Failure Analysis of SGI XFS File System

Krishna Pradeep Tamma, Shreepadma Venugopalan

pradeep@cs.wisc.edu, vshree@cs.wisc.edu
Computer Sciences Department
University of Wisconsin, Madison

Abstract

Commodity file systems expect a fail stop disk. But todays disks fail in unexpected ways. Disks exhibit latent sector errors, silent corruption and transient failures to name a few. In this paper we study the behavior of SGI XFS to such errors. File systems play a key role in handling most data and hence the failure handling policy of the file system plays a major role in ensuring the integrity of the data. XFS is a highly scalable journaling file system developed by SGI. We analyze the failure handling policy of XFS file system. We fail reads and writes of various blocks that originate from the file system and analyze the file system behavior. Some such blocks include super block, journal header block, commit block, data block, index blocks. We classify our errors according to the extended IRON Taxonomy. We see that XFS is vulnerable to these failures and does not possess a uniform failure handling policy. Further the file system at times silently corrupts data. We also see that XFS has very little internal redundancy even for critical structures such as the super block and B+Tree root. Finally we see that XFS, like most commodity file systems, fails to recover from these failures putting the data at risk.

1 Introduction

Traditional disks are considered fail-stop in which the disk either works or stops working. Further it is assumed that detecting such a failure is easy. Failure model of modern disks is changing from this traditional scenario. Modern disks exhibit more complex behavior. Latent sector errors involve a small portion of disk becoming temporarily or permanently unavailable. Some data corruption errors include, misdirection errors in which data is written to a wrong location, and phantom writes that result in incomplete writes. Modern disks comes with huge firmware. Bugs in the firmware can lead to write and read errors. Lastly the exhibited errors can be transient.

File systems form the central part of any operating system kernel and controls the disk. Thus, failure handling policy of the file system plays a major role in ensuring the integrity of the data. But most file systems assume the traditional failure model of disk behavior, and thus not tolerant to modern disk faults [6]. Failure analysis of a file system involves analysing the failure handling policy to different disk errors. Failure analysis of ReiserFS [9], IBM JFS [1] and ext3 [11] in [6]. We adopt the techniques developed in that framework and apply to XFS. XFS is a scalable file system developed by SGI. XFS comes with lot of guarantees, some of which include large file size, file system size, directory size and high I/O performance. XFS is designed around early 90s with techniques that todays file systems are adopting. XFS is mature compared to ReiserFS, JFS and ext3. Further XFS is developed in a commercial setting which involves lot of testing before deploying the system. We feel analyzing the failure handling policy of such a file system gives good insight into the behavior of modern file systems with modern disk faults.

In our experiments we fail reads and writes of various blocks that originate from the file system and analyze the file system behavior. Some such blocks include super block, journal header block, commit block, data block, index blocks. We run various workloads to generate various block accesses and fail such access when hitting the disk. By analyzing the aftermaths we conclude on the behavior file system.

We see that XFS is vulnerable to these failures and does not possess a uniform failure policy. Further the file system at times silently corrupts data. We also see that XFS has very little internal redundancy even for critical structures such as the super block and B+Tree root. Finally we see that XFS like most commodity file systems fails to recover from these failures putting the data at risk.

Our main contributions has been to extend the existing fault injection framework to support the semantics of XFS. We also analyzed XFS behavior on read and write errors under the POSIX system calls that exercise the file system.

The rest of this paper is organized as follows. In Section 2 we describe the architecture of XFS file system. Then we discuss the architecture of disk fault injection in section 3. In Section4, we explain the IRON taxonomy that we adopted to model the file system failure handling policy. We present our results in section 5 and we will conclude in section 6.

2 XFS Architecture

XFS [10] is a scalable file system developed by Silicon Graphics Inc (SGI) as part of the IRIX operating system. XFS was ported to the Linux operating system [2] and licensed under GPL in 1999. We perform disk fault injection on the Linux version of XFS. In this section we describe major features of XFS that are different from traditional file systems. XFS supports large (contiguous) files, large number of files, large directories, large file-systems, sparse files, fast crash recovery and high performance I/O. The overall architecture of XFS is shown in Figure 1. Here, we briefly describe various features of XFS. The reader is directed to [10] for further information.

