OpenZFS: a Community of Open Source ZFS Developers

Matthew Ahrens
Delphix
San Francisco, CA
mahrens@delphix.com

***Abstract*—OpenZFS is a collaboration among open source ZFS developers on the FreeBSD, illumos, Linux, and Mac OSX platforms. OpenZFS helps these developers work together to create a consistent, reliable, performant implementation of ZFS. Several new features and performance enhancements have been developed for OpenZFS and are available in all open-source ZFS distributions.**

I. Introduction

In the past decade, ZFS has grown from a project managed by a single organization to a distributed, collaborative effort among many communities and companies. This paper will explain the motivation behind creating the OpenZFS project, the problems we aim to address with it, and the projects undertaken to accomplish these goals.

II. History of ZFS Development

A. Early Days

In the early 2000's, the state of the art in filesystems was not pretty. There was no defense from silent data corruption introduced by bit rot, disk and controller firmware bugs, flaky cables, etc. It was akin to running a server without ECC memory. Storage was difficult to manage, requiring different tools to manage files, blocks, labels, NFS, mountpoints, SMB shares, etc. It wasn't portable between different operating systems (e.g. Linux vs Solaris) or processor architectures (e.g. x86 vs SPARC vs ARM). Storage systems were slow and unscalable. They limited the number of files per filesystem, the size of volumes, etc. When available at all, snapshots were limited in number and performance. Backups were slow, and remote-replication software was extremely specialized and difficult to use. Filesystem performance was hampered by coarse-grained locking, fixed block sizes, naive prefetch, and ever-increasing fsck times. These scalability issues were mitigated only by increasingly complex administrative procedures.

There were incremental and isolated improvements to these problems in various systems. Copy-on-write filesystems (e.g. NetApp's WAFL) eliminated the need for fsck. High-end storage systems (e.g. EMC) used special disks with 520-byte sectors to store checksums of data. Extent-based filesystems (e.g. NTFS, XFS) worked better than fixed-block-size systems when used for non-homogenous workloads. But there was no serious attempt to tackle all of the above issues in a general-purpose filesystem.

In 2001, Matt Ahrens and Jeff Bonwick started the ZFS project at Sun Microsystems with one main goal: to end the suffering of system administrators who were struggling to manage complex and fallible storage systems. To do so, we needed to re-evaluate obsolete assumptions and design an integrated storage system from scratch. Over the next 4 years, they and a team of a dozen engineers implemented the fundamentals of ZFS, including pooled storage, copy-on-write, RAID-Z, snapshots, and send/receive. A simple administrative model based on hierarchical property inheritance made it easy for system administrators to express their intent, and made high-end storage features like checksums, snapshots, RAID, and transparent compression accessible to non-experts.

B. Open Source

As part of the OpenSolaris project, in 2005 Sun released the ZFS source code as open source software, under the CDDL license. This enabled the ports of ZFS to FreeBSD, Linux, and Mac OSX, and helped create a thriving community of ZFS

users. Sun continued to enhance ZFS, bringing it to enterprise quality and making it part of the Solaris operating system and the foundation of the Sun Storage 7000 series (later renamed the Oracle ZFS Storage Appliance). The other platforms continually pulled changes from OpenSolaris, benefiting from Sun's continuing investment in ZFS. Other companies started creating storage products based OpenSolaris and FreeBSD, making open-source ZFS an integral part of their products.

However, the vast majority of ZFS development happened behind closed doors at Sun. At this time, very few core enhancements were made to ZFS by non-Sun contributors. Thus although ZFS was Open Source and multi-platform, it did not have an open development model. As long as Sun continued maintaining and enhancing ZFS, this was not necessarily an impediment to the continued success of products and community projects based on open-source ZFS – they could keep getting enhancements and bug fixes from Sun.

