Reliability analysis of ZFS

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

Abstract

The reliability of a file system depends upon how it deals with on disk data corruptions. An ideal file system would 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 FAT. It also makes strong claims about its reliability.In this paper we will 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 100 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

A computer system is defined by the data stored in it. The integrity and long term availability of its data is therefore of prime importance. 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 []. Further, bugs in the software stack such as in device drivers and the filesytems themselves also lead to data corruption []. ZFS[] claims to provide novel and 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 and also evaluate the need of further types reliability analyses. 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.

Background on ZFS
- Overview

ZFS is an object based filesystem and is very differently organized from most regular file systems. ZFS provides transactional consistency and an always on-disk consistency with the help of copy on write semantics and strong checksums which are stored at a different location than the disk 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 stroage pool allocator which manages this pool is responsible for data integrity amongst its host of other functions. This allocator uses a strong checksum mechanism for detection of corruption. For each node (of the tree structure) in the storage pool, the 64 bit checksum of all its children(block pointers) are stored. This mechanism has several advantages like eliminating separate I/O for fetching checksums, 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 to all user data and 64-bit fourth order Fletcher checksums to all metadata. Using this checksum mechanism, ZFS is able to detect and correct bit rot, misdirected reads, phantom writes, misdirected writes and some user errors. In addition to the above mechanism, 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 [**?**]. By providing these comprehensive reliability mechanisms, ZFS claims to eliminate the need of fsck. To summarize, ZFS presents a data integrity model comprising of an always valid on-disk state therby giving "no-window of vulnerability".

We look at some of the features and specification regarding reliability and also take a look on the disk specification of ZFS. This help us in understanding the corruption experiments and results in a better manner. ZFS presents a pooled storage model which is unique. The pool comprises of various filesystems and organizes itself in the form of a tree structure. ZFS, thus also performs the function of an Logical Volume Manager which is significant. ¡picture of pool ?¿ ZFS is transcational based object file system. Every structure in ZFS space is an object. The operations on ZFS objects are grouped and is referred to as an transaction group. Another important feature is that all data and metadata blocks are checksummed. The checksums are stored in the parent blocks of the tree like structure, thereby increasing their reliability. 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. This also rules out the chances of silent corruptions creeping in. Also worth noting is the fact that all modifications to the ZFS structures are copy on write(COW). This always maintains an on-disk consistent state for ZFS. 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.

ZFS organization

Entire ZFS file system is represented in terms of either objects or object sets. The objects are represented on disk as dnode$_phys_tstructurewhileanobjectsetisrepresentedasanarrayofdnodephys[]Thesestructurescontainabonusbufferdescribed$
$pictureofblkptr_t--maybemoreimportantforapointercorruptionpaper.>$

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

To organize the above information in terms of a filesystem or pool, 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. To delete a snapshot, we need to first delete all its associated clones. A dataset is a collection of multiple objectsets along with spacemaps 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.

ZFS Figure

The figure 4 represents the ZFS ondisk structure with one filesystem 'myfs'. The figure illustrates the traversal starting from the vdev (virtual device) labels to the data blocks. The figure is divided in two portions using dotted lines. The top half represents the pool wide metadata while the lower half represents the file system metadata and data. As discussed above the pool wide metadata has three copies of the same block. All metadata is compulsarily compressed in ZFS. Only data compression can be disabled here. Also, all blocks are checksummed in ZFS and stored with the parent. As explained above, the block pointers contain checksum information which is compared with the block being read. Upon checksum mismatch, ZFS reads the shadow copy of the corrupted block.

ZFS On-disk Structures

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. vdev labels are the only contents in the pool that have fixed location ondisk (two on front of disk and two on back) and do not have copy on write semantics. ZFS does two staged transactional updates 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 pool information, we need to locate the latest uberblock. Semantically, uberblock is like linux superblock, with a catch that uberblock contains information about the entire pool and not just one file system. Each vdev contains an array of 128 uberblocks each of size 1K each. ZFS locates the latest uberblock using the transaction group number. The uberblocks are located in vdev in the uberblock array at transaction group number mod 128 th location. The uberblock contains pointer to the root block pointer also known as the metaobject set (MOS). This represents the top of a pool wide mechanism for describing and managing relationships between and properties of various object sets. This is referred to as DataSet and Snapshot layer. There is only one MOS per pool pointed to by the root block pointer. The MOS also contains the pointer to ZIL or the ZFS Intent Log which contains the transaction records for ZFS's intent logging mechanism.