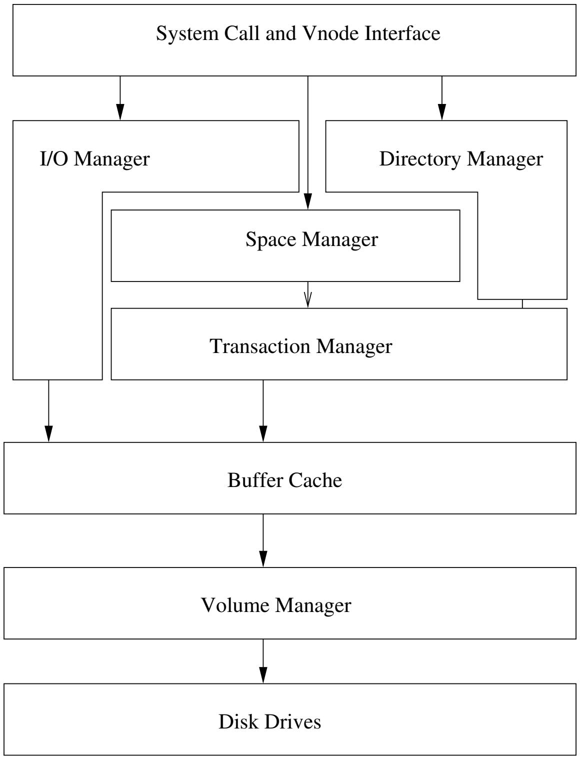

Figure 1: **XFS Architecture.**

2.1 Major Components

The logical structure of XFS is similar to traditional file systems except the transaction manager and the volume manager. XFS is modularized into several components as shown in Figure 1. The space manager, which forms the major part of the file system, deals with allocation of inodes and allocation of data blocks with in individual files. Further it manages free space in the system by employing B+Trees. The I/O manager satisfies the I/O requests from the applications. The directory manager implements the XFS file system name space. The buffer cache is used to cache frequently accessed blocks similar to the way in traditional file systems. The transaction manager is used to make all updates to metadata

ATOMIC, which enables quick recovery during a crash. The volume manager provides a layer of abstraction between XFS and its underlying disks. It deals with mirroring, striping, concatenation operations.

2.2 Major Features

The scalability of XFS is achieved due to its several features, some of which include 64-bit counters, File allocation groups, parallelism in I/O, Direct I/O, pervasive use of B+Trees, Dynamic Inode Allocation, and Delayed Allocation.

2.2.1 Allocation Groups

Each file system in XFS is partitioned into regions called allocation groups similar to cylinder groups in FFS [3]. Each allocation group size typically ranges from 0.5GB to 4 GB. These are used in achieving scalability by exploiting parallelism. Always a new directory is placed in different AG than its parent. Further inodes in that directory and the blocks for those inodes are allocated contiguously.

2.2.2 B+Trees

The major part of scalability in XFS should be attributed to its pervasive use of B+Trees. Several other features like dynamic inode allocation, free space management, large file support, large directory support is made possible due to B+Tree index structure.

2.2.3 Dynamic Inode Allocation

Unlike traditional file systems, where a part of disk space is reserved for inodes, XFS considers dynamic allocation of inodes and keeps track of such inodes using B+Trees. Each allocation group uses a B+Tree to index the locations of inodes in it. This allows to create millions of inodes in each allocation group and thus supporting large number of files.

2.2.4 Free Space Management

Traditional file systems use bitmaps to index free blocks on disks. XFS again uses B+Tree to index free space. Further XFS doesn't record free space at block level granularity but at extent level. An extent is a contiguous free space. Each allocation group keeps track of two B+Trees; one keeps track of starting location of free space and the other keeping track of size of free space. This double indexing allows very flexible and efficient searching of free extents based on type of allocation being performed.

2.2.5 Contiguous Files

XFS allocates space in terms of extents rather than blocks. These extents are tracked using extent maps similar to block maps in FFS, but are much smaller in size. Each entry in the extent map consists of block offset of the entry in the file, the length of the extent in blocks and the starting block of the extent in the file system.

