Model-Based Failure Analysis of Journaling File Systems

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

University of Wisconsin, Madison
Computer Sciences Department
1210, West Dayton Street, Madison, Wisconsin

Abstract

We propose a novel method to measure the robustness of journaling file systems under disk write failures. In our approach, we build models of how journaling file systems order disk writes under different journaling modes and use these models to inject write failures during file system updates. Using our technique, we analyze if journaling file systems maintain on-disk consistency in the presence of disk write failures. We apply our technique to three important Linux journaling file systems: ext3, Reiserfs, and IBM JFS. From our analysis, we identify several design flaws and correctness bugs in these file systems, which can cause serious file system errors ranging from data corruption to unmountable file systems.

1 Introduction

Disks fail. Hence, to ensure file system integrity and reliability, modern file systems and storage systems must include internal machinery to cope with such failures.

Unfortunately, the way disks fail is changing. Most traditional systems assume that disks are *fail-stop* [16]; under such an assumption, a disk is either working or it is not, and if not, the failure is easily detectable. However, as disk complexity increases, and as the pressures of time-to-market and cost increase as well, new disk failure modes are becoming more common. Specifically, *latent sector faults* may occur [4], in which a specific block becomes faulty (either in a transient or permanent manner), rather than the disk as a whole. Similarly, disks can *stutter* sometimes, exhibiting transient performance problems [1]. Hence, viewing the disk as either working or not may no longer be appropriate.

In this paper, we investigate how modern file systems cope with this new class of fault on disk writes. Most modern file systems are journaling systems [2, 15, 18, 21]. By logging data in a separate journal before writing them to their fixed locations, these file systems maintain file system integrity despite the presence of crashes.

To analyze such file systems, we develop a novel *model-based* fault-injection technique. Specifically, for the file system under test, we develop an abstract model of its update behavior, *e.g.*, how it orders writes to disk to maintain file system consistency. By using such a model, we can inject faults at various "interesting" points during a file system transaction, and thus monitor how the system reacts to such failures. In this paper, we focus only on write failures because file system writes are those that change the on-disk state and can potentially lead to corruption if not properly handled.

We use this fault-injection methodology to test three widely used Linux journaling file systems: ext3 [21], Reiserfs [15] and IBM JFS [2]. From our analysis, we find several design flaws with these file systems that can catastrophically affect on-disk data.

Specifically, we find that both ext3 and IBM JFS are not designed to handle sector write failures. Under such failures, both of these file systems sometimes commit failed transactions to disk; doing so can lead to serious problems, including an unmountable file system. In contrast, we find that Reiserfs, for the most part, is paranoid about write failures; specifically, Reiserfs crashes the system when a write to the journal fails. By crashing in this manner, Reiserfs ensures that file system integrity is maintained, at the cost of a (potentially expensive) restart. However, in certain configurations, Reiserfs does not abide by its general policy, and can be coerced into committing failed transactions and also can result in a corrupted file system. Further, Reiserfs assumes such failures are transient; repeated failure of a particular block will result in repeated crashes and restarts.

The rest of the paper is organized as follows. First, we give a brief introduction to journaling file systems (§2). Following that, we explain our methodology for analyzing journaling file systems (§3), and then discuss the results of our analysis of ext3, Reiserfs, and JFS (§4). We present related work (§5) and finally, conclude (§6).

2 Background

When a file system update takes place, a set of blocks is written to the disk. Unfortunately, if the system crashes in the middle of the sequence of writes, the file system is left in an inconsistent state. To repair the inconsistency, earlier systems such as FFS and ext2 scan the entire file system and perform integrity checks using fsck [13] before mounting the file system again. This scan is a time-consuming process and can take several hours for large file systems.

Journaling file systems avoid this expensive integrity check by recording some extra information on the disk in the form of a write-ahead log or a journal [6]. Journal writes are followed by a commit block write. Once these writes are successfully committed to the log, they can be transferred to their final, fixed locations on the disk. The process of transferring the writes from the log to the fixed location on disk is called *checkpointing*. If a crash occurs in the middle of checkpointing, the file system can recover the data from the log and write them to their fixed locations. Journaling file systems update a special block called the journal super block periodically to mark the size of the log and the end of checkpointing.

Many modern file systems provide different flavors of journaling, which have subtle differences in their update behavior to disk. We discuss the three different approaches: *data journaling*, *ordered journaling*, and *writeback journaling*. These journaling modes differ from each other by the kind of integrity they provide, by the type of data they write to the log, and the order in which the data is written.

Data journaling provides the strongest data integrity of the three. Every block that is written to the disk, irrespective of whether it is a data or metadata block, is first written to the log. Once the transaction is committed, the journaled data can be written to their fixed file system locations.

Writeback journaling logs only the file system metadata. However, it does not enforce any ordering between data writes and journal writes. Hence, while ensuring metadata consistency, writeback journaling provides no guarantee as to *data consistency*. Specifically, if a file's metadata is updated in-place before its data reaches disk, the file will contain data from the old contents of that data block.

