Reliability Analysis of ZFS

Asim Kadav Abhishek Rajimwale
University of Wisconsin-Madison
kadav@cs.wisc.edu abhi@cs.wisc.edu

Abstract

The reliability of a file system considerably depends upon how it deals with on-disk data corruption. A file system should ideally be able to detect and recover from all kinds of data corruptions on disk. ZFS is a new filesystem that arrives almost a generation after the introduction of other desktop filesystems like ext and NTFS and makes strong claims about its reliability mechanisms.In this paper we examine how ZFS deals with on disk data corruptions. We use the knowledge of on disk structures of ZFS to perform corruptions on different types of ZFS objects. This "type aware" corruption enables us to perform a systematic exploration of the data protection mechanism of ZFS from various types of data corruptions. This paper performs 90 experiments on ten different ZFS disk objects. The results of these corruption experiments give us information about the classes of on disk faults handled by ZFS. We also get a measure of the robustness of ZFS and gain valuable lessons on building a reliable and robust file system.

1. Introduction

The integrity and long term availability of the data is of prime importance for any computer system. Unfortunately, the persistent data of a computer system stored on hard disks is susceptible to corruptions. There are various reasons for on disk corruptions and many artifacts of literature discuss such on disk corruptions [8]. Further, bugs in the software stack such as in device drivers and the filesytems themselves also lead to data corruption. ZFS [3] claims to provide robust reliability mechanisms like using checksums in a self validating tree structure. In this project, we aim to perform a type aware corruption analysis of ZFS. In particular, by performing type aware corruption analysis we reduce the large search space of random data corruption possible on disk to a representative subset of interesting faults possible and analyse the file system's policy to deal with them. Section 2 discusses the ZFS organization and on disk structures. It also discusses in brief, their relationships and hierarchy. Section 3 details our corruption framework and methodology. Sections 4 and 5 discuss the analysis and results which is followed by conclusions and future work.

2. Background on ZFS

2.1. Overview

ZFS is an object based filesystem and is very differently organized from most regular file systems. ZFS provides transactional consistency and is always on-disk consistent due to copy-on-write semantics and strong checksums which are stored at a different location than the data blocks.

ZFS uses the concept of a common storage pools for different filesystems on a system instead of using the traditional concept of volumes. The pool can be viewed as a giant tree sized structure comprising of data at its leaf nodes and metadata at its interior nodes. The storage pool allocator which manages this pool is responsible for data integrity amongst its host of other functions. This allocator uses a checksum mechanism for detection of corruption. For each node (of the tree structure) in the storage pool, the 64 bit checksum of its child is stored with its parent block pointer. This mechanism has several advantages like eliminating separate I/O for fetching checksums and fault isolation by separating data and checksum. It also gives a mechanism to store checksum of checksums because of the inherent tree nature of the storage pool. ZFS uses 64-bit second order Fletcher checksums for all user data and 64-bit fourth order Fletcher checksums for all metadata. Using this checksum mechanism, ZFS is able to detect bit rot, misdirected reads, phantom writes, misdirected writes and some user errors. It is significant here that all blocks are checksummed, since checksums are considered traditionally expensive and ZFS is one of the the first commercial file system to have end to end checksums. In addition to the above mechanisms, ZFS also provides automatic repairs in mirrored configurations and also provides a disk scrubbing facility to detect the latent sector errors while they are recoverable [3]. By providing these comprehensive reliability mechanisms, ZFS claims to eliminate the need of fsck.

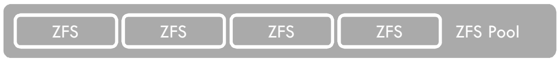

Figure 1. ZFS pool organization

To summarize, ZFS presents a data integrity model consisting of an always consistent on-disk state therby giving "no-window of vulnerability". Another feature of ZFS is that all the data and metadata blocks are compressed in ZFS. While one can disable the data compression, metadata cannot be stored decompressed. This is understandable for this 128 bit, copy-on-write filesystem which duplicates (and even triplicates) all its metadata for reliability purposes.

We now look at some of the features and specification regarding reliability and also review the on-disk structures of ZFS. This helps us in understanding the corruption experiments and results.

2.2. ZFS organization

ZFS presents a unique pooled storage model for mounting multiple instances of filesystem. This pool structure enables ZFS to perform the role of an Logical Volume Manager. This pool structure is represented in Figure 1.