As we traverse towards the filesystem metadata, we move from MOS to object directory. The object directory is always fixed and pointed to by 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 in this ZAP. Common references as indicated by the figure include - a)root$_dataset -$
$Thisisapointertotherootdatasetdirectory.Thisdatasetdirectory(asd$
$Thisisaname/valuelistdescribingthepoolvdevconfiguration.c)sync$
$Thisobjectcontainsthelistofblockpointersthatneedtobefreedduring$
We traverse the myfs object from its reference in the objectset and get a pointer to the myfs active dataset. From this active dataset, we get a blockpointer into the filesystem and we now move towards file system specific metadata. All the above references are stored as objects(dnode$_phys_t)intheobjectsetarraywhichhasthreecopiesinthesy$

As we move towards the file system metadata(object set) from the filesystem's block pointer, we encountered a tree of seven indirect blocks. As our pool and

Column1	Column2	Column3
foo1	1.1	1.2
foo2	3.1	1.2
foo3	3.1	3.3

Table 1. This is an example table.

more importantly filesystem just consisted of few files we found it to be pointing to just one objectset, as pointed to by the left most leaf of the chain of indirect pointers. There are two redundant copies of this filesystem object set. We not start traversing this objectset in a similar fashion as we traversed the pool object set.The second entry in this object set contains the master node object. This master node object contains various properties in its ZAP(much similar to how the second entry in pool's object set contained properties like root$_dataset, configetc.).TheZAPheregivesusareference(infilesystem)forobjectsetoftherootdirectoryre$

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. There metadata corruption also restricts our type aware corruption scope. ZFS is also copy on write, which helps it always maintain an on-disk consistent state. There is also an additional scrub facility provided, which is user initiated and traverses all blocks in the file system and corrects the corrupted blocks based on checksum. In our next section we describe our methodology to test this strong reliability framework followed by results and conclusion. We also describe additional future work based on additional areas of reliability testing based on the knowledge gained from this project.

2. Design

Good structure is the key to a well written report. First introduce the overall design in the first paragraph. Introduce the different aspects to the design. Describe each aspect in separate sub-sections. In a final paragraph summarize the interactions between the subsections and allude to the implementation issues.

2.1. Design piece1

Dummy text.

2.2. Design piece2

Dummy text.

2.3. Design piece3

In this subsection I describe some presentation guidelines on tables. Table 1 shows an example table.

- Table headers must be in bold
- Do not use more than 2 digits of precision
- Use a descriptive caption.

Summary paragraph.

3. Implementation

We develop a pseudo device driver called "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 exposes a raw character device.

We create a storage pool on the block device minor node exposed by the "corruptor". 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 also develop an application called the"analyser" which uses the device nodes exposed by driver. The "analyser" uses the block device minor node for reading the on-disk ZFS data. The "analyser" 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 "analyser" to selectively corrupt objects. For corrupting an on-disk object, the "analyser" finds the physical disk block and 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 table of corruption. For performing the corruption, the driver always perform a check on reads and looks for a match in its table of corruption. In case of a match, the driver modifies the given offset with the new value and sends this corrupted block to the calling process.

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. This behavior is representative of the transient errors on disks. Also, 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

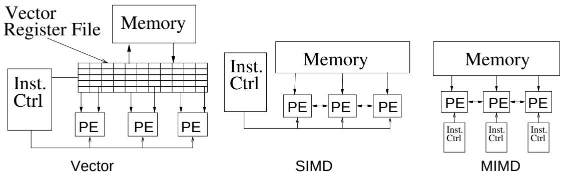

Figure 1. Types of machine organization

analyze the behavior of ZFS to recover from faults injected by our driver. To analyse the behavior of ZFS, we see the return values, system logs and also trace the the system calls.

Corruption Experiments ZFS uses compression on all its metadata while data compression for files and directories is an optional feature. So, any corruption in the ZFS metadata 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 analysed the behaviour of ZFS in response to the corruptions. For file and directory data, we corrupted the uncompressed data blocks on disk. We also do not use ditto blocks for the file and directory data. The following is a list of the different objects on which the corruption experiments were performed:

4. Results

Describe results here.

