IRON FILE SYSTEMS

by

Vijayan Prabhakaran

B.E. Computer Sciences (Regional Engineering College, Trichy, India) 2000
M.S. Computer Sciences (University of Wisconsin-Madison) 2003

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
in
Computer Sciences

University of Wisconsin-Madison

2006

Committee in charge:
Andrea C. Arpaci-Dusseau (Co-chair)
Remzi H. Arpaci-Dusseau (Co-chair)
David J. DeWitt
Mary K. Vernon
Mikko H. Lipasti

Abstract

IRON FILE SYSTEMS
Vijayan Prabhakaran

Disk drives are widely used as a primary medium for storing information. While commodity file systems trust disks to either work or fail completely, modern disks exhibit complex failure modes such as latent sector faults and block corruptions, where only portions of a disk fail.

In this thesis, we focus on understanding the failure policies of file systems and improving their robustness to disk failures. We suggest a new *fail-partial failure model* for disks, which incorporates realistic localized faults such as latent sector faults and block corruption. We then develop and apply a novel *semantic failure analysis* technique, which uses file system *block type* knowledge and *transactional semantics*, to inject interesting faults and investigate how commodity file systems react to a range of more realistic disk failures.

We apply our technique to five important journaling file systems: Linux ext3, ReiserFS, JFS, XFS, and Windows NTFS. We classify their failure policies in a new taxonomy that measures their *Internal RObustNess (IRON)*, which includes both failure detection and recovery techniques. Our analysis results show that commodity file systems store little or no redundant information, and contain failure policies that are often inconsistent, sometimes buggy, and generally inadequate in their ability to recover from partial disk failures.

We remedy the reliability short comings in commodity file systems by addressing two issues. First, we design new low-level redundancy techniques that a file system can use to handle disk faults. We begin by qualitatively and quantitatively evaluating various redundancy information such as checksum, parity, and replica, Then, in order to account for spatially correlated faults, we propose a new probabilistic model that can be used to construct redundancy sets Finally, we describe two update strategies: a overwrite and no-overwrite approach that a file system can use to update its data and parity blocks atomically without NVRAM support. Over-

all, we show that low-level redundant information can greatly enhance file system robustness while incurring modest time and space overheads.

Second, to remedy the problem of failure handling diffusion, we develop a modified ext3 that unifies all failure handling in a *Centralized Failure Handler* (CFH). We then showcase the power of centralized failure handling in ext3$_C$, a modified IRON version of ext3 that uses CFH by demonstrating its support for flexible, consistent, and fine-grained policies. By carefully separating policy from mechanism, ext3$_C$ demonstrates how a file system can provide a thorough, comprehensive, and easily understandable failure-handling policy.

To my advisors, Andrea and Remzi, who showed me research can be fun.

Acknowledgements

I'd like to express my heart-felt thanks to my advisors, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau, for without them this Ph.D. would have been a distant dream. When even my parents were doubtful about my intentions to pursue a doctorate, Andrea and Remzi encouraged and supported my decision.

The initial sparks of interest in research were created by Andrea when I took the Advanced Operating Systems course under her. Her thoroughness and deep thoughts (to interpret which, I usually need Remzi's help) never cease to amaze me. When I often get buried in small trivial details, she rescues me out with her broad and high-level perspectives.

I didn't realize that one can do serious research and have fun at the same time until I met Remzi. To me, he is the "*Master in the art of living*". He listens patiently to my muddled ideas, sharpens my thought process, guides me throughout our projects, helps me write interesting papers, and more importantly, somehow makes me feel as though I did everything, while in reality, without his guidance there'd have been a lot of crazy ideas and not one solid work.

Andrea and Remzi are the source of inspiration and role models to me. Great many times in our weekly meetings (several of which never lasted more than 10 minutes, perhaps due to my lack of capacity to keep a conversation going), I was awed by the clarity, breadth, and depth of their thoughts. At the end of our meetings, I usually end-up wondering if I can ever raise to their level of thinking. In fact, I enjoy working with them so much that sometimes I wish I can go back in time and start fresh again, avoid all the mistakes I did while trying to learn more from them and hopefully, have longer meetings.

I'm grateful to my dissertation committee members: David DeWitt, Mary Vernon, and Mikko Lipasti for their insightful comments and suggestions. Their feedback during my preliminary and final exams improved this thesis greatly.

I'd also like to express my sincere gratitude to Paul Barford. Although, I studied under him only for a semester, he played an important role in motivating me to do Ph.D. I thank him for his kind encouraging words and reference letters.

I'm also indebted to all those who gave me useful feedbacks on the dissertation topic during my interview process. My discussions with Chandhu Thekkath at Microsoft Research, Yasushi Saito at Google, Erik Riedel at Seagate, Ram Rajmony at IBM-Austin, the IBM-Almaden interview panel, Fred Douglis at IBM-Watson, and Mary Baker at HP Labs were fruitful.

I'm fortunate to work with wonderful colleagues at UW: Nitin Agrawal, Lakshmi Bairavasundaram, Nathan Burnett, Timothy Denehy, Haryadi Gunawi, Todd Jones, Florentina Popovici, and Muthian Sivathanu. Our late-night work during conference deadlines were lot of fun (with free pizzas, thanks to our advisors). Here, I'd like to thank Muthian Sivathanu specially. He was always eager to discuss my problems (both technical and personal) and offer his help. There were numerous occasions when I was surprised at his complex yet clear analytical reasonings. I'd also like to thank Himani Apte and Meenali Rungta for letting me participate in their course project. While trying to answer their in-depth questions, I learnt a lot about the internals of ext3.

I'd like to thank my friends at Madison. I don't have any of my family members in United States. But, when I'm with my friends I never felt bad about being away from home. To name a few: Arini Balakrishnan, Gogul Balakrishnan, Vinod Deivasigamany, Aditi Ganesan, Karthik Jayaraman, Vignesh Kannappan, Indirajith Krishnamurthy, Ramanathan Palaniappan, Prabu Ravindran, Muthian Sivathanu, Sanjay Sribalusu, and Joseph Stanley. They were happy even at my small successes and supported me during difficult times. I'm going to miss all the fun we had: screening late-night movies, watching Disney channel with Gogul, Prabu's witty comments, and evening coffee conversations with Arini (whose concerns about my eating habits reminds me of my Mother).

I'm grateful to my family back in India. My Grandpa's regular phone calls (in spite of his old age) were a great source of encouragement. Whenever I felt low during these years, I'd call up my sister and her family. A few minutes of laugh with them helped me relax more than anything. Finally, I'm grateful to my parents. They had faith in me when I went against their wishes to pursue the doctorate. I've always been an irresponsible son pursuing his own desires than supporting the family. But, they neither complained nor showed me their personal struggles. Instead, they rejoiced even at my smallest successes, showering me with all their love, and kept me always feeling great like a little prince. Thanks Ma and Pa, I hope I've made you all proud.

Contents

Chapter 1

Introduction

The importance of building dependable systems cannot be overstated. One of the fundamental requirements in computer systems is to store and retrieve information reliably. Disk drives have been widely used as a primary storage medium for several decades in many systems including but not limited to, personal computers (desktops, laptops), distributed file systems, database systems, high end storage arrays, archival systems, and mobile devices.

Unfortunately, disk failures can occur. Traditionally, systems have been built with the assumption that disks operate in a "fail stop" manner [99]; within this classic model, the disks either are working perfectly, or fail absolutely and in an easily detectable manner. Based on this assumption, storage systems such as RAID arrays have been built to tolerate whole disk failures [81]. For example, a file system or database system can store its data on a RAID array and withstand failure of an entire disk drive.

