The Zettabyte File System

Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, Mark Shellenbaum

Jeff.Bonwick@sun.com

Abstract

In this paper we describe a new file system that provides strong data integrity guarantees, simple administration, and immense capacity. We show that with a few changes to the traditional high-level file system architecture — including a redesign of the interface between the file system and volume manager, pooled storage, a transactional copy-on-write model, and self-validating checksums — we can eliminate many drawbacks of traditional file systems. We describe a general-purpose production-quality file system based on our new architecture, the Zettabyte File System (ZFS). Our new architecture reduces implementation complexity, allows new performance improvements, and provides several useful new features almost as a side effect.

1 Introduction

Upon hearing about our work on ZFS, some people appear to be genuinely surprised and ask, "Aren't local file systems a solved problem?" From this question, we can deduce that the speaker has probably never lost important files, run out of space on a partition, attempted to boot with a damaged root file system, needed to repartition a disk, struggled with a volume manager, spent a weekend adding new storage to a file server, tried to grow or shrink a file system, mistyped something in `/etc/fstab`, experienced silent data corruption, or waited for `fsck` to finish. Some people are lucky enough to never encounter these problems because they are handled behind the scenes by system administrators. Others accept such inconveniences as inevitable in any file system. While the last few decades of file system research have resulted in a great deal of progress in performance and recoverability, much room for improvement remains in the areas of data integrity, availability, ease of administration, and scalability.

Today's storage environment is radically different from that of the 1980s, yet many file systems continue to adhere to design decisions made when 20 MB disks were considered big. Today, even a personal computer can easily accommodate 2 terabytes of storage — that's one ATA PCI card and eight 250 GB IDE disks, for a total cost of about $2500 (according to http://www.pricewatch.com) in late 2002 prices. Disk workloads have changed as a result of aggressive caching of disk blocks in memory; since reads can be satisfied by the cache but writes must still go out to stable storage, write performance now dominates overall performance[8, 16]. These are only two of the changes that have occurred over the last few decades, but they alone warrant a reexamination of file system architecture from the ground up.

We began the design of ZFS with the goals of strong data integrity, simple administration, and immense capacity. We decided to implement ZFS from scratch, subject only to the constraint of POSIX compliance. The resulting design includes pooled storage, checksumming of all on-disk data, transactional copy-on-write update of all blocks, an object-based storage model, and a new division of labor between the file system and volume manager. ZFS turned out to be simpler to implement than many recent file systems, which is a hopeful sign for the long-term viability of our design.

In this paper we examine current and upcoming problems for file systems, describe our high level design goals, and present the Zettabyte File System. In Section 2 we explain the design principles ZFS is based on. Section 3 gives an overview of the implementation of ZFS. Section 4 demonstrates ZFS in action. Section 5 discusses the design tradeoffs we made. Section 6 reviews related work. Section 7 describes future avenues for research, and Section 8 summarizes the current status of ZFS.

2 Design principles

In this section we describe the design principles we used to design ZFS, based on our goals of strong data integrity, simple administration, and immense capacity.

2.1 Simple administration

On most systems, partitioning a disk, creating a logical device, and creating a new file system are painful and slow operations. There isn't much pressure to simplify and speed up these kinds of administrative tasks because they are relatively uncommon and only performed by system administrators. At the same time though, mistakes or accidents during these sorts of administrative tasks are fairly common, easy to make, and can destroy lots of data quickly[3]. The fact that these tasks are relatively uncommon is actually an argument for making them easier — almost no one is an expert at these tasks precisely *because* they are so uncommon. As more and more people become their own system administrators, file systems programmers can no longer assume a qualified professional is at the keyboard (which is never an excuse for a painful user interface in the first place).

Administration of storage should be simplified and automated to a much greater degree. Manual configuration of disk space should be unnecessary. If manual configuration is desired, it should be easy, intuitive, and quick. The administrator should be able to add more storage space to an existing file system without unmounting, locking, or otherwise interrupting service on that file system. Removing storage should be just as easy. Administering storage space should be simple, fast, and difficult to screw up.

The overall goal is to allow the administrator to state his or her intent ("make a new file system") rather than the details to implement it (find unused disk, partition it, write the on-disk format, etc.). Adding more layers of automation, such as GUIs, over existing file systems won't solve the problem because the file system is the unit of administration. Hiding the division of files into file systems by covering it up with more layers of user interface won't solve underlying problems like files that are too large for their partitions, static allocation of disk space to individual file systems or unavailable data during file system repair.

2.2 Pooled storage

One of the most striking design principles in modern file systems is the one-to-one association between a file system and a particular storage device (or portion thereof). Volume managers do virtualize the underlying storage to some degree, but in the end, a file system is still assigned to some particular range of blocks of the logical storage device. This is counterintuitive because a file system is intended to virtualize physical storage, and yet there remains a fixed binding between a logical namespace and a specific device (logical or physical, they both look the same to the user).

To make the problem clearer, let's look at the analogous problem for main memory. Imagine that every time an administrator added more memory to a system, he or she had to run the "`formatmem`" command to partition the new memory into chunks and assign those chunks to applications. Administrators don't have to do this because virtual addressing and dynamic memory allocators take care of it automatically.

Similarly, file systems should be decoupled from physical storage in much the same way that virtual memory decouples address spaces from memory banks. Multiple file systems should be able to share one pool of storage. Allocation should be moved out of the file system and into a storage space allocator that parcels out permanent storage space from a pool of storage devices as file systems request it. Contrast the traditional volume approach in Figure 1 with the pooled storage approach in Figure 2. Logical volumes are a small step in this direction, but they still look like a range of bytes that must be partitioned before use.

