Improving File System Reliability and Availability with Continuous Checker and Repair

Haryadi S. Gunawi

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-88
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-88.html

August 4, 2011

Improving File System Reliability and Availability with Continuous Checker and Repair

Haryadi S. Gunawi
University of California, Berkeley

1 Introduction

Despite the best efforts of the file and storage system community, file system images become corrupt and require repair. In particular, problems with many different parts of the file and storage system stack can corrupt a file system image: disk media, mechanical components, drive firmware, the transport layer, bus controller, OS drivers, and the buggy file system code itself [2, 9, 17, 23].

Traditionally, file systems rely on an offline checker utility, fsck [16], to repair all inconsistencies caused by the corruption. Unfortunately, as the name suggests, offline fsck can only work when the file system is not running. Furthermore, it has a bad reputation of being an extremely slow process [13]. Since file system downtime is usually avoided in reality, offline fsck is run very rarely (*e.g.*, every 30 mounts).

As a result, the occasionality of offline fsck is risky for reliability; corruptions are not detected early in time, and hence, corrupt data may potentially be used by the file system. Worse, as file system data structures are highly inter-connected, a corruption could easily spread to other structures, creating a more corrupt file system. In extreme cases, we have observed that a corruption could spread such that the file system becomes unmountable [17].

Therefore, to improve file system reliability and availability, the file system should be armed with a continuous checker and repair utility. The checker guarantees that the file system does not use corrupt data structures, while the repair restores the file system consistency without the need to shut down the file system. Unfortunately, most of today's file systems lack such a utility [10, 13, 17]. To build one, several challenges must be addressed.

First, to detect a corruption, the consistency of each data structure and all of its fields needs to be verified against the rest of the file system. This is an expensive process since the whole file system must be scanned in order to run the cross-checks. One way to alleviate the cost is to use a fast corruption detection such as checksumming. However, checksumming is often done at a coarse-grained level (*e.g.*, sector- or block-level). Such coarse-grained corruption detection only reports that a block is corrupt, but does not pinpoint which data structures are corrupt within the block. The implication is that all data structures within the corrupt block have to be repaired.

After corrupt data structures have been detected, one can use existing redundancy to repair them on-the-fly. However, the corruption detection and the redundancy could appear in different levels of the storage stack (*e.g.*, checksumming at the file system level, and parity at the RAID-level). Ideally, the file system should be able to cooperate with the storage subsystem to repair the corruption. Unfortunately, the current storage interface prohibits such cooperation (*e.g.*, the file system cannot repair the block from the parity set because the parity blocks are excluded by the interface).

Third, since redundancy is not always available, not all important metadata can be repaired. For example, commonly directory names are not replicated. A corrupt directory name causes its subdirectories untraversable.

Lastly, when all forms of fast repair cannot fix the corruption, a full online fsck is needed. Designing a full online fsck is tricky because it could unsafely modify data structures that are being used by the file system and the application. If not designed carefully, a complex management of in-kernel data-structures is required [13]. Moreover, since the online fsck most likely mimics the procedures in offline fsck, it must also be designed robustly; our recent experience shows that existing offline fsck design is remarkably complex and unreliable [11].

By addressing all the problems above, this paper proposes how a continuous checker and repair should be built, and how the file system has to be redesigned to make it more amenable to our continuous checker and repair.

To detect a corruption in a fine-grained manner, we recommend the use of *data-structure checksumming*, with which the file system can easily retain non-corrupt data structures and repair only the corrupt ones. To repair the corrupt data structures, we propose three solutions to continuous repair. These solutions are stackable, and can be implemented independently. First, the file system should use existing redundancy in the storage stack. To do that, the storage interface has to support *cooperative repair*. Specifically, a small interface is added such that the file system can delegate the repair to the underlying storage subsystem. Second, we introduce a *summary database* which stores partial redundancy of important file system metadata. Metadata copies can be added or removed flexibly depending on the level of availability needed. Moreover, the repair process can be made fast since we only need to scan the compact database rather than the whole

file system. Third, we design a full online fsck that could distinguish safe and unsafe repairs. To achieve that, the file system adds a *repair bit* for each file system data structure. The bit is set when the corresponding data structure is corrupt. With this marker, our online fsck cannot perform unsafe repair (*e.g.*, removal) on data structures with the repair bit off. Hence, complex management of in-use structures is unnecessary.

