An Introduction to ZFS – The Next Generation in File System Technology

Introduction

Sun's recent file system addition (ZFS) to the Solaris 10 operating system is considered an innovative, ground-up redesign of the traditional UNIX file systems. Engineers from Sun and the open source community have drawn their ideas for the new file system from some of the best products currently on the market. Network Appliances' snapshots and Veritas' object-based storage management, transactions, and checksumming features influenced ZFS. Further, the engineers involved in the project contributed their own new ideas and expertise to develop a streamlined, cohesive approach to file system design. With ZFS, Sun addresses the important issues of integrity and security, scalability, and difficulty of administration that often impact some of the other UNIX file systems.

ZFS highlights

- Provable data integrity
- Detects and corrects silent data corruption
- Immense capacity (the world's first 128-bit file system)
- Simple administration
- Performance
- RAID-Z

Challenges with existing local file system solutions

- No defense against silent data corruption. Any defect associated with a disk, controller, cable, driver, or firmware can corrupt the data (silently). This is like running a server without ECC memory
- Hard to manage. Administrators have to deal with disk labels, partitions, and volumes. File systems have to be manually manipulated to allow grow or shrink operations
- Lots of file system limits such as file system and/or volume size, file size, number of files per file system, files per directory, number of snapshots
- Performance issues. Some of the issues are the linear-time constraint on create, locks, fixed block sizes, inefficient prefetch operations, slow random writes, or dirty region logging
- File systems and volume managers are sometimes sold as separate products/entities
- Inherent problems in the file system and volume interface can not be fixed

- ***Quote SUN Microsystems: An industry grew up around the file system and volume model***

ZFS objectives and design principles

- Design an integrated system from scratch
- Blow away 20 years of obsolete assumptions
- Pooled storage. This concept completely eliminates the antique notion of volumes. According to SUN, this feature does for storage what the VM did for the memory subsystem
- End-to-end data integrity. A feature that was historically considered as being too expensive. It turned out that it is not, and that the alternative is unacceptable
- Everything in the system is transactional. This keeps the data always consistent on disk, removes almost all constraints on I/O order, and allows for huge performance gains
- Design a RAID solution (RAID-Z) that circumvents the RAID-5 write whole and the (slow) partial RAID-5 stripe write issues

ZFS Data Integrity and Security

Among the most important components of any file system are data integrity and security. It is imperative that information on the disk not suffer from bit rot, silent corruption, or even malicious or accidental tampering. In the past, file systems have had various problems overcoming these challenges and providing reliable and accurate data. Most UNIX based file systems built on older technologies (such as UFS or HFS) and overwrite blocks when modifying in-use data. If a power failure occurs during a write operation, the data is corrupted, and the file system may lose some of the block pointers. To circumvent that issue, the fsck command scans for dirty blocks and reconnects the information where possible. Unfortunately, the fsck operation scans the entire file system, resulting in runtimes measured in minutes or hours to complete (depending on the size of the file system). To accelerate the recovery process, many file systems incorporate a journaling or logging feature. But in the case of a corrupt journal, fsck still has to be invoked to repair the file system. Further, most journaled file systems do not log actual user data (an overhead issue).

For reliability purposes, organization moved to disk or file system mirroring (by utilizing a volume management solution). In case some corruption occurs, one half of the mirror resyncs to the other (even if only a few blocks are in question). Not only is I/O performance degraded during the resync operation, but also the system can not always accurately predict which copy of the data is uncorrupted. Sometimes the system chooses the wrong mirror to trust, and the bad data overwrites the good data. To address the performance issues, some volume managers introduced dirty region logging (DRL). DRL allows to only resync the areas where write operations were in flight at the time of the power loss. This addresses the performance issue, but it does not address the problem with detecting which side of the mirror has the valid data. ZFS tackles these issues by processing transaction-based copy-on-write modifications and constantly checksumming every in-use block in the file system.

Copy-on-write transactional model

The ZFS design represents a combination of a file system and a volume manager. The file system commands require no concept of the underlying physical disks (because of the storage pool virtualization). All of the high-level interactions occur through the data management unit (DMU), a concept that is similar to a memory management unit (MMU) for disks instead of memory (see Figure 1). All of the transactions committed through the DMU are atomic and therefore the data is never left in an inconsistent state. In addition to being a transaction-based file system, ZFS only performs copy-on-write operations. This implies that the blocks containing the data that is in use on disk are never modified. The changed information is written to alternate blocks, and the block pointer to the data in use is only moved once the write transactions are completed. This scenario holds true all the way up the file system block structure to the top block, which is labeled the *uberblock*. In the case that the system encounters a power outage while processing a write operation, no corruption occurs as the pointer to the good data is not moved until the entire write operation completes. It has to be pointed out that the pointer to the data is the only entity that is moved. This eliminates the need for journaling or logging, as well as for an fsck or mirror resync when a machine reboots unexpectedly. To summarize, ZFS uses a copy-on-write, transactional object model. All block pointers within the file system contain a 256-bit checksum of the target block, which is verified when the block is read. Blocks containing active data are never overwritten in place; instead, a new block is allocated, modified data is written to it, and then any metadata blocks referencing it are similarly read, reallocated, and written. To reduce the overhead of this process, multiple updates are grouped into transaction groups, and an intent log is used when synchronous write semantics are required.