ZFS is a object based transactional file system. Every structure in ZFS space is an object. The write operations on ZFS objects are grouped and are referred to as a transaction group commit. Entire ZFS file system is represented in terms of either objects or object sets. The objects are represented on disk as *dnode_phys_t* structure while an object set is represented as an *objset_phys_t* structure. The object set data pointed to by this *objset_phys_t* structure is an array of *dnode_phys_t* structures. This *dnode_phys_t* structure contains a bonus buffer in the *dn_bonus* field which describes specific information about each different ZFS object. This is analogus to the use of objects and templates in a programming language where each template type(*dnode_phys_t)* can be instantiated to different object types (using *dn_bonus* field). These objects are stored on disk in terms of blocks and these objects connect to one another using block pointers. Block pointers are not traditional pointers in ZFS but are represented by a specific structure called as *blkptr_t*. This pointer gives us a fair idea about what to expect about the block (or sequence of blocks) being read. The pointer tells us the checksum and compression algorithm information of the next block being read. It also tells us the sizes before and after compression of the logical blocks and whether the blocks are gang blocks. Gang blocks provide a tree of runs(block pointers) to store more data in blocks than would be stored by any single contiguous run of blocks. The addresses of the location of the blocks on disk are calculated using a combination of fields in the block pointer. This address is called as the Data Virtual Address(DVA). The block pointer can store upto three copies of the data each pointed by a unique DVA. These blocks are referred to as "ditto" blocks in ZFS terminology. The ability of block pointers to store multiple valid DVA's is referred to as the wideness of the block pointer. This wideness is three for pool wide metadata, two for file system wide metadata and one for data. The wideness for the file system data is configurable.

2.3. Other structures

Another important structure worth describing is the ZAP (ZFS attribute processor) object. The ZAP objects are used to store various attributes to handle arbitary (name, object) associations within an object set. Common attributes include contents of a directory or specific pool wide metadata information.

To organize the above information in filesystem terminology, we summarize as follows. Everything in ZFS is an object. The objects when represented as an array form object sets. Filesystems, volumes, clones and snapshots are each examples of objectsets. A snapshot is point in time copy of the filesystem while a clone is a child copy of a snapshot. A dataset is a collection of multiple objectsets along with spacemaps(free space management information) and snapshot information. A dataset directory groups datasets and provides properties such as quotas, compression etc. Various dataset relationships are also encapsulated using a dataset directory.

2.4. ZFS On-disk Structures

A complete description of individual ZFS On-disk structures can be found at the ZFS website. [6] In this section we use the knowledge of on-disk structures to perform a complete traversal of the on-disk structures that lead to file and directory data. We thus attempt to create a top-to-bottom picture of ZFS on disk data. The illustration in figure 2 represents the ZFS on-disk structure with one filesystem "myfs". The figure illustrates the traversal starting from the vdev (virtual device) labels to the data blocks of files in "myfs". The figure is divided in two portions by the dotted line. The top half represents the pool wide metadata while the lower half represents the file system metadata and data. As discussed in the section 2.2, the pool wide metadata has three copies of the same block. All metadata is always compressed in ZFS while data compression is configurable and is disabled by default. Also, all blocks are checksummed in ZFS and stored with the parent as mentioned in section 2.1. The block pointers contain checksum information which is compared with the calculated checksum of the