In the next two sections, Section 2 and 3, we present our design of continuous read checker and repair. Section 4 describes the status of this work. Finally, we discuss related work and conclude in Section 5 and 6 respectively.

2 Continuous Read Checker

The goal of the read checker is to detect corruption caused by low-level failures. One approach is to mimic traditional fsck, that is, the consistency of each data structure and all of its fields is verified against the rest of the file system. For example, to ensure that a given directory entry is not corrupt, all of its fields (inode number, record length, directory name, etc.) must be consistent: the inode that it is referring to must not be a free inode, no other directory points to the same inode, the subdirectory names should be unique within the same directory, etc. This is an expensive process since the whole file system must be scanned in order to run the cross-checks. We believe this is the reason why many commodity file systems skip such a process and potentially use corrupt data [17].

A faster solution is the use of checksumming, which is no longer considered expensive and has been deployed in some of the recent file systems [9, 19, 21]. However, most of the deployed checksumming is done at a coarse-grained level (*e.g.*, sector- or block-level). Such coarse-grained corruption detection is not suitable for data structure repair because it only reports that the block is corrupt, but does not pinpoint the corrupt data structures within the block. For example, in ext3, a 4-KB block contains 32 inodes, and since coarse-grained checksumming does not pinpoint which inodes are corrupt, all the 32 inodes must be repaired. Thus, we advocate that file systems should employ *data-structure checksumming*, with which we can easily retain non-corrupt data structures, and repair only the corrupt ones.

To support data-structure checksumming, the location of each data structure in a block must be at a known offset. Unfortunately, not all existing file systems structure the content in a *location-independent* manner. For example, in ext3, variable-length directory entries are placed next to each other [6], thus, the location of succeeding entries depend on the preceding ones. In this design, adding a checksum to each directory entry is useless because if a directory entry is corrupted, succeeding entries are unreachable and their checksums cannot be checked. Such design hinders a good repair process; the ext3 fsck purges all unreachable directory entries (although they could be valid). As an alternative, one can place directory entries in a location-independent manner such as in ReiserFS. In ReiserFS, directory entries are represented with fixed-size directory headers in the beginning of a directory block, and the variable-length directory names are added in reverse order at the end of the block [5]. In this location-independent design, a checksum can be added in the fixed-length header.

3 Continuous Read Repair

To improve availability, corrupt data structures should be repaired on-the-fly. We propose three solutions for performing continuous repair. These solutions are stackable, that is, if the first solution cannot fix the corruption, the second one can pick it up, and so on. However, each solution can be implemented and used independently.

3.1 Leverage Storage System Redundancy

In the first approach, the repair depends on certain redundancy such as mirroring or parity to exist at any level in the storage stack. In such case, the repair can simply reconstruct the corrupt block from the redundancy.

However, one problem arises: the corruption detection and the redundancy could appear in different levels of the storage stack. For example, checksumming is done at the file system level, but parity at the RAID-level. Ideally, we want the file system to be able to repair the corrupt parity set after detecting a corrupt block within the set. Unfortunately, the current storage interface prohibits *cooperative repair*; the file system cannot cooperate with the underlying storage system because the current storage interface does not expose the redundancy. For example, a RAID-5 system only exposes a linear array of blocks in which parity blocks are excluded. Therefore, we advocate that the thin interface between the storage and the file system should expose more information and control [7, 8], in this case to support a cooperative repair.

To extend the interface, we need to decide which layer should perform the repair: the file system or the storage subsystem. If the repair is done by the file system, there are two drawbacks. First, the interface must be extended such that the file system is able to read all the redundancy set. Second, all kinds of repair algorithm (*e.g.*, parity calculation, erasure-coded) must be implemented in the file system layer. A solution to both drawbacks will break modularity. Thus, we believe the correct solution is to delegate the repair to the underlying storage system.