2.2.6 Delayed Allocation

Delayed allocation involves applying lazy techniques to disk space allocation. XFS reserves blocks in the file system for the data buffered in memory. A virtual extent is built in memory for the reserved blocks. Delaying allocation provides the allocator much better knowledge on the eventual size of the file when it makes its decision to be written. Further temporary files will never get allocated real disk blocks.

2.2.7 Fast Crash Recovery

XFS uses write ahead logging [5] scheme that allows atomic updates to metadata. Other modern file systems with similar features are termed as journaling file systems. XFS is different from these file systems in having an incore log and disk log. The incore log is asynchronously dumped to disk.

2.2.8 Direct I/O

XFS supports Direct I/O to allow applications exploit raw disk bandwidth. Direct I/O allows a program to read or write a file without first passing the data through the system buffer cache. The data is moved directly between the user program's buffer and the disk array using DMA. This avoids the overhead of copying the data into or out of the buffer cache, and it also allows the program to better control the size of the requests made to the underlying devices.

2.2.9 Clustering Write Requests

XFS uses aggressive write clustering. Dirty file data is buffered in memory in chunks of 64 kilobytes and when a chunk is chosen to be flushed from memory, it is clustered with other contiguous requests to form a larger request. This write behind mechanism combined with delayed allocation gives better I/O performance.

2.2.10 Parallelism

The transaction log is the only centralized resource in the XFS file system. All other resources are made independent across allocation groups. This allows allocation of inodes and disk blocks, parallel throughout the file system. Further the most contentious resource, log, is parallelized by making the processor performing the transaction also do the copy.

3 Fault Injection Architecture

In the previous section we motivated the reason behind choosing XFS for performing failure analysis. In this section we describe the actual methodology we employed in injecting the disk faults. We adopted the framework developed in [7] and is shown in Figure . We treat the file system as a black box. The framework involves two major components, the coordinator and the fault injection driver.

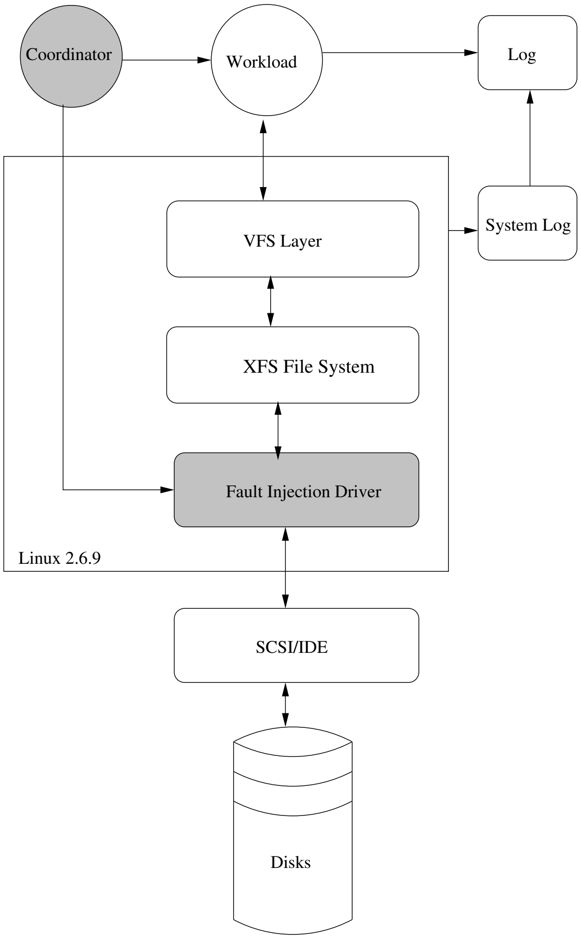

Figure 2: **Fault Injection Architecture**

3.1 Fault Injection Driver

The Fault injection driver is a pseudo driver placed between the file system and the disk. The driver exposes a logical device to the file system. It observes all the traffic between the file system and the disk. The driver has three main functions. It classifies the blocks seen in the traffic based on the type of block. The block type can be data block, inode block, super block, journal block etc. Here we employ semantic block analysis techniques developed in [8] to classify the block. Next the driver models what the file system above is doing. This involves finding the journaling mode of the file system. We found that XFS uses ordered journaling mode and discuss the techniques used in detection in the next section. Lastly, the driver injects the actual fault in the I/O stream i.e. it informs the file system that particular block operation has failed.