The fault model presented by modern disk drives, however, is much more complex. For example, modern drives can exhibit *latent sector faults* [14, 28, 45, 60, 100], where a block or set of blocks are inaccessible. Under latent sector fault, the sector fault occurs sometime in the past but the fault is detected only when the sector is accessed for storing or retrieving information [59]. Blocks sometimes become *corrupted* [16] and worse, this can happen *silently* without the disk being able to detect it [47, 74, 126]. Finally, disks sometimes exhibit *transient* performance problems [11, 115].

There are several reasons for these complex disk failure modes. First, a trend that is common in the drive industry is to pack more bits per square inch (BPS) as the areal densities of disk drives are growing at a rapid rate [48]. As density increases, errors such as bit spillovers on adjacent tracks may occur and the effects of such errors become larger at higher areal densities as they can corrupt more bits [4]. In addition, increased density can also increase the complexity of the logic, that is the firmware that manages the data [7], which can result in increased number of bugs. For example, buggy firmwares are known to issue *misdirected* writes [126], where correct data is placed on disk but in the wrong location.

Second, increased use of low-end desktop drives such as the IDE/ATA drives worsens the reliability problem. Low cost dominates the design of personal storage drives [7] and therefore, they are less tested and have less machinery to handle disk errors [56]. It is important we consider the ATA drives here because they are not only used in our laptops and desktops but also in high-end storage arrays such as EMC Centera [36, 50] and large-scale internet clusters like Google [40]. While personal storage drives are designed for active use only several hours per day, they are less likely to perform reliably under operational stresses of enterprise systems [7].

Finally, amount of software used on the storage stack has increased. Firmware on a desktop drive contains about 400 thousand lines of code [33]. Moreover, the storage stack consists of several layers of low-level device driver code that have been considered to have more bugs than the rest of the operating system code [38, 113]. As Jim Gray points out in his study of Tandem Availability, "As the other components of the system become increasingly reliable, software necessarily becomes the dominant cause of outages" [44].

Developers of high-end systems have realized the nature of these disk faults and built mechanisms into their systems to handle them. For example, many redundant storage systems incorporate a background *disk scrubbing* process [59, 100] to proactively detect and subsequently correct latent sector faults by creating a new copy of inaccessible blocks (although, auditing the disk for latent sector faults can itself increase the rate of visible or latent faults [14]); some recent storage arrays incorporate extra levels of redundancy to lessen the potential damage of undiscovered latent faults [28]. Finally, highly-reliable systems (*e.g.*, Tandem NonStop) utilize end-to-end checksums to detect when block corruption occurs [16].

The above said failure characteristics (latent sector faults and block corruption), trends (*e.g.*, increased use of cheap drives), and our knowledge about reliable high-end systems raise the question: *how do commodity file systems handle disk failures?* We derived the answer to the above question from our file system analysis and it led us to the second question: *how can we improve the robustness of commodity file systems?* Our research efforts to answer the above two questions form two broad parts of this dissertation. We discuss more about each of these parts in the following sections.

1.1 Analysis of File System Robustness

In this dissertation, the first question we pose is: how do modern commodity file systems react to failures that are common in modern disks? Although the same question is applicable to other data management systems such as databases, distributed file systems, and even RAID storage arrays, we focus and limit the scope of the thesis to local file systems that we run on our personal computers for three reasons. First, commodity file systems are important; they are not only pervasive in the home environment, storing valuable (and often non-archived) user data such as photos, home movies, and tax returns, but also used in high end storage arrays such as EMC Centera [36, 50] and within distributed file systems such as NFS [95]. Understanding how commodity file systems handle disk failures and improving their robustness will shed light into the robustness of other complex systems that use commodity file systems. Second, commodity file systems, specifically open source file systems, may often be designed and developed by inexperienced and free lance programmers, who might not be aware of complex disk failure modes. Therefore, it is necessary to analyze these file systems to unearth the bugs, design flaws, and assumptions made by the developer. Finally, commodity file systems are often used with cheap hardware devices without any hardware redundancy such as multiple disks or expensive hardware support such as NVRAM (both of which are usually available in high-end systems), and this raises new challenges that have not been addressed before. Among the commodity file systems, our analysis centers around journaling file systems because most modern file systems such as Linux ext3 [121], ReiserFS [89], JFS [19], XFS [112], and Windows NTFS [109] implement journaling in order to maintain file system integrity.

As a first step to understand disk failure handling in file system, we aggregate knowledge from the research literature, industry, and field experience to form a new model for disk failure. We label our model the *fail-partial failure model* to emphasize that portions of the disk can fail, either through block errors or data corruption.

With the model in place, we develop and apply an automated *failure-policy fingerprinting* framework, to inject more realistic disk faults beneath a file system. The goal of fingerprinting is to unearth the failure policy of each system: how it detects and recovers from disk failures. Instead of injecting faults randomly in file system traffic, we develop and apply a unique *Semantic Failure Analysis (SFA)* technique wherein we fail particular file system blocks based on their *types* and *transactional states*. Specifically, for write failure analysis, we develop a novel *transactional state specific* fault-injection technique, wherein we build an abstract model of file system 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. For read failure analysis, we simply fail different block types. Our semantic failure analysis approach leverages gray-box knowledge [9, 108] of file system data structures to meticulously exercise file system access paths to disk across a wide range of about 30 different workloads.

To better characterize failure policy, we develop an *Internal RObustNess (IRON)* taxonomy, which catalogs a broad range of detection and recovery techniques that a file system can use to handle disk failures. For example, a file system can use error codes from lower layers to detect if its read request has completed successfully and use redundant information such as a replica to recover from the failure. The output of our fingerprinting tool is a broad categorization of which IRON techniques a file system uses across its constituent data structures.

Our study focuses on four important and substantially different open-source file systems, ext3 [121], ReiserFS [89], IBM's JFS [19], and XFS [112] and one closed-source file system, Windows NTFS [109]. From our analysis results, we find that the technology used by high-end systems (*e.g.*, checksumming, disk scrubbing, and so on) has not filtered down to the realm of commodity file systems. Across all platforms, we find *ad hoc* failure handling and a great deal of *illogical inconsistency* in failure policy, often due to the *diffusion* of failure handling code through the kernel; such inconsistency leads to substantially different detection and recovery strategies under similar fault scenarios, resulting in unpredictable and often undesirable fault-handling strategies. Moreover, failure handling diffusion makes it difficult to examine any one or few portions of the code and determine how failure handling is supposed to behave. Diffusion also implies that failure handling is *inflexible*; policies that are spread across so many locations within the code base are hard to change. In addition, we observe that failure handling is quite *coarse-grained*; it is challenging to implement nuanced policies in the current system.

We also discover that most systems implement portions of their failure policy *incorrectly*; the presence of bugs in the implementations demonstrates the difficulty and complexity of correctly handling certain classes of disk failure. We observe little tolerance to transient failures; most file systems assume a single temporarily-inaccessible block indicates a fatal whole-disk failure. We show that none of the file systems can recover from partial disk failures, due to a lack of *in-disk redundancy*.

Finally, in addition to fingerprinting the failure policy, our analysis also helps us find several bugs within file systems that can catastrophically affect on-disk data. For example, under certain write failures, all the analyzed file systems commit failed transactions to disk; doing so can lead to serious problems, including an unmountable file system.