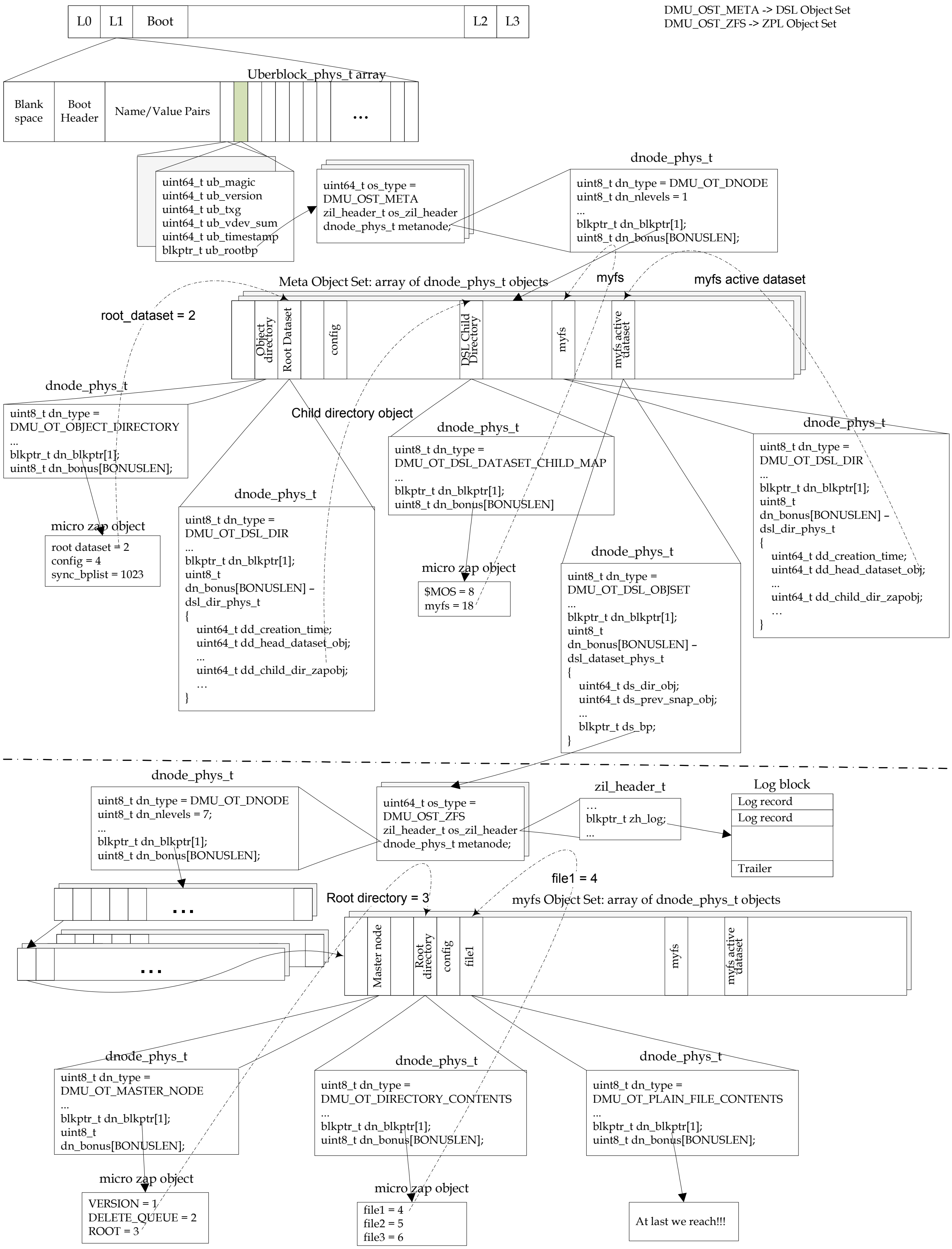

Figure 2. ZFS on disk structures

block being read. Upon checksum mismatch, ZFS resorts to the correct ditto copy of the block.

The storage pool (ZFS formatted disk) contains four copies of vdev labels. These labels are identical in contents and can be interchangabily used to verify pool contents. These vdev labels are the only contents in the pool that have fixed location on-disk (two labels at the starting blocks of the disk and two labels towards the end) and do not have copy on write semantics. ZFS performs a two staged transactional update to ensure on-disk consistency. To overwrite any label data, the even labels are updated first followed by update of the odd labels. To access the pool information, we need to locate the "active" uberblock. The active uberblock is the latest copy of the uberblock. Semantically, uberblock is like a ext3 superblock, with the difference that uberblock contains information about the entire pool and not just one file system. Each vdev contains an array of 128 uberblocks of size 1K each. ZFS locates the latest uberblock using the transaction group number(txg number). The uberblocks are located in the uberblock array at the txg number mod 128 location. The uberblock contains a pointer to the metaobject set (MOS). The MOS contains pool wide information for describing and managing relationships between and properties of various object sets. There is only one MOS per pool pointed to by the block pointer in the uberblock. The MOS also contains the pointer to ZIL or the ZFS Intent Log which contains the transaction records for ZFS's intent logging mechanism.

To traverse towards the filesystem metadata, we must read the contents of the object directory. The object directory is always in a fixed location and in the second element of the dnode array. The object directory is a ZAP object and contains references to various other objects which can be located by traversing through the object references given in this ZAP object. As indicated in Figure 2, common object references in the object directory ZAP include the following :

a)*root_dataset* - This is a pointer to the root dataset directory. This dataset directory in contains references to all top level datasets in the pool.

b)*config* -This is a name/value list describing the pool vdev configuration.

c)*sync_bplist* - This object contains the list of block pointers that need to be freed during the next transaction.

The root dataset directory contains further child dataset directories representing different file systems in the pool. The root dataset directory always contains a default dataset directory called as $MOS. Along with this it also contains a dataset directory for our file system "myfs'. From this "myfs" dataset directory we locate its active dataset. The active dataset object contains a block pointer which points to the "myfs" *objset_phys_t structure*.