The interface between the file system and the volume manager puts allocation on the wrong side of the interface, making it difficult to grow and shrink file systems, share space, or migrate live data.

2.3 Dynamic file system size

If a file system can only use space from its partition, the system administrator must then predict (i.e., guess) the maximum future size of each file system

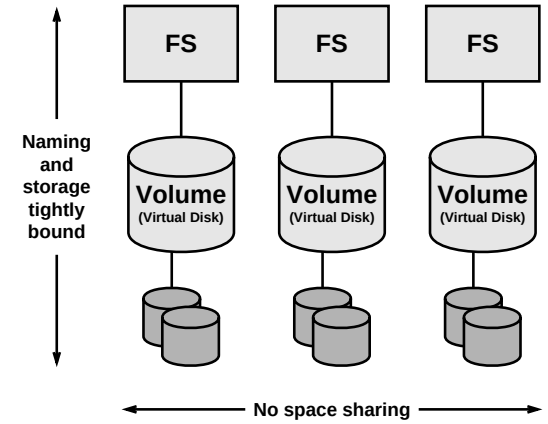

Figure 1: Storage divided into volumes

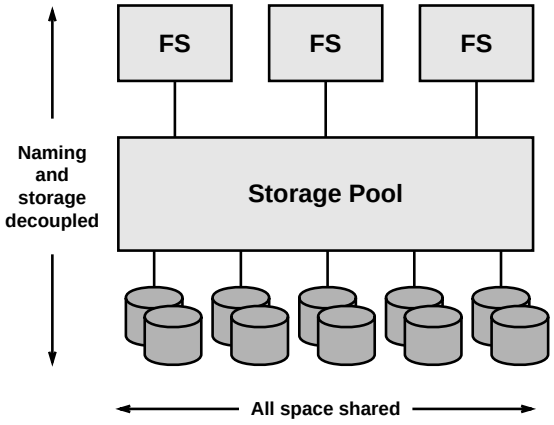

Figure 2: Pooled storage

at the time of its creation. Some file systems solve this problem by providing programs to grow and shrink file systems, but they only work under certain circumstances, are slow, and must be run by hand (rather than occurring automatically). They also require a logical volume manager with the capability to grow and shrink logical partitions.

Besides the out-of-space scenario, this also prevents the creation of a new file system on a fully partitioned disk.[1] Most of the disk space may be unused, but if no room is left for a new partition, no new file systems may be created. While the growth of disk space has made this limitation less of a concern — users can afford to waste lots of disk space because many systems aren't using it anyway[5] — partitioning of disk space remains an administrative headache.

Once the storage used by a file system can grow and shrink dynamically through the addition and removal of devices, the next step is a file system that can grow and shrink dynamically as users add or remove data. This process should be entirely automatic and should not require administrator intervention. If desired, the administrator should be able to set quotas and reservations on file systems or groups of file systems in order to prevent unfair usage of the storage pool.

Since file systems have no fixed size, it no longer makes sense to statically allocate file system metadata at file system creation time. In any case, administrators shouldn't be burdened with a task that the file system can perform automatically. Instead, structures such as inodes[2] should be allocated and freed dynamically as files are created and deleted.

2.4 Always consistent on-disk data

Most file systems today still allow the on-disk data to be inconsistent in some way for varying periods of time. If an unexpected crash or power cycle happens while the on-disk state is inconsistent, the file system will require some form of repair

[1] Some might wonder why multiple file systems sharing pooled storage is preferable to one large growable file system. Briefly, the file system is a useful point of administration (for backup, mount, etc.) and provides fault isolation.

[2] An inode is the structure used by a file system to store metadata associated with a file, such as its owner.

on the next boot. Besides FFS-style file systems that depend on repair from `fsck`[13], metadata logging file systems require a log replay[20], and soft updates leave behind leaked blocks and inodes that must be reclaimed[12]. Log-structured file systems periodically create self-consistent checkpoints, but the process of creating a checksum is too expensive to happen frequently[16, 17].

To see why repair after booting isn't an acceptable approach to consistency, consider the common case in which a bootloader reads the root file system in order to find the files it needs to boot the kernel. If log replay is needed in order to make the file system consistent enough to find those files, then all the recovery code must also be in the bootloader. File systems using soft updates don't have this problem, but they still require repair activity after boot to clean up leaked inodes and blocks[12]. Soft updates are also non-trivial to implement, requiring "detailed information about the relationship between cached pieces of data,"[18] roll-back and roll-forward of parts of metadata, and careful analysis of each file system operation to determine the order that updates should appear on disk[12, 18].

The best way to avoid file system corruption due to system panic or power loss is to keep the data on the disk self-consistent at all times, as WAFL[8] does. To do so, the file system needs a simple way to transition from one consistent on-disk state to another without any window of time when the system could crash and leave the on-disk data in an inconsistent state. The implementation of this needs to be relatively foolproof and general, so that programmers can add new features and fix bugs without having to think too hard about maintaining consistency.

2.5 Immense capacity

Many file systems in use today have 32-bit block addresses, and are usually limited to a few terabytes. As we noted in the introduction, a commodity personal computer can easily hold several terabytes of storage, so 32-bit block addresses are clearly already too small. Some recently designed file systems use 64-bit addresses, which will limit them to around 16 exabytes (2^{64} bytes = 16 EB). This seems like a lot, but consider that one petabyte (2^{50} bytes) datasets are plausible today[7], and that storage capacity is currently doubling approximately every 9–12 months[21]. Assuming that the rate of growth remains the same, it takes only 14 more doublings to get from 2^{50} bytes to 2^{64} bytes, so 16 EB datasets might appear in only 10.5 years. The lifetime of an average file system implementation is measured in decades, so we decided to use 128-bit block addresses.