1.2 Building Robust File Systems

This behavior under realistic disk failures led us to our second question: how can we change file systems to better handle modern disk failures? The file system should not view the disk as an utterly reliable component. For example, if blocks can become corrupt, the file system should apply measures to both detect and recover from such corruption, even when running on a single disk. Our approach is an instance of the end-to-end argument [94]: at the top of the storage stack, the file system (or other data management system) is fundamentally responsible for reliable management of its data and metadata. In this thesis, we present a number of techniques (some of which are already implemented in high-end systems) that we design, implement, and evaluate in a commodity file system (*i.e.*, Linux ext3). The challenge in building such a file system is that one has to use the limited resources available on a typical desktop to implement the various techniques and not to overly reduce the performance which could make the system less attractive.

There are two main challenges in building a robust file system. First, it is important to design and implement the *low-level redundancy machinery* efficiently such that the overheads are not prohibitive. Second, even if the necessary redundant information is present, it is harder to incorporate them in the current file system framework due to the diffused failure handling. To solve the first problem, we design new redundancy techniques that can be implemented efficiently on a low-end system. To handle the second problem, we rearchitect commodity file systems with a centralized failure handling framework that unifies the low-level redundancy machinery with high-level failure policies.

In our initial efforts, we develop a family of prototype IRON file systems, all of which are robust variants of the Linux ext3 file system. Within our IRON ext3 (ixt3), we investigate the costs of using checksums to detect data corruption, replication to provide redundancy for metadata structures, and a simple parity protection for user data. We show that these techniques incur modest space and time overheads while greatly increasing the robustness of the file system to latent sector faults and data corruption. By implementing detection and recovery techniques from the IRON taxonomy, a system can implement a well-defined failure policy and subsequently provide vigorous protection against the broader range of disk failures.

Our next step in building robust file systems is to address two specific problems that arise in the realm of personal computers with single disk drives: *spatially local sector errors* [59] and *lack of hardware support* like non-volatile RAM (NVRAM).

In this context, we design, implement, and evaluate new techniques to store redundant information on disk, focusing primarily on parity as the redundant information due to its simplicity and wide-spread use.

First, we address the problem of spatially local errors by proposing a probabilistic disk failure model that characterizes disk block errors with spatial correlation and use this model to layout blocks such that the probability of two or more errors affecting related blocks is low. Second, we address the lack of NVRAM support in low-end systems by developing new parity update techniques. Specifically, we design two block update techniques where one technique updates the data blocks in the traditional fashion by *overwriting* the old contents while the other radical approach, called *no-overwrite*, issues writes to different locations on the disk and changes the block pointers to point to the new locations.

We evaluate the validity of the probabilistic model by comparing it with results from fault injection experiments. The probability of multiple failures as predicted by the failure model matches closely with the results from the fault injection experiments. We also evaluate the performance overheads of parity update techniques and show that both overwrite and no-overwrite techniques incur performance overhead when compared to ext3 and the cost varies from less than 1 percent for read-only workloads to about 50 percent for highly synchronous, write intensive workloads. Also, no-overwrite technique can fragment a file and therefore running a de-fragmenter periodically can improve performance.

In the last part of this dissertation, we seek to improve the state of the art of failure-handling in modern file systems by solving the following problem we noticed from our analysis: failure handling in commodity file systems is diffused resulting in ad hoc, inconsistent, and coarse-grained failure policies. Our approach here begins with the age-old wisdom of separating policy and mechanism [66], which we achieve with the design and implementation of a *Centralized Failure Handler* (CFH). All important failure-handling decisions are routed through the CFH, which is thus responsible for enforcing the specific failure-handling policy of the system. All relevant information is also made available within the CFH, hence enabling the construction of the desired failure-handling policies.

We demonstrate the power of centralized failure handling in a modified version of the Linux ext3 file system (ext3$_C$). We first show how ext3$_C$ is *flexible* by implementing a broad range of vastly different policies. For example, we show how ext3$_C$ can mimic the failure-handling behavior of different file systems such as ReiserFS and NTFS, all with changes to only a few lines of code. We also demonstrate that ext3$_C$ is inherently *consistent*; because policies are specified within a single localized portion of the code, the desired failure handling is enacted. Because of the presence of block-type information within the CFH, ext3$_C$ also enables *fine-grained policies* to be developed. For example, we show that ext3$_C$ can implement a highly paranoid policy for its own metadata while passing user data errors onto the application. The end result is a file system that handles disk failures in a comprehensive, thorough, and easily understood fashion; as a result, ext3$_C$ reacts in the expected manner even when an unexpected disk failure occurs.

1.3 Contributions

The contributions of this dissertation are as follows:

- We aggregate knowledge from various sources and define a more realistic failure model for modern disks, which we call the fail-partial model. We also formalize the techniques that a file system can use to detect and recover from disk errors under the IRON taxonomy.

- We develop a fingerprinting framework to determine the failure policy of a file system. This framework is useful both for academic study of said file systems as well as to developers who could apply it to test their systems and improve their robustness. We use this framework to analyze five popular commodity file systems to discover how they handle disk errors.

- We explore effectiveness and cost of different redundant information such as checksums, parity, and replica by building a prototype version of an IRON file system (ixt3). We apply redundancy to both file system data and metadata in various combinations and analyze the file system robustness to disk failure and its performance characteristics.

- We propose a probabilistic disk block failure model that considers spatial correlation in block errors to derive the probability of 2 or more failures. We use this model to separate redundant information in block layouts to avoid spatially local errors.

- We develop two different data update approaches, overwrite and no-overwrite, that differ in the location of block writes and their interaction with the file system journaling.

- We design a centralized failure handler (CFH) to unify failure handling in file system. We evaluate CFH and show that it enables flexible, consistent, and fine-grained failure policies in file system.

1.4 Outline

The rest of this dissertation is organized as follows. In Chapter 2, we present the common causes of disk failures following it with a discussion on fail-partial model and IRON taxonomy. In Chapter 3, we give a brief introduction to journaling file systems and different commodity file systems we analyze. Chapter 4 explains our failure policy analysis. where we discuss our semantic failure analysis methodology and results. In Chapter 5, we present the details of building a robust file system. We start with a broad exploration of different redundant information followed by a discussion of the system evaluation. Then, we explain our techniques to address spatially correlated errors and lack of hardware support. Chapter 6 presents our centralized failure handler for file systems. Related work is discussed in Chapter 7, and conclusion and future directions are presented in Chapter 8.

Chapter 2

Fail-Partial Failure Model

In order to understand how commodity file systems handle disk failures, first, we must understand the storage system architecture and the ways it can fail. Second, we need a standard taxonomy of various detection and recovery techniques that a file system can use to handle disk failures in order to classify their failure handling policies.

In this chapter, we first explain the components of a storage stack, which is a complex layered collection of electrical, mechanical, firmware and software components (Section 2.2). There are many reasons that the file system may see faults in the storage system below and we discuss common causes of disk failures (Section 2.3). We then present a new, more realistic *fail-partial model* for disks and discuss various aspects of this model (Section 2.4). We follow this with a discussion about the techniques that an IRON file system can use to detect and recover from disk errors (Section 2.5). Finally, we conclude the chapter by reasoning why failure handling must be performed at the file system (Section 2.5.4) and why RAID is not a complete solution to improve storage reliability (Section 2.5.5).

2.1 Terminology

Before we proceed to the details, we will clarify certain terminologies in order to make the following discussion unambiguous and easy to follow. We follow the IEEE Standard Software Engineering Terminology [22] and use the following definitions for *error*, *fault*, and *failure*.

- **Error:** 1. The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. 2. An incorrect step, process, or data definition. 3. An incorrect result.

