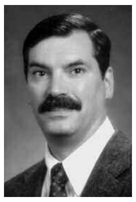
PAWEL JAKUB DAWIDEK AND
MARSHALL KIRK MCKUSICK

porting the Solaris ZFS file system to the FreeBSD operating system

Pawel Jakub Dawidek is a FreeBSD committer. In the FreeBSD project he works mostly in the storage subsystems area (GEOM, file systems), security (disk encryption, opencrypto framework, IPsec, jails), but his code is also in many other parts of the system. Pawel currently lives in Warsaw, Poland, running his small company.

pjd@FreeBSD.org



Dr. Marshall Kirk McKusick writes books and articles, teaches classes on UNIX- and BSD-related subjects, and provides expert-witness testimony on software patent, trade secret, and copyright issues, particularly those related to operating systems and file systems. While at the University of California at Berkeley, he implemented the 4.2BSD fast file system and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD.

mckusick@mckusick.com

THE ZFS FILE SYSTEM MADE REVOLUtionary (as opposed to evolutionary) steps forward in filesystem design, with its authors claiming that they threw away 20 years of obsolete assumptions to design an integrated system from scratch. In this article, we describe the porting of ZFS to FreeBSD, along with describing some of the key features of the ZFS file system.

Features of ZFS

ZFS is more than just a file system. In addition to the traditional role of data storage, ZFS also includes advanced volume management that provides pooled storage through a collection of one or more devices. These pooled storage areas may be used for ZFS file systems or exported through a ZFS Emulated Volume (ZVOL) device to support traditional file systems such as UFS.

POOLED STORAGE MODEL

File systems created by ZFS are not tied to a specified device, volume, partition, or disk but share the storage assigned to a pool. The pool may be constructed from storage ranging from a single partition up to farms composed of hundreds of disks. If more storage is needed, new disks can be added at run time and the space is automatically made available to all the file systems sharing the pool. Thus, there is no need to manually grow or shrink the file systems when space allocation requirements change. There is also no need to create slices or partitions. When working with ZFS, tools such as fdisk(8), bsdlabel(8), newfs(8), tunefs(8), and fsck(8) are no longer needed.

COPY-ON-WRITE DESIGN

File systems must be in a consistent state to function in a stable and reliable way. Unfortunately, it is not easy to guarantee consistency if a power failure or a system crash occurs, because most file system operations are not atomic. For example, when a new hard link to a file is created, the file system must create a new directory entry and increase the link count in the inode. These changes usually require writing two different disk sectors. Atomicity of disk writes can only be guaranteed on a per-sector basis. Two techniques have been used to maintain filesystem consistency when multiple sectors must be updated:

- Checking and repairing the file system with the fsck utility on boot [11], a technique that has lost favor as disk systems have grown. Starting with FreeBSD 5.0, it is possible to run the fsck program in the background, significantly reducing system downtime [9].
- To allow an immediate reboot after a crash, the file system uses soft updates to guarantee that the only inconsistency the file system might experience is resource leaks stemming from unreferenced blocks or inodes [5, 10].

McKusick added the ability to create snapshots to UFS, making background fsck possible [12]. Unfortunately, filesystem snapshots have a few disadvantages, because during one step of a snapshot all write operations to the file system are blocked. Luckily, this step does not depend on filesystem size and takes only a few seconds. However, the time of the step that sets up the snapshot grows linearly with the size of the file system and generates heavy I/O load. So even though the file system continues to operate, its performance is degraded while the snapshot is being prepared.

Once a snapshot is taken, all writes to the file system must be checked to see whether an older copy needs to be saved for the snapshot. Because of the design of snapshots, copies are rarely needed and thus do not appreciably slow down the system. A slowdown does occur when removing many small files (i.e., any file less than 96 kilobytes whose last block is a fragment) that are claimed by a snapshot. In addition, checking a file system in the background slows operating system performance for many hours because of its added demands on the I/O system. If the background fsck fails (usually because of hardware-based disk errors) the operating system needs to be rebooted and the file system must be repaired in the foreground. When a background fsck has failed, it means that the system has been running with an inconsistent file system, which implies undefined behavior.

