ZFS and RAID-Z: The Über-FS?

Brian Hickmann and Kynan Shook
University of Wisconsin – Madison
Electrical and Computer Engineering Department
Madison, WI 53706
{bjhickmann and kashook} @ wisc.edu

Abstract

ZFS has been touted by Sun Microsystems as the "last word in file systems". It is a revolutionary file system with several new features that improve its reliability, manageability, and performance. ZFS also contains RAID-Z, an integrated software RAID system that claims to solve the RAID-5 write hole without special hardware. While these claims are impressive, very little information about ZFS and RAID-Z has been published and no work has been done to verify these claims. In this work, we investigate RAID-Z and ZFS by examining the on-disk file layout. We also use a kernel extension to intercept reads and writes to the backing store and use this temporal information to further understand the file system's behavior. With these tools, we were able to discover how RAID-Z lays files out on the disk. We also verified that RAID-Z both solved the RAID-5 write hole and maintains its full-stripe write semantics even when the volume is full or heavily fragmented. Finally, during our testing, we uncovered serious performance degradation and erratic behavior in ZFS when the disk is near full and heavily fragmented.

1. Introduction

ZFS, a file system recently developed by Sun Microsystems, promises many advancements in reliability, scalability, and management. It stores checksums to detect corruption and allows for the easy creation of mirrored or RAID-Z volumes to facilitate correcting errors. It is a 128-bit file system, providing the ability to expand well beyond inherent limitations in existing file systems. Additionally, ZFS breaks the traditional barriers between the file system, volume manager, and RAID controller, allowing file systems and the storage pool they use to grow flexibly as needed.

RAID-Z was also developed by Sun as an integral part of ZFS. It is similar to RAID-5, with one parity block per stripe, rotated among the disks to prevent any one disk from being a bottleneck. There is also the double-parity RAID-Z2, which is similar to RAID-6 in that it can recover from two simultaneous disk failures instead of one. However, unlike the standard RAID levels, RAID-Z allows a variable stripe width. This allows data that is smaller than a whole stripe to be written to fewer disks. Along with the copy-on-write behavior of ZFS, this allows RAID-Z to avoid the read-modify-write of parity that RAID-5 performs when updating only a single block on disk. Because of the variable stripe width, recovery necessitates reconstructing metadata in order to find where the parity blocks are located. Although RAID-Z requires a more intelligent recovery algorithm, this also means that recovery time is much shorter when a volume has a lot of free space. In RAID-5, because the RAID controller is separate from the file system and has no knowledge of what is on the disk, it must recreate data for the entire drive, even if it is unallocated space.

While RAID-Z and ZFS make several exciting claims, very little information is published on ZFS, and even less on RAID-Z. Sun has published a draft On-Disk Specification [3], however it barely mentions RAID-Z. Nothing is published to indicate how data is spread out across disks or how parity is written. Our objective in this paper is to show some of the behavior of RAID-Z that has not yet been documented.

In this paper, we cover a variety of topics relating to the behavior of RAID-Z. We look at file and parity placement on disk, including how RAID-Z always performs full-stripe writes, preventing RAID-5's read-modify-write when writing a single block of data. We discuss the RAID-5 write hole, and show how RAID-Z avoids this problem. Additionally, we investigate the behavior of RAID-Z when full or when free space is fragmented, and we show how performance can degrade for a full or fragmented volume.

2. RAID-Z and ZFS Background Information

There are two pieces of ZFS's and RAID-Z's architecture that are important to this work. The first is ZFS's Copy-on-Write (CoW) transactional model. In this model any update to the file system, including both metadata and data, is written in a CoW fashion to an empty portion of disk. Old data or metadata is never immediately overwritten in ZFS when it is being updated. Once the updates have been propagated up the file system tree and eventually committed to disk, the 512-byte uberblock (akin to the superblock in ext3) is updated in a single atomic write. This ensures that the updates are applied atomically to the file system, solving many consistent update problems. RAID-Z must also follow this model and therefore guarantees to always perform full-stripe writes of new data to empty locations on the disk.