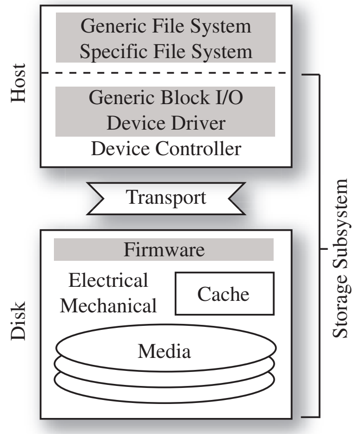


Figure 2.1: **The Storage Stack.** *We present a schematic of the entire storage stack. At the top is the file system; beneath are the many layers of the storage subsystem. Gray shading implies software or firmware, whereas white (unshaded) is hardware.*

- **Fault:** 1. A defect in a hardware device or component *e.g.*, a defective disk sector. 2. An incorrect step, process, or data definition in a computer program (*e.g.*, software bugs). In common usage, the terms "error" and "bug" are used to express the same meaning as "fault" [22]. In the context of this thesis, we use the terms "error" and "fault" interchangeably.

- **Failure:** The inability of a system or component to perform its required functions within specified performance requirements. Simply put, the result of a fault is a failure. For example, a disk failure occurs if the storage stack is unable to recover a faulty sector.

As we mentioned in Section 1.1, in this thesis we focus and limit our scope only to local file systems. For the rest of the thesis unless otherwise stated, we use the term "file system" to refer to a specific file system on top of the storage stack (*i.e.*, the end-point on the storage stack).

2.2 The Storage Subsystem

Figure 2.1 presents a typical layered storage subsystem below the file system (similar and perhaps, more complex layered storage architectures are used by other data management systems like databases and distributed file systems). The storage stack consists of three main layers: the host, the transport layer, and the disk. An error can occur in any of these layers and propagate itself to the file system above.

At the bottom of the "storage stack" is the disk itself; beyond the magnetic storage media, there are mechanical (*e.g.*, the motor and arm assembly), electrical components (*e.g.*, buses), and cache. A particularly important component is firmware – the code embedded within the drive to control most of its higher-level functions, including caching, disk scheduling, and error handling. This firmware code is often substantial and complex (*e.g.*, a modern Seagate drive contains roughly 400,000 lines of code [33]).

Connecting the drive to the host is the transport. In low-end systems, the transport medium is often a bus (*e.g.*, SCSI), whereas networks are common in higher-end systems (*e.g.*, FibreChannel). If a hardware RAID is used instead of a single disk, an additional layer of RAID controller is added between the transport layer and disk drives. A hardware RAID controller manages multiple disk drives and exports a standard interface (*e.g.*, SCSI) to the transport layer.

At the top of the stack is the host. Herein there is a hardware controller that communicates with the device, and above it a software device driver that controls the hardware. Block-level software forms the next layer, providing a generic device interface and implementing various optimizations (*e.g.*, request reordering). Modern operating systems such as Windows and Linux provide support to stack multiple layers of software components such as volume managers or software RAID layers beneath the file system and above the device specific drivers, resulting in a more complex storage stack.

Above all other software is the file system. This layer is often split into two pieces: a high-level component common to all file systems, and a specific component that maps generic operations onto the data structures of the particular file system. A standard interface (*e.g.*, Vnode/VFS [63]) is positioned between the two.

2.3 Why Disks Fail?

To motivate our failure model, we first describe how errors in the layers of the storage stack can cause failures. In our discussion, we begin at the bottom of the stack and work our way up to the file system.

Media: There are two primary errors that occur in the magnetic media. First, the classic problem of "bit rot" occurs when the magnetism of a single bit or a few bits is flipped. This type of problem can often (but not always) be detected and corrected with low-level ECC embedded in the drive. Second, physical damage can occur on the media. The quintessential "head crash" is one culprit, where the drive head contacts the surface momentarily. A media scratch can also occur when a particle is trapped between the drive head and the media [100]. Thermal asperity is a spike in the read signal caused by disk asperities or contaminant particles, which can cause the disk heads to lose their reading capabilities temporarily [124]. Such dangers are well-known to drive manufacturers, and hence modern disks park the drive head when the drive is not in use to reduce the number of head crashes; SCSI disks sometimes include filters to remove particles [7]. Media errors most often lead to permanent failure or corruption of individual disk blocks.

Mechanical: "Wear and tear" eventually leads to failure of moving parts. A drive motor can spin irregularly or fail completely. Erratic arm movements can cause head crashes and media flaws; inaccurate arm movement can misposition the drive head during writes, leaving blocks inaccessible or corrupted upon subsequent reads.

Electrical: A power spike or surge can damage in-drive circuits and hence lead to drive failure [116]. Thus, electrical problems can lead to entire disk failure.

Drive firmware: Interesting errors arise in the drive controller, which consists of many thousands of lines of real-time, concurrent firmware. For example, disks have been known to return correct data but circularly shifted by a byte [67] or have memory leaks that lead to intermittent failures [116]. Other firmware problems can lead to poor drive performance [97]. Drive manufacturers often introduce new firmware versions to fix existing problems on production drives and such fixes can inadvertently increase the failure rates of the drive [103]. Some firmware bugs are well-enough known in the field that they have specific names; for example, *misdirected* writes are writes that place the correct data on the disk but in the wrong location, and *phantom* writes are writes that the drive reports as completed but that never reach the media [126]. Phantom writes can be caused by a buggy or even misconfigured cache (*i.e.*, write-back caching is enabled). In summary, drive firmware errors often lead to sticky or transient block corruption but can also lead to performance problems.

Transport: The transport connecting the drive and host can also be problematic. For example, a study of a large disk farm [115] reveals that most of the systems tested had interconnect problems, such as bus timeouts. Parity errors also occurred with some frequency, either causing requests to succeed (slowly) or fail altogether. Thus, the transport often causes transient errors for the entire drive.

Bus controller: The main bus controller can also be problematic. For example, the EIDE controller on a particular series of motherboards incorrectly indicates completion of a disk request before the data has reached the main memory of the host, leading to data corruption [125]. A similar problem causes some other controllers to return status bits as data if the floppy drive is in use at the same time as the hard drive [47]. Others have also observed IDE protocol version problems that yield corrupt data [40]. In summary, controller problems can lead to transient block failure and data corruption.

Low-level drivers: Recent research has shown that device driver code is more likely to contain bugs than the rest of the operating system [27, 38, 113]. While some of these bugs will likely crash the operating system, others can issue disk requests with bad parameters, data, or both, resulting in data corruption.

2.4 The Fail-Partial Failure Model

From our discussion of the many root causes for failure, we are now ready to put forth a more realistic model of disk failure. In our model, failures manifest themselves in three ways:

- **Entire disk failure:** The entire disk is no longer accessible. If permanent, this is the classic "fail-stop" failure.
- **Block failure:** One or more blocks are not accessible; often referred to as *latent sector faults* [59, 60].
- **Block corruption:** The data within individual blocks is altered. Corruption is particularly insidious because it is silent – the storage subsystem simply returns "bad" data upon a read.

We term this model the *Fail-Partial Failure Model*, to emphasize that pieces of the storage subsystem can fail. We now discuss some other key elements of the fail-partial model, including the transience, locality, and frequency of failures, and then discuss how technology and market trends will impact disk failures over time.

2.4.1 Transience of Failures

In our model, failures can be "sticky" (permanent) or "transient" (temporary). Which behavior manifests itself depends upon the root cause of the problem. For example, a low-level media problem portends the failure of subsequent requests. In contrast, a transport or higher-level software issue might at first cause block failure or corruption; however, the operation could succeed if retried.