The second technique used requires storing all filesystem operations (or only metadata changes) first in a special journal. Once the entire operation has been journaled, filesystem updates may be made. If a power failure or a system crash occurs, incomplete entries in the journal are removed and partially completed filesystem updates are finished by using the completed entries stored in the journal. Filesystem journaling is currently the most popular way of managing filesystem consistency [1, 17, 18].

The ZFS file system needs neither fsck nor journals to guarantee consistency. Instead it takes an alternate copy-on-write (COW) approach. COW means that ZFS never overwrites valid data. Instead, ZFS always writes data into a free area. When the data is safely stored, ZFS switches a single pointer in the block's parent. With this technique, block pointers never point at inconsistent blocks. This design is similar to the WAFL file system design [6].

END-TO-END DATA INTEGRITY AND SELF-HEALING

Another important ZFS feature is end-to-end data integrity. All data and metadata undergoes checksum operations using one of several available algorithms (fletcher2, fletcher4 [4], or SHA256 [14]). ZFS can detect silent data corruption caused by any defect in disk, controller, cable, driver, or firmware. There have been many reports from Solaris users of silent data corruption that has been successfully detected by ZFS. If the storage pool has been configured with some level of redundancy (RAID-Z or mirroring) and data corruption is detected, ZFS not only reconstructs the data but also writes valid data back to the component where corruption was originally detected.

SNAPSHOTS AND CLONES

Snapshots are easy to implement for file systems such as ZFS that store data using a COW model. When new data are created, the file system simply does not free the block with the old data. Thus, snapshots in ZFS are cheap to create (unlike UFS2 snapshots). ZFS also allows the creation of a clone, which is a snapshot that may be written. Finally, ZFS has a feature that allows it to roll back a snapshot, forgetting all modifications introduced after the snapshot was created.

ZFS supports compression at the block level. Currently, Jeff Bonwick's variant of the Lempel-Ziv compression algorithm and the gzip compression algorithm are supported. Data encryption is also a work in progress [13].

Porting ZFS to FreeBSD

We describe work done by Pawel Jakub Dawidek in porting ZFS to FreeBSD in the remainder of this article. This task seemed daunting at first, as a student had spent an entire Summer of Code project looking at porting ZFS to Linux and had made little progress. However, a study of the ZFS code showed that it had been written with portability in mind. The ZFS code is clean, well commented, and self-contained. The source files rarely include system headers directly. Most of the time, they include only ZFS-specific header files and a special zfs_context.h header where system-specific includes are placed. Large parts of the kernel code can be compiled in a user process and run by the ztest utility for regression and stress testing.

So, Dawidek felt a fresh start on doing a port seemed appropriate, this time taking the approach of making minimal changes to the ZFS code base itself. Instead, Dawidek built a set of software compatibility modules to convert from the FreeBSD internal interfaces to those used by Solaris and expected by ZFS. Using this approach, he had an initial port up and running with just ten days of effort.

SOLARIS COMPATIBILITY LAYER

When a large project such as ZFS is ported from another operating system, it is important to keep modifications of the original code to a minimum. Having fewer modifications makes porting easier and makes the importation of new functionality and bug fixes much less difficult.

To minimize the number of changes, Dawidek created a Solaris-compatible application programming interface (API) layer. The main goal was to implement the Solaris kernel functions that ZFS expected to call. These functions were implemented by using the FreeBSD kernel programming interface (KPI). Many of the API differences were simple, involving different function names, slightly different arguments, or different return values. For other APIs, the functionality needed to be fully implemented from scratch. This technique proved to be quite effective. For example, after these stubs were built, only 13 files out of 112 of the core ZFS implementation directory needed to be modified.

The following milestones were defined to port the ZFS file system to FreeBSD:

1. Create a Solaris-compatible API using the FreeBSD API.
2. Port the user-level utilities and libraries.
3. Define connection points in ZFS where FreeBSD makes its service requests. These service requests include:
   - ZFS POSIX Layer, which has to be able to communicate with the virtual filesystem (VFS) layer
   - ZFS Emulated Volume (ZVOL), which has to be able to communicate with the FreeBSD volume-management subsystem (GEOM)
   - /dev/zfs, a control device that communicates with the ZFS user-level utilities and libraries
4. Define connection points in ZFS where the storage pool virtual device (VDEV) needs to make I/O requests to FreeBSD.