Ordered journaling adds data consistency to writeback mode. It does so by enforcing an ordering constraint on writes, such that the data blocks are written (called ordered data writes) to their fixed locations before the metadata blocks are committed. This ordering constraint ensures that no file system metadata points to any corrupt data.

3 Methodology

In this section, we describe the overall methodology we use for testing the reliability of journaling file systems. Our basic approach is quite simple: we inject "disk faults" beneath the file system at certain key points during its operation and observe its resultant behavior.

Our testing framework is shown in Figure 1(a). It consists of two main components; a device driver called the *fault-injection driver* and a user-level process labeled the *coordinator*. The driver is positioned between the file system and the disk and is used to observe I/O traffic from the file system and to inject faults at certain points in the I/O stream. The coordinator monitors and controls the entire process by informing the driver as to which specific fault to insert, running workloads on top of the file system, and then observing the resultant behavior.

A flow diagram of the benchmarking process is shown in Figure 1(b). We now describe the entire process in detail.

3.1 The Fault-Injection Driver

The fault-injection driver (or just "driver") is a pseudo-device driver, and hence appears as a typical block device to the file system. Internally, it simply interposes upon all I/O requests to the real underlying disk.

The driver has three main roles in our system. First, it must classify each block that is written to disk based on its *type*, *i.e.*, what specific file-system data structure this write represents. We have developed techniques to perform this classification elsewhere [14], and simply employ those techniques herein.

Second, the driver must "model" what the journaling file system above is doing. Specifically, such a model represents the correct sequence of states that a transaction must go through in committing to disk. By inserting failures at specific points within the transaction sequence, we can observe how the file system handles different types of faults and better judge if it correctly handles the faults we have injected.

Third, the driver is used to inject faults into the system. These faults are specified to occur at various state transitions (as based on the journaling model) in the I/O stream.

3.2 The Coordinator

The coordinator monitors the entire benchmarking process. It first inserts the fault-injection driver into the Linux kernel. Then, the coordinator constructs the file system, passes a fault specification to the driver, spawns a child process to run the workload, and looks for errors.

Errors can manifest themselves in numerous locales, so we must log all such errors and have the coordinator collate them. Specifically, the child process may receive errors from the file system, the driver may observe errors in the sequence of state transitions, and the coordinator itself must look through system logs to look for other errors reported

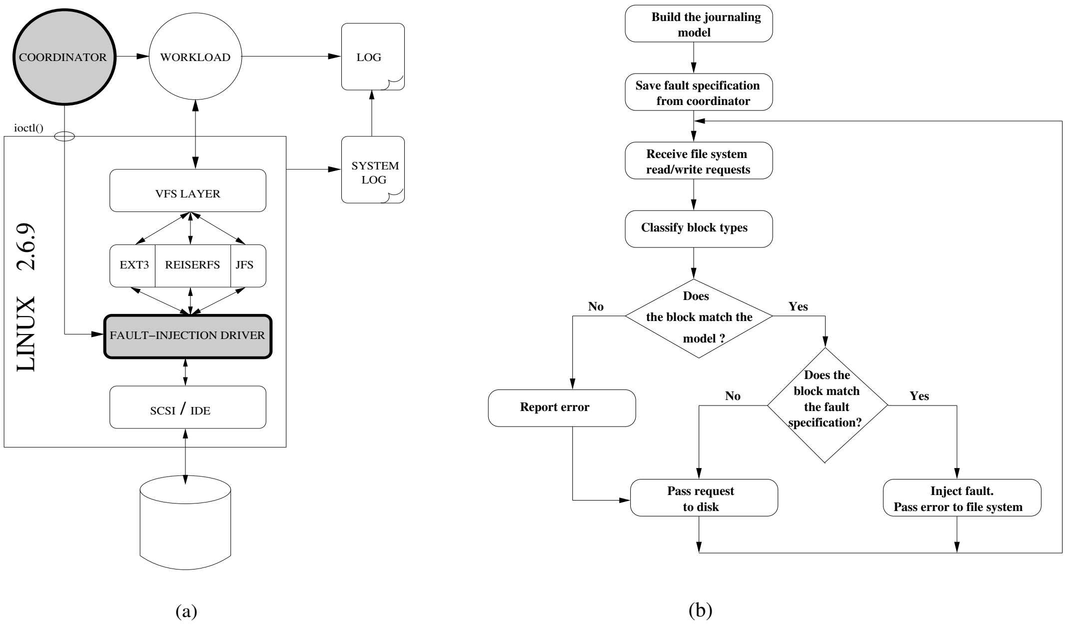


Figure 1: **Benchmarking Framework and Algorithm Flow.** *Figure (a) shows the benchmarking framework we use to measure the fault tolerance of journaling file systems to write failures. The two main components in the figure are the user level process that issues the fault and the fault-injection driver that classifies blocks and injects faults. Figure (b) shows a simplified flowchart of our benchmarking algorithm that is implemented by the fault-injection driver.*

by the file system but not reflected to the calling child process.