2.4.2 Locality of Failures

Because multiple blocks of a disk can fail, one must consider whether such block failures are dependent. The root causes of block failure suggest that some forms of block failure do indeed exhibit spatial locality [60]. For example, a scratched surface or thermal asperity can render a number of contiguous blocks inaccessible. However, all failures do not exhibit locality; for example, a corruption due to a misdirected write may impact only a single block.

2.4.3 Frequency of Failures

Block failures and corruptions do occur – as one commercial storage system developer succinctly stated, "Disks break a lot – all guarantees are fiction" [52]. However, one must also consider how frequently such errors occur, particularly when modeling overall reliability and deciding which failures are most important to handle. Unfortunately, as Talagala and Patterson point out [115], disk drive manufacturers are loathe to provide information on disk failures; indeed, people within the industry refer to an implicit industry-wide agreement to not publicize such details [5]. Not surprisingly, the actual frequency of drive errors, especially errors that do not cause the whole disk to fail, is not well-known in the literature. Previous work on latent sector faults indicates that such errors occur more commonly than absolute disk failure [60], and a related research work estimates that such errors may occur five times more often than absolute disk failures [100]. Recently, more efforts have been made to quantify the disk error rates. Specifically, Gray *et al.* measure about 30 uncorrectable bit errors at the file system while moving about 2 petabytes of data [45].

In terms of relative frequency, block failures are more likely to occur on reads than writes, due to internal error handling common in most disk drives. For example, failed writes to a given sector are often remapped to another (distant) sector, allowing the drive to transparently handle such problems [56]. However, remapping does not imply that writes cannot fail. A failure in a component above the media (*e.g.*, a stuttering transport), can lead to an unsuccessful write attempt; the move to network-attached storage [42] serves to increase the frequency of this class of failures. Also, for remapping to succeed, free blocks must be available; a large scratch could render many blocks unwritable and quickly use up reserved space. Reads are more problematic: if the media is unreadable, the drive has no choice but to return an error.

2.4.4 Trends

In many other areas (*e.g.*, processor performance), technology and market trends combine to improve different aspects of computer systems. In contrast, we believe that technology trends and market forces may combine to make storage system failures occur *more* frequently over time, for the following three reasons.

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 or more blocks	Detect corruption in end-to-end way

Table 2.1: **The Levels of the IRON Detection Taxonomy.**

First, reliability is a greater challenge when drives are made increasingly more dense; as more bits are packed into smaller spaces, drive logic (and hence complexity) increases [7].

Second, at the low-end of the drive market, cost-per-byte dominates, and hence many corners are cut to save pennies in IDE/ATA drives [7]. Low-cost "PC class" drives tend to be tested less and have less internal machinery to prevent failures from occurring [56]. The result, in the field, is that ATA drives are observably less reliable [115]; however, cost pressures serve to increase their usage, even in server environments [40].

Finally, the amount of software is increasing in storage systems and, as others have noted, software is often the root cause of errors [44]. In the storage system, hundreds of thousands of lines of software are present in the lower-level drivers and firmware. This low-level code is generally the type of code that is difficult to write and debug [38, 113] – hence a likely source of increased errors in the storage stack.

2.5 The IRON Taxonomy

In this section, we outline strategies for developing an IRON file system, *i.e.*, a file system that detects and recovers from a range of modern disk failures. Our main focus is to develop different strategies, not *across* disks as is common in storage arrays, but *within* a single disk. Such Internal RObustNess (IRON) provides much of the needed protection within a file system.

To cope with the failures in modern disks, an IRON file system includes machinery to both *detect* (Level D) partial faults and *recover* (Level R) from them.

Level	Technique	Comment
R_{Zero}	No recovery	Assumes disk works
$R_{Propagate}$	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}	Repair data structs	Could lose data
R_{Remap}	Remaps block or file to different locale	Assumes disk informs FS of failures
$R_{Redundancy}$	Block replication or other forms	Enables recovery from loss/corruption

Table 2.2: **The Levels of the IRON Recovery Taxonomy.**

Tables 2.1 and 2.2 present our IRON detection and recovery taxonomies, respectively. Note that the taxonomy is by no means complete. Many other techniques are likely to exist, just as many different RAID variations have been proposed over the years [3, 127].

The detection and recovery mechanisms employed by a file system define its *failure policy*. Currently, it is difficult to discuss the failure policy of a system. With the IRON taxonomy, one can describe the failure policy of a file system, much as one can already describe a cache replacement or a file-layout policy.

2.5.1 Levels of Detection

Level D techniques are used by a file system to detect that a problem has occurred (*i.e.*, that a block cannot currently be accessed or has been corrupted).

• $Zero$: The simplest detection strategy is none at all; the file system assumes the disk works and does not check return codes. As we will see in Section 4.3, this approach is surprisingly common (although often it is applied unintentionally).

• $ErrorCode$: A more pragmatic detection strategy that a file system can implement is to check return codes provided by the lower levels of the storage system.

• $Sanity$: With sanity checks, the file system verifies that its data structures are consistent. This check can be performed either within a single block or across blocks.

When checking a single block, the file system can either verify individual fields (*e.g.*, that pointers are within valid ranges) or verify the *type* of the block. For example, most file system superblocks include a "magic number" and some older file systems such as Pilot even include a header per data block [88]. By checking whether a block has the correct type information, a file system can guard against some forms of block corruption.

Checking across blocks can involve verifying only a few blocks (*e.g.*, that a bitmap corresponds to allocated blocks) or can involve periodically scanning all structures to determine if they are intact and consistent (*e.g.*, similar to `fsck` [72]). Even journaling file systems can benefit from periodic full-scan integrity checks. For example, a buggy journaling file system could unknowingly corrupt its on-disk structures; running `fsck` in the background could detect and recover from such problems.

• *Redundancy*: The final level of the detection taxonomy is redundancy. Many forms of redundancy can be used to detect block corruption. For example, *checksumming* has been used in reliable systems for years to detect corruption [16] and has recently been applied to improve security as well [79, 110]. Checksums are useful for a number of reasons. First, they assist in detecting classic "bit rot", where the bits of the media have been flipped. However, in-media ECC often catches and corrects such errors. Checksums are therefore particularly well-suited for detecting corruption in higher levels of the storage system stack (*e.g.*, a buggy controller that "misdirects" disk updates to the wrong location or does not write a given block to disk at all). However, checksums must be carefully implemented to detect these problems [16, 126]; specifically, a checksum that is stored along with the data it checksums will not detect such misdirected or phantom writes.

Higher levels of redundancy, such as block mirroring [20], parity [78, 81] and other error-correction codes [69], can also detect corruption. For example, a file system could keep three copies of each block, reading and comparing all three to determine if one has been corrupted. However, such techniques are truly designed for correction (as discussed below); they often assume the presence of a lower-overhead detection mechanism [81].

2.5.2 Detection Frequency

All detection techniques discussed above can be applied *lazily*, upon block access, or *eagerly*, perhaps scanning the disk during idle time. We believe IRON file systems should contain some form of lazy detection and should additionally consider eager methods.

For example, *disk scrubbing* is a classic eager technique used by RAID systems to scan a disk and thereby discover latent sector faults [60]. Disk scrubbing is particularly valuable if a means for recovery is available, that is, if a replica exists to repair the now-unavailable block. To detect whether an error occurred, scrubbing typically leverages the return codes explicitly provided by the disk and hence discovers block failure but not corruption. If combined with other detection techniques (such as checksums), scrubbing can discover block corruption as well.