ZFS POSIX LAYER

The ZFS POSIX layer receives requests from the FreeBSD VFS interface. This interface was the hardest part of the entire port to implement. The VFS interface has many complex functions and is quite system-specific. Although the Solaris and FreeBSD VFS interfaces had a common heritage twenty years ago, much has changed between them over the years. VFS on Solaris seems to be cleaner and a bit less complex than FreeBSD's.

ZFS EMULATED VOLUME

A ZFS VDEV managed storage pool can serve storage in two ways, as a file system or as a raw storage device. ZVOL is a ZFS layer responsible for exporting part of a VDEV-managed storage pool as a disk device.

FreeBSD has its own GEOM layer, which can also be used to manage raw storage devices either to aggregate them with RAID or by striping, or to subdivide them using partitioning. GEOM can also be used to provide compression or encryption (see [12], pp. 270–276, for details on GEOM).

To maximize the flexibility of ZVOL, a new ZVOL provider-only GEOM class was created. As a GEOM provider, the ZVOL storage pool is exported as a device in /dev/ (just like other GEOM providers). So, it is possible to use a ZFS storage pool for a UFS file system or to hold a swap-space partition.

ZFS VIRTUAL DEVICES

A ZFS VDEV-managed storage pool has to use storage provided by the operating system [16]. The VDEV has to be connected to storage at its bottom layer. In Solaris there are two types of storage used by VDEVs: storage from disks and storage from files. In FreeBSD, VDEVs can use storage from any GEOM provider (disk, slice, partition, etc.). ZFS can access files by making them look like disks using an md(4) device.

Rather than interfacing directly to the disks, a new VDEV consumer-only GEOM class was created to interface ZFS to the GEOM layer in

FreeBSD. In its simplest form, GEOM just passes an uninterpreted raw disk to ZFS. But all the functionality of the GEOM layer can be used to build more complex storage arrangements to pass up to a VDEV-managed storage pool.

EVENT NOTIFICATION

ZFS has the ability to send notifications on various events. Those events include information such as storage pool imports as well as failure notifications (I/O errors, checksum mismatches, etc.). Dawidek ported this functionality to send notifications to the devd(8) daemon, which seemed to be the most suitable communication channel for those types of messages. In the future, a dedicated user-level daemon to manage messages from ZFS may be written.

KERNEL STATISTICS

Solaris exports various statistics (mostly about ZFS-cache and name-cache usage) via its kstat interface. This functionality was directed to the FreeBSD sysctl(9) interface. All statistics can be printed using the following command:

```
# sysctl kstat
```

ZFS AND FREEBSD JAILS

ZFS works with Solaris zones [15]. In our port, we make it work with FreeBSD jails [7], which have many of the same features as zones. A useful attribute of ZFS is that once it has constructed a pool from a collection of disks, new file systems can be created and managed from the pool without requiring direct access to the underlying disk devices. Thus, a jailed process can be permitted to manage its own file system since it cannot affect the file systems of other jails or of the base FreeBSD system. If the jailed process were permitted to directly access the raw disk, it could mount a denial-of-service attack by creating a file system with corrupted metadata and then panicking the kernel by trying to access that file system.

ZFS fits into the jail framework well. Once a pool has been assigned to a jail, the jail can operate on its own file system tree. For example:

```
main# zpool create tank mirror da0 da1
main# zfs create tank/jail
main# zfs set jailed=on tank/jail
main# zfs jail 1 tank/jail
jail# zfs create tank/jail/home
jail# zfs create tank/jail/home/pjd
jail# zfs create tank/jail/home/mckusick
jail# zfs snapshot tank/jail@backup
```

FreeBSD Modifications

There were only a few FreeBSD modifications needed to port the ZFS file system.

