Porting the ZFS file system to the FreeBSD operating system

Pawel Jakub Dawidek
pjd@FreeBSD.org

1 Introduction

The ZFS file system makes a revolutionary (as opposed to evolutionary) step forward in file system design. ZFS authors claim that they throw away 20 years of obsolute assumptions and designed an integrated system from scratch.

The ZFS file system was developed by Sun Microsystems, Inc. and was first available in Solaris 10 operating system. Although we cover some of the key features of the ZFS file system, the primary focus of this paper is to cover how ZFS was ported to the FreeBSD operating system.

FreeBSD is an advanced, secure, stable and scalable UNIX-like operating system, which is widely deployed for various internet functions. Some argue that one of the largest challenges facing FreeBSD is the lack of a robust file system. Porting ZFS to FreeBSD attempts to address these short comings.

2 ZFS file system and some of its features

Calling ZFS a file system is not precise. ZFS is much more than only file system. It integrates advanced volume management, which can be utilized by the file system on top of it, but also to provide storage through block devices (ZVOLs). ZFS also has many interesting features not found in other file systems. In this section, we will describe some of the features we find most interesting.

2.1 Pooled storage model

File systems created by ZFS are not tied to a specified block device, volume, partition or disk. All file systems within the same ”pool”, share the whole storage assigned to the ”pool”. A pool is a collection of storage devices. It may be constructured from one partition only, as well as from hundreds of disks. If we need more storage we just add more disks. The new disks are added at run time and the space is automatically available to all file systems. 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, one can simply forget about tools like fdisk(8), bsdlabel(8), newfs(8), tunefs(8) and fsck(8) when working with ZFS.

2.2 Copy-on-write design

To ensure the file system is functioning in a stable and reliable manner, it must be in a consistent state. Unfortunately it is not easy to guarantee consistency in the event of a power failure or a system crash, because most file system operations are not atomic. For example when a new hard link to a file is created, we need to create a new directory entry and increase link count in the inode, which means we need two writes. Atomicity around disk writes can only be guaranteed on a per sector basis. This means if a write operation spans more than a single sector, there can be no atomicity guarantees made by the disk device. The two most common methods to manage consistency of file system are:

- Checking and repairing file system with fsck [McKusick1994] utility on boot. This is very inefficient method, because checking large file systems can take serval hours. Starting from FreeBSD 5.0 it is possible to run fsck program in the background [McKusick2002], significantly reducing system downtime. To make it possible, UFS [McKusick1996] gained ability to create snapshots [McKusick1999]. Also file system has to use Soft updates [Ganger] guarantee that the only

inconsistency the file system would experience is resource leaks steming from unreferenced blocks or inodes. Unfortunately, file system snapshots have few disadvantages. One of the stages of performing a snapshot blocks all write operations. This stage should not depend on file system size and should not take too long. The time of another stage, which is responsible for snapshot perparation grows linearly with the size of the file system and generates heavy I/O load. Once snapshot is taken, it should not slow the system down appreciably except when removing many small files (i.e., any file less than 96Kb whose last block is a fragment) that are claimed by a snapshot. In addition checking file system in the background slows operating system performance for many hours. Practice shows that it is also possible for background fsck to fail, which is a really hard situation, because operating system needs to be rebooted and file system repaired in foreground, but what is more important, it means that system was working with inconsistent file system, which implies undefined behaviour.

- Store all file system operations (or only metadata changes) first in a special "journal", once the whole operation is in the journal, it is moved to the destination area. In the event of a power failure or a system crash incomplete entires in the journal are removed and not fully copied entries are copied once again. File system journaling is currently the most popular way of managing file system consistency [Tweedie2000, Best2000, Sweeney1996].

The ZFS file system does not need fsck or jounrals to guarantee consistency, instead takes an alternate "Copy On Write" (COW) approach. This means it never overwrites valid data - it writes data always into free area and when is sure that data is safely stored, it just switches pointer in block's parent. In other words, block pointers never point at inconsistent blocks. This design is similar to the WAFL [Hitz] file system design.