2.5.3 Levels of Recovery

Level R of the IRON taxonomy facilitates recovery from block failure within a single disk drive. These techniques handle both latent sector faults and block corruptions.

• $Zero$: Again, the simplest approach is to implement no recovery strategy at all, not even notifying clients that a failure has occurred.

• $Propagate$: A straightforward recovery strategy is to propagate errors up through the file system; the file system informs the application that an error occurred and assumes the client program or user will respond appropriately to the problem.

• $Stop$: One way to recover from a disk failure is to stop the current file system activity. This action can be taken at many different levels of granularity. At the coarsest level, one can crash the entire machine. One positive feature is that this recovery mechanism turns all *detected* disk failures into fail-stop failures and likely preserves file system integrity. However, crashing assumes the problem is transient; if the faulty block contains repeatedly-accessed data (*e.g.*, a script run during initialization), the system may repeatedly reboot, attempt to access the unavailable data, and crash again. At an intermediate level, one can kill only the process that triggered the disk fault and subsequently mount the file system in a read-only mode. This approach is advantageous in that it does not take down the entire system and thus allows other processes to continue. At the finest level, a journaling file system can abort only the current transaction. This approach is likely to lead to the most available system, but may be more complex to implement.

• $Guess$: As recently suggested by Rinard *et al.* [91], another possible reaction to a failed block read would be to manufacture a response, perhaps allowing the system to keep running in spite of a failure. Such failure oblivious computing has received attention from other researchers as well. For example, Qin *et al.* use similar techniques to transparently recover from software failures [87]. The negative is that an artificial response may be less desirable than failing.

• *Retry*: A simple response to failure is to retry the failed operation and recent work shows that file systems do recover from most number of disk errors by simply retrying [45]. Retry can appropriately handle transient errors, but wastes time retrying if the failure is indeed permanent.

• *Repair*: If an IRON file system can detect an inconsistency in its internal data structures, it can likely repair them, just as `fsck` would. For example, a block that is not pointed to, but is marked as allocated in a bitmap, could be freed. As discussed above, such techniques are useful even in the context of journaling file systems, as bugs may lead to corruption of file system integrity.

• *Remap*: IRON file systems can perform block remapping. This technique can be used to fix errors that occur when writing a block, but cannot recover failed reads. Specifically, when a write to a given block fails, the file system could choose to simply write the block to another location. More sophisticated strategies could remap an entire "semantic unit" at a time (*e.g.*, a user file), thus preserving logical contiguity.

• *Redundancy*: Finally, redundancy (in its many forms) can be used to recover from block loss. The simplest form is *replication*, in which a given block has two (or more) copies in different locations within a disk. Another redundancy approach employs parity to facilitate error correction. Similar to RAID 4/5 [81], by adding a parity block per block group, a file system can tolerate the unavailability or corruption of one block in each such group. More complex encodings (*e.g.*, Tornado codes [69]) could also be used, a subject worthy of future exploration.

However, redundancy within a disk can have negative consequences. First, replicas must account for the spatial locality of failure (*e.g.*, a surface scratch that corrupts a sequence of neighboring blocks); hence, copies should be allocated across remote parts of the disk, which can lower performance. Second, in-disk redundancy techniques can incur a high space cost; however, in many desktop settings, drives have sufficient available free space [32].

2.5.4 Why IRON in the File System?

One natural question to ask is: why should the file system implement detection and recovery instead of the disk? Perhaps modern disks, with their internal mechanisms for detecting and recovering from errors, are sufficient.

In our view, the primary reason for detection and recovery within the file system is found in the end-to-end argument [94]; even if the lower-levels of the system implement some forms of fault tolerance, the file system must implement them as well to guard against all forms of failure. For example, the file system is the *only* place that can detect corruption of data in higher levels of the storage stack (*e.g.*, within the device driver or drive controller).

A second reason for implementing detection and recovery in the file system is that the file system has exact knowledge of how blocks are currently being used. Thus, the file system can apply detection and recovery intelligently across different block types. For example, the file system can provide a higher level of replication for its own metadata, perhaps leaving failure detection and correction of user data to applications (indeed, this is one specific solution that we explore in Section 5.2). Similarly, the file system can provide machinery to enable application-controlled replication of important data, thus enabling an explicit performance/reliability trade-off.

A third reason is performance: file systems and storage systems have an "unwritten contract" [98] that allows the file system to lay out blocks to achieve high bandwidth. For example, the unwritten contract stipulates that adjacent blocks in the logical disk address space are physically proximate. Disk-level recovery mechanisms, such as remapping, break this unwritten contract and cause performance problems. If the file system instead assumes this responsibility, it can itself remap logically-related blocks (*e.g.*, a file) and hence avoid such problems.

However, there are some complexities to placing IRON functionality in the file system. First, some of these techniques require new persistent data structures (*e.g.*, to track where redundant copies or parity blocks are located). Second, some mechanisms require control of the underlying drive mechanisms. For example, to recover on-disk data, modern drives will attempt different positioning and reading strategies [7]; no interface exists to control these different low-level strategies in current systems. Third, a file system may not know the exact disk geometry, which makes it harder to place redundant copies such that they tolerate spatially local errors.

2.5.5 Doesn't RAID Make Storage Reliable?

Another question that must be answered is: can't we simply use RAID techniques [81] to provide reliable and robust storage? We believe that while RAID can indeed improve storage reliability, it is not a complete solution, for the following three reasons.

First, not all systems incorporate more than one disk, the *sine qua non* of redundant storage systems. For example, desktop PCs currently ship with a single disk included; because cost is a driving force in the marketplace, adding a $100 disk to a $300 PC solely for the sake of redundancy is not a palatable solution.

Second, RAID alone does not protect against failures higher in the storage stack, as shown in Figure 2.1. Because many layers exist between the storage subsystem and the file system, and errors can occur in these layers as well, the file system must ultimately be responsible for detecting and perhaps recovering from such errors. Ironically, a complex RAID controller can consist of millions of lines of code [127], and hence be a source of faults itself.

Third, depending on the particular RAID system employed, not all types of disk faults may be handled. For example, lower-end RAID controller cards do not use checksums to detect data corruption, and only recently have some companies included machinery to cope with latent sector faults [28].

Hence, we believe that IRON techniques within a file system are useful for all single-disk systems, and even when multiple drives are used in a RAID-like manner. Although we focus on single-disk systems in this paper, we believe there is a rich space left for exploration between IRON file systems and redundant storage arrays.

Chapter 3

Background

Our failure policy analysis entirely focuses on journaling file systems as most modern file systems perform journaling. To understand the analysis methodology, it is necessary that we understand the general concepts of journaling, transaction, and checkpointing and some specific file system details like how the on-disk structures are updated.

Journaling file systems, in order to maintain the file system integrity, use the concept of transactions [46] and follow certain write ordering constraints when they issue their updates. Depending upon the file system, transactions may be composed of whole file system blocks or only modified records. In our analysis, we examine five different journaling file systems. Four of them are open source: Linux ext3 [121], ReiserFS [89], JFS [19], and XFS [112] and Windows NTFS [109] is closed source. There are two reasons for selecting these file systems. First, they are widely used being default file systems in several operating system configurations (*e.g.*, ext3 is the default file system in Red Hat Linux, ReiserFS in SuSE Linux distribution, and NTFS in Windows NT, 2000, and XP). Second, in addition to being used in personal computers, they are also used in high-end storage arrays [36, 50].

