely code reviews and assisting with the integration of new features, particularly any platform-specific code. FreeBSD's continued status as a "must-be-working-on" platform depends upon this vigorous engagement. Should the FreeBSD community not uphold these expectations, new features will no longer have to pass continuous integration on FreeBSD to be merged into OpenZFS, which would have a deleterious effect on FreeBSD and its future. FreeBSD must be a partner that actively moves OpenZFS forward, not one that holds it back.

[16] https://github.com/zfsonlinux/zfs/wiki/dRAID-HOWTO

[17] https://github.com/zfsonlinux/zfs/pull/8853