2.3 End-to-end data integrity and self-healing

Another very important ZFS feature is end-to-end data integrity - all data and metadata undergoes checksum operations using one of several available algorithms (fletcher2 [fletcher], fletcher4 or SHA256). This allows to detect with very high probability silent data corruptions cased by any defect in disk, controller, cable, driver, or firmware. Note, that ZFS metadata are always checksummed using SHA256 algorithm. There are already many reports from the users experiencing silent data corruptions successfully detected by ZFS. If some level of redundancy is configured (RAID1 or RAID-Z) and data corruption is detected, ZFS not only reads data from another replicated copy, but also writes valid data back to the component where corruption was originally detected.

2.4 Snapshots and clones

A snapshot is a read-only file system view from a given point in time. Snapshots are fairly easy to implement for file system storing data in COW model - when new data is stored we just don't free the block with the old data. This is the reason why snapshots in ZFS are very cheap to create (unlike UFS2 snapshots). Clone is created on top of a snapshot and is writable. It is also possible to roll back a snapshot forgetting all modifications introduced after the snapshot creation.

2.5 Built-in compression and encryption

ZFS supports compression at the block level. Currently (at the time this paper is written) only one compression algorithm is supported - lzjb (this is a variant of Lempel-Ziv algorithm, jb stands for his creator - Jeff Bonwick). There is also implementation of gzip algorithm support, but it is not included in the base system yet. Data encryption is a work in progress [Moffat2006].

2.6 Portability

A very important ZFS characteristic is that the source code is written with portability in mind. This is not an easy task, especially for the kernel code. ZFS code is very portable, clean, well commented and almost self-contained. The source files rarely include system headers directly. Most of the times, they only include ZFS-specific header files and a special `zfs_context.h` header, where one should place system-specific includes. Big part of the kernel code can be also compiled in user-land and used with ztest utility for regression and stress testing.

3 ZFS and FreeBSD

This section describes the work that has been done to port the ZFS file system over to the FreeBSD operating system.
The code is organized in the following parts of the source tree:

- `contrib/opensolaris/` - userland code taken from OpenSolaris, used by ZFS (ZFS control utilities, libraries, etc.),
- `compat/opensolaris/` - userland API compatibility layer (implementation of Solaris-specific functions in FreeBSD way),
- `cddl/` - Makefiles used to build userland utilities and libraries,
- `sys/contrib/opensolaris/` - kernel code taken from OpenSolaris, used by ZFS,
- `sys/compat/opensolaris/` - kernel API compatiblity layer,
- `sys/modules/zfs/` - Makefile for building ZFS kernel module.

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

- Created Solaris compatible API based on FreeBSD API.
- Port userland utilities and libraries.
- Define connection points in the ZFS top layers where FreeBSD will talk to us and those are:
  - ZPL (ZFS POSIX Layer) which has to be able to communicate with VFS,
  - ZVOL (ZFS Emulated Volume) which has to be able to communicate with GEOM,
  - `/dev/zfs` control device, which actually only talks to ZFS userland utilities and libraries.
- Define connection points in the ZFS buttom layers where ZFS needs to talk to FreeBSD and this is only VDEV (Virtual Device), which has to be able to communicate with GEOM.

3.1 Solaris compatibility layer

When a large project like ZFS is ported from another operating system one of the most important rules is to keep number of modifications of the original code as small as possible, because the fewer modifications, the easier porting new functionality and bug fixes is. The programmer that does the porting work is not the only one responsible for number of changes needed, it also depends on how portable the source code is.

To minize the number of changes, a Solaris API compatability layer was created. The main goal was to implement Solaris-specific functions, structures, etc. using FreeBSD KPI. In some cases, functions needed to be renamed, while in others, functionality needed to be fully implemented from scratch. This technique proved to be very effective (not forgetting about ZFS code portability). For example looking at files from the `uts/common/fs/zfs/` directory and taking only non-trivial changes into account, only 13 files out of 112 files were modified.