3.2 Journaling Modes

Major Journaling modes involve data journaling, ordered journaling and write back journaling. In data journaling both data blocks and metadata blocks are written to log before they are written to the actual locations. In write-back journaling mode, only metadata is written to log and file system don't bother about data writes. This ensures file system consistency while file consistency may be compromised. The third journaling mode is ordered journaling, in which file system delays metadata journaling till the data is written to the actual location. Different file systems support different journaling modes. For example, ext3 can be configured using any type of journaling mode during the creation of the file system. We found that XFS uses ordered journaling by our experiments. We delayed the data blocks in the SBA driver and observed that the corresponding metadata writes also delayed by the same time. This shows that metadata is not getting written till the data is written to the actual location thus showing that the mode is ordered journaling.

3.3 Coordinator

The coordinator is a user level process, and monitors the entire benchmarking process. The coordinator informs the driver, the type of block to fail and the corresponding operation which can be either read or write. The coordinator moves the file system to a clean state and then it forks a new process. The new process involves some workload and the causes various block accesses. The coordinator logs all the errors from the file system for later analysis.

3.4 Type aware Fault Injection

We inject faults in the I/O stream to emulate disk faults. Randomly injecting faults to uncover file system behavior can be time consuming. Hence to exercise the file system for most block types, we perform type aware injection. The pseudo driver performs the block classification and the coordinator specifies the block type to fail to the driver.By injecting faults in the driver we emulate a whole range of possible faults in the I/O stack. Semantic Block Level Analysis requires us to interpret the contents of the disk block traffic to identify the block type. For example, we look at the journal record header to infer the commit record, start record and the number of records in the transaction. Further we infer the location of the journal from the super block and inodes from the journal writes. Using these techniques we were able to classify blocks as: super block, B+tree root, inode, indirect, data, journal header, journal descriptor, journal commit and journal data.To emulate a write fault, we fail the request and don't issue the request to the disk. Read faults are emulated on up stream, because the type information needed for block classification is available only after the disk read. We read the block from the disk and fail the request if it matches the fault specification. One drawback of our approach is we don't model how the lower layers react to the faults. But since we are interested in knowing how the file system reacts to faults, we believe that this is the correct layer for fault injection.

4 IRON Taxonomy

IRON Taxonomy allows to define the failure policy adopted by the file system. Although the taxonomy is not complete, it covers most of the failure handling mechanisms adopted by the file systems. We extend the IRON taxonomy in those places where it fails to capture the XFS failure handling policy. IRON separates the failure handling policy into detection and recovery classes. We discuss these classes in this section.

Table 1: The levels of IRON Detection Taxonomy

Level	Technique	Comment
D_{Zero}	No detection	Assumes disk works
$D_{ErrorCode}$	Check return codes from lower levels	Assumes lower level can detect errors
D_{Sanity}	Check data structures for consistency	May require extra space per block
$D_{Redundancy}$	Redundancy over one one or more blocks	Detect corruption in end-to-end way
$D_{Unmount}$	Detection during unmount	

4.1 Levels of Detection

Table 3 shows the various classes in the detection taxonomy. Here we briefly discuss various classes and emphasize the ones that are related to XFS failure handling policy. The reader is directed to [6] for detailed description of the taxonomy.

Zero: The file system assume disk always works and don't care if a request fails. It safely assumes that the operation is succeeded.

ErrorCode: The file system checks the return codes provided by the disk and the lower level storage systems.

Sanity: With sanity checks, the file system verifies that its on-disk data structures are consistent.

Redundancy: The file system employs different forms of redundancy like check-summing to detect block corruption. Higher levels of redundancy include block mirroring, parity and other error correction codes.
Unmount: The file system remains silent when the fault occurs, but detects the error during the unmount.This is observed in XFS and not classified in IRON Taxonomy.