C. Turmoil

In 2010, Oracle acquired Sun Microsystems, stopped contributing source code changes to ZFS, and began dismantling the OpenSolaris community. This raised big concerns about the future of open-source ZFS – without its primary contributor, would it stagnate? Would companies creating products based on ZFS flounder without Sun's engineering resources behind them? To address this issue for both ZFS and OpenSolaris as a whole, the Illumos project was created. Illumos took the source code from OpenSolaris (including ZFS) and formed a new community around it. Where OpenSolaris development was controlled by one company, illumos creates common ground for many companies to contribute on equal footing. ZFS found a new home in Illumos, with several companies basing their products on it and contributing code changes. FreeBSD and Linux treated Illumos as their upstream for ZFS code. However, there was otherwise not much interaction between platform-specific communities. There continued to be duplicated efforts between platforms, and surprises when code changes made on one platform were not easily ported to others. As the pace of ZFS development on FreeBSD and Linux increased, fragmentation between the platforms became a real risk.

III. The OpenZFS Collaboration

A. Goals

The OpenZFS project was created to accomplish three goals:

1) Open communication: We want everyone working on ZFS to work together, regardless of what platform they are working on. By working together, we can reduce duplicated effort and identify common goals.

2) Consistent experience: We want users' experience with OpenZFS to be high-quality regardless of what platform they are using. Features should be available on all platforms, and all implementations of ZFS should have good performance and be free of bugs.

3) Public awareness: We want to make sure that people know that open-source ZFS is available on many platforms (e.g. illumos, FreeBSD, Linux, OSX), that it is widely used in some of the most demanding production environments, and that it continues to be enhanced.

B. Activities

We have undertaken several activities to accomplish these goals:

1) Website: The http://open-zfs.org website (don't forget the dash!) publicizes OpenZFS activities such as events, talks, and publications. It acts as the authoritative reference for technical work, documenting both usage and how ZFS is implemented (e.g. the on-disk format). The website is also used as a brainstorming and coordination area for work in progress. To facilitate collaboration, the website is a Wiki which can be edited by any registered user.

2) Mailing list: The OpenZFS developer mailing list[1] serves as common ground for developers working on all platforms to discuss work in progress, review code changes, and share knowledge of how ZFS is implemented. Before its existence, changes made on one platform often came as a surprise to developers on other platforms, and sometimes introduced platform compatibility issues or required new functions to be implemented in the Solaris Porting Layer. The OpenZFS mailing list

allows these concerns to be raised and addressed during code review, when they can easily be addressed. Note that this mailing list is not a replacement for platform-specific mailing lists, which continue to serve their role primarily for end users and system administrators to discuss how to use ZFS, as well for developers to discuss platform-specific code changes.

3) Office hours: Experts in the OpenZFS community hold online office hours[2] approximately once a month. These question and answer sessions are hosted by a rotating cast of OpenZFS developers, using live audio/video/text conferencing tools. The recorded video is also available online.

4) Conferences: Since September 2013, 6 OpenZFS developers have presented at 8 conferences. These events serve both to increase awareness of OpenZFS, and also to network with other developers, coordinating work in person. Additionally, we held the first OpenZFS Developer Summit[3] in November 2013 in San Francisco. More than 30 individuals participated, representing 14 companies and all the major platforms. The two-day event consisted of a dozen presentations and a hackathon. Ten projects were started at the hackathon, including the "best in show": a team of 5 who ported the TestRunner test suite from illumos to Linux and FreeBSD. Slides from the talks and video recordings are available on the open-zfs.org website[3].

C. New features

In this section we will share some recent improvements to the OpenZFS code. These changes are available on all OpenZFS platforms (e.g. Illumos, FreeBSD, Linux, OSX, OSv).

1) Feature flags: The ZFS on-disk format was originally versioned with a linear version number, which was incremented whenever the on-disk format was changed. A ZFS release that supported a given version also must understand all prior versions.