In this chapter, first we give a general introduction to journaling file systems (Section 3.1) and then briefly discuss different commodity journaling file systems (Sections 3.1.1 to 3.1.5).

3.1 Journaling File Systems

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

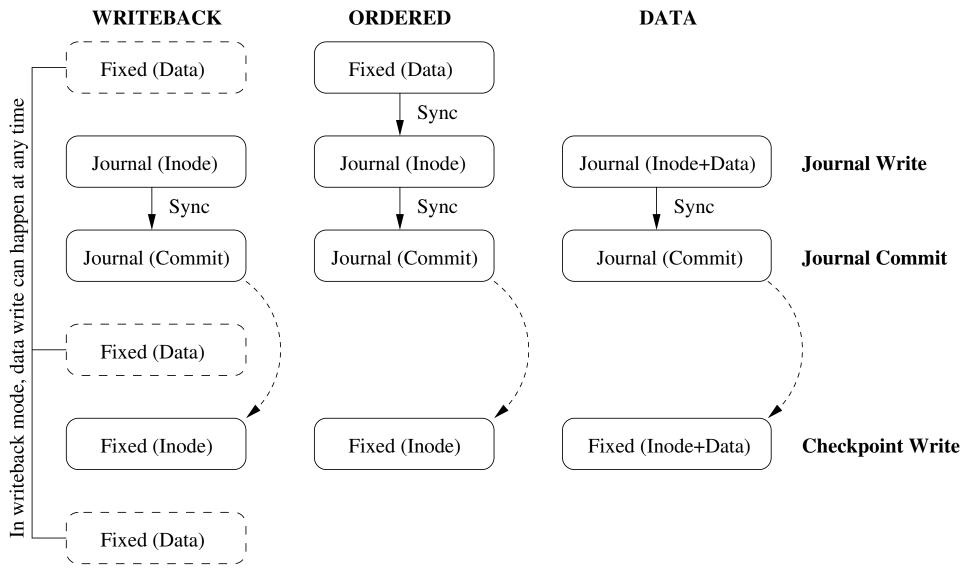


Figure 3.1: **Journaling Modes.** *The diagram depicts the three different journaling modes supported in modern file systems: writeback, ordered, and data. In the diagram, time flows downward. Boxes represent updates to the file system, e.g., "Journal (Inode)" implies the write of an inode to the journal; the other destination for writes is labeled "Fixed", which is a write to the fixed in-place ext2 structures. An arrow labeled with a "Sync" implies that the two blocks are written out in immediate succession synchronously, hence guaranteeing the first completes before the second. A curved arrow indicates ordering but not immediate succession; hence, the second write will happen at some later time. Finally, for writeback mode, the dashed box around the "Fixed (Data)" block indicates that it may happen at any time in the sequence. In this example, we consider a data block write and its inode as the updates that need to be propagated to the file system; the diagrams show how the data flow is different for each of the journaling modes.*

system is left in an inconsistent state. To repair the inconsistency, earlier systems such as FFS [71] and ext2 [26] scan the entire file system and perform integrity checks using fsck [72] 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 [46]. By forcing journal updates to disk *before* updating complex file system structures, this write-ahead logging technique enables efficient crash recovery; a simple scan of the journal and a redo of any incomplete committed operations bring the file system to a consistent state. During normal operation, the journal is treated as a circular buffer; once the necessary information has been propagated to its fixed location structures, journal space can be reclaimed.

Journaling Modes: Modern journaling file systems such as ext3 and ReiserFS present three flavors of journaling: *writeback mode*, *ordered mode*, and *data journaling mode*; Figure 3.1 illustrates the differences between these modes. The choice of mode is made at mount time and can be changed via a remount.

In *writeback mode*, only file system metadata is journaled; data blocks are written directly to their fixed location. This mode does not enforce any ordering between the journal and fixed-location data writes, and because of this, writeback mode has the weakest integrity and consistency semantics of the three modes. Although it guarantees integrity and consistency for file system metadata, it does not provide any corresponding guarantees to the data blocks.

For example, assume a process appends a block to a file; internally, the file system allocates a new pointer within the inode I and a new data block at address D to hold the data. In writeback mode, the file system is free to write D at any time, but is quite careful in how it updates I. Specifically, it will force a journal record to disk containing I, followed by a commit block. Once the commit block is written, the file system knows it can safely recover from a crash, and at some time later will write I to its fixed location structures. However, if I makes it to the log successfully, and a crash happens *before* D gets to disk, upon recovery I will point to D but the contents of D will not be correct.

In *ordered journaling mode*, again only metadata writes are journaled; however, data writes to their fixed location are ordered *before* the journal writes of the metadata. In contrast to writeback mode, this mode provides more sensible integrity semantics where a metadata block is guaranteed not to point to a block that does not belong to the file.

In continuing from our example above, D will be forced to its fixed location before I is written to the journal. Thus, even with an untimely crash, the file system will recover in a reasonable manner.

However, in ordered journaling mode, consistency between metadata and data might be affected. For example, if a data block is overwritten and the system crashes before writing the inode block to the journal, after recovery, the file system inode timestamps will be inconsistent with the recency of the data.

In full *data journaling mode*, the file system logs *both* metadata and data to the journal. This decision implies that when a process writes a data block, it will typically be written out to disk *twice*: once to the journal, and then later to its fixed location. Data journaling mode provides the strongest integrity and consistency guarantees; however, it has different performance characteristics, in some cases worse, and surprisingly, in some cases, better [84].

Transactions: Journaling file systems write their updates as part of a transaction to disk. However, instead of considering each file system update as a separate transaction, most modern file systems group many updates into a single *compound transaction* that is periodically committed to disk. This approach is relatively simple to implement [121]. Compound transactions may have better performance than more fine-grained transactions when the same structure is frequently updated in a short period of time (*e.g.*, a free space bitmap or an inode of a file that is constantly being extended) [51].

When file systems write their updates to the journal, they either write a full block, called *physical journaling* or write only the modified portion of the block, called *logical journaling*. Physical journaling incurs more journal traffic because even if a single bit is modified in a metadata block, the entire block is written to the journal. On the other hand, logical journaling carefully writes only modified records of file system structures and therefore, use the journal more efficiently. However, fixed-location structures can become corrupt, for example, if the file system is checkpointing the metadata at the precise moment when the power goes down (memory is more sensitive to power failures and can send garbages to disk if the power fails while data is being DMA'ed to the disk). Physical journaling has the advantage that if the fixed-location structures become corrupt, they can be recovered by simply scanning and replaying the successfully committed transactions [119].

Checkpointing: The process of writing journaled metadata and data to their fixed-locations is known as *checkpointing*. Checkpointing is triggered when various thresholds are crossed, *e.g.*, when file system buffer space is low, when there is little free space left in the journal, or when a timer expires.

Crash Recovery: Crash recovery is straightforward in many journaling file systems; a basic form of *redo logging* is used. Because new updates (whether to data or just metadata) are written to the log, the process of restoring in-place file system structures is easy. During recovery, the file system scans the log for committed complete transactions; incomplete transactions are discarded. Each update in a completed transaction is simply replayed into the fixed-place structures.