Each of our fault injection experiment proceeds as follows: The file system to be tested is freshly created and the files and directories needed for the testing are created in it. Depending on the type of the block to fail, the coordinator constructs a fault specification (which contains the attributes described in §3.4) and passes it to the fault-injection driver. Then, the coordinator runs a controlled workload (*e.g.*, creating a file or directory) as a child process that would generate the block write to be failed. When the expected block is written by the file system, the driver injects the fault by failing that block write. The driver also records any file system writes that violate the journaling model.

Once the fault is injected, the coordinator collects the error logs from the child process, system log and the driver. Although all the above process is automated, the error logs have to be interpreted manually to figure out the extent to which the file system can be damaged by extraneous writes.

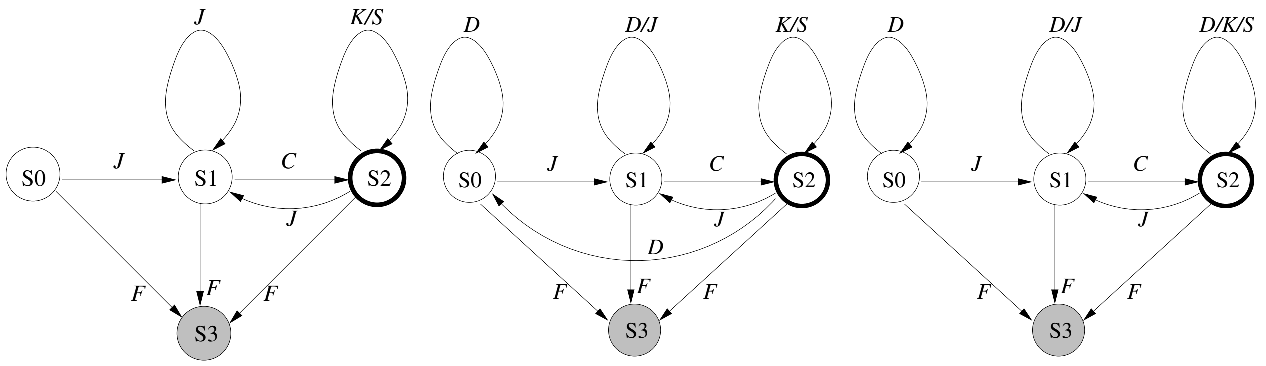
Our fault injection experiments are not statistical. Instead, we carefully choose the fault injection points. We inject faults at 5 main points: ordered data writes, journal writes, commit writes, checkpoint writes and journal superblock writes. Within journal writes, we perform fault injection to both journal metadata and journal data blocks. In some of our analysis, we force a system crash after fault injection to see if the file system recovers correctly.

3.3 Journaling Models

We now describe how we model journaling file systems. As explained in Section 2, there are three different journaling modes. Each of these journaling modes differs from the other by the type of data it journals and the order in which it writes the blocks. We build a model for each of the journaling modes based on its functionality. The models represent the journaling modes by specifying the type of data they accept and the order in which the data must be written. For example, the model for ordered journaling mode specifies that ordered data must be written before the metadata is committed to the log.

We build the models as follows. First, we construct a regular expression for each journaling mode. We use regular expressions because they can represent the journaling modes concisely and they are easy to construct and understand. Then, we build a model based on the regular expression. Figure 2 shows the models for each journaling mode. The journaling models consist of different states. These states represent the state of on-disk file system. The on-disk file system moves from one state to another based on the type of write it receives from the file system. We keep track of this state change by moving correspondingly in the model.

We explain briefly the regular expression for each journaling mode. Let, J represent journal writes, D represent data writes, C represent journal commit writes, S repre-



(a) Data Journaling Model (b) Ordered Journaling Model (c) Writeback Journaling Model

Figure 2: **Journaling Models.** *This figure shows the models that we use for verifying the different journaling modes. Each model is built based on a regular expression and then the state $S3$, which represents the state that is reached on any write failure, is added to it. In the above models, J represents journal writes, D represents data writes, C represents journal commit writes, K represents checkpoint writes, S represents journal super block writes, and F represents any write failure.*

sent journal super block writes, K represent checkpoint data writes and F represent any write failures.

Data Journaling: Data journaling can be expressed by the following regular expression:

$$((J^+C)^+(K^*S^*)^*)^+.$$

In data journaling mode, all the file system writes are journaled (represented by J) and there are no ordered or unordered writes. After writing one or more journal blocks, a commit block (represented by C) is written by the file system to mark the end of the transaction. The file system could write one or more such transactions to the log. Once the transactions are committed, the file system might write the checkpoint blocks (represented by K) to their fixed locations or the journal super block (represented by S) to mark the new head and tail of the journal. We convert this regular expression to a state diagram as shown in Figure 2(a) and add the failure state $S3$ to it.

Ordered Journaling: Ordered journaling can be expressed by the following regular expression:

$$(((D^*J^+D^*)^+C)^+(K^*S^*)^*)^+.$$

In ordered mode, data (D) must be written before metadata blocks are *committed* to the journal. Note that the data blocks can be issued in parallel with the journal writes (J) but all of those writes must be complete before the commit block (C) is written. Once the commit block is written, the transaction is over. There could be one or more such transactions. Similar to data journaling, the file system can write the checkpoint blocks (K) or the journal super block (S) after the transactions. This regular expression is converted into a state diagram and a failure state $S3$ is added to it as shown in Figure 2(b).