As we move towards the file system metadata the "myfs" *objset_phys_t* strucutre points to several layers of indirect objects which eventually lead to a large logical object set array. There are two redundant copies of this filesystem object set. We now start traversing this objectset in a similar fashion as we traversed the meta object set.The second entry in this object set array contains the master node object. The ZAP here gives us a reference(in "myfs" objectset) to the root directory. The is the root directory of the "myfs" filesystem. The root directory ZAP object can be traversed to find further child directories and files in the "myfs" file system. The file objects contain the block pointers to their corresponding data blocks.

To summarize, ZFS offers layers of object based indirection which gives sufficient opportunity to perform corruption on this file system. ZFS offers significant reliability features like end to end checksum, concept of wideness and ditto blocks. The metadata compression also restricts our type aware corruption scope. The copy-on-write semantics always helps ZFS maintain an on-disk consistent state. We now describe our corruption framework to corrupt the above mentioned ZFS structures.

3. Methodology

We now describe the corruption framework that we developed for performing the analysis followed by a description of the corruption experiments.

3.1. Corruption Framework

We introduce a pseudo device driver called the "corrupter" driver which is a standard Solaris layered driver exposing the LDI (Layered Driver Interface). This pseudo layered driver creates two minor nodes in the /devices tree. One device node is used for block I/O while the other has exposes a raw character device necessary to perform corruption related ioctl calls.

We create a storage pool on the block device minor node exposed by the "corrupter". The corrupter thus interposes between the ZFS Virtual devices (VDEVs) and the disk driver beneath. The kernel interface of the LDI exposed by the "corrupter" driver is called in by the VDEVs to perform I/O for the pool. The "corrupter" in turn calls into the actual disk driver using LDI for performing the I/O.

We have also developed an application called the "analyzer" which uses the device nodes exposed by the driver. The "analyzer" uses the block device minor node for reading the on-disk ZFS data. It reads the on-disk structures and uses the knowledge of these structures to selectively corrupt different types of ZFS on-disk objects. It presents the on-disk picture to the user who can then instruct the "analyzer" to selectively corrupt objects. For corrupting an on-disk object, the "analyzer" finds the physical disk block and

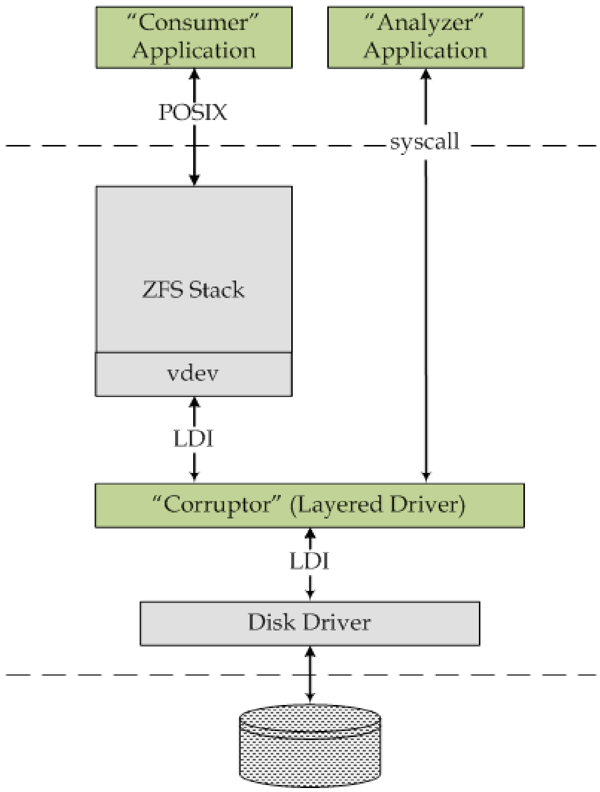

Figure 3. Corruption analysis framework

offset to corrupt. It then uses the raw character device node to call into the driver via an ioctl. The ioctl passes a structure to the driver containing the physical block, offset, new-value for corruption and the driver adds this to its corruption table. For performing the corruption, the driver always perform a check on reads and looks for a match in its corruption table. In case of a match, the driver modifies the given offset with the new value and sends this modified block to the caller.

We use an application as the "consumer" which calls into ZFS using the POSIX filesystem APIs. For different filesystem operations performed by the "consumer" such as reads and writes, the corrupter driver injects the faults in the blocks read by ZFS. It does this corruption only once for each block. To simulate errors on disk, the driver performs corruption only once for the selected block. When writes are performed back to the identified blocks, the corrupter driver monitors these writes and doesn't further corrupt them on reads. We then analyze the behavior of ZFS to recover from faults injected by our driver. To analyse this behavior, we see the return values, system logs and also trace the system calls. Figure 3 describes this corruption framework developed.