The second relevant part of ZFS to this work is how it manages free space. Since ZFS is a 128-bit file system, it can scale to extremely large volumes. Traditional methods of managing free space, such as bitmaps, do not scale to these large volumes due to their large space overhead. ZFS therefore uses a different structure called a space map to manage free space [9]. Each device in a ZFS volume is split into several hundred metaslabs, and each metaslab has an associated space map that contains a log of all allocations and frees performed within that metaslab. At allocation time, this log is replayed in order to find free space available in the metaslab. The space map is also condensed at this time and written to disk. Space maps save significant space, especially when the metaslab is nearly full; a completely full metaslab is represented by a single entry in the log. Their downside, however, is the time overhead needed to replay the log to find free space and condense it for future use. This overhead becomes noticeable when the disk is nearly full.

3. Methodology

In order to investigate the properties of RAID-Z we used binary files as backing stores, instead of actual disks. This had several advantages. First, we did not need to use separate disks or disk partitions to create our RAID-Z volumes. This allowed us to easily test configurations with different numbers or sizes of backing stores. Second, it also allowed us to easily observe the backing stores while they were in use with a normal binary editor. Finally, the use of binary files as backing stores allowed us to place them on a special disk image and write a kernel extension to intercept all reads and writes to this disk image.

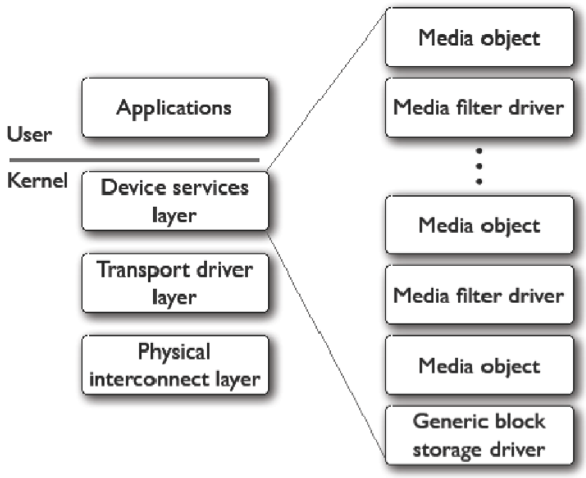

Figure 1: Storage Driver Stack [1]

Our kernel extension fits in the storage driver hierarchy pictured in Figure 1 and is based off of sample code from the Apple Developer Connection website [7]. It acts as a media filter driver, consuming and producing a media object which represents a single write or read on the disk. It sits below the file system, so there is no concept of files or directories. The kernel matches this type of driver based on one of several properties of the media objects. Our kernel extension matched on the content hint key, which is set by the disk formatting utility when the volume is created. We then created a custom disk image with our content hint so that the kernel extension would be called only for traffic to this disk image.

This kernel extension allowed us to easily monitor file system activity to the ZFS backing stores. However, it does have various limitations. Because writing to a file from within the kernel is very difficult due to the potential for deadlock, we used the system logging facility to record the output of our kernel extension. However, this has a limited buffer size, causing some events to get missed during high periods of activity. To avoid this, we only printed messages relevant to our reads and writes, which we recognized from a repeating data pattern. Also, because the driver is in the direct path of all reads and writes, if the extension takes too long to run, it can cause the file system to become too slow to even mount in a reasonable amount of time. In one particular example, we attempted to scan through all the data being written to the disk image, but this caused the mount operation to take longer than an hour.

In order to facilitate directly examining the backing stores, we wrote repeating data patterns to the disk that we could search for using a binary file editor. We always wrote the pattern in multiples of 512-byte chunks to match the logical block size of the "disks" in our system. In order to be able to distinguish the parity blocks from the data blocks in our test files, we used a 32-bit pattern that was rotated and periodically incremented after each write to guarantee that the parity block would always be unique. The use of a 32-bit repeating pattern also allowed us to filter the reads and writes reported by our kernel extension by looking for writes or reads that had a 4-byte pattern repeating for at least 512 bytes.

We performed our testing on two separate environments. The first was Mac OS 10.5 using the recently released ZFS beta version 1.1. We also developed our kernel extension described above for MacOS 10.5, which was enabled during nearly all of our tests. The second environment we used was OpenSolaris 10 Developer Edition running inside a VMWare Workstation 6.0.1 virtual machine. We repeated several of our tests within this environment to try and eliminate any effects that could be caused by implementation-specific details or bugs. This environment did not have the benefit of the kernel extension and so all tests done here simply examined the backing store.