4.2 Levels of Recovery

Similar to the levels of detection, there are various levels of recovery classes in the IRON taxonomy. Once again we discuss those that are related to XFS. Table 2 shows different levels of recovery accroding to IRON taxonomy.

Zero: The file system assumes that the disk works, and doesn't take any action on failures.

Propagate: The file system propagates the error to the application and assumes the user will respond appropriately to the problem.

Stop: The current file system activity is stopped. Such stopping can be performed at different granularities. At the coarsest level it can crash the entire machine. At an intermediate level it can kill the process that triggered the disk fault or at the finest level it can abort the transaction that caused the fault.

Retry: The file system retries the failed operation. Retry handles transient errors.

Repair: The file system tries to repair the detected error. Such errors include fixing a bitmap, freeing a block etc.

Redundancy File system can use redundancy to recover from block loss. Such redundancy can be applied using replication, parity encodings etc.
Remount File system silently does recovery on remount. This is observed in XFS and not classified in IRON Taxonomy.

5 XFS Failure Policy: Results

In this section, we present the results of failure analysis of XFS. Background information on the XFS architecture has already been presented in section 2. We discuss the general failure policy of XFS under various workloads. We also graphically depict our results, showing the XFS failure policy for each of

Table 2: The levels of IRON Recovery Taxonomy

Level	Technique	Comment
R_{Zero}	No recovery	Assumes disk works
$\mathrm{R}_{Propagate}$	Propagate Error	Informs User
R_{Stop}	Stop Activity crash,prevent writes	Limit amount of damage
R_{Guess}	Return ”guess” at block contents	Could be wrong; failure hidden
R_{Retry}	Retry read or write	Handles failures that are transient
R_{Repair} R_{Remap}	Repair data structs Remaps block or file to different locale	Could lose data Assumes disk informs FS of failures
$\mathrm{R}_{Redundancy}$	Block replication or other forms	Enables recovery from loss/corruption
$\mathrm{R}_{Remount}$	Recovery on Remount	Data may be inconsistent

the block type/system call pair. These results are shown in Figures(3,4,5,6).We summarize the results below. The detailed behavior of XFS under each of the workloads can be obtained from Figures(3,4,5,6).

5.1 Write Failures

To analyze the effects of write failures, we fail read requests from the file system for different block types. Methods to infer block types has been described in section 3 and elsewhere [8]. We present the behavior of the file system on write failures for each of the block types.

Super Block: As evident from the graph, XFS recovers on super block failure in most cases. But we see that the file system is not shut down in some cases. This means there can be further corruption and its up to the user not to issue further requests. We also see that the file system ignores the error in a few cases and doesn’t attempt any recovery in few.

B-Tree Root: In case of a B Tree write failure, XFS shuts down the file system to prevent further damage and propagates the error to the user. But the file system attempts no recovery and can leave the Meta data index information in an inconsistent state. For e.g. an allocated inode may appear unallocated due to the write failure. On continued operations this can lead to data loss.

Inode: Inode write failures are handled much like the B Tree root. An error is propagated to the user, but no recovery is attempted. A failed inode write can lead to data loss as the on disk structure doesn’t point to the newly allocated data. Some data blocks may be orphaned but can be recovered using a program like fsck [4].

Indirect Block: Indirect blocks are not handled as well as the inode. In some cases the file system ignores the errors. But in most other cases the error is propagated to the user and file system halted to prevent further damage. But we see that in some cases the file system is not halted, which can lead to further corruption. Since no recovery takes place, the newly allocated file data would be lost.

Data: We see that data errors are mostly ignored or little action is taken other than informing the user of the error. In most cases data loss occurs silently without the knowledge of the user.

Journal Commit: The journal commit record is written at the end of a transaction. No recovery is performed on journal write errors. But the file system is halted to reduce the damage. Since no recovery is performed, the checkpoint data is partially written to the disk violating transactional consistency i.e., transactions are partially committed.

Journal Start: The journal start record indicates the transaction start. No recovery is performed on write errors. Even though the file system is halted, transactional consistency is violated as part of the data is check pointed. Partially committing transactions can lead to serious meta data inconsistencies and data loss.