File systems that want to handle 16 EB of data need more than bigger block addresses, they also need scalable algorithms for directory lookup, metadata allocation, block allocation, I/O scheduling, and all other routine[3] operations. The on-disk format deserves special attention to make sure it won't preclude scaling in some fundamental way.

It may seem too obvious to mention, but our new file system shouldn't depend on `fsck` to maintain on-disk consistency. `fsck`, the file system checker that scans the entire file system[10], has already fallen out of favor in the research community[8, 9, 16, 20], but many production file systems still rely on `fsck` at some point in normal usage.[4] O(data) operations to repair the file system must be avoided except in the face of data corruption caused by unrecoverable physical media errors, administrator error, or other sources of unexpected external havoc.

2.6 Error detection and correction

In the ideal world, disks never get corrupted, hardware RAID never has bugs, and reads always return the same data that was written. In the real world, firmware has bugs too. Bugs in disk controller firmware can result in a variety of errors, including misdirected reads, misdirected writes, and phantom writes.[5] In addition to hardware failures, file system corruption can be caused by software or administrative errors, such as bugs in the disk driver or turning the wrong partition into a swap device. Validation at the block interface level can only catch a subset of the causes of file system corruption.

[3] "Routine" includes operations to recover from an unexpected system power cycle, or any other activity that occurs on boot.

[4] For example, RedHat 8.0 supports ext3, which logs metadata operations and theoretically doesn't need `fsck`, but when booting a RedHat 8.0 system after an unexpected crash, the init scripts offer the option of running `fsck` on the ext3 file systems anyway. File systems using soft updates also still require `fsck` to be run in the background to reclaim leaked inodes and blocks[12].

[5] Phantom writes are when the disk reports that it wrote the block but didn't actually write it.

Traditionally, file systems have trusted the data read in from disk. But if the file system doesn't validate data read in from disk, the consequences of these errors can be anything from returning corrupted data to a system panic. The file system should validate data read in from disk in order to detect many kinds of file system corruption (unfortunately, it can't detect those caused by file system bugs). The file system should also automatically correct corruption, if possible, by writing the correct block back to the disk.

2.7 Integration of the volume manager

The traditional way to add features like mirroring is to write a volume manager that exports a logical block device that looks exactly like a physical block device. The benefit of this approach is that any file system can use any volume manager and no file system code has to be changed. However, emulating a regular block device has serious drawbacks: the block interface destroys all semantic information, so the volume manager ends up managing on-disk consistency much more conservatively than it needs to since it doesn't know what the dependencies between blocks are. It also doesn't know which blocks are allocated and which are free, so it must assume that all blocks are in use and need to be kept consistent and up-to-date. In general, the volume manager can't make any optimizations based on knowledge of higher-level semantics.

Many file systems already come with their own volume managers (VxFS and VxVM, XFS and XLV, UFS and SVM). The performance and efficiency of the entire storage software stack should be improved by changing the interface between the file system and volume manager to something more useful than the block device interface. The resulting solution should be lightweight enough that it imposes virtually no performance penalty in the case of a storage pool containing a single plain device.

2.8 Excellent performance

Finally, performance should be excellent. Performance and features are not mutually exclusive. By necessity, we had to start from a clean slate with ZFS, which allowed us to redesign or eliminate crufty old interfaces accumulated over the last few decades. One key aspect of performance is the observation that file system performance is increasingly dominated by write performance[16, 8]. In general, block allocation algorithms should favor writes over reads, and individual small writes should be grouped together into large sequential writes rather than scattered over the disk.

3 The Zettabyte File System

The Zettabyte File System (ZFS) is a general purpose file system based on the principles described in the last section. ZFS is implemented on the Solaris operating system and is intended for use on everything from desktops to database servers. In this section we give a high level overview of the ZFS architecture. As we describe ZFS, we show how our design decisions relate to the principles we outlined in the last section.

3.1 Storage model

The most radical change introduced by ZFS is a redivision of labor among the various parts of system software. The traditional file system block diagram looks something like the left side of Figure 3. The device driver exports a block device interface to the volume manager, the volume manager exports another block device interface to the file system, and the file system exports vnode operations[6] to the system call layer.

The ZFS block diagram is the right side of Figure 3. Starting from the bottom, the device driver exports a block device to the Storage Pool Allocator (SPA). The SPA handles block allocation and I/O, and exports virtually addressed, explicitly allocated and freed blocks to the Data Management Unit (DMU). The DMU turns the virtually addressed blocks from the SPA into a transactional object interface for the ZFS POSIX Layer (ZPL). Finally, the ZPL implements a POSIX file system on top of DMU objects, and exports vnode operations to the system call layer.

The block diagrams are arranged so that roughly equivalent functional blocks in the two models are

[6] Vnode operations are part of the VFS (Virtual File System interface), a generic interface between the operating system and the file system. An example vnode operation is the `vop_create()` operation, which creates a new file.