This model was designed initially for the single-vendor model, and was copacetic with the OpenSolaris goals of community development while maintaining control over the essentials of the product. However, in the open development model of OpenZFS, we want different entities to be able to make on-disk format changes independently, and then later merge their changes together into one codebase that understands both features. In the version-number model, two companies or projects each working on their own new on-disk features would both use version N+1 to denote their new feature, but that number would mean different things to each company's software. This would make it very difficult for both companies to contribute their new features into a common codebase. The world would forever be divided into ZFS releases that interpreted version N+1 as company A intended, and those that interpreted it as company B intended.

To address this problem, we designed and implemented "feature flags" to replace the linear version number. Rather than having a simple version number, each storage pool has a list of features that it is using. Features are identified with strings, using a reverse-DNS naming convention (e.g. *com.delphix:background_destroy*). This enables on-disk format changes to be developed independently, and later be integrated into a common codebase.

With the old version numbers, once a pool was upgraded to a particular version, it couldn't be accessed by software that didn't understand that version number. This accomplishes the goal of on-disk versioning, but it is overly restrictive. With OpenZFS feature flags, if a feature is enabled but not actually used, the on-disk information reflects this, so software that doesn't understand the feature can still access the pool. Also, many features change the on-disk format in a way that older software can still safely read a storage pool using the new feature (e.g. because no existing data structures have been changed, only new structures added). OpenZFS feature flags also supports this use case.

2) LZ4 compression: ZFS supports transparent compression, using the LZJB and GZIP algorithms. Each block (e.g. 128KB) is compressed independently, and can be stored as any multiple of the disk's sector size (e.g. 68.5KB). LZJB is fairly fast and provides a decent compression ratio, while GZIP is slow but provides a very good compression ratio. In OpenZFS, we have also implemented the LZ4 compression algorithm, which is faster than LZJB (especially at decompression) and provides a somewhat better compression ratio (see Figure 1). For many workloads, using LZ4 compression

Fig. 1. Compression speeds and ratios compared (single core)

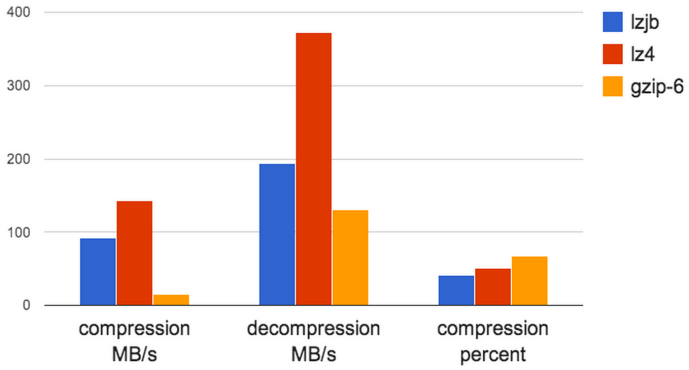
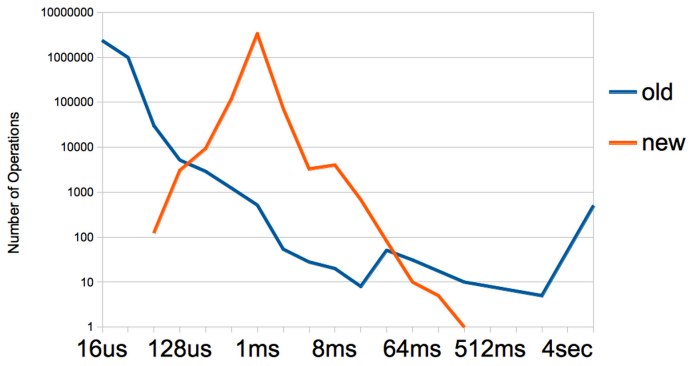


Fig. 2. Histogram of write latencies (log/log graph)



is actually faster than not compressing, because it reduces the amount of data that must be read and written.