The mountd(8) program was modified to work with multiple export files. This change allows the zfs(1) command to manage private export files stored in /etc/zfs/exports.

The vnode-pointer-to-file-handle (VPTOFH) operation was switched from one based on the filesystem type (VFS_VPTOFH) to one based on the vnode type (VOP_VPTOFH). Architecturally, the VPTOFH translation should always have been a vnode operation, but Sun first defined it as a filesystem operation, so BSD did the same to be compatible. Solaris changed it to a vnode operation years ago, so it made sense for FreeBSD to do so as well. This change allows VPTOFH to support multiple node types within one file system. For example, in ZFS the v_data field from the vnode structure can point at two different structures (either znode_t or zfsctl_node_t). To be able to recognize which structure it references, two different vop_vptofh functions are defined for those two different types of vnodes.

The lseek(2) API was extended to support the SEEK_DATA and SEEK_HOLE operation types [2]. These operations are not ZFS-specific. They are useful on any file system that supports holes in files, as they allow backup software to identify and skip holes in files.

The jail framework was extended to support "jail services." With this extension, ZFS can register itself as a jail service and attach a list of assigned ZFS datasets to the jail's in-kernel structures.

User-level Utilities and Libraries

User-level utilities and libraries communicate with the kernel part of ZFS via the /dev/zfs control device. We needed to port the following utilities and libraries:

- zpool: utility for storage pools configuration
- zfs: utility for ZFS file systems and volumes configuration
- ztest: program for stress testing most of the ZFS code
- zdb: ZFS debugging tool

- libzfs: the main ZFS user-level library used by both the zfs and zpool utilities
- libzpool: test library containing most of the kernel code, used by ztest

To make it work, we also ported libraries (or implemented wrappers) they depend on: libavl, libnvpair, libutil, and libumem.

Testing FileSystem Correctness

It is quite important and very hard to verify that a file system works correctly. The file system is a complex beast and there are many corner cases that have to be checked. If testing is not done right, bugs in a file system can lead to application misbehavior, system crashes, data corruption, or even security failure. Unfortunately, we did not find freely available filesystem test suites that verify POSIX conformance. Instead, Dawidek wrote the fstest test suite [3]. The test suite currently contains 3438 tests in 184 files and tests all the major filesystem operations including chflags, chmod, chown, close, link, mkdir, mkfifo, open, read, rename, rmdir, symlink, truncate, and unlink.

Filesystem Performance

Below are some performance numbers that compare the current ZFS version for FreeBSD with various UFS configurations. Note that all file systems were tested with the atime option turned off. The ZFS numbers are measured with checksumming enabled, as that is the recommended configuration.

Untarring src.tar archive four times one by one:

UFS	256s
UFS+soft-updates	207s
UFS+gjournal+async	127s
ZFS	237s

Removing four src directories one by one:

UFS	230s
UFS+soft-updates	94s
UFS+gjournal+async	48s
ZFS	97s

Untarring src.tar by four processes in parallel:

UFS	345s
UFS+soft-updates	333s
UFS+gjournal+async	158s
ZFS	199s

Removing four src directories by four processes in parallel:

UFS	364s
UFS+soft-updates	185s
UFS+gjournal+async	111s
ZFS	220s

Executing dd if=/dev/zero of=/fs/zero bs=1m count=5000:

UFS	78s
UFS+soft-updates	77s
UFS+gjournal+async	200s
ZFS	111s

Status and Future Directions

After about six months of work, the ZFS port is almost finished. About 98% of the functionality is ported and tested. The primary work that remains is to tune its performance.

Here are some missing functionalities:

- ACL support. Currently ACL support has not been ported. ACL support is difficult to implement because FreeBSD has only support for POSIX.1e ACLs, whereas ZFS implements NFSv4-style ACLs. Porting NFSv4-style ACLs to FreeBSD requires the addition of system calls, updating system utilities to manage ACLs, and preparing procedures on how to convert from one ACL type to another.
- Allowing ZFS to export ZVOLs over iSCSI. At this point there is no iSCSI target daemon in the FreeBSD base system, so there is nothing with which to integrate this functionality.
- Code optimization. Many parts of the code were written quickly but inefficiently.