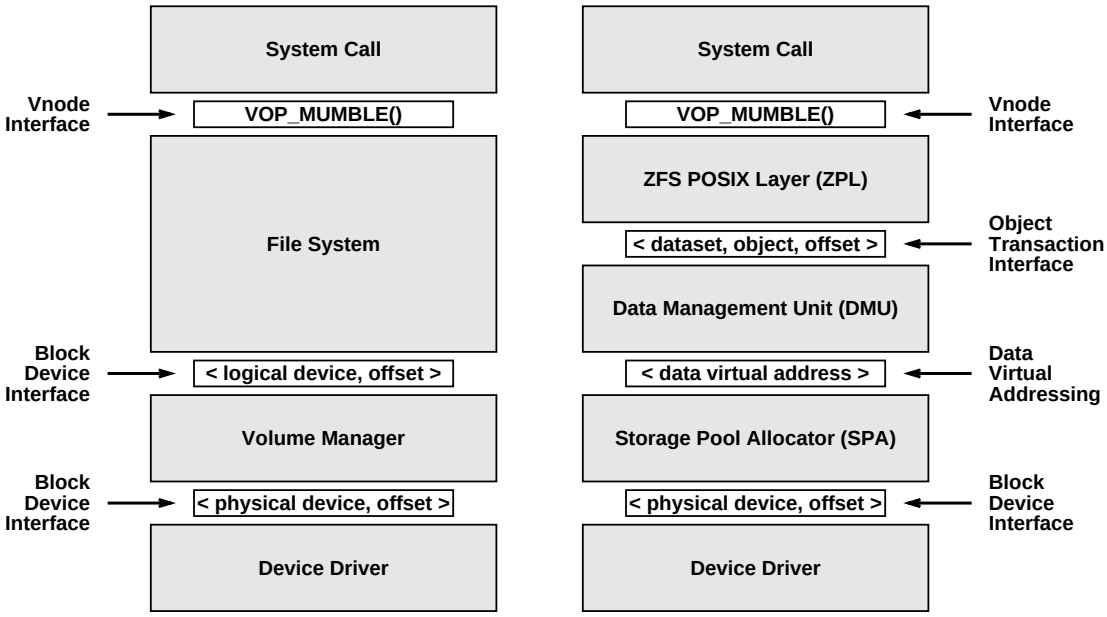

Figure 3: Traditional file system block diagram (left), vs. the ZFS block diagram (right).

lined up with each other. Note that we have separated the functionality of the file system component into two distinct parts, the ZPL and the DMU. We also replaced the block device interface between the rough equivalents of the file system and the volume manager with a virtually addressed block interface.

3.2 The Storage Pool Allocator

The Storage Pool Allocator (SPA) allocates blocks from all the devices in a storage pool. One system can have multiple storage pools, although most systems will only need one pool. Unlike a volume manager, the SPA does not present itself as a logical block device. Instead, it presents itself as an interface to allocate and free virtually addressed blocks — basically, `malloc()` and `free()` for disk space. We call the virtual addresses of disk blocks **data virtual addresses** (DVAs). Using virtually addressed blocks makes it easy to implement several of our design principles. First, it allows dynamic addition and removal of devices from the storage pool without interrupting service. None of the code above the SPA layer knows where a particular block is physically located, so when a new device is added, the SPA can immediately start allocating new blocks from it without involving the rest of the file system code. Likewise, when the user requests the removal of a device, the SPA can move allocated blocks off the disk by copying them to a new location and changing its translation for the blocks' DVAs without notifying anyone else.

The SPA also simplifies administration. System administrators no longer have to create logical devices or partition the storage, they just tell the SPA which devices to use. By default, each file system can use as much storage as it needs from its storage pool. If necessary, the administrator can set quotas and reservations on file systems or groups of file systems to control the maximum and minimum amount of storage available to each.

The SPA has no limits that will be reached in the next few decades. It uses 128-bit block addresses, so each storage pool can address up to 256 billion billion billion billion blocks and contain hundreds of thousands of file systems.[7] From the current state of knowledge in physics, we feel confident that 128-bit addresses will be sufficient for at least a few more decades.[8]

[7] The number of file systems is actually constrained by the design of the operating system (e.g., the 32-bit `dev_t` in the `stat` structure returned by the `stat(2)` family of system calls) rather than any limit in ZFS itself.

[8] Using quantum mechanics, Lloyd[11] calculates that a device operating at sub-nuclear energy levels (i.e., it's still in

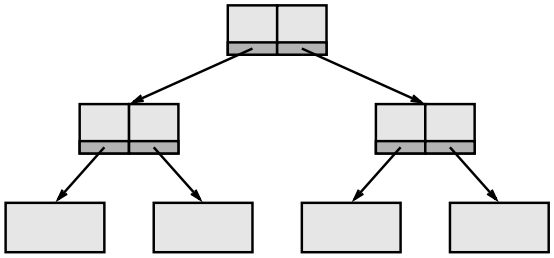

Figure 4: ZFS stores checksums in parent indirect blocks; the root of the tree stores its checksum in itself.

3.2.1 Error detection and correction

To protect against data corruption, each block is checksummed before it is written to disk. A block's checksum is stored in its parent indirect block (see Figure 4). As we describe in more detail in Section 3.3, all on-disk data and metadata is stored in a tree of blocks, rooted at the **überblock**. The überblock is the only block that stores its checksum in itself. Keeping checksums in the parent of a block separates the data from its checksum on disk and makes simultaneous corruption of data and checksum less likely. It makes the checksums self-validating because each checksum is itself checksummed by its parent block. Another benefit is that the checksum doesn't need to be read in from a separate block, since the indirect block was read in to get to the data in the first place. The checksum function is a pluggable module; by default the SPA uses 64-bit Fletcher checksums[6]. Checksums are verified whenever a block is read in from disk and updated whenever a block is written out to disk. Since all data in the pool, including all metadata, is in the tree of blocks, everything ZFS ever writes to disk is checksummed.

Checksums also allow data to be self-healing in some circumstances. When the SPA reads in a block from disk, it has a high probability of detecting any data corruption. If the storage pool is mirrored, the SPA can read the good copy of the data and repair the damaged copy automatically (presuming the storage media hasn't totally malfunctioned).

the form of atoms) could store a maximum of 10^{25} bits/kg. The minimum mass of a device capable of storing 2^{128} bytes would be about 272 trillion kg; for comparison, the Empire state building masses about 500 billion kg.