4. Investigating File Placement in RAID-Z

The first aspect of RAID-Z that we investigated was how it places the file data on disk. Here we wanted to see how RAID-Z laid out data and parity on blocks on the disk, how it chose to rotate parity across the disks, and how it guaranteed to always perform full-stripe writes. To investigate this, we tried writing files with known data patterns to the RAID-Z volume and examined the resulting on-disk contents. We experimented with writing small, medium, and large files using the rotating pattern described above in a simple C program using a single write() system call. We also tested using several write() system calls, but this had no effect on our results due to the caching within ZFS. Most of our testing was performed using a 5-disk system, however we also verified our results using 3-disk and 7-disk systems.

4.1. Results

The basic layout of a medium-sized file on a 5-disk RAID-Z system is depicted in Figure 2. In this particular example, we wrote a single file with sixteen 512-byte blocks. As can be seen in the figure, the data for the file is split evenly across the 4 data disks in the

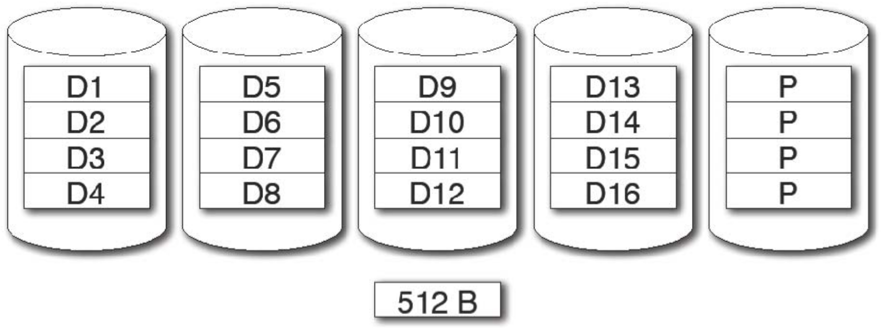

Figure 2: Medium File Layout

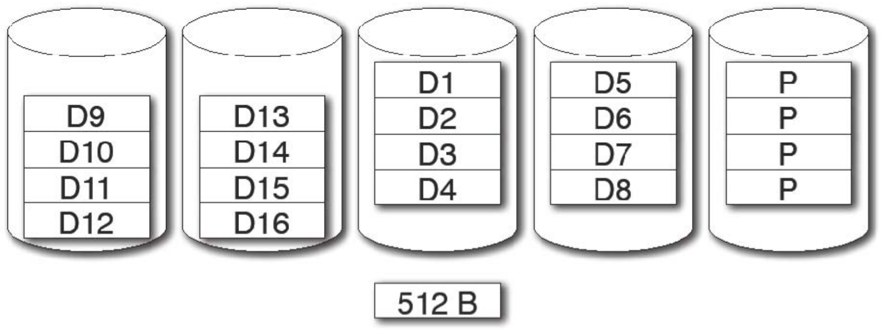

Figure 3: Realistic Medium File Layout

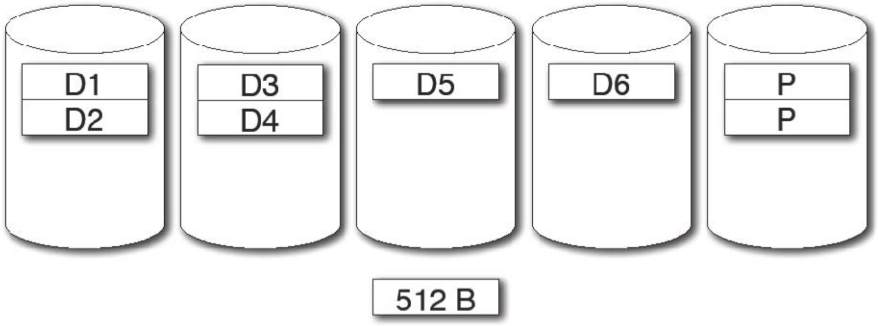

Figure 4: Small File Layout

system. On each disk, the data is written using sequential 512-byte blocks from the file. The final disk contains all of the file's parity blocks. The first parity block protects data blocks 1, 5, 9, 13, the second protects data blocks 2, 6, 10, 14, and so on.