3.2. Corruption Experiments

We must first briefly look at the setup and assumptions used to perform the experiments. In our setup, we have used ZFS version 6 in a Solaris 10 virtual machine on VMWare server. This is used only to facilitate the experimentation process and does not interfere with any of the results. The experiments bear the same significance as on a real hardware. We have simplified the storage pool by using a single disk device. So the SPA layer deals with a single vdev and does not employ any replication or redundancy at the disk device level. This is representative of the common case where commodity systems use a single hard disk and avoid any device replication to minimize storage cost. At the file system level, ZFS uses compression on all its metadata while data compression for files and directories is an optional feature. In our experiments we have disabled compression for file system data. Further, we have used the ZFS default configuration for ditto blocks. ZFS uses 3 ditto blocks for pool level metadata, 2 ditto blocks for file system level metadata and no replication for file system data.

We use a type aware fault injection mechanism where the analyzer application presents the user with the on-disk picture consisting of all ZFS objects traversed on the disk to read a specific file in a mounted file system. It facilitates the selection of a specific object for corruption by the user. The analyzer uses the block device minor node of the corrupter and reads the on disk ZFS structures. It then presents a summary of all the objects traversed and the corresponding physical device block numbers to the user. Once the user selects an object for corruption, the analyzer application uses the raw character device node to inform the driver about the specific physical block on disk. It uses an ioctl to send a structure to the corrupter informing it about the specific block, offset in the block and the new value to use for corruption. As an example, the application can ask the driver to corrupt the nth block by changing the value of the kth byte to zero. For all the ZFS metadata including the pool and the file system level metadata, the corruption can be effectively viewed as a block level corruption on the compressed metadata. In the experiments we performed, we corrupted the compressed ZFS metadata objects and analyzed the behaviour of ZFS in response to these corruptions. For file and directory data, we corrupted the uncompressed data blocks on disk. In all cases, we tried to corrupt random offsets in the physical blocks occupied by the selected object. The values to use of corruption are not relevant; we used the corrupted value as zero.

The following is a list of the different ZFS objects on which the corruption experiments were performed along with the number of corruption tests performed (in parantheses).

- vdev labels (5)

- uberblocks (10)
- MOS object (*objset_phys_t*) (10)
- MOS Object set array (10)
- File System object (*objset_phys_t*) (10)
- Indirect objects to FS object set data (10)
- FS Object set array (10)
- FS directory data (10)
- File data (10)
- ZFS Intent Log (5)

We performed a total of 90 corruption tests on the 10 objects listed. Each set of tests are performed and the corresponding observations are explained in section 4 .

4. Experimental Analysis

The set of tests performed and their results have been explained below in terms of the class of objects corrupted.

4.1. Vdev Labels

4.1.1. Corrupting the labels

There are 4 vdev labels in each physical vdev in the storage pool. These are named as L0, L1, L2 and L3 and are self checksummed. Each label update causes these labels to be overwritten in place. ZFS uses a 2 stage transaction update for the labels. The even labels (L0 and L2) are written out to disk in a single atomic transaction followed by the odd labels (L1 and L3). We performed corruptions on different vdev labels. On performing corruptions to vdev labels, ZFS does not detect these corruptions. We unmount and mount the file system ("myfs") in order to make ZFS read the labels on disk. On a mount, ZFS is able to read the correct vdev label (one with the correct checksum) as long as a correct vdev label exists on disk. However on performing corruptions on all 4 vdev labels and then unmounting and remounting the file system, ZFS is unable to recover from this error and causes a panic. It is peculiar that ZFS does not use the correct the vdev label to fix the corrupted labels even on a transaction group commit. We are not sure when these labels are written out to disk.

4.1.2. Corrupting the uberblock

Uberblocks are stored in a 128 slot array in each of the 4 vdev labels. In our tests for the uberblocks, we performed the following experiments -

- Corrupting the active uberblock in 1, 2, 3 and all 4 vdev label(s).
- Corrupting a large number of uberblocks (including the active one) in 1,2,3 and all 4 vdev label(s).
- Performing the above corruptions when file system is mounted and when file system is unmounted.

We first performed the corruptions with "myfs" mounted. We observed that as per the specification, uberblocks are never overwritten. A new modified version of the uberblock is written in the uberblock array whenever a txg commit occurs. The corruption experiments did not cause any failure in ZFS due to the copy-on-write property. Even on corrupting all copies of uberblocks, ZFS was oblivious to the corruptions because it uses the in-memory copy of the uberblock. In order to cause ZFS to access the corrupted on-disk copy of uberblock, we attempted to unmount the "myfs" file system from the pool. The unmount causes a txg commit and a new copy of the uberblock is written to the disk using the correct in-memory copy. This uberblock becomes the active uberblock because it is written out in the latest txg commit. In this way, the old corrupted uberblock is never accessed. Instead of unmounting "myfs", we tried to create a new file in the myfs file system. This causes exactly the same behavior because it causes a txg commit to happen and ZFS writes a correct copy of the uberblock to the disk using its in-memory copy.