End-to-End Checksumming

To avoid accidental data corruption ZFS provides memory-based end-to-end checksumming. Most checksumming file systems only protect against bit rot, as they use self-consistent blocks where the checksum is stored with the block itself. In this case, no external checking is done to verify validity. This style of checksumming will not prevent issues such as:

- Phantom write operations where the write is dropped
- Misdirected read or write operations where the disk accesses the wrong block
- DMA parity errors between the array and server memory (or from the device driver). The issue here is that the checksum only validates the data inside the array
- Driver errors where the data is stored in the wrong buffer (in the kernel)
- Accidental overwrite operations such as swapping to a live file system

With ZFS, the checksum is not stored in the block but next to the pointer to the block (all the way up to the uberblock). Only the uberblock contains a self-validating SHA-256 checksum. All block checksums are done in memory, hence any error that may occur up the tree is caught. Not only is ZFS capable of identifying these problems, but in a mirrored or RAID-Z configuration, the data is self-healing.

ZFS Scalability

While data security and integrity is paramount, a file system has to perform well. The ZFS designers either removed or greatly increased the limits imposed by modern file systems by using a 128-bit architecture, and by making all metadata dynamic. ZFS further supports data pipelining, dynamic block sizing, intelligent prefetch, dynamic striping, and built-in compression to improve the performance behavior.

The 128-Bit Architecture

Current trends in the industry reveal that the disk drive capacity roughly doubles every nine months. If this trend continues, file systems will require 64-bit addressability in about 10 to 15 years. Instead of focusing on 64-bits, the ZFS designers implemented a 128-bit file system. This implies that the ZFS design provides 16 billion times more capacity than the current 64-bit file systems. According to Jeff Bonwick (ZFS chief architect) *"Populating 128-bit file systems would exceed the quantum limits of earth-based storage. You couldn't fill a 128-bit storage pool without boiling the oceans."*

Dynamic Metadata

In addition to being a 128-bit based solution, the ZFS metadata is 100 percent dynamic. Hence, the creation of new storage pools and file systems is extremely efficient. Only 1 to 2 percent of the write operations to disk are metadata related, which results in large (initial) overhead savings. To illustrate, there are no static inodes, therefore the only restriction to the number of inodes that (theoretically) can be used is the size of the storage pool.

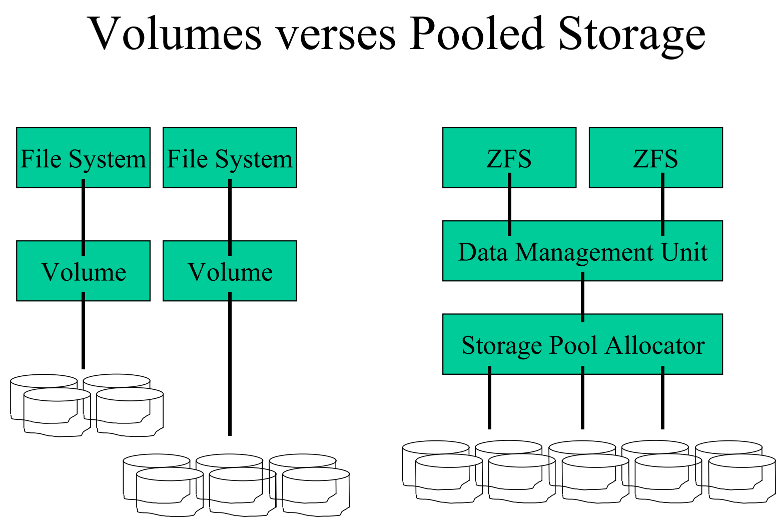
RAID-Z

ZFS implements RAID-Z, a solution that is similar to RAID-5 but uses a variable stripe width to circumvent RAID-5 write hole issue. With RAID-5, write operations are performed to two or more independent devices, and the parity block is written as a component of each stripe. Since these write operations are non-atomic, a power failure between the data and parity transactions results in the possibility of data corruption. Some vendors address this issue with parity region logging (a concept that is similar to dirty region logging, only for the parity disk) or via battery-backed NVRAM. With the NVRAM solution, the data is written into the NVRAM, and afterwards the data and parity writes are made to disk. Finally, the contents of the NVRAM is released. NVRAM is expensive, and can sometimes turn into the bottleneck component. Unlike with RAID-5, all RAID-Z write operations are full-stripe writes, implying that they involve all the disks. There is no read-modify-write overhead, no RAID-5 write hole issue, and no need for a NVRAM based solution.