3.1.1 Atomic operations

There are a bunch of atomic operations implemented in FreeBSD (atomic(9)), but there are some that exist in Solaris and have no equivalents in FreeBSD. The missing operations in pseudo-code look like this:

```
<type>
atomic_add_<type>_nv(<type> *p, <type> v)
{
    return (*p += v);
}

<type>
atomic_cas_<type>(<type> *dst, <type> cmp, <type> new)
{
    <type> old;

    old = *dst;
    if (old == cmp)
        *dst = new;
    return (old);
}
```

Another missing piece is that FreeBSD implements 64bit atomic operations only on 64bit architectures and ZFS makes heavy use of such operations on all architectures.

Currently, atomic operations are implemented in assembly language located in the machine dependant portions of the kernel. As a temporary work around, the missing atomic operations were implemented in C, and global mutexes were used to guarantee atomicity. Looking forward, the missing atomic operations may be imported directly from Solaris.

3.1.2 Sleepable mutexes and condition variables

The most common technique of access synchronization to the given resources is locking. To guarantee exclusive access FreeBSD and Solaris use mutexes. Unfortunately we cannot use FreeBSD mutex(9) KPI to im-

plement Solaris mutexes, because there are some important differences. The biggest problem is that sleeping with FreeBSD mutex held is prohibited, on Solaris on the other hand such behaviour is just fine. The way we took was to implement Solaris mutexes based on our shared/exclusive locks - sx(9), but only using exclusive locking. Because of using sx(9) locks for Solaris mutex implementation we also needed to implement condition variables (condvar(9)) to use Solaris mutexes.

3.2 FreeBSD modifications

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

The sleepq_add(9) function was modified to take `struct lock_object *` as an argument instead of `struct mtx *`. This change allowed to implement Solaris condition variables on top of sx(9) locks.

The mountd(8) program gained ability to work with multiple exports files. With this change we can automatically manage private exports file stored in `/etc/zfs/exports` via zfs(1) command.

The VFS_VPTOFH() operation was turned into VOP_VPTOFH() operation. As confirmed by Kirk McKusick, vnode to file handle translation should be a VOP operation in the first place. This change allows to support multiple node types within one file system. For example in ZFS v_data field from the vnode structure can point at two different structures (znode_t or zfsctl_node_t). To be able to recognize which structure it is, we define different functions as vop_vptofh operation for those two different vnodes.

lseek(2) API was extended to support SEEK_DATA and SEEK_HOLE [Bonwick2005] operation types. Those operations are not ZFS-specific. They are useful mostly for backup software to skip ”holes” in files. ”Holes” like those created with truncate(2).

3.3 Userland utilities and libraries

Userland utilities and libraries communicate with the kernel part of the ZFS via `/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 userland library used by both 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`, `libuutil` and `libumem`.

3.4 VDEV_GEOM

ZFS have to use storage provided by the operating system, so at the bottom layers it has to be connected to disks. In Solaris there are two ”leaf” VDEVs (Virtual Devices) that allow to use storage from disks (VDEV_DISK) and from files (VDEV_FILE). We don’t use those in FreeBSD. The interface to talk to disks in FreeBSD is totally incompatible with what Solaris has. That’s why we decided to create a FreeBSD-specific leaf VDEV - VDEV_GEOM. VDEV_GEOM was implemented as consumer-only GEOM class, which allows to use **any** GEOM provider (disk, partition, slice, mirror, encrypted storage, etc.) as a storage pool component. We find this solution very flexible, even more flexible than what Solaris has. We also decided not to port VDEV_FILE, because files can always be accessed via md(4) devices.

3.5 ZVOL

ZFS can serve the storage in two ways - as a file system or as a raw storage device. ZVOL (ZFS Emulated Volume) is a ZFS layer responsible for managing raw storage devices (GEOM providers in FreeBSD) backed by space from a storage pool. It was implemented in FreeBSD as a provider-only GEOM class to fit best in FreeBSD current architecture (all storage devices in FreeBSD are GEOM providers). This way we can put a UFS file system or swap on top of a ZFS volume or we can use ZFS volumes as components in other GEOM tranformations. For example we can encrypt ZFS volume with GELI class.