While the file depicted in Figure 2 was written with the first blocks on the first disk, the more common case is for the file to start on one of the other disks in the RAID-Z volume as depicted in Figure 3. In this case, the data is again split equally across all disks and written in a sequential fashion on each disk. However, since we now start writing on a different disk, when the data wraps around to the first disk in the system a 512-byte offset is introduced as shown in the figure. Generally, this small offset area is filled with other data or metadata, and hence the data on these disks is offset by one sector. These small chunks of data are possible in RAID-Z due to its partial-stripe write policy where a write does not need to span all disks in the system. The other diagrams in this paper will ignore this small offset for simplicity.

Next, Figure 4 depicts how a small file with six 512-byte blocks is laid out on a RAID-Z volume. An important feature to notice about this example is that the number of blocks in the file is not a multiple of the number of data disks used and hence illustrates how RAID-Z does partial stripe writes. Here, RAID-Z divides the data blocks as evenly as possible among all data disks and any remaining data blocks are spread evenly across a subset of the data disks. Within each data disk, data is again written as sequentially as possible, with data blocks 1 and 2 being written sequentially to disk 1 in this example. Here, parity block 1 covers data blocks 1, 3, 5, and 6 while parity block 2 covers blocks 2 and 4 only. This second parity block represents a partial stripe write that covers disks 1, 2, and 5 while allowing disks 3 and 4 to be used for a different stripe.

Finally, the results of writing a single large file can be seen in Figure 5. Please note that in this figure, the size of a single block has changed from 512 bytes to 32 kilobytes. Another important observation is that if

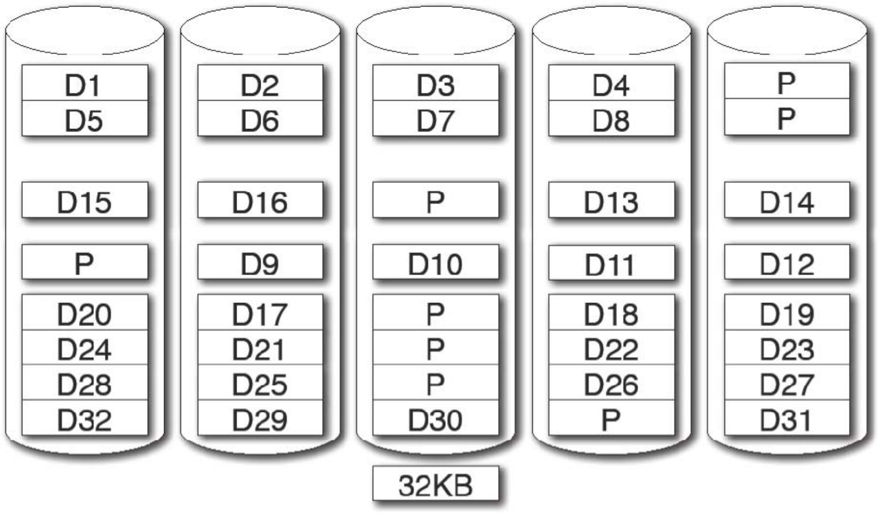

Figure 5: Large File Layout

sufficient data is written to a RAID-Z volume, it will perform writes in multiples of 128KB chunks which coincides with the default logical block size of ZFS. These 128KB chunks are split evenly across the data disks in the system. For 5 disks, this implies that each disk receives a 32KB sequential piece of the chunk. If the 128KB chunk is not evenly divisible across the number of disks, some of the data disks will have pieces that are 512 bytes shorter than other disks in the system in order to handle the remainder from the division across the disks. Also, even though each 128KB chunk is written sequentially, separate 128KB chunks can be scattered across the disk. We were unable to determine any pattern of how these 128KB chunks were distributed on disk, but we presume that 128KB are placed around the disk to ensure that no single metaslab becomes too full.

Another important feature to notice in Figure 5 is that parity is rotated around the disks, but without any distinguishable pattern. We ran several tests to investigate how the parity disk was chosen and determined that it does not directly relate to either the current disk offset or the current file offset. We also examined the source code for RAID-Z found on the OpenSolaris website [8] and the code seems to imply that the parity disk should rotate based on the disk offset of the write. However, there were also in-line comments implying that the algorithm in the code was incorrect and had created an “implicit on-disk format” of rotating the parity disk if a write was larger than 1 MB. Thus the chosen parity disk depends on both the offset of a write and the size of the write. We believe that since data and metadata may be combined into a single write to disk, the location of the parity disks depends on how RAID-Z structures writes internally and hence that is why the parity disk seems to be randomly selected when looking only at the output of data blocks on disk.