Next, we unmounted "myfs" file system from the pool and then performed the corruptions. After performing each corruption we mounted the file system to check how ZFS reacts. In all test cases, ZFS remained oblivious to the corruptions because the in-memory copy of the active uberblock is always present as long as the storage pool exists. Even after unmounting "myfs" and corrupting all the copies of uberblocks, ZFS does not detect any error on a mount because, it still uses the in-memory active uberblock. The next commit that follows puts a correct copy of this in-memory uberblock on disk.

4.2. Meta Object Set

4.2.1. Corrupting the MOS *objset_phys_t* object

The uberblock contains a pointer(*blkptr_t*) to a compressed object which is the MOS *object_phys_t*. This pointer has wideness of three which means that there are three ditto blocks containing the same *objset_phys_t* structure of MOS. We performed the following set of corruptions on this object -

- Corrupting 1, 2 and all 3 ditto blocks of this object.
- Performing the above corruptions when file system is mounted and when file system is unmounted.

We first performed the corruptions with "myfs" mounted. All the corruption tests go undetected by ZFS because it uses the in-memory copy of the *object_phys_t* object. On the next txg commit this consistent in-memory copy is used to write new objects to disk. As a side effect, even the on-disk corruption of all three ditto blocks does not cause any error in ZFS. We tried to unmount "myfs" which again causes a txg commit and ZFS writes new objects to disk using its in-memory copy of the *objset_phys_t* object. However, a crash at this stage will cause ZFS to read the corrupted objects after reboot, and ZFS would fail to mount the storage pool. Next, we performed the corruptions with myfs unmounted. We observed that ZFS still uses the in-memory copy of the *objset_phys_t* object which is a pool wide object. However, we observed that immediately after a mount ZFS does a txg commit which writes new MOS objects on disk. The corruptions induced cause no errors in ZFS and go undetected because of the in-memory copy.

4.2.2. Corrupting the MOS *dnode_phys_t array*

The MOS *objset_phys_t* object contains a pointer (*blkptr_t*) to the compressed MOS object array. This pointer has wideness of three which means that there are three ditto blocks containing the same MOS object array. We performed similar set of corruptions on this object as we did on the MOS *objset_phys_t* object.

All the results obtained were similar to the results obtained for the set of tests performed on the MOS *objset_phys_t object*.

4.3. File System Object Set

4.3.1. Corrupting the myfs *objset_phys_t* Object

A file system is contained as a dataset. A dataset has an associated object set. A blkptr in dataset points to the object containing the *objset_phys_t* structure for the object set. This blkptr has wideness=2 and the block containing *objset_phys_t* is compresssed. So in effect there are two copies of *objset_phys_t* for myfs which are both compressed. We conducted the following corruptions on the *objset_phys_t* object -

- Corrupting a single copy of this object.
- Corrupting both copies of this object.
- Performing the above corruptions when file system is mounted and when file system is unmounted

We first performed the corruptions when myfs is mounted. As in previous experiment, the corruptions done on the object on disk are not visible to ZFS because it uses the in-memory object it caches. Even the corruption of both the ditto blocks is not detected. On trying to create a new file in myfs or unmounting myfs, a txg commit is done and new objects are written on disk.

Next we performed corruptions on *objset_phys_t* objects when myfs is unmounted. We first corrupt a single copy of the object and mount the myfs filesytem. ZFS does not report this corruption. It silently used the other copy and then the next txg commit causes new objects to be created and new copies for both these objects are created. Next, we corrupt both the ditto blocks of the "myfs" *objset_phys_t* object and try to mount myfs. In this case ZFS, fails to mount the file system and reports an "I/O error" on mount. In short, two single bit errors on disk can cause the file system to be corrupted as is obvious with having two replicas.

4.3.2. Corrupting the indirect block pointers myfs Object Set

ZFS assumes the file system object set to be a large object set and hence creates seven levels of compressed indirect objects which eventually lead to the object set array consisting of the file systems objects like files and directories. The indirect blocks were not present in case of the Meta Object Set because that object set is small enough to be pointed to directly. The *objset_phys_t* structure corresponding to myfs points to indirect objects with block pointers (*blkptr_t*) leading to the object set data. All these pointers have wideness of two. We performed similar set of tests to these indirect objects and obtained the same results as with the "myfs" *objset_phys_t* object.