Writeback Journaling: Writeback journaling is given by the following regular expression:

$$(((D^*J^+D^*)^+C)^+(K^*D^*S^*)^*)^+.$$

In writeback journaling mode, the data (D) can be written at any time by the file system. It can be written before the journal writes (J) or after them. Once the journal writes are complete, a commit block (C) is written. After the transaction is committed, the file system can write the journal super block (S) or the checkpoint blocks (K) or the unordered writes (D). The writeback journaling model in Figure 2(c) is obtained by taking this regular expression and adding the state $S3$ to it.

3.4 Error Model

In our error model, we assume that the latent errors originate from the storage subsystem. These errors can be accurately modeled through software-based fault injection because in Linux, all such low-level errors are reported to the file system in a uniform manner as "I/O errors" at the device-driver layer.

Write failures can be due to several reasons. The media on the disk could be corrupt or it could be damaged by the moving parts in the disk (*e.g.*, head crash) and this can cause write failures. In a study by Talagala [19], it is found that the interconnecting bus can be faulty, leading to timeouts; this can result in transient or permanent write errors. Finally, the low level device drivers can be buggy and can cause write failures.

The errors we inject into the block write stream have three different attributes, similar to the classification of faults injected to the Linux kernel by Gu *et al.* [7]. The coordinator passes the fault specification to the fault-injection driver with the following attributes:

What: This specifies the file system to test. The driver currently understands ext3, Reiserfs, and IBM JFS file system semantics.

Which: This attribute specifies the block type and it determines which request in a given traffic stream must be failed. The fault-injection driver can identify the following list of block types: ordered data, journal metadata such as the commit block or the journal super block, journal data, and checkpoint data. Not all request types are supported by all the file systems, and therefore this attribute can change depending on the file system. The request to be failed can be a dynamically-typed one (like a journal commit block) or a statically typed one (like a journal super block).
How long: This determines whether the fault that is injected is a transient error (*i.e.*, fails for the next N requests, but then succeeds), or a permanent one (*i.e.*, fails upon all subsequent requests).

3.5 Failure Classification

We now classify the different ways in which the file system can fail due to the write failures. The type of losses one might incur after the write failure are as follows:
No Loss: The file system handles the write failure properly and prevents its data from getting corrupted or lost.
Data Corruption: In this case, write failures lead to data corruption but not metadata corruption. Although the file system metadata structures remain consistent, data from files can get corrupted. This type of failure can occur if the data block pointers from metadata blocks point to invalid contents on the disk. Note that these type of errors cannot be detected by fsck.
Data Loss: In this type of failure, file data is lost due to transient or permanent write failures. Data loss can occur if the data block pointers are not updated correctly.
Files and Directories Loss: In this case, file system metadata is corrupted. This can result in lost files or directories.
Unmountable File System: When write failures happen, the file system can corrupt its important metadata blocks like the super block or group descriptors and as a result can become unmountable.
File System Crash: Write failures can lead to some serious file system reactions, such as a system-wide crash. This failure could be initiated by an explicit call such as `panic` or due to other reasons such as dereferencing a null pointer.

3.6 Why Semantic Fault Injection?

One question we have to address is why the fault injection technique has to be file-system aware. Is it possible to conduct a similar analysis without semantic knowledge?

The device driver that we use to fail the disk writes understands the various file system block types and transaction boundaries. Without this high-level information, the driver does not know the type of block it receives and therefore it cannot determine if it is failing a journal block or a data

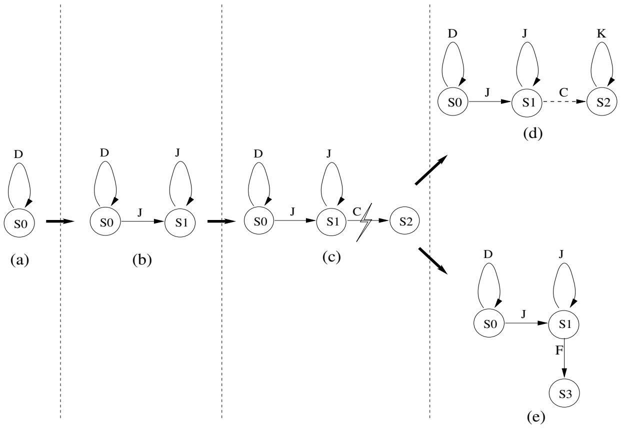


Figure 3: **Fault Injection Example.** *This figure shows the sequence of steps followed by the fault-injection driver to track transaction writes in ordered journaling mode and fail the commit block write.*

block. This information is important because file systems behave differently on different block-write failures. For example, Reiserfs crashes on journal write failures and does not crash on data-block write failures. Moreover, depending on the type of block failed, the file system errors can vary from data corruption to unmountable file systems. With file-system knowledge, it is possible to answer *why* the file system fails in a certain way. Having higher-level semantic knowledge also enables us to identify several design flaws that would not have been identified if the fault injection was performed without any semantic information, as we will see in our analysis.