ZFS was recently merged into the FreeBSD base system. Indeed, it may be ready for version 7.0 release. There are no plans to merge ZFS to the RELENG_6 branch.

The UFS file system supports system flags—chflags(2). There is no support for those in the ZFS file system, but it would be easy to add support for system flags to ZFS.

There is no encryption support in ZFS itself, but there is an ongoing project to implement it. It may be possible to cooperate with Sun developers to help finish this project. With a properly defined interface within ZFS, it would be easy to integrate encryption support provided by the opencrypto framework [8].

ACKNOWLEDGMENTS

We would like to thank the ZFS developers, who created a great file system, and the FreeBSD Foundation (www.FreeBSDFoundation.org) for their support. The machines from the FreeBSD Netperf Cluster (www.freebsd.org/projects/netperf/cluster .html)were used for much of the development work. Pawel Jakub Dawidek would like to thank Wheel LTD (www.wheel.pl). He was able to do this work during his day job. Finally, we would like to thank the FreeBSD community for their never-ending support and warm words.

REFERENCES

[1] S. Best, "JFS Overview" (2000): http://www-128.ibm.com/developerworks/linux/library/l-jfs.html.

[2] J. Bonwick, "SEEK_HOLE and SEEK_DATA for Sparse Files" (2005): http://blogs.sun.com/bonwick/entry/seek_hole_and_seek_data.

[3] P. Dawidek, "File System Test Suite" (2007): http://people.freebsd.org/~pjd/fstest/.

[4] J. Fletcher, "Fletcher's Checksum" (1990): http://en.wikipedia.org/wiki/Fletcher's_checksum.

[5] G. Ganger, M.K. McKusick, C. Soules, and Y. Patt, "Soft Updates: A Solution to the Metadata Update Problem in File Systems," *ACM Transactions on Computer Systems* 18(2) (2000): 127–153.

[6] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," *USENIX Association Conference Proceedings* (Berkeley, CA: USENIX Association, 1994), pp. 235–246.

[7] P. Kamp and R. Watson, "Jails: Confining the Omnipotent Root," *Proceedings of the Second International System Administration and Networking Conference (SANE)* (2000): http://docs.freebsd.org/44doc/papers/jail/.

[8] S. Leffler, "Cryptographic Device Support for FreeBSD," *Proceedings of BSDCon 2003* (Berkeley, CA: USENIX, 2003), pp. 69–78.

[9] M.K. McKusick, "Running Fsck in the Background," *Proceedings of the BSDCon 2002 Conference*, pp. 55–64.

[10] M.K. McKusick and G. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Proceedings of the FREENIX Track at the 1999 USENIX Annual Technical Conference* (Berkeley, CA: USENIX Association, 1999), pp. 1–17.

[11] M.K. McKusick and T.J. Kowalski, "Fsck: The UNIX File System Check Program," in *4.4BSD System Manager's Manual* (Sebastopol, CA: O'Reilly & Associates, 1994), vol. 3, pp. 1–21.

[12] M.K. McKusick and G. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System* (Reading, MA: Addison-Wesley, 2005).

[13] D. Moffat, "ZFS Encryption Project" (2006): www.opensolaris.org/os/project/zfs-crypto/files/zfs-crypto.pdf.

[14] NIST, "SHA Hash Functions" (1993): http://en.wikipedia.org/wiki/SHA-256.

[15] D. Price and A. Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," *Proceedings of LISA '04: 18th Large Installation System Administration Conference* (Berkeley, CA: USENIX Association, 2004), pp. 241–254.

[16] Sun Microsystems, "ZFS Source Tour" (2007): http://www.opensolaris.org/os/community/zfs/source/.

[17] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," *USENIX 1996 Annual Technical Conference Proceedings* (Berkeley, CA: USENIX Association, 1996), pp. 1–14.

[18] S. Tweedie, "EXT3, Journaling Filesystem," Ottawa Linux Symposium (2003): http://ssrc.cse.ucsc.edu/PaperArchive/ext3.html.