4.2. Derived File Layout Algorithm

From the figures above as well as additional testing, we believe we are able to accurately describe RAID-Z’s algorithm for placing file data onto the disk. First, we assume that for each write we know the currently selected parity disk number and which disk and disk offset where we want to start writing. There are three cases based on the number of 512-byte blocks that must be written. After the execution of each case, the current disk, disk offset, and parity disk may need to be updated before performing the next write.

The first case is if the number of blocks to be written is less than the number of data disks in the system, where the number of data disks is the total number of disks in the system minus the one disk for parity. In this case, we check and see if we will reach the current parity disk during our write of this small file. If not, we write the parity to our current disk, otherwise we start writing the file data to the disks in the system in succession and write the parity when we reach the current parity disk.

The second case is when the number of blocks to write is greater than the number of disks in the system but less than 128KB. In this case, we do an integer division of the number of blocks in the write with the number of data disks. This gives us the number of data blocks N we will write sequentially to each disk. We also calculate the remainder from this division and use this number to handle writing out any extra blocks. Next we loop through the disks starting with the currently selected disk. If the current disk matches the current parity disk number, we simply write the parity information for the entire file. If not, we write the data blocks. If there are any blocks left from our remainder calculation, we write N+1 data blocks from the file to the current disk and decrement the remainder number, otherwise we write just N data blocks. If, while looping through the disks, we wrap around to the first disk, we increment the current disk address by 512 bytes to avoid overwriting any data that was written previously.

The final case is when the write is larger than 128KB. Here we write the first 128KB split as evenly as possible across the data disks in the system using an algorithm similar to case 2. Once this first 128KB is written we then check the number of blocks that still need to be written and use either case 1, case 2, or case 3 to complete the write.

5. Investigating the RAID-5 Write Hole

Next, we wanted to investigate the claim that RAID-Z solves the so called RAID-5 write hole. The next two sections describe the RAID-5 write hole, our methodology for investigating it with RAID-Z, and our results and conclusions.

5.1. Problem Statement and Methodology

The RAID-5 write hole or consistent update problem is a well known problem with RAID designs. A good description of this problem along with an investigation into the vulnerability of RAID system to this problem can be found in [4]. The problem stems from the inability of RAID systems to update all disks

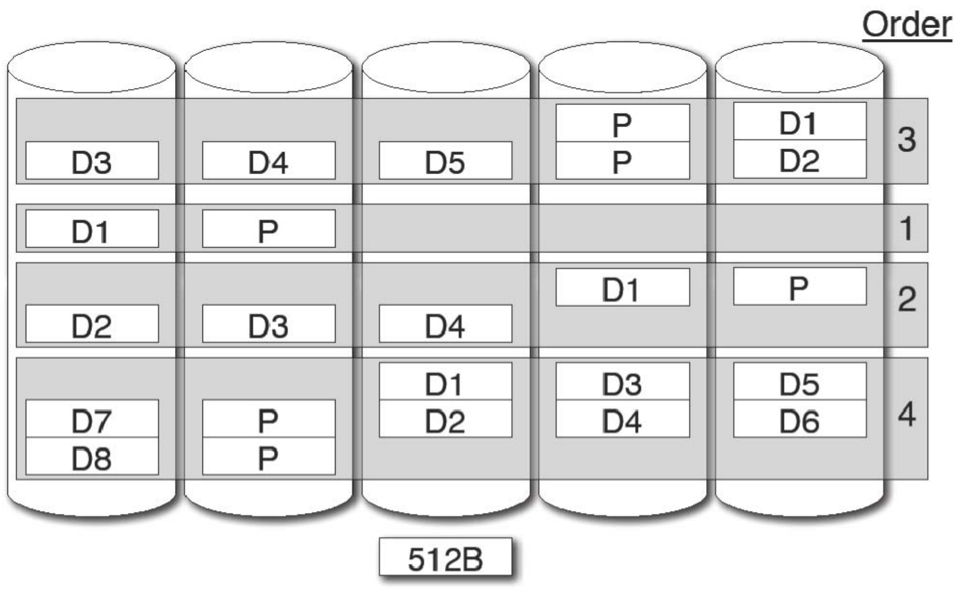

Figure 6: Medium File Layout with Multiple fsync() Calls

in the system atomically. Therefore, if there is a system crash or power failure during an update to a RAID system, the parity information in the RAID system can become inconsistent with the data. To solve this problem, RAID system manufacturers often include expensive hardware such as NVRAM or battery-backed caches to ensure that all issued writes actually reach the disk platters in the case of a system crash.