Journal Header: The journal header marks the start of a transaction and is much like the super block for the records that follow. The file system doesn't recover the write errors to the file system journal header. This can lead the journal records unreadable during recovery. Hence upon recovery from a crash the journaled transactions can't be read.

Journal Data: Write errors to the journal data are propagated to the user but no recovery is attempted. We also see that journal data loss can slow down crash recovery as a part of the journal may be unavailable.

We see that XFS doesn't handle all the write errors well. In most cases no recovery is done and this can leave the file system in an inconsistent state and lead to data loss. Even some Meta data errors are ignored, which can lead to data loss. The failure to handle the errors uniformly causes data corruption, losses and inconsistencies.

5.2 Read Failures

To analyze the effects of read failures, we fail read requests from the file system for different block types. Methods to infer block types has been described in section 3 and elsewhere [8]. We present the behavior of the file system on read failures for each of the block types.

Super Block: Super block is read only during mount and a read error leads to an unmountable file system. XFS maintains copies of the super block in each of the allocation groups, but doesn't read the redundant copy to mount the file system. On a write to the super block only the main copy is updated to avoid unnecessary disk traffic. XFS probably doesn't use the copies as they may not be up-to-date.

B-Tree Root: Read errors to the root of the B Tree renders the entire file system unusable. The inode are located using the B Tree and a read error to the root makes the entire file system inaccessible. We see no redundancy in XFS for these critical structures.

Inode: Inode read errors are reported to the user. Inode read errors make the entire file/directory inaccessible. This is particularly serious for the root directory as the entire file system cannot be accessed. XFS has no internal redundancy to deal with such failures.

Indirect Block: Indirect block errors are reported to the user and no recovery is attempted. In the absence of redundancy these errors make file and directories partially inaccessible. But since the indirect blocks point to a chunk of the data in large files, these errors can lead to substantial data loss in large files.

Data: Data read errors are reported to the user. No recovery is attempted and these lead to partial data loss on files and loss of files on directories.

Journal Commit: Journal reads occur at the time of mount and recovery.The failure to read a journal commit record causes the recovery system to partially commit transaction, thus violating transactional consistency. During crash recovery partial commits may lead to inconsistencies in the file system along with data loss.

Journal Start: Read failures to the start record causes the recovery system to violate transactional consistency. Partial commits can lead to meta data inconsistencies in the file system which can lead to data loss.

Journal Header: Journal header contains data needed to interpret the rest of the records in the transaction. Failures to the journal header cause the recovery miss the entire transaction. This can lead the file system inconsistent and lead to data loss.

Journal Data: Read errors to the journal can cause the crash recovery to partially commit transactions and miss entire transactions. These errors can lead to Meta data inconsistencies and data loss in the "recovered" file system.