To delegate the repair to the storage subsystem, we propose a new simple interface, `repair(B)`, where `B` is the block that should be repaired. Before beginning the repair process, the storage subsystem should wait for any ongoing write to complete, and buffer new writes to the redundant set until the repair finishes. The idea is the redun-

DirEntryTable

ino	entryNum	entryIno	entryName
2	1	2	.
2	2	2	..
2	3	35	parent
35	1	35	.
35	2	2	..
35	3	40	child_1
35	4	50	child_2

Table 1: **Summary database for directory entries.** *The table shows the partial content of a directory entry table.* `ino`*,* `entryNum`*,* `entryIno` *and* `entryName` *represent a directory's inode number and its subdirectory's entry number, inode number, and name respectively. Inode number 2 represents the root inode (*`/`*). Entries number 1 and 2 are always reserved for the "*`.`*" and "*`..`*" entries.*

dancy set should not be interfered by any writes. After the repair finishes successfully, the file system can read the block again. We believe this simple interface can be easily introduced to support a powerful cooperative repair.

3.2 Partial Repair with Summary Database

Redundancy is not always available for repairing a corrupt data structure; it may not exist in the storage stack or it could also be corrupt. Fortunately, a corrupt data structure can be repaired from the *implicit redundancy* stored within the file system. For example, suppose we have a `/parent/...` directory path, and the `parent`'s directory block is corrupt such that we cannot traverse to the subdirectories. This corrupt content can be reconstructed by physically scanning all directories and finding all directories whose "`..`" entries point to the `parent` directory. Another example is when we want to reconstruct a corrupt inode. One of the tasks in reconstructing the inode is to ensure that the pointers to its data blocks are valid, specifically, by verifying that the bitmap marks those blocks as used, and no other inodes pointing to the same blocks.

The examples above suggest that repair is feasible even in the absence of explicit redundancy. However, both examples reveal two problems. First, not all important metadata has implicit redundancy (*e.g.*, directory name, inode's time). In such case, the repair can manufacture a safe value [18]. For example, corrupt characters in directory name can be changed to valid ones (*e.g.*, `par#!$` to `par__`), inode time can be reset to current time, etc. Nevertheless, this kind of repair might not be suitable for important metadata such as directory names.

The second problem is more challenging: repair could involve lots of cross-checks that require whole file system scan. In the first example, all directory blocks are scanned in order to find the orphan directories, while in the second, all inodes are scanned to prevent duplicate blocks. This makes continuous repair slow, intrusive, and unattractive.

Our solution to the problems above is to add a small

```
SELECT  S.*
FROM    DirEntryTable P, DirEntryTable S
WHERE   // P.entryIno will be
        // the /parent inode number
        P.ino       = 2             AND
        P.entryName = 'parent'      AND
        // Find the i-th subdir of
        // the /parent directory
        S.ino       = P.entryIno    AND
        S.entryNum  = i
```

Figure 1: **Reconstructing a corrupt subdirectory entry.** *This query returns the i-th subdirectory of the* `/parent` *directory from the summary database.*

summary database managed by the file system. The summary database stores partial metadata redundancy. As an example, the directory entry table shown in Table 1 replicates the directory hierarchy. With the summary database, metadata copies can be added or removed flexibly depending on the level of availability needed. For example, one might want to replicate the directory hierarchy so that users can still traverse the file system in the midst of corruption, or one does not want to replicate any metadata if the underlying storage subsystem already employs a certain redundancy.

The summary database also enables a fast repair since the repair only needs to scan the compact database rather than the whole file system. For example, if the data-structure checksum indicates that the i-th directory entry of the `/parent` directory is corrupt, we can repair it by running the query shown in Figure 1. The query obtains the necessary information that will be used to reconstruct the corrupt directory entry. Furthermore, by writing the repair code in a declarative query language, we found that the repair code can be made robust [11].

A potential drawback of storing extra partial redundancy is that extra writes are added. Depending on the amount of redundancy, the overhead of the extra writes could be negligible. For example, if we only store directory hierarchy in the database, non-directory file operations will not impose any overhead. Furthermore, to improve spatial locality, each record in the summary database is stored closely to the information it describes.

In an extreme and unfortunate case, the summary database itself could be corrupt. In this case, the last and only way to repair the corrupt data structure is to cross-check the whole file system, which is an intrusive solution for the sake of repairing a single data structure. Thus, instead of repairing it directly, the file system maintains a statistic of unrepairable data structures, which will be used as an indicator to run the final approach of continuous repair, *i.e.*, the full online fsck.