3.2.2 Virtual devices

The SPA also implements the usual services of a volume manager: mirroring, striping, concatenation, etc. We wanted to come up with a simple, modular, and lightweight way of implementing arbitrarily complex arrangements of mirrors, stripes, concatenations, and whatever else we might think of. Our solution was a building block approach: small modular virtual device drivers called **vdevs**. A vdev is a small set of routines implementing a particular feature, like mirroring or striping. A vdev has one or more children, which may be other vdevs or normal device drivers. For example, a mirror vdev takes a write request and sends it to all of its children, but it sends a read request to only one (randomly selected) child. Similarly, a stripe vdev takes an I/O request, figures out which of its children contains that particular block, and sends the request to that child only. Most vdevs take only about 100 lines of code to implement; this is because on-disk consistency is maintained by the DMU, rather than at the block allocation level.

Each storage pool contains one or more top-level vdevs, each of which is a tree of vdevs of arbitrary depth. Each top-level vdev is created with a single command using a simple nested description language. The syntax is best described with an example: To make a pool containing a single vdev that is a two-way mirror of `/dev/dsk/a` and `/dev/dsk/b`, run the command "`zpool create mirror(/dev/dsk/a,/dev/dsk/b)`". For readability, we also allow a more relaxed form of the syntax when there is no ambiguity, e.g., "`zpool create mirror /dev/dsk/a /dev/dsk/b`". Figure 5 shows an example of a possible vdev tree constructed by an administrator who was forced to cobble together 100 GB of mirrored storage out of two 50 GB disks and one 100 GB disk — hopefully an uncommon situation.

3.2.3 Block allocation strategy

The SPA allocates blocks in a round-robin fashion from the top-level vdevs. A storage pool with multiple top-level vdevs allows the SPA to use **dynamic striping**[9] to increase disk bandwidth. Since a new block may be allocated from any of the top-level vdevs, the SPA implements dynamic

[9] The administrator may also configure "static" striping if desired.

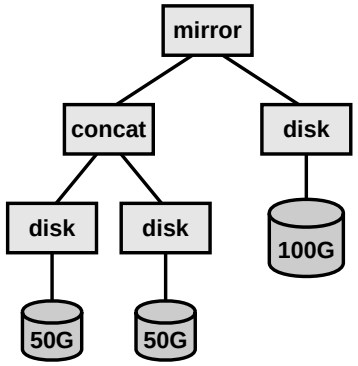

Figure 5: Example vdev with the description `mirror(concat(/dev/dsk/a,/dev/dsk/b),/dev/dsk/c)` where disks `a` and `b` are the 50 GB disks and disk `c` is the 100 GB disk.

striping by spreading out writes across all available top-level vdevs at whatever granularity is convenient (remember, blocks are virtually addressed, so the SPA don't need a fixed stripe width to calculate where a block is located). As a result, reads also tend to spread out across all top-level vdevs. When a new device is added to the storage pool, the SPA immediately begins allocating blocks from it, increasing the total disk bandwidth without any further intervention (such as creating a new stripe group) from the administrator.

The SPA uses a derivative of the slab allocator[2] to allocate blocks. Storage is divided up into metaslabs, which are in turn divided into blocks of a particular size. We chose to use different sizes of blocks rather than extents in part because extents aren't amenable to copy-on-write techniques and because the performance benefits of extents are achievable with a block-based file system[14]. For good performance, a copy-on-write file system needs to find big chunks of contiguous free space to write new blocks to, and the slab allocator already has a proven track record of efficiently preventing fragmentation of memory in the face of variable sized allocations without requiring a defragmentation thread. By contrast, log-structured file systems require the creation of contiguous 512KB or 1MB segments of free disk space[16, 17], and the overhead of the segment cleaner that produced these segments turned out to be quite significant in some cases[19] (although recent work[22] has reduced the cleaning load). Using the slab allocator allows us much more freedom with our block allocation strategy.

3.3 The Data Management Unit

The next component of ZFS is the Data Management Unit (DMU). The DMU consumes blocks from the SPA and exports objects (flat files). Objects live within the context of a particular **dataset**. A dataset provides a private namespace for the objects contained by the dataset. Objects are identified by 64-bit numbers, contain up to 2^{64} bytes of data, and can be created, destroyed, read, and written. Each write to (or creation of or destruction of) a DMU object is assigned to a particular transaction[10] by the caller.

The DMU keeps the on-disk data consistent at all times by treating all blocks as copy-on-write. All data in the pool is part of a tree of indirect blocks, with the data blocks as the leaves of the tree. The block at the root of the tree is called the **überblock**. Whenever any part of a block is written, a new block is allocated and the entire modified block is copied into it. Since the indirect block must be written in order to record the new location of the data block, it must also be copied to a new block. Newly written indirect blocks "ripple" all the way up the tree to the überblock. See Figures 6–8.

When the DMU gets to the überblock at the root of the tree, it rewrites it in place, in effect switching atomically from the old tree of blocks to the new tree of blocks. In case the rewrite does not complete correctly, the überblock has an embedded checksum that detects this form of failure, and the DMU will read a backup überblock from another location. Transactions are implemented by writing out all the blocks involved and then rewriting the überblock once for the entire transaction. For efficiency, the DMU groups many transactions together, so the überblock and other indirect blocks are only rewritten once for many data block writes. The threshold for deciding when to write out a group of transactions is based on both time (each transaction waiths a maximum of a few seconds) and amount of changes built up.