4.3.3. Corrupting the myfs Object Set dnode_phys_t array

The indirect objects pointed to by "myfs" *objset_phys_t* object further lead to the compressed myfs object set dnode_phys_t array. The indirect objects have pointers with wideness of two. We performed the same set of tests on this object as we did for the "myfs" *objset_phys_t* object and obtained the same results.

4.4. File System Data

The individual objects in the file system are represented by *dnode_phys_t* objects in the file system object set. The dnode_phys_t objects contain pointers (*blkptr_t*) to the disk blocks containing the corresponding file or directory data. The directory data is formatted as a ZAP object containing name value pairs for the files contained in the directory. ZFS treats directories as file system metadata and has 2 ditto blocks for them. However, unlike all other metadata, directory data is not compressed by default. For our corruption tests, we used the default configuration in ZFS which keeps only one ditto block (wideness of *blkptr_t*= 1) for file data, two ditto blocks for directory data. We performed the following tests -

- Corrupting the file data.
- Corrupting the directory data.
- Performing the above corruptions when file system is mounted and when file system is unmounted.

Object Type	Detection	Recovery	Correction
vdev label	Yes/Checksum	Yes/Replica	No
uberblock	Yes/Checksum	Yes/Replica	No/COW
MOS Object	Yes/Checksum	Yes/Replica	No/COW
MOS Object Set	Yes/Checksum	Yes/Replica	No/COW
FS Object	Yes/Checksum	Yes/Replica	No/COW
FS Indirect Objects	Yes/Checksum	Yes/Replica	No/COW
FS Object Set	Yes/Checksum	Yes/Replica	No/COW
ZIL	Yes/Checksum	No	No
Directory Data	Yes/Checksum	No/Configurable	No/COW
File Data	Yes/Checksum	No/Configurable	No/COW

Table 1. Results of the type aware corruption on ZFS.

We first performed the corruptions on file data when myfs is mounted. In this case, ZFS uses the cached blocks and does not detect any error on corrupting the file data. However, on unmounting and subsequent mounting of myfs, ZFS detects the corruption and reports an "I/O error". On performing the corruption with myfs unmounted and then mounting it, ZFS detects and reports an "I/O error".

Next we performed the corruptions on directory data. We performed the corruptions first with myfs mounted. On corrupting a single ditto block of directory data, ZFS is unable to detect the error immediately. However on unmounting and subsequent mounting myfs and then accessing the directory, ZFS detects this as a checksum error but does not do any recovery. It still has a correct ditto block of the directory contents and uses that for further operations. Only the next write to the directory causes a txg commit and new objects are written to the disk. A mount or unmount of the file system without any writes to the directory has no effect and the data remains corrupted. Further, on corrupting both the ditto blocks and unmounting and remounting "myfs", ZFS detects the error as a "checksum error" but is unable recover from it. Further operations on the infected directory lead to further "checksum errors" on the pool and ZFS reports an "I/O error" to the user. Directory deletion is also not possible.

We further perform the directory data corruptions with myfs unmounted. On performing corruptions on a single ditto block of directory data and then accessing the directory after mounting "myfs", ZFS is able to detect the error as a "checksum error" but doesn't recover from it as stated earlier. Corrupting both the ditto blocks also has the same effect as discussed earlier.

4.5. ZFS Intent Log

ZFS uses an intent log to provide synchronous write guarantees to applications. When an application issues a synchronous write, ZFS writes this transaction in the intent log (ZIL) and request for the write returns. When there is sufficiently large data to write on to the disk, ZFS performs a txg commit and writes all the data at once. The ZIL is not used to maintain consistency of on-disk structures; it is only to provide synchronous guarantees. We have done some initial corruption tests on ZIL. ZIL blocks are uncompressed and do not have any ditto blocks. In our experiments, we did not simulate a crash after corrupting the intent log. So the results of these tests are not conclusive. However, our tests suggest that because there is no replication, the corruption of ZIL followed by a crash will cause ZFS to detect the ZIL checksum error on the next mount. This will cause ZFS to skip replaying the log and will result in a bit old, yet consistent on-disk data.

The results of the experiments performed and the observations can be summarized as shown in Table 1.

5. Result Summary

ZFS is able to detect all corruptions performed in the tests because of its use of checksums in *blkptr_t*. On detecting such corruption, ZFS checks for ditto blocks. For metadata, ZFS is able to access the correct copy of the ditto block and hence recover from the error. For file data, there are no ditto blocks by default and ZFS is not able to recover from any errors in it. It flags an I/O error in such cases. Earlier work done on the reliability analysis of file systems in IRON File System [8] describes a taxonomy for reliability of file systems. For detection of errors, ZFS uses redundancy in the from of checksums in parent block pointers, thus incorporating the best level of detection described. For recovery of errors, ZFS again uses redundancy in the form of ditto blocks pointed to by parent block pointers, incorporating the best level of recovery as per the taxonomy.