3) Smooth write throttle: If the disks can't keep up with the application's write rate, then the filesystem must intervene by causing the application to block, delaying it so that it can't queue up an unbounded amount of dirty data.

ZFS batches several seconds worth of changes into a *transaction group*, or *TXG*. The dirty data that is part of each TXG is periodically synced to disk. Before OpenZFS, the throttle was implemented rather crudely: once the limit on dirty data was reached, all write system calls (and equivalent NFS, CIFS, and iSCSI commands) blocked until the currently syncing TXG completed. The effect is that ZFS performed writes with near-zero latency, until it got "stuck" and all writes blocked for several seconds.[4]

We rewrote the write throttle in OpenZFS to provide much smoother, more consistent latency, by delaying each write operation a little bit. The trick was to find a good way of computing how large the delay should be. The key was to measure the amount of dirty data in the system, incrementing it as write operations came in and decrementing it as write i/o to the storage completes. The delay is a function of the amount of dirty data (as a percentage of the overall dirty data limit). As more write operations come in, the amount of dirty data increases, thus increasing the delay. For a given workload, this algorithm will seek a stable amount of dirty data and thus a stable delay. Crucial for easy understanding of the system, this works without taking into account historical behavior or trying to predict the future. This makes the algorithm very responsive to changing workloads; it can't get "stuck" doing the wrong thing because of a temporary workload anomaly.

As a result of this work, we were able to reduce the latency outliers for a random write workload by 300x, from 10 seconds to 30 milliseconds (meaning that 99.9% of all operations completed in less than 30 milliseconds). (See Figure 2.)

IV. Further Work

Here we will outline some of the projects that are in progress.

A. Platform-independent code repository

Currently, code is shared between platforms on an ad-hoc basis. Generally, Linux and FreeBSD pull changes from illumos. This process is not as smooth as it could be. Linux and FreeBSD must maintain fairly tricky porting layers to translate the interfaces that the ZFS code uses on illumos to equivalent interfaces on Linux and FreeBSD. It is rare that changes developed on other platforms are integrated into illumos, in part because of the technical challenges that newcomers to this platform face in setting up a development environment, porting, building, etc.

We plan to create a platform-independent code repository of OpenZFS source code that will make it much easier to get changes developed on one platform onto every OpenZFS platform. The goal is that all platforms will be able to pull the exact code in the OpenZFS repo into their codebase, without having to apply any diffs.

We will define the interfaces that code in the OpenZFS repo will use, by explicitly wrapping all external interfaces. For example, instead of calling *cv_broadcast(kcondvar_t *)*, OpenZFS code would call *zk_cv_broadcast(zk_condvar_t *)*. Each platform would provide wrappers which translate from the OpenZFS *zk_* interfaces to platform-specific routines and data structures. This will allow the "Solaris Porting Layers" to be simplified.

The OpenZFS repo will only include code that is truly platform-independent, and which can be tested on any platform in userland (using the existing *libzpool.so* mechanism). Therefore it will include the DMU, DSL, ZIL, ZAP, most of the SPA, and userland components (/sbin/zfs, libzfs, etc). It will not include the ZPL, ZVOL, or vdev_disk.c, as these have extensive customizations for each platform. A longer-term goal is to split the ZPL into platform-independent and platform-dependent parts, and include the platform-independent part in the OpenZFS repo.

For more information, see the slides and video from the talk at the 2013 OpenZFS Developer Summit[3].

B. Conferences

Continuing the very successful 2012 ZFS Day and 2013 OpenZFS Developer Summit conferences, we plan to hold more OpenZFS-centered events. This will include annual OpenZFS Developer Summits, as well as more casual local meet-ups. We will also continue evangelizing OpenZFS at general technology conferences.

C. Resumable send and receive

ZFS send and receive is used to serialize and transmit filesystems between pools. It can quickly generate incremental changes between snapshots, making it an ideal basis for remote replication features. However, if the connection between send and receive processes is broken (e.g. by a network outage or one of the machines rebooting), then the send must re-start from the beginning, losing whatever data was already sent.