3.3 Full Online Fsck

The third approach of repair, a full online fsck, is initiated when the number of unrepairable data structures in

the second approach has exceeded a certain threshold (or when a certain period of time has elapsed).

There are two big challenges in designing an online fsck. First, while the online fsck is running, file system data structures can be in-use (*e.g.*, by open descriptor, for directory cache, etc.). However, fsck has the power to modify any metadata, including metadata removal. This means that the online fsck could be in conflict with the file system and the application. For example, if a `/lost+found` exists as a file, it will be deleted since fsck wants to use that path as a directory. If an application coincidentally created and is currently using the `lost+found` file, the online fsck will be in conflict with the application. This implies that an online fsck should be able to identify the type of repair that can be performed safely on-the-fly. Without such ability, an online fsck will require complex management of in-kernel data-structures, which is one of the big reasons why a full online fsck is hard to design [13].

Second, fsck is complex. For example, the ext3 fsck, written in 20,000 lines of low-level C code, performs 121 repairs and can identify and return 269 different error codes. Its checks and repairs range from simple (*e.g.*, examining individual structures in isolation) to complex ones (*e.g.*, cross-checking all instances of multiple structures at the same time). Hence, a robust fsck is hard to design and implement. In our recent work [11], we confirmed that existing offline fsck has many weaknesses: fsck sometimes performs inconsistent repairs that can corrupt the file system image, fsck also sometimes does not use all available information and can lose portions of the directory tree, and many more. This accentuates that a robust online fsck should also be designed carefully.

To solve the first challenge, our online fsck follows one important rule: it does not perform removal of data structures that could be in-use. We implement the rule by adding a *repair bit* in each of the file system data structures. The bit is set when the corresponding data structure is found corrupt (*i.e.*, the checksum is wrong). Thus, this marker guarantees that the file system can only use non-corrupt data structures. Our online fsck then performs all types of repair (update, addition, and removal) on data structures that have been marked, but could only perform update and addition (but not removal) on those that can be in-use. Without the repair bit, an online fsck cannot distinguish which data structures are safely repairable on-the-fly.

It is possible that our online fsck has to delete an in-use data structure. This implies that its checksum is valid but the content is not consistent with the rest of the file system. This case could only arise due to two factors: a bug in the file system, or a complex failure scenario (*e.g.*, lost write, misdirected write [20]) has occured in the storage subsystem. Although this inconsistency cannot be repaired online, it cannot be ignored. Therefore, our online fsck also keeps another statistic of the unrepairable in-use data structures. This statistic will be used as a better indicator to run the offline fsck, rather than using an arbitrary number (*e.g.*, on every 30 mounts).

To solve the second challenge, we leverage our recent work in creating a more robust offline fsck [11]. The key to our robust offline fsck is to use a high-level declarative language and clearly separate each of the fsck components (scanner, checker, repair, and flusher). In this design, the scanner scans the metadata from the disk and loads them to a temporary database, on which the repair will take place. The checks and repairs are built as a collection of robust declarative queries, similar to the one in Figure 1. We found that a declarative query language is an excellent match for the cross-checks that must be made across the different structures of a file system. Unlike existing offline fsck, the repair is reflected to the temporary database first. After all inconsistencies have been repaired, the flusher reflects all the database updates to their home location.

We design our online fsck similar to our robust offline fsck, but with slight modifications. First, since this process is online, before scanning the disk and loading the temporary database, we wait for ongoing file operations to finish and freeze new ones temporarily. The scanning works similar to a disk scrubbing utility, but it is metadata-driven [3]. It also sets the repair bit of all data structures with incorrect checksums. After all metadata has been scanned and loaded into the temporary database, the halted file operations can proceed, and the repair phase can begin. During the repair, any modification is only reflected to the temporary database rather than to the file system. Since we are keeping two copies of metadata (one each in the file system and the temporary database), we must keep both copies synchronized. Thus, each file system operation is also reflected onto the database. For example, if there is a foreground operation that deletes a directory during the repair process, the directory records in the temporary database will also be removed. Finally, all data structures that have been repaired in the temporary database are reflected to their home location (the summary database is also updated).