The DMU fulfills several more of our design principles. The transactional object-based interface provides a simple and generic way for any file system

[10] A transaction is a group of changes with the guarantee that either of all of the changes will complete (be visible on-disk) or none of the changes will complete, even in the face of hardware failure or system crash.

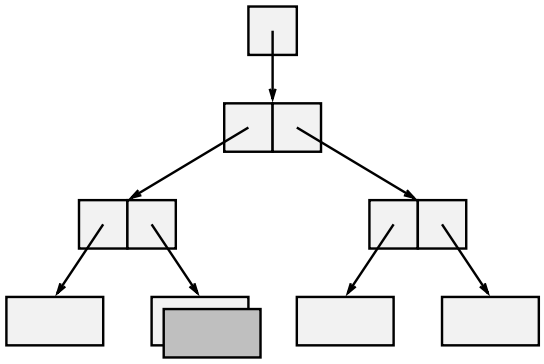

Figure 6: Copy-on-write the data block.

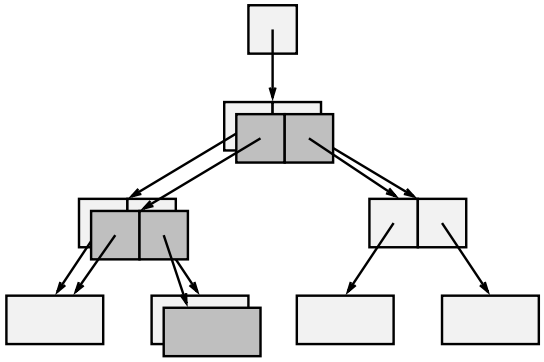

Figure 7: Copy-on-write the indirect blocks.

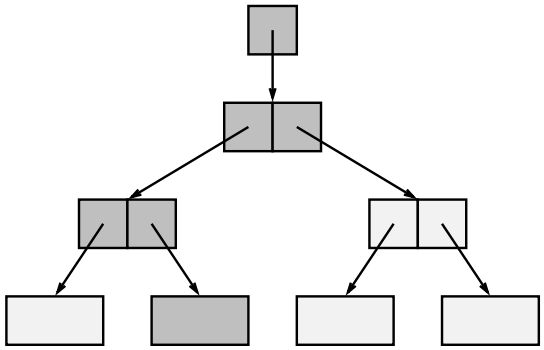

Figure 8: Rewrite the überblock in place.

implemented on top of it to keep its on-disk data self-consistent. For example, if a file system needs to delete a file, it would (1) start a transaction, (2) remove the directory entry by removing it from the directory's object, free the object containing the on-disk inode, and free the object containing the file's data, *in any order*, and (3) commit the transaction.[11] In ZFS, on-disk consistency is implemented exactly once for all file system operations.

The DMU's generic object interface also makes dynamic allocation of metadata much easier for a file system. When the file system needs a new inode, it simply allocates a new object and writes the inode data to the object. When it no longer needs that inode, it destroys the object containing it. The same goes for user data as well. The DMU allocates and frees blocks from the SPA as necessary to store the amount of data contained in any particular object.

The DMU also helps simplify administration by making file systems easy to create and destroy. Each of the DMU's objects lives in the context of a dataset that can be created and destroyed independently of other datasets. The ZPL implements an individual file system as a collection of objects within a dataset which is part of a storage pool. Many file systems can share the same storage, without requiring the storage to be divided up piecemeal beforehand or even requiring any file system to know about any other file system, since each gets its own private object number namespace.

3.4 The ZFS POSIX Layer

The ZFS POSIX Layer (ZPL) makes DMU objects look like a POSIX file system with permissions and all the other trimmings. Naturally, it implements what has become the standard feature set for POSIX-style file systems: `mmap()`, access control lists, extended attributes, etc. The ZPL uses the DMU's object-based transactional interface to store all of its data and metadata. Every change to the on-disk data is carried out entirely in terms of creating, destroying, and writing objects. Changes to individual objects are grouped together into transactions by the ZPL in such a way that the on-disk structures are consistent when a transaction is completed. Updates are atomic; as long as the ZPL groups related changes into transactions correctly, it will never see an inconsistent on-disk state (e.g., an inode with the wrong reference count), even in

[11] This is, of course, only one way to implement a file system using DMU objects. For example, both the inode and the data could be stored in the same DMU object.

the event of a crash.

Instead of using the `mkfs` program to create file systems, the ZPL creates each new file system itself. To do so, the ZPL creates and populates a few DMU objects with the necessary information (e.g., the inode for the root directory). This is a constant-time operation, no matter how large the file system will eventually grow. Overall, creating a new file system is about as complicated and resource-consuming as creating a new directory. In ZFS, file systems are cheap.

The attentive reader will have realized that our implementation of transactions will lose the last few seconds of writes if the system unexpectedly crashes. For applications where those seconds matter (such as NFS servers), the ZPL includes an intent log to record the activity since the last group of transactions was committed. The intent log is not necessary for on-disk consistency, only to recover uncommitted writes and provide synchronous semantics. Placing the intent log at the level of the ZPL allows us to record entries in the more compact form of "create the file '`foo`' in the directory whose inode is 27" rather than "set these blocks to these values." The intent log can log either to disk or to NVRAM.

4 ZFS in action

All of this high-level architectural stuff is great, but what does ZFS actually look like in practice? In this section, we'll use a transcript (slightly edited for two-column format) of ZFS in action to demonstrate three of the benefits we claim for ZFS: simplified administration, virtualization of storage, and detection and correction of data corruption. First, we'll create a storage pool and several ZFS file systems. Next, we'll add more storage to the pool dynamically and show that the file systems start using the new space immediately. Then we'll deliberately scribble garbage on one side of a mirror while it's in active use, and show that ZFS automatically detects and corrects the resulting data corruption.