To reiterate, RAID-Z reflects a data/parity scheme similar to RAID-5, but RAID-Z utilizes a dynamic stripe width based approach. Hence, every block reflects a RAID-Z stripe, regardless of the block-size. This implies that every RAID-Z write represents a *full-stripe write operation*. Combined with the copy-on-write transactional semantics of ZFS, this approach completely eliminates the RAID-5 write hole issue. RAID-Z may further perform more efficiently than a RAID-5 setup, as there is no read-modify-write cycle.

The challenge faced by RAID-Z revolves around the *reconstruction process* though. As the stripes are all of different sizes, an *all the disks XOR to zero* based approach (such as with RAID-5) is not feasible. In a RAID-Z environment, it is necessary to traverse the file system metadata to determine the RAID-Z geometry. It has to be pointed out that this technique would not be feasible if the file system and the actual RAID array were separate products. Traversing all the metadata to determine the geometry may be slower than the traditional approach though (especially if the storage pool is used-up close to capacity). Nevertheless, traversing the metadata implies that ZFS can validate every block against the 256-bit checksum (in memory). Traditional RAID products are not capable of doing this; they simply XOR the data together. Based on this approach, RAID-Z supports a self-healing data feature. In addition to whole-disk failures, RAID-Z can also detect and correct silent data corruption. Whenever a RAID-Z block is read, ZFS compares it against its checksum. If the data disks do not provide the expected checksum, ZFS (1) reads the parity, and (2) processes the necessary *combinatorial reconstruction* to determine which disk returned the bad data. In a 3d step, ZFS repairs the damaged disk, and returns *good data* to the application.

Figure 1: ZFS Storage Pool



Storage pools

Unlike a traditional UNIX file system that either resides on a single device or uses multiple devices and a volume manager, ZFS is implemented on top of virtual storage pools, labeled the *zpools* (see Figure 1). A pool is constructed from virtual devices (*vdevs*), each of which is either a raw device, a mirror (RAID 1), or a RAID-Z group. The storage capacity of all the *vdevs* are available to all of the file systems in the *zpool*. To limit the amount of space a file system can occupy, a quota can be applied, and to guarantee that space will be available to a specific file system, a reservation can be set.

Snapshots

The ZFS copy-on-write model has another powerful advantage: when ZFS writes new data, instead of releasing the blocks containing the old data, it can instead retain them, creating a snapshot version of the file system. ZFS snapshots are created quickly, as all the data comprising the snapshot is already stored. This approach is also space efficient, as any unchanged data is shared among the file system and its snapshots. Writable snapshots (clones) can also be created, resulting in two independent file systems that share a set of blocks. As changes are made to any of the clone file systems, new data blocks are created to reflect those changes, but any unchanged blocks continue to be shared, no matter how many clones exist.

Dynamic striping

Dynamic striping across all devices to maximize throughput implies that as additional devices are added to the *zpool*, the stripe width automatically expands to include them, thus all disks in a pool are used, which balances the write load across them.

Variable block sizes

ZFS utilizes variable-sized blocks of up to 128 kilobytes. The currently available ZFS code allows tuning the maximum block size, as certain workloads do not perform well with large blocks. Automatic tuning to match workload characteristics is contemplated. If compression is enabled, variable block sizes are used. If a block can be compressed to fit into a smaller block size, the smaller size is used on the disk to use less storage and improve IO throughput. This is accomplished though at the cost of increased CPU usage for the compression and decompression operations.

Additional capabilities

- Explicit I/O priority with deadline scheduling
- Globally optimal I/O sorting and aggregation
- Multiple independent prefetch streams with automatic length and stride detection
- Parallel, constant-time directory operations

Limitations

- ZFS is currently not available as a root file system since there is no ZFS boot support. The ZFS Boot project is currently working on adding root file system support.
- ZFS lacks transparent encryption, a la NTFS, although there is an OpenSolaris project underway.
- ZFS does not support per-user or per-group quotas. Instead, it is possible to create user-owned file systems, each with its own size limit. The low overhead of ZFS file systems makes this practical even with many users.

Notes & References

The material presented in this document was compiled by Dominique Heger (dom@fortuitous.com) based on information received from Sun Microsystems, collected from the WWW (developers blogs), and personal experience with ZFS and OpenSolaris.