We see that there is little internal redundancy

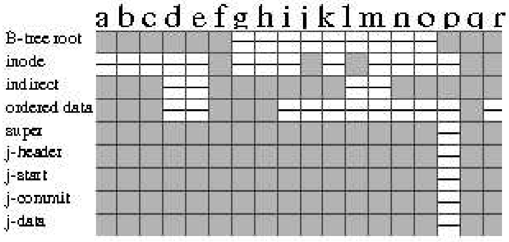

Figure 3: **Detection under Read Failure**

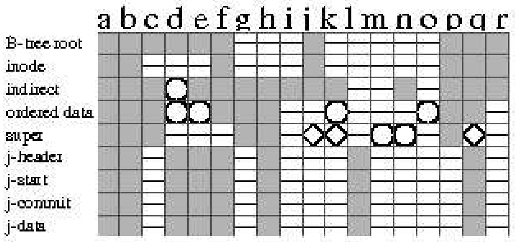

Figure 5: **Detection under Write Failure**

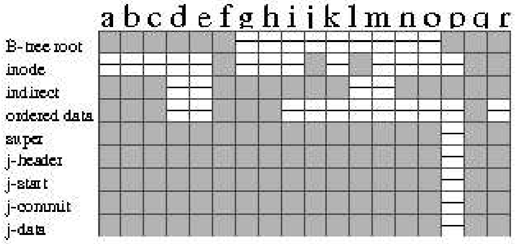

Figure 4: **Recovery under Read Failure**

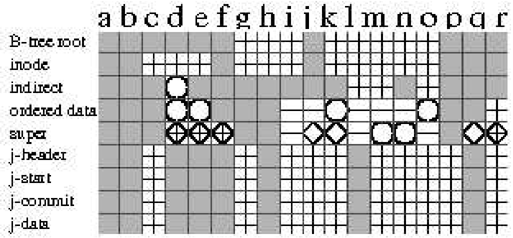

Figure 6: **Recovery under Write Failure**

and most read errors cause a portion of the file system to become unusable. Absence of redundancy for critical structures like the root inode, B Tree root can render the entire file system inaccessible. We also see that no recovery is attempted on most errors and much is left to the user. A uniform failure handling policy is needed to prevent such losses.

Figures (3,4,5,6) indicate detection and recovery policies of XFS file system for read and write faults injected for each block type across a range of workloads. The workloads are a:path traversal, b:(access,chdir,chroot,stat,statfs,lstat,open), c:(chmod,chown,utimes), d:read e:readlink, f:getdirentries, g:creat, h:link, i:mkdir, j:rename, k:symlink, l:write, m:truncate, n:rmdir, o:unlink, p:mount, q:(fsync,sync), r:umount. A gray box indicates that the workload is not applicable for the block type. When multiple recovery mechanisms are observed, the symbols are superimposed in the figure.

Table 3: Key for Interpreting Graphs

Key for Detection	**Key for Recovery**
o D_{Zero}	o R_{Zero}
- $D_{ErrorCode}$	- $R_{Propagate}$
◇ $D_{Unmount}$	| R_{Stop}
	◇ $R_{ReMount}$

6 Conclusions

Modern day file system designers have paid little attention to localized disk failures. More generally commodity operating systems assume the presence of reliable hardware. As disks increasingly fail in a non fail stop manner, we believe there is a need for a more uniform failure handling policy. File systems should be paranoid and should not trust the layers below. One solution that has been recently proposed is to maintain internal redundancy for important structure and checksum over the data. This way the file system will be able to detect and recover from most errors. Internal redundancy for critical structures like the root inode, inode B+trees is needed to prevent sector faults from rendering the entire file system inaccessible. So does the solution lie in an IRON XFS? If so, what should be the level of redundancy? What low-overhead detection and recovery mechanisms can these systems employ? Many of these need to be answered, as we try and build more resilient and robust file systems.

7 Acknowledgements

We thank Remzi H Arpaci-Dusseau for the motivation and guidance through out this work. Further we thank Vijayan Prabhakaran and Nitin Agarwal for helping us understand the fault injection driver. We also thank Lakshmi Bairavasundaram for the useful discussions. Finally we thank the people on SGI mailing list and other Linux kernel mailing lists.

References

[1] S. Best. JFS. http://www.ibm.com/developerworks/library/1-jfs.html.

[2] W. E. Jim Mostek and D. Koren. Porting the SGI XFS File System to Linux. In *Linux Showcase*, Atlanta, Georgia, USA, October 1999.

[3] M. Mckusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for Unix. In *ACM Transactions on Computer Systems*, August 1984.

[4] M. McKusick, W. Joy, S. Leffler, and R. Fabry. Fsck - The UNIX File System Check Program. In *4.3 BSD VAX-11 UNIX Manual*, April 1986.

[5] C. Mohan, D. Hederle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. In *ACM Transactions on Database Systems(TODS)*, March 1992.

[6] V. Prabhakaran, N. Agarwal, L. Bairavasundaram, H. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Under Submission*.

[7] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model Based Failure Analysis of Journaling File Systems.

[8] 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)*, April 2005.

[9] H. Reiser. ReiserFS. http://www.namesys.com/, 2004.

[10] A. Sweeney. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, USA, January 1996.

[11] S. Tweedie. Journaling the Linux extfs File system. In *In the Fourth annual Linux Expo*, Durham, North Carolina, USA, May 1998.