First, let's create a storage pool named "home" using a mirror of two disks:

```
# zpool create home mirror /dev/dsk/c3t0d0s0 \
  /dev/dsk/c5t0d0s0
```

Now, look at the pool we just created:

```
# zpool info home
Pool                    size   used  avail capacity
home                     80G   409M    80G     1%
```

Let's create and mount several file systems, one for each user's home directory, using the "`-c`" or "create new file system" option:

```
# zfs mount -c home/user1 /export/home/user1
# zfs mount -c home/user2 /export/home/user2
# zfs mount -c home/user3 /export/home/user3
```

Now, verify that they've been created and mounted:

```
# df -h -F zfs
Filesystem  size used avail use%  Mounted on
home/user1   80G   4K   80G   1%  /export/home/user1
home/user2   80G   4K   80G   1%  /export/home/user2
home/user3   80G   4K   80G   1%  /export/home/user3
```

Notice that we just created several file systems without making a partition for each one or running `newfs`. Let's add some new disks to the storage pool:

```
# zpool add home mirror /dev/dsk/c3t1d0s0 \
  /dev/dsk/c5t1d0s0
# df -h -F zfs
Filesystem  size used avail use%  Mounted on
home/user1  160G   4K  160G   1%  /export/home/user1
home/user2  160G   4K  160G   1%  /export/home/user2
home/user3  160G   4K  160G   1%  /export/home/user3
```

As you can see, the new space is now available to all of the file systems.

Let's copy some files into one of the file systems, then simulate data corruption by writing random garbage to one of the disks:

```
# cp /usr/bin/v* /export/home/user1
# dd if=/dev/urandom of=/dev/rdsk/c3t0d0s0 \
  count=10000
10000+0 records in
10000+0 records out
```

Now, diff[12] all of the files with the originals in /usr/bin:

```
# diff /export/home/user1/vacation /usr/bin/vacation
[...]
# diff /export/home/user1/vsig /usr/bin/vsig
```

No corruption! Let's look at some statistics to see how many errors ZFS detected and repaired:

[12] Using a version of diff that works on binary files, of course.

```
# zpool info -v home
                          I/O per sec   I/O errors
vdev  description         read  write  found  fixed
   1  mirror(2,3)             0      0     82     82
   2    /dev/dsk/c3t0d0s0     0      0     82      0
   3    /dev/dsk/c5t0d0s0     0      0      0      0
   4  mirror(5,6)             0      0      0      0
   5    /dev/dsk/c3t1d0s0     0      0      0      0
   6    /dev/dsk/c5t1d0s0     0      0      0      0
```

As you can see, 82 errors were originally found in the first child disk. The child disk passed the error up to the parent mirror vdev, which then reissued the read request to its second child. The checksums on the data returned by the second child were correct, so the mirror vdev returned the good data and also rewrote the bad copy of the data on the first child (the "`fixed`" column in the output would be better named "`fixed by this vdev`").

5 Design tradeoffs

We chose to focus our design on data integrity, recoverability and ease of administration, following the lead of other systems researchers[15] calling for a widening of focus in systems research from performance alone to overall system availability. The last two decades of file system research have taken file systems from 4% disk bandwidth utilization[13] to 90–95% of raw performance on an array of striped disks[20]. For file systems, good performance is no longer a feature, it's a requirement. As such, this paper does not focus on the performance of ZFS, but we will briefly review the effect of our design decisions on performance-related issues as part of our discussion on design tradeoffs in general.

Copy-on-write of every block provides always consistent on-disk data, but it requires a much smarter block allocation algorithm and may cause nonintuitive out-of-space errors when the disk is nearly full. At the same time, it allows us to write to any unallocated block on disk, which gives us room for many performance optimizations, including coalescing many small random writes into one large sequential write.

We traded some amount of performance in order to checksum all on-disk data. This should be mitigated by today's fast processors and by increasingly common hardware support for encryption or checksumming, which ZFS can easily take advantage of. We believe that the advantages of checksums far outweigh the costs; however, for users who trust their disks,[13] ZFS allows them to select cheaper checksum functions or turn them off entirely. Even the highest quality disks can't detect administration or software errors, so we still recommend checksums to protect against errors such as accidentally configuring the wrong disk as a swap device.

Checksums actually speed up ZFS in some cases. They allow us to cheaply validate data read from one side of a mirror without reading the other side. When two mirror devices are out of sync, ZFS knows which side is correct and can repair the bad side of the mirror. Since ZFS knows which blocks of a mirror are in use, it can add new sides to a mirror by duplicating only the data in the allocated blocks, rather than copying gigabytes of garbage from one disk to another.

One tradeoff we did *not* make: sacrificing simplicity of implementation for features. Because we were starting from scratch, we could design in features from the beginning rather than bolting them on later. As a result, ZFS is simpler, rather than more complex, than many of its predecessors. Any comparison of ZFS with other file systems should take into account the fact that ZFS includes the functionality of a volume manager and reduces the amount of user utility code necessary (through the elimination of `fsck`,[14] `mkfs`, and similar utilities).

As of this writing, ZFS is about 25,000 lines of kernel code and 2,000 lines of user code, while Solaris's UFS and SVM (Solaris Volume Manager) together are about 90,000 lines of kernel code and 105,000 lines of user code. ZFS provides more functionality than UFS and SVM with about 1/7th of the total lines of code. For comparison with another file system which provides comparable scalability and capacity, XFS (without its volume manager or user utility code) was over 50,000 lines of code in 1996[20].

6 Related work

The file system that has come closest to our design principles, other than ZFS itself, is WAFL[8], the file system used internally by Network Appliance's NFS

[13] And their cables, I/O bridges, I/O busses, and device drivers.