RAID-Z, on the other hand, is supposed to solve the consistent update problem without the need for expensive hardware solutions. It does so by leveraging the Copy-on-Write (CoW) transactional semantics of ZFS. RAID-Z will write all the needed data, metadata, and parity information to new locations on disk using full-stripe writes, never performing a read-modify-write as is done in RAID-5. Once this data has reached the disk, the uberblock is then written to disk. Since the uberblock is only a single 512-byte structure that is written to only one device in the system, this can be done atomically, also making the updated data live in a single atomic action. If we have a system crash at any point before the uberblock is written, then the file system is still consistent because the updated data did not overwrite live data.

To verify that RAID-Z maintains the necessary CoW semantics and always performs full-stripe writes, we ran tests to observe how RAID-Z handled the CoW semantics of ZFS. To perform this investigation, we added additional fsync() calls in between calls to write() in our C program. The program also waited for 30 seconds after each fsync() call to ensure we bypassed any internal buffering within ZFS. We ran this program for various file sizes to ensure that full-stripe write and the CoW semantics were maintained in all cases.

5.2. Results and Conclusions

The results from one of our experiments can be seen in Figure 6. In this experiments, we called fsync() after writing each 512-byte block of an eight block file. The figure shows only four of the CoW copies for brevity. Region 1 shows the copy of the file that resulted from calling fsync() after writing only one data block, region 2 is after writing blocks one through four, region 3 is after writing blocks one through five, and region 4 is the final live version of the data placed on disk after writing all eight data blocks.

In both the example above and in our other tests, RAID-Z always maintained its full-stripe write semantics even when fsync() is called often, using partial stripes to handle writes smaller than the number of disks. Also, we always found all of the copies of the data that would be expected given the CoW semantics of ZFS. These tests, as well as information published about ZFS [2,3], indicate that RAID-Z appears to solve the consistent update problem without the need for expensive hardware solutions.

6. Investigating Disk Fragmentation and Full Disk Behavior

Finally, we wanted to investigate how RAID-Z maintained its guarantee of full-stripe writes when the disk was heavily fragmented or nearly full. In order to see how RAID-Z handled a fragmented file system, we first created a new empty RAID-Z volume. We then wrote several thousand small files to the disk, usually 512 bytes or 1 KB in size. Once these files had been copied onto the volume, we then went back and deleted a widely-scattered subset of these files. This left many small holes of free space throughout the file system. We then attempted to write several larger files to the volume to investigate how ZFS maintained its full-stripe write semantics. We repeated this experiment for different size volumes, different numbers of small files, and different levels of fullness.

6.1. Fragmentation Results

The primary result from our tests is that RAID-Z does in fact maintain full-stripe writes in all of our fragmentation tests by reordering the data on the disks. Using our kernel extension, we were able to observe that RAID-Z rewrote the previously fragmented data to different locations on disk in order to create space for the new write. In our tests, RAID-Z always created holes in multiples of 128KB, the logical block size of the volume, and wrote the new data using full-stripe writes to these locations.

In order to investigate if there was a threshold after which this cleaning was done, we performed a similar experiment in which we started with a fragmented volume and slowly filled it with files a few megabytes in size. After writing each large file we examined the output of our kernel extension to see if cleaning had been performed. We performed this experiment for a few different volume sizes.

As the result of these experiments, we determined that there is no single threshold point after which cleaning is performed. The cleaning process was always performed when space for the new file was being allocated; we never observed the cleaning process working in the background while the disk volume was idle. There was very little pattern as to when cleaning was performed and appeared to happen regardless of how full the disk was, although it was more likely to happen as disk got near full. It appears that cleaning is performed if the metaslab where space is allocated is heavily fragmented, and is not based on how full the disk is.