3.6 ZPL

ZPL (ZFS POSIX Layer) is the layer that VFS interface communicates with. This was the hardest part of the enitre ZFS port. The VFS interfaces are most of the time very system-specific and also very complex. We belive

that VFS is one of the most complex subsystem in the entire FreeBSD kernel.
There are many differences in VFS on Solaris and FreeBSD, but they are still quite similar. VFS on Solaris seems to be cleaner and a bit less complex than FreeBSD's.

3.7 Event notification

ZFS has the ability to send notifications on various events. Those events include information like storage pool imports as well as failure notifications (I/O errors, checksum mismatches, etc.). This functionality was ported to send notifications to the devd(8) daemon, which seems to be the most suitable communication channel for those kind of messages. We may consider creating dedicated userland daemon to manage messages from ZFS.

3.8 Kernel statistics

Various statistics (mostly about ZFS cache usage) are exported via kstat Solaris interface. We implemented the same functionality using FreeBSD sysctl interface. All statistics can be printed using the following command:

```
# sysctl kstat
```

3.9 Kernel I/O KPI

The configuration of a storage pool is kept on its components, but in addition configuration of all pools is cached in `/etc/zfs/zpool.cache` file. When the pools are added, removed or modified `/etc/zfs/zpool.cache` file is updated. It was not possible to access files from the kernel easly (without using VFS internals), so we created KPI that allows to perform simple operations on files from the kernel. We called the KPI "kernio". Below are the list of operations supported. All functions are equivalents of userland functions, the only difference is that they operate on vnode, not file descriptor.

struct vnode *kio_open(const char *file, int flags, int cmode)

- Opens or creates a file returning a pointer to a vnode related to the file. Returns NULL if file can't be opened or created.

void kio_close(struct vnode *vp)

- Close the file related to the given vnode.

ssize_t kio_pread(struct vnode *vp, void *buf, size_t size, off_t offset)

- Reads data at the given offset. Returns number of bytes read or -1 if the data cannot be read.

ssize_t kio_pwrite(struct vnode *vp, void *buf, size_t size, off_t offset)

- Writes data at the given offset. Returns number of bytes written or -1 if the data cannot be written.

int kio_fstat(struct vnode *vp, struct vattr *vap)

- Obtains informations about the given file. Return 0 on success or error number on failure.

int kio_fsync(struct vnode *vp)

- Causes all modified data and file attributes to be moved to a permanent storage device. Return 0 on success or error number on failure.

int kio_rename(const char *from, const char *to)

- Renames file **from** to a name **to**. Return 0 on success or error number on failure.

int kio_unlink(const char *name)

- Removes file **name**. Return 0 on success or error number on failure.

4 Testing file system correctness

It is very important and very hard to verify that file system works correctly. File system is a very 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 applications misbehaviour, system crashes, data corruptions or even security holes. Unfortunately we didn't find freely available file system test suits, that verify POSIX conformance. Because of that, during the ZFS port project the author developed fstest test suite [fstest]. At the time this paper is written, the test suite contains 3438 tests in 184 files and tests the following file system operations: chflags, chmod, chown, link, mkdir, mkfifo, open, rename, rmdir, symlink, truncate, unlink.

5 File system performance

Below we present some performance numbers to compare current ZFS version for FreeBSD with various UFS configurations. All file systems were tested with the `atime` option turned off.
Untaring src.tar archive four times one by one:

UFS	256s
UFS+SU	207s
UFS+gjournal+async	127s
ZFS	237s

Removing four src directories one by one:

UFS	230s
UFS+SU	94s
UFS+gjournal+async	48s
ZFS	97s

Untaring src.tar by four processes in parallel:

UFS	345s
UFS+SU	333s
UFS+gjournal+async	158s
ZFS	199s

Removing four src directories by four processes in parallel:

UFS	364s
UFS+SU	185s
UFS+gjournal+async	111s
ZFS	220s

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

UFS	78s
UFS+SU	77s
UFS+gjournal+async	200s
ZFS	111s

6 Status and future directions

6.1 Port status

ZFS port is almost finished. 98% of the whole functionality is already ported. We still need to work on performance. Here are some missing functionalities:

- ACL support. Currently ACL support is not ported. This is more complex problem, because FreeBSD has only support for POSIX.1e ACLs. ZFS implements NFSv4-style ACLs. To be able to port it to FreeBSD, we must add required system calls, teach system utilities how to manage ACLs and prepare procedures on how to convert from one ACL-type to another on copy, etc. (if possible).