3.7 Putting it All Together: An Example of Fault Injection

We conclude this methodology section with an example of how a fault is injected using the journaling model. Figure 3 shows the sequence of steps followed by the fault-injection driver to track the file system writes and inject the fault. In this example, we consider failing a commit block write of a transaction in ordered journaling mode. Each step in the figure captures a transition to a different state. Initially, the transaction starts with a set of ordered data writes (Figure 3a). After the data writes, the journal blocks are logged (Figure 3b). The commit block is written after all the data and journal writes are complete and it is failed (Figure 3c). The file system can be oblivious to this commit block failure and continue to checkpoint the journaled blocks (Figure 3d). Or, the file system can recognize this failure and take steps to prevent file system corruption by moving to state $S3$ (Figure 3e). In state $S3$, the file system could abort the failed transaction, or do bad block remapping, or remount itself read-only, or crash the system.

From the example, we can see that it is not sufficient to know just the block types to inject faults in file system requests. Without the model, the fault-injection driver cannot determine if the requests following a write failure belong to the failed transaction or to new transactions from the file system. By keeping track of the writes using the journaling model, the fault-injection driver can explain *why* a particular block write failure leads to certain file system errors.

4 Analysis

In this section, we explain the failure analysis for three Linux based journaling file systems: ext3, Reiserfs, and IBM JFS.

4.1 Ext3 Analysis

Ext3 is a journaling file system based on the ext2 file system [21]. By default, ext3 logs the file system metadata blocks in the journal. It uses different types of journal metadata blocks to keep track of the transactions and the blocks that are logged. Journal descriptor blocks store the fixed location block numbers of the journaled data. Journal revoke blocks prevent the file system from replaying some data that should not be replayed during recovery. Journal commit blocks mark the end of transactions. Journal super block stores information about the journal such as the head, tail, next transaction ID and so on. Apart from these journal metadata blocks, the log also stores journal data blocks that are journaled versions of the fixed location blocks.

Ext3 is designed such that its journal metadata blocks such as the journal super block, descriptor block, revoke block and commit block contain a magic number that identifies them as journal metadata blocks. The journal metadata block also contains a sequence number that denotes the transaction number of the particular transaction in which it occurs. During recovery, if the block read from the journal does not have the correct magic number, it is treated as a journal data block. If it has the magic number and if its sequence number does not match the next transaction id that is expected, then those blocks are skipped. Based on our ext3 analysis, we found the following design flaws in handling write failures.

4.1.1 Committing Failed Transactions

When a write in a transaction fails, ext3 continues to write the transaction to the log and commits it. This can affect file system integrity. For example, when an ordered data write fails in ordered journaling mode, we expect the file system to abort the transaction, because if it commits the transaction, the metadata blocks will end up pointing to wrong or old data contents on the disk. This problem occurs in ext3 where failure of an ordered write can cause data corruption.

4.1.2 Checkpointing Failed Transactions

When a write in a transaction fails, the file system must not checkpoint the blocks that were journaled as part of that transaction, because during checkpointing if a crash occurs, the file system cannot recover the failed transaction. This can result in a corrupted file system. Ext3 commits a transaction even after a transaction write fails. After committing the failed transaction, ext3 checkpoints the blocks that were journaled in that transaction. Depending on the journaling mode, the checkpointing can be either partial or complete as described below.

Partial Checkpointing: In certain cases, ext3 only checkpoints some of the blocks from a failed transaction. This happens in data journaling mode when a journal descriptor block or journal commit block write fails. In these cases, during checkpointing, *only* the file system metadata blocks of the transaction are checkpointed and the data blocks are not checkpointed. For example, in data journaling mode, when a file is created with some data blocks, if the transaction's descriptor block fails, then only the metadata blocks like the file's inode, data bitmap, inode bitmap, directory data, and directory inode blocks are written to their fixed locations. The data blocks of the file, which are also journaled in data journaling mode, are not written. Since the data blocks are not written to their fixed locations, the metadata blocks of the file end up pointing to old or incorrect contents on the disk.

Complete Checkpointing: In ordered and writeback journaling mode, only file system metadata blocks are journaled and no data blocks are written to the log. In these modes, ext3 checkpoints all the journaled blocks even from a failed transaction. Below we describe a generic case where it can cause file system corruption.

Let there be two transactions T_1 and T_2, where T_1 is committed first followed by T_2. Let block B_1 be journaled in T_1 and blocks B_1 and B_2 be journaled in T_2. Assume transaction T_2 fails and that the file system continues to checkpoint blocks B_1 and B_2 of the failed transaction T_2. If a crash occurs after writing blocks B_1 and B_2 to their fixed locations, the file system log recovery runs during next mount. During the recovery only transaction T_1 will be recovered because T_2 is a failed transaction. When T_1 is recovered, the contents of block B_1 will be overwritten by old contents from T_1. After the recovery, the file system will be in an inconsistent state where block B_1 is from transaction T_1 and block B_2 is from transaction T_2.

The above explained problem can occur in ext3 when the write of a journal metadata block like descriptor block, revoke block, or commit block fails. This can lead to file system corruptions resulting in loss of files, inaccessible directories and so on.