6.2. Full Disk Results

We also investigated how RAID-Z and ZFS behaved as the disk became full. We first ran a very basic performance test of copying a 100 MB file to both a fragmented and unfragmented volume, varying

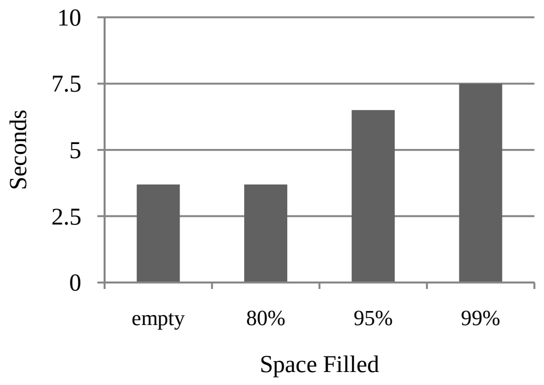

Figure 7: Time to Complete 100MB Copy on an Unfragmented Disk

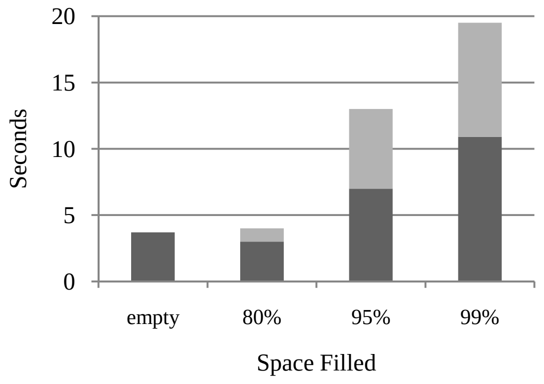

Figure 8: Time to Complete 100MB Copy on a Fragmented Disk

how full the RAID-Z volume was. The results of these tests can be found in Figures 7 and 8. In the unfragmented case in Figure 7 we can see that it can take up to twice as long to perform the copy when the disk is 99% full as compared to when the disk is empty. When the disk is fragmented, however, this can take up to four times as long as when the disk is empty as shown in Figure 8. We believe that the lengthy copy times are most likely due to the need to replay the allocation log in order to find the many small chunks of free space, as well as overheads in moving data to make space for full-stripe writes. In addition to the lengthened copy time, when the disk is fragmented the copy times become extremely erratic as seen in Figure 8, ranging from 10.9 to 19.5 seconds when the disk is 99% full. We believe this erratic behavior stems from the need to move varying amounts of fragmented data around in order to make space for full-stripe writes.

An additional performance problem that we noticed is that failed writes can take an extremely long time to return with a failure message. In one case, we attempted to write a 1 MB file to a nearly full fragmented volume and the file system took 14 minutes to return a failure message. During this time we noticed several tens of megabytes of writes to the disk as the file system tried to find space for the file. Other failed writes, while not as extreme, also took a significant amount of time to fail. Performing a large write on a standard Mac OS 10.5 HFS+ volume with a small amount of fragmented free space caused the write to fail immediately.

Finally, during all of our tests when the disk was nearly full, we also noticed that the free space reported was extremely erratic and often did not match the actual space available. Specifically, any failed write always caused the file system to report that it had no free space available. Despite this, subsequent small writes actually succeeded and reset the free space to be a non-zero value. This made it extremely difficult to judge the actual amount of free space available. Noël Dellofano, a developer on Apple's ZFS team, indicated that this may be a result of assumptions made elsewhere in the OS that doesn't apply to ZFS's pool-based storage model. However, ZFS itself reports more free space than can be filled with user data. We also observed a single write of 50 KB failing repeatedly, while we were able to successfully write six 25 KB files to the file system afterwards. Furthermore, some of these writes did not reduce the amount of space reported to be available. From these tests, we conclude that ZFS has difficulty determining the exact amount of free space available on a volume. This may be a performance optimization, since there is no central data structure keeping track of free space, and replaying hundreds of logs is probably not a reasonable way to determine the free space remaining.

7. Recurring Data Pattern Investigation

During the course of our investigations, we also discovered a recurring data pattern on disk, written almost every time any operation was performed on the disk. The pattern was a series of 16 bytes of 0xFC42 repeating, followed by a single byte of 0xFF. This pattern would usually repeat several times. We never saw more than several hundred bytes of this pattern, however, it was scattered widely across the disk. A very lightly-used test disk used in a 5-disk RAID-Z had over 28 thousand copies of the 0xFC42 pattern. By corrupting these, then scrubbing the volume to restore consistency, we found that about 7 thousand of these were part of active data.