[14] We will provide a tool for recovering ZFS file systems that have suffered damage, but it won't resemble `fsck` except in the most superficial way.

server appliances. WAFL, which stands for Write Anywhere File Layout, was the first commercial file system to use the copy-on-write tree of blocks approach to file system consistency. Both WAFL and Episode[4] store metadata in files. WAFL also logs operations at the file system level rather than the block level. ZFS differs from WAFL in its use of pooled storage and the storage pool allocator, which allows file systems to share storage without knowing anything about the underlying layout of storage. WAFL uses a checksum file to hold checksums for all blocks, whereas ZFS's checksums are in the indirect blocks, making checksums self-validating and eliminating an extra block read. Finally, ZFS is a general purpose UNIX file system, while WAFL is currently only used inside network appliances.

XFS and JFS dynamically allocate inodes[1, 20], but don't provide a generic object interface to dynamically allocate other kinds of data. Episode appears[15] to allow multiple file systems (or "filesets" or "volumes") to share one partition (an "aggregate"), which is one step towards fully shared, dynamically resizeable pooled storage like ZFS provides. Although it doesn't use true pooled storage, WAFL deserves recognition for its simple and reliable method of growing file systems, since it only needs to grow its inode and block allocation map files when more space is added.

As with any other file system, ZFS borrows many techniques from databases, cryptography, and other research areas. ZFS's real contribution is integrating all these techniques together in an actual implementation of a general purpose POSIX file system using pooled storage, transactional copy-on-write of all blocks, an object-based storage model, and self-validating checksums.

7 Future work

ZFS opens up a great number of possibilities. Databases or NFS servers could use the DMU's transactional object interface directly. File systems of all sorts are easily built on top of the DMU — two of our team members implemented the first usable prototype of the ZPL in only six weeks. The zvol driver, a sparse volume emulator that uses a DMU object as backing store, was implemented in a matter of days. Encryption at the block level will be trivial to implement, once we have a solution to the more difficult problem of a suitable key management infrastructure. Now that file systems are cheap and easy to create, they appear to be a logical administrative unit for permissions, ACLs, encryption, compression and more. ZFS supports multiple block sizes, but our algorithm for choosing a block size is in its infancy. We would like to automatically detect the block size used by applications (such as databases) and use the same block size internally, in order to avoid read-modify-write of file system blocks. Block allocation in general is an area with many possibilities to explore.

[15] Changes of terminology and seemingly conflicting descriptions make us unsure we have interpreted the literature correctly.

8 Status

The implementation of ZFS began in January of 2002. The transcript in Section 4 was created in October 2002, showing that we implemented some of what appear to be the most extravagant features of ZFS very early on. As of this writing, ZFS implements all of the features described in Sections 3.1–3.4 with the exception of removal of devices from the storage pool, quotas and reservations, and the intent log. We plan to deliver all these features plus snapshots in our first release of ZFS, sometime in 2004.

9 Conclusion

Current file systems still suffer from a variety of problems: complicated administration, poor data integrity guarantees, and limited capacity. The key architectural elements of ZFS that solve these problems are pooled storage, the movement of block allocation out of the file system and into the storage pool allocator, an object-based storage model, checksumming of all on-disk blocks, transactional copy-on-write of all blocks, and 128-bit virtual block addresses. Our implementation was relatively simple, especially considering that we incorporate all the functionality of a volume manager (mirroring, etc.). ZFS sets an entirely new standard for data integrity, capacity, and ease of administration in file systems.

References

[1] Steve Best. How the journaled file system cuts system restart times to the quick. http://www-106.ibm.com/developerworks/linux/library/l-jfs.html, January 2000.

[2] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the 1994 USENIX Summer Technical Conference*, 1994.

[3] Aaron Brown and David A. Patterson. To err is human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY '01)*, 2001.

[4] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode file system. In *Proceedings of the 1992 USENIX Winter Technical Conference*, 1992.

[5] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 1999.

[6] John G. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, COM-30(1):247–252, 1982.

[7] Andrew Hanushevsky and Marcia Nowark. Pursuit of a scalable high performance multi-petabyte database. In *IEEE Symposium on Mass Storage Systems*, 1999.

[8] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference*, 1994.

[9] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas. DEcorum file system architectural overview. In *Proceedings of the 1990 USENIX Summer Technical Conference*, 1990.

[10] T. J. Kowalski and Marshall K. McKusick. Fsck - the UNIX file system check program. Technical report, Bell Laboratories, March 1978.

[11] Seth Lloyd. Ultimate physical limits to computation. *Nature*, 406:1047–1054, 2000.

[12] M. Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the 1999 USENIX Technical Conference - Freenix Track*, 1999.

[13] M. Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[14] Larry W. McVoy and Steve R. Kleiman. Extent-like performance from a UNIX file system. In *Proceedings of the 1991 USENIX Winter Technical Conference*, 1991.

[15] David A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science Technical Report, 2002.

[16] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[17] Margo Seltzer, Keith Bostic, Marshall K. McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the 1993 USENIX Winter Technical Conference*, 1993.

[18] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Technical Conference*, 2000.

[19] Margo Seltzer, Keith Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata N. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the 1995 USENIX Winter Technical Conference*, 1995.

[20] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Michael Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Technical Conference*, 1996.

[21] Jon William Toigo. Avoiding a data crunch. *Scientific American*, May 2000.

[22] Jun Wang and Yiming Hu. WOLF—a novel reordering write buffer to boost the performance of log–structured file systems. In *Proceedings of the 2002 USENIX File Systems and Storage Technologies Conference*, 2002.