4.1.3 Not Replaying Failed Checkpoint Writes

Checkpointing is the process of writing the journaled blocks from the log to their fixed locations. When a checkpoint write fails, the file system must either attempt to write again or mark the journal such that the checkpoint write will happen again during the next log replay. Ext3 does not replay failed checkpoint writes. This can cause data corruption, data loss, or loss of files and directories.

4.1.4 Not Replaying Transactions

Journaling file systems maintain a state variable to mark the log as dirty or clean. When the file system is mounted, if the log is dirty, the transactions from the log are replayed to their fixed locations. Usually journaling file systems update this state variable before starting a transaction and after checkpointing the transaction. If the write to update this state variable fails, two things can possibly happen; one, the file system might replay a transaction that need not be replayed; two, it might fail to replay a transaction that needs recovery. Replaying the same transaction again does not cause any integrity problems. But the second possibility (*i.e.*, not replaying the journal contents) can lead to corruption, or loss of data.

Ext3 maintains its journal state in the journal super block. Ext3 clears this field and writes the journal super block to indicate a clean journal. To mark the journal as dirty, the journal super block is written with a non-zero value in this field. When the journal super block write fails, ext3 does not attempt to write it again or save the super block in other locations. Moreover, even after the journal super block failure, ext3 continues to commit transactions to the log. If the journal super block written to mark the journal as dirty is failed, the journal appears clean on next mount. If any transaction needs replay due to a previous crash, ext3 fails to replay them. This can result in lost files and directories.

4.1.5 Replaying Failed Transactions

When a journal data block write fails, that transaction must be aborted and not replayed. If the transaction is replayed, journal data blocks with invalid contents might be read and written to the fixed location. If not handled properly, this can lead to serious file system errors.

As said earlier, ext3 does not abort failed transactions. It continues to commit them to the log. Therefore during recovery, it can write invalid contents to file system fixed location blocks. This can corrupt important file system metadata and even result in an unmountable file system.

4.1.6 Ext3 Summary

Overall, we find that ext3 is designed only with the whole system crash in mind; ext3 does not effectively handle single block write failures. Some features in ext3 are well designed. First, ext3 does not crash the entire system on failed writes. Second, by using magic numbers and transaction IDs on journal metadata blocks, ext3 prevents replay of invalid contents. The main weakness in the ext3 design is that it does not abort failed transactions but continues to commit them. This can lead to serious file system errors ranging from data corruption to unmountable file systems. We also found that ext3 sometimes logs empty transactions – transactions that do not have any blocks in them other than the commit block. Although it does not affect integrity, this can result in unnecessary disk traffic.

4.2 Reiserfs Analysis

Journaling in Reiserfs is similar to that of ext3 [15]. Reiserfs uses a circular log to capture journal writes and logs file system writes at block level. Reiserfs supports all the three different journaling modes. It also uses journal metadata blocks like the journal descriptor block, journal commit block, and journal super block to describe the transactions and the fixed location blocks. The journal metadata blocks in Reiserfs also contain a magic number and a transaction number similar to that of ext3. Based on our analysis, we found the following design flaws in Reiserfs.

4.2.1 Crashing File System

When a write fails in Reiserfs, most of the time it crashes the whole file system by issuing a `panic` call. This necessitates the entire system to be rebooted before using it again. Not only does this affect processes using Reiserfs, but it also affects all other processes running in the same system.

However, crashing the entire file system on single write error also has a benefit. When a journal write (journal data or journal metadata) fails, the system crashes and therefore the failed transaction does not get committed to the disk. When the system boots up and mounts the file system, Reiserfs performs recovery. During recovery, it replays all transactions that were successfully committed before the failed transaction. Since no failed transactions are committed, they are not replayed and the file system remains in a consistent state after recovery. This avoids some of the problems that we saw in ext3 such as checkpointing failed transactions, not replaying successful transactions, and replaying failed transactions. In other words, Reiserfs converts all failures, whether transient or sticky, local or global, into entire disk failure. However, if a particular journal block write always fails, Reiserfs repeatedly crashes the whole system.

Reiserfs also crashes the system when a checkpoint write fails. After the crash, recovery takes place and the failed checkpoint write is replayed properly. Note that this works correctly for transient write failures but for permanent write errors, Reiserfs requires fsck to be run to handle replay failures. Crashing on checkpoint write failures prevents the problem of not replaying failed checkpoint writes that happens in ext3.

4.2.2 Committing Failed Transactions

On certain write failures, Reiserfs does not crash but continues to commit the failed transaction. In ordered journaling mode, when an ordered data block write fails, Reiserfs journals the transaction and commits it without handling the write error. This can result in corrupted data blocks because on such failed transactions the metadata blocks of the file system will end up pointing to invalid data contents. Reiserfs does not have a uniform failure handling policy; it crashes on some write failures and not on others. File system corruption would have been prevented if Reiserfs was crashing the system even on ordered write failures.