There is no published work that mentions this pattern in ZFS, nor is there any reference to it on the public internet. According to Jeff Bonwick, who leads Sun's ZFS development, this pattern is an artifact of the ZFS compression algorithm he designed, LZJB. In the current implementation, metadata written to disk is always compressed, and a series of zeros is compressed to this particular pattern. He also noted that this is an area that is still being developed, and that they are working on improving the algorithm to compress a long series of zeros more efficiently.

8. Related Work

There is relatively little published academic work on ZFS at this time. However, there are several papers that discuss reliability issues that are interesting in a RAID-Z context.

A presentation [2] written by Sun's CTO of Storage Systems, Jeff Bonwick, gives a good introduction of ZFS and RAID-Z. It covers administration, reliability, and several important concepts about the ZFS design that differ from existing storage systems. The ZFS On-Disk Specification [3] includes many details of ZFS, but is limited to the on-disk format, so the actual behavior of ZFS is not discussed.

In [4], a number of failure modes are enumerated. They discuss some commonly used protections against corruption, including RAID, checksums, and scrubbing, all of which are part of ZFS. They do not specifically address the mechanisms that ZFS uses, however RAID-Z would appear to solve many of the issues related to consistency by writing checksums separately from data, and by replicating important metadata. Sun has reported that they have forced over a million crashes without losing data integrity or leaking a single block [2]. They do not claim zero data loss, however. We expect that occasional data loss from ZFS is possible, such as when data has been written to disk, but the uberblock or other metadata in the tree of pointers leading to the new data has not been rewritten. The CoW mechanisms of ZFS simply prevent such a scenario from corrupting the existing data.

The insertion of kernel-level code to intercept traffic between the file system and disk has been performed by others, such as [5]. This paper also demonstrates how a disk with more knowledge about what the file system is doing can implement additional features. This is similar in many ways to the way ZFS breaks the usual barriers between a file system, volume manager, and RAID controller, sharing information that might not be shared in other systems to enable improved features, performance and reliability.

9. Conclusions

We have uncovered some of the details about the algorithm that ZFS uses to place data and parity on a RAID-Z volume. These details have not been exposed through Sun's specifications or other sources. We verified the use of full-stripe writes for a variety of data sizes and verified that RAID-Z only writes as many blocks of data as are needed using partial stripe writes. Although it does not perform a read-modify-write of parity, it will read a data block smaller than 128 KB if adding data to it to avoid writing too many small fragments on disk. When a volume's remaining free space is too fragmented to allow a whole 128 KB block to be written as a stripe, we found that the file system will clean up the fragments to get larger fragments of free space. Finally, we also discovered that ZFS performance can degrade when the storage pool is nearly full. This performance degradation is especially severe when the free space is very fragmented. Performance on a fragmented volume is also highly variable when full. These observations shed some light on the behaviors of RAID-Z, which were previously largely unpublished.

10. References

[1] Apple Inc., Mass Storage Device Driver Programming Guide. Cupertino, CA : 2007.

[2] J. Bonwick, "ZFS: The last word in file systems," Available: http://www.opensolaris.org/os/community/zfs/docs/zfs_last.pdf

[3] Sun Microsystems, Inc., ZFS On-Disk Specification. Santa Clara, CA : 2007.

[4] A. Krioukov, L. Bairavasundaram, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. "Parity Lost and Parity Regained," Proceedings of FAST '08: 6th USENIX Conference on File and Storage Technologies, to be published February 2008.

[5] M. Sivathanu, V. Prabhakaran, F. Popovici, T. Denehy, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. "Semantically-Smart Disk Systems," Proceedings of FAST '03: 2nd USENIX Conference on File and Storage Technologies, pp. 73-88, March 2003.

[6] T. Denehy, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. "Journal-guided Resynchronization for Software RAID," Proceedings of FAST '05: 4th USENIX Conference on File and Storage Technologies, pp. 87-100, December 2005.

[7] Apple Developer Connection, "SampleFilterScheme" Cupertino, CA: Apple Inc., 2006. Available: http://developer.apple.com/samplecode/SampleFilterScheme/

[8] Sun Microsystems, Inc., OpenSolaris Community Website, "ZFS Source Tour". Available: http://www.opensolaris.org/os/community/zfs/source/

[9] J. Bonwick, "Space Maps," Available: http://blogs.sun.com/bonwick/entry/space_maps