4 Status

We plan to build a continuous checker and repair utility for the ext2 file system. Adding a checksum and a repair bit to each of the ext2 data structures should be straightforward. A little more work needs to be done for changing the ext2 directory structure to be location-independent. Extending the storage interface to support a cooperative repair should also be straightforward. In particular, we plan to leverage a previous work done in our group [8]. In that work, the storage interface is extended to support cooperation between file system journaling and RAID syn-

chronization. To implement the summary database and the full online fsck, we will reuse much of the components of our recent work in redesigning the ext2 offline fsck [11]. In that work, we have implemented the scanner, and rewritten all the 121 repairs in SQL. The only things left are to add a lightweight database support (*e.g.*, SQLite [1]) into the file system, and a synchronization support between the file system and the database.

5 Related Work

Our first solution to continuous repair is similar to ZFS [19]. In particular, ZFS performs an online repair from the redundancy stored in RAID-Z [3]. However, the cooperation between ZFS and RAID-Z requires a fully transparent interface; as Bonwick stated, the repair is "impossible if the file system and the RAID array were separate products" [3]. On the other hand, we propose a more generic solution by adding a simple repair interface. Furthermore, ZFS's repair fully depends on existing redundancy; ZFS does not propose a full online fsck by which corruptions are repaired in the absence of redundancy.

Partial online fsck has been proposed before. Chunkfs supports a partial online fsck but only on partitions that can be offline for a while [12]. McKusick suggests a background fsck [15], however, it could only repair simple inconsistencies such as lost resources, but not complex ones such as directory corruption.

To flexibly add partial redundancy, we suggested I/O shepherding in recent research [10]. However, since repairing is the focus of this paper, we believe using the summary database is a better approach; a repair could require lots of cross-checks which are more robust if implemented with database techniques. Integrating a database into the file system is not a new concept. Amino [22] integrates a full database engine to support extensibility and complete ACID properties. We only intend to add a lightweight database that sufficiently supports our purpose.

In this paper, we only discuss continuous read checking, which could not catch file system bugs early in time. Continuous write checking can be performed similar to our online fsck. In particular, for each write, the file system cross-checks the update with the rest of the file system. Since performing such a check for each write imposes a great overhead, the check can be implemented as dynamic distributed probes [14].

6 Conclusion

File system reliability is important, but availability is also highly demanded. It is not uncommon that some systems sacrifice reliability in favor of availability [4]. Such choice should not be an option for file systems. Thus, file systems should be armed with a continuous checker and repair utility. Furthermore, such utility should be carefully designed such that it is robust, fast, and safe. To achieve that, we have proposed three stackable continuous repair solutions, which when combined will increase the file system reliability and availability significantly.

References

[1] http://www.sqlite.org/.

[2] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08*, pages 223–238, San Jose, California, February 2008.

[3] Jeff Bonwick. RAID-Z. http://blogs.sun.com/bonwick/entry/raid_z, November 2005.

[4] Aaron Brown and David A. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *USENIX '00*, pages 263–276, San Diego, CA, June 2000.

[5] Florian Buchholz. The structure of the Reiser file system. http://homes.cerias.purdue.edu/∼florian/reiser/reiserfs.php, January 2006.

[6] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

[7] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *USENIX '02*, pages 177–190, Monterey, CA, June 2002.

[8] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *FAST '05*, pages 87–100, San Francisco, CA, December 2005.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*, pages 29–43, Bolton Landing, NY, October 2003.

[10] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *SOSP '07*, pages 283–296, Stevenson, WA, October 2007.

[11] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. Under Submission to OSDI '08.

[12] Val Henson. The Many Faces of fsck. http://lwn.net/Articles/248180/, September 2007.

[13] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *HotDep II*, Seattle, Washington, Nov 2006.

[14] Emre Kiciman and Benjamin Livshits. Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SOSP '07*, Stevenson, WA, October 2007.

[15] Marshall Kirk McKusick. Running 'fsck' in the Background. In *BSDCon '02*, San Fransisco, CA, February 2002.

[16] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[17] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.

[18] Martin Rinard, Christian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI '04*, San Francisco, CA, December 2004.

[19] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[20] Rajesh Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html, February 2006.

[21] Wikipedia. ext4. en.wikipedia.org/wiki/Ext4, 2008.

[22] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID Semantics to the File System Via Ptrace. *ACM Transactions on Storage (TOS)*, 3(2):1–42, June 2007.

[23] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.