4.2.3 Reiserfs Summary

Overall, we find that Reiserfs avoids many of the mistakes done by ext3, but *expensively* at the cost of crashing the entire file system. Basically, Reiserfs converts a fail-stutter system in to a fail-stop one to handle the write errors. We find that not committing a failed transaction, as done in Reiserfs, is a desirable design decision because it would solve many of the problems that we saw in ext3. However, if the block write errors are permanent, then Reiserfs might make the system unusable by repeated crashing. We also used our model to find a bug in Reiserfs on Linux 2.6.7. The data journaling mode in that version was behaving like ordered journaling mode. Thus our journaling model can also be used to find such bugs where the semantics of journaling is violated.

4.3 JFS Analysis

IBM JFS works only in ordered journaling mode [2]. Unlike ext3 and Reiserfs, it does not support data and write-back journaling modes. JFS also differs from ext3 and Reiserfs by the information that is written to the log. While ext3 and Reiserfs log whole blocks in to the journal, JFS writes records of modified blocks to the log. However, ordered data block writes are written as whole blocks similar to other file systems.

Since JFS does record level journaling, log blocks cannot be classified as journal data blocks or journal commit blocks. A single log write can contain both the journal data records and commit records. It is hard to separate commit record from other journal records as most of the transactions are small and so can fit in a single journal block. We modified our ordered journaling model to work under JFS record level journaling. We performed the same failure analysis on JFS and found the following problems.

4.3.1 Crashing File System

Similar to Reiserfs, JFS also crashes the file system on certain write failures. For example, the system crashes when the journal super block write fails during the mount operation. As said earlier, crashing the whole system affects all the processes running in that system. Also, crashing the whole system is not a graceful way to provide fault tolerance if the write errors are permanent.

4.3.2 Not Replaying Failed Checkpoint Writes

When a checkpoint write fails, JFS does not rewrite it or mark the transaction for a replay. JFS simply ignores this error, which can lead to a corrupted file system. This behavior is similar to that of ext3. Since both these file systems do not record failed checkpoint writes, they have no way of identifying which transactions must be replayed again.

4.3.3 Committing Failed Transactions

We found that all the three journaling file systems commit a transaction on an ordered block write failure. JFS does not notify the application of an ordered write failure and commits the transaction. This can lead to data corruption.

4.3.4 Failing to Recover

When a journal block write fails, JFS does not abort the failed transaction but commits it. If a crash happens after a journal write failure, the `logredo` routine of JFS fails because of unrecognized log record type. This can lead to unmountable file system.

4.3.5 JFS Summary

JFS has some of the design flaws we saw in ext3 and Reiserfs. For example, JFS commits failed transactions and does not replay failed checkpoint writes. It also crashes the file system like Reiserfs on journal super block write failures. We also found a bug in JFS. JFS does not flush the blocks associated with a file even after a sync call. We created a zero sized file and called `fsync` on the file descriptor. The `fsync` call returned without flushing any blocks to the journal while one would expect the file system to write the metadata blocks associated with the file to the disk.

	Ext3	Reiserfs	IBM JFS
Committing Failed Transactions	×	×	×
Checkpointing Failed Transactions	×		
Not Replaying Failed Checkpoint Writes	×		×
Not Replaying Transactions	×		
Replaying Failed Transactions	×		
Crashing File System		×	×

Table 1: **Design Flaws.** *This table gives a summary of the type of design flaws we have identified in ext3, Reiserfs and IBM JFS.*

	Ext3					Reiserfs					IBM JFS				
Block Type	DC	DL	FDL	UFS	CR	DC	DL	FDL	UFS	CR	DC	DL	FDL	UFS	CR
Journal Descriptor Block	×	×	×							×	–	–	–	–	–
Journal Revoke Block	×	×	×							–				×	
Journal Commit Block	×	×	×							×				×	
Journal Super Block	×	×	×							×					×
Journal Data Block	×	×	×	×						×				×	
Checkpoint Block	×	×	×							×	×	×	×		
Data Block	×					×					×				

Table 2: **Analysis Summary.** *This table presents the summary of the type of failures that can occur in ext3, Reiserfs and IBM JFS when block writes fail. Data block represents both ordered and unordered writes in ext3 and Reiserfs whereas it represents only ordered writes in JFS. DC means "Data Corruption", DL means "Data Loss", FDL means "Files and Directory Loss", UFS means "Unmountable File System", and CR means "Crash". A "–" means that the corresponding block type is not available in that file system. Although JFS does not have separate commit or revoke blocks, it has records of that type.*

4.4 Analysis Summary

The summary of our analysis is presented in Table 1 and Table 2. Table 1 lists the various design flaws we identified in Linux journaling file systems. Table 2 gives the different types of file system failures that can happen when block writes fail. Overall, we find that Linux journaling file systems need a more careful design of their failure handling policies.

5 Related Work

In this section, we discuss related work. We first talk about related work in fault injection in general and then about specific work on file and storage systems testing.

Fault Injection: Fault injection has been used for a long time to measure the robustness of systems. Koopman argues that faults injected directly into the modules under test do not give representative results for dependability evaluation [11]. He says that the fault must be injected in the external environments of the module under test and the fault must be activated by inputs during real execution. This is similar to our approach. We inject faults external to the file system module and activate them by running workloads on top of the file system.