It is peculiar though that ZFS does not perform any correction immediately when it detects checksum errors. It silently uses the correct copy and waits for a transaction group commit to happen which has copy-on-write semantics, thus causing new correct objects to be written out to the disk.

6. Related work

There has been significant research in failure and corruption detection for file systems and many fault injection methodologies and techniques have been developed [2] [7] . The techniques range from dynamic injection of errors [5] to static and model checking [9] for detection of failure points and scenarios. Our study of ZFS uses type information for fault injection to understand the failure behaviour of file systems. This study of ZFS is related to previous fault injection based failure behavior analyses [8] [1] from ADSL research group in our deparment. These analyses also use type information for fault injection in order to understand the failure behavior of file systems. ZFS also provides a ztest utility to perform various types of tests including data corruption [4]. These tests are aimed at user level functional testing and do not perform a fine grained type aware block corruption as is performed in this paper.

7. Future work

The copy on write method of ZFS to fix errors raises questions on the integrity of snapshots. We look forward to exploring this area in our future studies on reliability analysis of ZFS. We also look forward to explore the striping, mirroring and RAID-Z configurations of ZFS and seek to understand the reliability policies and mechanisms associated with these multiple device configurations.

8. Conclusions

There are some useful conclusions that we can draw from our analysis of ZFS and from the corruption experiments performed. These conclusions can also be viewed as lessons for building reliable file systems.

- Use of one generic pointer object with checksums and replication

  ZFS uses a single type of pointer (blkptr_t) structure for all the on disk objects. This pointer is used to store the checksum of the object being pointed to thus enabling the physical separation of objects from their checksums. The pointer structure also points to ditto blocks i.e. replicas of the object. This whole organization creates a Merkel tree structure for the entire storage pool which makes the file system robust.

- Use of replication and compression in commodity file systems

  Earlier work on file systems [8] has pointed towards the use of replication to provide robustness against disk corruption and the use of compression for achieving high disk density in terms of cost per usable byte. However, commodity file systems have been conservative in the use of these techniques for performance and space considerations. ZFS effectively uses both replication and compression while still providing the performance required of large scale production file systems.

- Copy-on-write as an effective tool

  While copy-on-write is used by ZFS as a means to achieve always consistent on-disk structures, it also enables some useful side effects. ZFS does not perform any immediate correction when it detects errors in checksums of objects. It simply takes advantage of the COW mechanism and waits for the next transaction group commit to write new objects on disk. This technique provides for better performance while relying on the frequency of transaction group commits. In terms of robustness, this technique does provide weaker guarantees than eager correction of errors.

Acknowledgments

We would like to thank Shweta Krishnan for providing initial resources required for this project. We also would like to thank Lakshmi Bairavasundaram for his continued support at various stages in the project. We are also grateful to Remzi Arpaci-Dusseau who has guided this project and also provided the infrastrcuture for carrying out our experiments. Finally, we thank Chao Xie(Asim's office mate) for his continued absence that always gave us an extra desk while working together.

References

[1] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Dependability analysis of virtual memory systems. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 355–364, Washington, DC, USA, 2006. IEEE Computer Society.

[2] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault Injection Experiments Using FIAT. In *IEEE Transactions on Computers*, 1990.

[3] J. Bonwick. ZFS(Zettabyte FileSystem). In *One pager on ZFS: http://www.opensolaris.org/os/community/arc/caselog/2002/240/onepager/*, 2002.

[4] M. Byrne. ZTEST. In *http://www.opensolaris.org/os/community/zfs/ztest/*.

[5] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of Linux Kernel Behavior under Errors, (DSN'03). In *2003 International Conference on Dependable Systems and Networks*, page 459, 2003.

[6] S. M. Inc. ZFS(Zettabyte FileSystem). In *ZFS On-Disk Specification:http://opensolaris.org/os/community/zfs/docs /ondiskformatfinal.pdf/*, 2006.

[7] G. Kanawati, N. Kanawati, and J. Abraham. FERRARI: a flexible software-based fault and error injection system. In *Transactions on Computers.*, 1995.

[8] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 206–220, New York, NY, USA, 2005. ACM.

[9] J. Yang, P. Twohey, B. Pfaff, C. Sar, and D. Engler. EXPLODE: A Lightweight, General Approach to Finding Serious Errors in Storage Systems. In *In OSDI '06, Seattle, WA*, November 2006.