Content: Your results should be cleanly tied to your design and implementation sections. You should describe the metrics of evaluation in this section. Some aspects of the methodology that you described in your proposal is best presented here, while some parts may best fit in the implementation section. This section should also include processor configurations used, technology models used if applicable, and benchmarks used.

In terms of your experimental results, it is important to describe overall trends. It also important to describe individual specific results and give the reader intuition behind why the results are the way they are. For example, determining some workload has poor cache behavior is a useful result. Even more useful is to tie that back to source code behavior and microarchitecture and explain why it has poor cache behavior.

Structure: Break your results into subsections based on type of analysis you are doing. For some projects these subsections may be very naturally tied to the subsections in the implementation section.

Presentation: Create readable graphs. Use a graph drawing tool that can generate PS/EPS or some form of vectorized graphics and not just raster graphics. Do not use screen capture and insert GIF/JPG/BMP to generate graphs. Figure 2 shows an example of a good graph (in terms of presentation only) and Figure 3 shows an example of a poor graph.

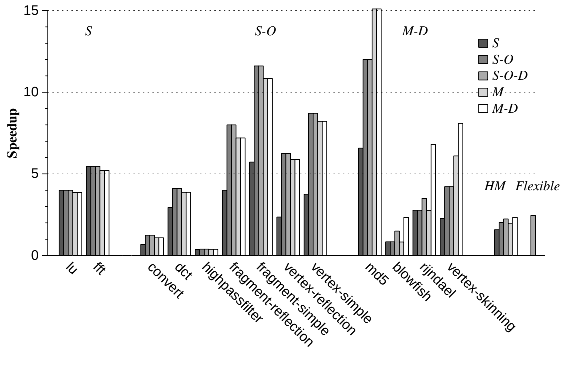

Figure 2. A good graph. Created using a PS generated from jgraph.

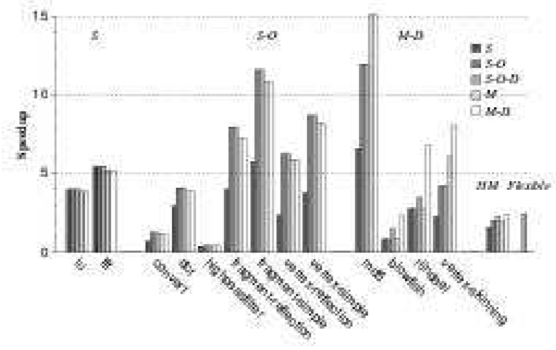

Figure 3. A bad graph. This was created by doing screen capture and converting a JPG to PS. You must use a graph drawing tool that can directly generate PS or EPS.

Conclude your results section with a summary paragraph that brings together individual detailed results you have explained in the previous paragraphs. Point to broad trends and summarize.

5. 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 [?] [?] . The techniques range from dynamic injection of errors [?] to static and model checking errors [?] 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. Our study of ZFS is also related to previous fault injection based failure behavior analyses [3, 19] from ADSL research group. These analyses also use type information for fault injection in order to understand the failure behavior of file systems. Type-aware corruption and some initial NTFS experiments are discussed in our workshop position paper [4].ZFS also provides a ztest utility to perform various types of tests including data corruption and is shipped with the file system. [**?**].

6. Conclusions

First summarize the paper. Include a paragraph on lessons learned. You can include future work as a paragraph in the conclusion section or use a separate section. Include any other concluding thoughts. For example, with respect to writing documents, I want to add the following:

- Spell check your document.
- Print your document, read it at least once, mark corrections using pen or pencil, and make the changes.
- I recommend reading the entire document at least two times.
- Avoid footnotes.
- Avoid parenthetical remarks.

End with a paragraph or a few sentences on the broad impact of the work.

Acknowledgments

We would like to thank Shweta Krishnan for providing initial resources required for this project. We would also like to thank Lakshmi Bairavasundaram for his continued support at various stages in the project. We are also thankful to Remzi who has guided this project and also provided the infrastrcuture for carrying out our experiments.

References

[1] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 281–292, June 2002.

[2] R. Espasa, M. Valero, and J. E. Smith. Out of order vector architectures. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 160–170, December 1997.

[3] J. Hennesy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc, 1996.

[4] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 399–409, June 2003.

[5] B. Moore, A. Padegs, R. Smith, and W. Bucholz. Concepts of th System/370 Architecture. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 282–292, June 1987.

[6] R. M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 22(1):64–72, January 1978.

A. Appendix

This is the appendix. Use an appendix if necessary.