We use software to simulate the effects of hardware faults and inject faults by dynamically determining the block types of the file system. FTAPE is a tool that performs dynamic workload measurements and inject faults by automatically determining time and location that will maximize fault propagation [20]. FIAT is one of the early systems to use fault injection techniques to simulate the occurrences of hardware errors by changing the contents of memory or registers [9]. FINE is a tool developed by Kao *et al.*, to inject hardware induced software faults into UNIX kernel and trace the execution flow of the kernel [12]. In a more recent work, fault injection techniques are used to test the Linux kernel behavior under errors [7].

File and Storage System Testing: Most of the file system testing tools test the file system API with various types of invalid arguments. Siewiorek *et al.* develop a benchmark to measure the system's robustness and use it to test the dependability of file system's libraries [17]. Similarly, Koopman *et al.* use the Ballista testing suite to find robustness problems in Safe/Fast IO (SFIO) library [5]. Another way to test file system robustness is to use model checking techniques and apply it to the file system code. In a more recent work, Yang *et al.* use model checking comprehensively to find bugs in three different file systems: ext3, Reiserfs and JFS [22]. They use formal verification techniques to systematically enumerate a set of file system states and verify them against valid file system states. Their work can be used to identify problems like deadlock, and NULL pointers whereas our work focuses mainly on how file systems handle latent sector errors.

Previous work has studied the reliability of storage systems. Brown *et al.* developed a method to measure the system robustness and applied it to measure the availability of software RAID systems in Linux, Solaris and Windows [3]. They use a PC to emulate a disk and use the disk emulator to inject faults. They test the software RAID systems while our work targets the file systems. Moreover, we use file system knowledge to carefully select and fail specific block types whereas they do not require any semantic information for fault injection. Other studies have evaluated RAID storage systems for reliability and availability [8, 10]. These studies have developed detailed simulation models of RAID storage arrays and network clusters and used them to obtain the dependability measures.

6 Conclusion

In this paper, we propose a new way to evaluate the robustness of journaling file systems under disk write failures. We build semantic models of different journaling modes and use them to inject faults into the file system disk requests. We evaluate three widely used Linux journaling file systems. From our analysis, we find that ext3 and IBM JFS violate journaling semantics on block write failures, which could result in corrupt file systems. In contrast, Reiserfs maintains file system integrity by crashing the entire system on most write failures. However, on permanent write failures, this will result in repeated crashes and restarts. Based on the analysis, we identify various design flaws and correctness bugs in these file systems that can catastrophically affect the on-disk data. Overall, we find that modern file systems need more design consideration placed into their failure handling policies.

Acknowledgments

We thank the members of the ADSL research group for their insightful comments. We also thank the anonymous reviewers for their thoughtful suggestions. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM and EMC.

References

[1] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *HotOS VIII*, pages 33–38, Schloss Elmau, Germany, May 2001.

[2] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2004.

[3] A. Brown and D. A. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 263–276, San Diego, California, June 2000.

[4] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 1–14, San Francisco, California, April 2004.

[5] J. DeVale and P. Koopman. Performance Evaluation of Exception Handling in I/O Libraries. In *Dependable Systems and Networks*, June 2001.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[7] W. Gu, Z. Kalbarczyk, I. K. Ravishankar, and Z. Yang. Characterization of linux kernel behavior under error. In *Dependable Systems and Networks*, pages 459–468, June 2003.

[8] Y. Huang, Z. T. Kalbarczyk, and R. K. Iyer. Dependability Analysis of a Cache-Based RAID System via Fast Distributed Simulation. In *The 17th IEEE Symposium on Reliable Distributed Systems*, 1998.

[9] Z. S. D. S. J.H. Barton, E.W. Czeck. Fault Injection Experiments Using FIAT. In *IEEE Transactions on Computers*, volume 39, pages 1105–1118, April 1990.

[10] M. Kaniche, L. Romano, Z. Kalbarczyk, R. K. Iyer, and R. Karcich. A Hierarchical Approach for Dependability Analysis of a Commercial Cache-Based RAID Storage Architecture. In *The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, June 1998.

[11] P. Koopman. What's wrong with fault injection as a dependability benchmark? In *Workshop on Dependability Benchmarking (in conjunction with DSN 2002)*, Washington DC, July 2002.

[12] W. lun Kao, R. K. Iyer, and D. Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. In *IEEE Transactions on Software Engineering*, pages 1105–1118, 1993.

[13] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[14] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, Anaheim, California, April 2005.

[15] H. Reiser. ReiserFS. www.namesys.com, 2004.

[16] F. B. Schneider. Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[17] D. Siewiorek, J. Hudak, B.-H. Suh, and Z. Segal. Development of a benchmark to measure system robustness. In *The Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993.

[18] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[19] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, San Juan, Puerto Rico, April 1999.

[20] T. K. Tsai and R. K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *8th Intl. Conf. On Modeling Techniques and Tools for Conp. Perf. Evaluation*, pages 26–40, Sept 1995.

[21] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[22] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.