We are working on an enhancement to this that will allow a failed send to resume where it left off. This involves having the receiving system remember what data has been received. This is fairly simple, because data is sent in (object, offset) order. Therefore the receiving system need only remember the highest (object, offset) that has been received. This information will then be used to restart the send stream from that point.

The one tricky part is that we need to enhance the checksum that is stored in the send stream. Currently the checksum is only sent at the end of the entire send stream, so if the connection is lost, the data that was already received has not been verified by any checksum. We will enhance the send stream format to transmit the checksum after each record, so that we can verify each record as it is received. This will also provide better protection against transmission errors in the metadata of the send stream.

D. Large block support

ZFS currently supports up to 128KB blocks. This is large compared to traditional filesystems, which typically use 4KB or 8KB blocks, but we still see some circumstances where even larger blocks would increase performance. Therefore, we are planning to add support for blocks up to at least 1MB in OpenZFS.

We expect to see an especially large performance benefit when using RAID-Z, especially with very wide stripes (i.e. many devices in the RAID-Z group). RAID-Z breaks each block apart and spreads it out across all devices in the RAID-Z group. Therefore, under a random read workload, RAID-Z can deliver the IOPS of only a single device, regardless of the number of devices in the RAID-Z group. By increasing the block size, we increase the size of each IO, which increases the effective bandwidth of the random read workload.

This is especially important when scrubbing or resilvering, which in the worst case creates a random read workload. By increasing the block size, we raise the lower bound of the scrub or resilver time. For example, consider a RAID-Z group with eight 1-TB disks that can do 100 random reads per second. With 128KB block size, in the worst case we could resilver one drive in 186 hours (1TB * 8 drives / 128KB block size / 100 IOPS). Whereas with 8MB block size, in the worst case we could resilver a drive in 2.8 hours. This corresponds to a rate of 104MB/second, which is close to the

typical maximum sequential transfer rate of hard drives, thus matching the performance of LBA-based resilver mechanisms.

V. Participation

OpenZFS exists because of contributions of every type. There are a number of ways you can get involved:

If you are working with ZFS source code, join the developer mailing list[1]. Post there to get design help and feedback on code changes.

If your company is making a product with OpenZFS, tell people about it. Contact *admin@open-zfs.org* to put your logo on the OpenZFS website. Consider sponsoring OpenZFS events, like the Developer Summit. If you have enhanced OpenZFS, work with the community to contribute your code changes upstream. Beside benefiting everyone using OpenZFS, this will make it much easier for you to sync up with the latest OpenZFS enhancements from other contributors, with a minimum of merge conflicts.

If you are using OpenZFS, help spread the word by writing about your experience on your blog or social media sites. Ask questions at the OpenZFS Office Hours events. And of course, keep sharing your suggestions for how OpenZFS can be even better (including bug reports).

VI. Conclusion

ZFS has survived many transitions, and now with OpenZFS we have the most diverse, and yet also the most unified, community of ZFS contributors. OpenZFS is available on many platforms: illumos, FreeBSD, Linux, OSX, and OSv. OpenZFS is an integral part of dozens of companies' products.[5] A diverse group of contributors continues to enhance OpenZFS, making it an excellent storage platform for a wide range of uses.

References

[1] Mailing list: *developer@open-zfs.org*, see http://www.open-zfs.org/wiki/Mailing_list to join.

[2] Office Hours, see http://www.open-zfs.org/wiki/OpenZFS_Office_Hours

[3] Developer Summit, see http://www.open-zfs.org/wiki/OpenZFS_Developer_Summit_2013

[4] Old ZFS write throttle, see http://blog.delphix.com/ahl/2013/zfs-fundamentals-write-throttle/

[5] Companies using OpenZFS, see http://www.open-zfs.org/wiki/Companies