- ZFS allows to export file systems over NFS (which is already implemented) and ZVOLs over iSCSI. At this point there is no iSCSI target deamon in the FreeBSD base system, so there is nothing to integrate this functionality with.

- Clean up some parts of the code that were coded temporarily to allow to move forward.

6.2 Future directions

Of course there is a plan to import ZFS into FreeBSD base system, it may be ready for 7.0-RELEASE. There is no plan to merge ZFS to the RELENG_6 branch.

One of the interesting things to try is to add jails [Kamp2000] support to ZFS. On Solaris, ZFS has support for zones [Price] and will be nice to experiment with allowing for ZFS file system creation and administration from within a jail.

FreeBSD UFS file system supports system flags - chflags(2). There is no support for those in the ZFS file system. We consider adding support for system flags to ZFS.

There is no encryption support in the 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 and to protect portability of the code, so we can easly integrate encryption support with the opencrypto [Leffler2003] framework.

7 Acknowledgments

I'd like to thank ZFS Developers who created this great file system.

I'd like to thank the FreeBSD Foundation [fbsdf] for their support. I'm using machines from the FreeBSD Netperf Cluster [netperf] for my development work. This paper was also first presented at AsiaBSDCon conference, where the FreeBSD Foundation covered my transportation costs.

I'd like to thank Wheel LTD [wheel]. I was able to do the work during my day job.

I'd like to thank the FreeBSD community for their never-ending support and warm words.

References

[McKusick1994] M. McKusick and T. Kowalski, *Fsck - The UNIX File System Check Program*, 1994

[McKusick2002] M. McKusick, *Running 'Fsck' in the Background*, 2002

[McKusick1996] M. McKusick, K. Bostic, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, 1996

[McKusick1999] M. McKusick, G. Ganger, *Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem*, 1999

[Ganger] G. Ganger, M. McKusick, C. Soules, and Y. Patt, *Soft Updates: A Solution to the Metadata Update Problem in File Systems*

[Tweedie2000] S. Tweedie, *EXT3, Journaling Filesystem*, 2000

[Best2000] S. Best, *JFS overview*, 2000

[Sweeney1996] A. Sweeney, *Scalability in the XFS File System*, 1996

[Hitz] D. Hitz, J. Lau, and M. Malcolm, *File System Design for an NFS File Server Appliance*,

[SHA-1] SHA-1 hash function,
`http://en.wikipedia.org/wiki/SHA-1`

[fletcher] Fletcher's checksum,
`http://en.wikipedia.org/wiki/Fletcher's_checksum`

[Moffat2006] D. Moffat, *ZFS Encryption Project*, 2006

[Bonwick2005] J. Bonwick, *SEEK_HOLE and SEEK_DATA for sparse files*, 2005

[Kamp2000] P. Kamp and R. Watson, *Jails: Confining the omnipotent root*, 2000

[Leffler2003] S. Leffler, *Cryptographic Device Support for FreeBSD*, 2003

[Price] D. Price and A. Tucker, *Solaris Zones: Operating System Support for Consolidating Commercial Workloads*

[fstest] File system test suite,
`http://people.freebsd.org/ pjd/fstest/`,
2007

[fbsdf] The FreeBSD Foundation,
`http://www.FreeBSDFoundation.org/`

[netperf] FreeBSD Netperf Cluster,
`http://www.freebsd.org/projects/netperf/cluster.html`

[wheel] Wheel LTD,
`http://www.wheel.pl`