3.1.1 Ext3

Linux ext3 [121, 122] is the default journaling file system in several distributions of Linux (*e.g.*, Red Hat), and it is built as an extension to the ext2 file system. In ext3,

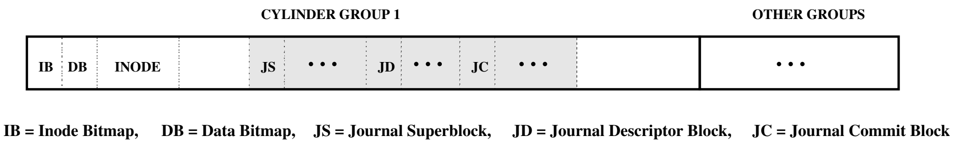


Figure 3.2: **Ext3 Layout.** *The picture shows the layout of an ext3 file system. The disk address space is broken down into a series of block groups (akin to FFS cylinder groups), each of which has bitmaps to track allocations and regions for inodes and data blocks. The ext3 journal is depicted here as a file within the first block group of the file system; it contains a superblock, various descriptor blocks to describe its contents, and commit blocks to denote the ends of transactions.*

data and metadata are eventually placed into the standard ext2 structures, which are the fixed-location structures. In this organization (which is loosely based on FFS [71]), the disk is split into a number of *block groups*; within each block group are bitmaps, inode blocks, and data blocks. The ext3 journal (or log) is commonly stored as a file within the file system, although it can be stored on a separate device or partition. Figure 3.2 depicts the ext3 on-disk layout.

Ext3 Journal Structure: Ext3 supports all the three journaling modes and uses additional metadata structures to track the list of journaled blocks. The *journal superblock* tracks summary information for the journal, such as the block size and head and tail pointers. A *journal descriptor block* marks the beginning of a transaction and describes the subsequent journaled blocks, including their final fixed on-disk location. In data journaling mode, the descriptor block is followed by the data and metadata blocks; in ordered and writeback mode, the descriptor block is followed by the metadata blocks. In all modes, ext3 logs full blocks (physical journaling), as opposed to differences from old versions; thus, even a single bit change in a bitmap results in the entire bitmap block being logged. Depending upon the size of the transaction, multiple descriptor blocks each followed by the corresponding data and metadata blocks may be logged. Finally, a *journal commit block* is written to the journal at the end of the transaction; once the commit block is written, the journaled data can be recovered without loss.

Ext3 Transactions: Ext3 maintains a single, system-wide, compound transaction into which all the file system updates are added and committed. Although group commit improves performance, it can potentially result in combining unrelated updates, which can lead to a tangled synchrony between asynchronous and synchronous traffic in the file system [84]. Ext3 commits and checkpoints transactions under the influence of three factors: application initiated synchronization calls, journal size, and commit timer settings.

3.1.2 ReiserFS

ReiserFS [89] is the default journaling file system under certain distributions of Linux such as SuSE. The general behavior of ReiserFS is similar to ext3. For example, both file systems support all the three journaling modes, both have compound transactions, and both perform physical journaling. However, ReiserFS differs from ext3 in three primary ways.

First, the two file systems use different on-disk structures to track their fixed-location data. Ext3 uses the same structures as ext2; for improved scalability, ReiserFS uses a B+ tree, in which data is stored on the leaves of the tree and the metadata is stored on the internal nodes. Since the impact of the fixed-location data structures is not the focus of this thesis, this difference is largely irrelevant.

Second, the format of the journal is slightly different. In ext3, the journal can be a file, which may be anywhere in the partition and may not be contiguous. The ReiserFS journal is not a file and is instead a contiguous sequence of blocks at the beginning of the file system; as in ext3, the ReiserFS journal can be put on a different device. Further, ReiserFS limits the journal to a maximum of 32 MB.

Third, ext3 and ReiserFS differ slightly in their journal contents. In ReiserFS, the fixed locations for the blocks in the transaction are stored not only in the descriptor block but also in the commit block. Also, unlike ext3, ReiserFS uses only one descriptor block in every compound transaction, which limits the number of blocks that can be grouped in a transaction.

3.1.3 JFS

JFS is a journaling file system developed by IBM, which was designed with the journaling support fully integrated from the start rather than adding it later. The journal is located, by default, at the end of the partition and is treated as a contiguous sequence of blocks [18]. One cannot specify any journaling mode during the file system mount.

In a previous work, we infer that JFS uses ordered journaling mode [84]. Due to the small amount of traffic to the journal, it was obvious that it was not employing data journaling. To differentiate between writeback and ordered modes, we observe that the ordering of writes matched that of ordered mode. That is, when a data block is written by the application, JFS orders the write such that the data block is written successfully before the metadata writes are issued.

JFS does logging at the record level (logical journaling). That is, whenever an inode, index tree, or directory tree structure changes, only that structure is logged instead of the entire block containing the structure. As a result, JFS writes fewer journal blocks than ext3 and ReiserFS for the same operations.

JFS does not by default group concurrent updates into a single compound transaction although, there are circumstances in which transactions are grouped: for example, if the write commit records are on the same log page.

Finally, there are no commit timers in JFS and the fixed-location writes happen whenever the Linux kernel timer (*i.e.*, the kupdate daemon) expires. However, the journal writes are never triggered by the timer: journal writes are indefinitely postponed until there is another trigger such as memory pressure or an unmount operation. This *infinite write delay* can limit reliability, as a crash can result in data loss even for data that was written minutes or hours before.

3.1.4 XFS

XFS is a 64-bit journaling file system developed by SGI. XFS supports only ordered journaling mode and data/writeback journaling modes are not present [112]. Similar to JFS, XFS journals only records in the log instead of writing whole blocks. Since only records are logged, XFS has no separate journal commit block; a journal block contains different types of records and a commit record is one of them. Also, there is no separate journal super block. Usually, the journal super block (or its equivalent) stores the 'dirty' state of the journal, which is used to determine whether the journal has to be replayed on a mount. In XFS, on every mount, the log is searched to find any transaction that needs to be replayed. On a clean unmount, XFS writes an 'unmount' record into the journal to mark the journal as clean.

The fixed-location structures of XFS consist of allocation groups where each allocation group manages its own inodes and free space. Since each allocation group is, in effective, an independent entity, the kernel can interact with multiple groups simultaneously. Internally, each allocation group uses B+ trees to keep track of data and metadata.

3.1.5 NTFS

Microsoft's NTFS is a widely used (and default) journaling file system on Windows operating systems such as NT, 2000, and XP. Although the source code or documentation of NTFS is not publicly available, tools for finding the NTFS file layout exist [2].

Every object in NTFS is a file. Even metadata is stored in terms of files. The journal itself is a file and is located almost at the center of the file system. In a related work, we find that NTFS does not implement data journaling [84]. We find that NTFS, similar to JFS and XFS, does logical journaling, where it journals metadata in terms of records. We also infer that NTFS performs ordered journaling by introducing artificial delays on data writes and noting the corresponding delays on metadata writes as well.

File	**Journaling Mode**			**Granularity**
System	**Data**	**Ordered**	**Writeback**	
Ext3	√	√	√	Physical
ReiserFS	√	√	√	Physical
JFS		√		Logical
XFS		√		Logical
NTFS		√		Logical

Table 3.1: **Journaling File Systems Summary.** *This table gives a summary of the journaling properties of commodity journaling file systems. Although ext3 and ReiserFS support all the three journaling modes, ordered journaling mode is the default one.*

3.1.6 File System Summary

The Table 3.1 summarizes the salient features of various journaling file systems. Although, ext3 and ReiserFS support all the three journaling modes, ordered journaling is the default mode in both the file systems.

Chapter 4

Failure Policy Analysis and Results

In this chapter, we describe the file system failure policy analysis. Broadly, there are two ways to unearth the failure policies. One, by examining the source code to understand what detection and recovery techniques are used by a file system under different failure scenarios. Two, by injecting failures and monitoring how failures are handled. The first approach is time consuming and error prone due to the large code base consisting of several thousands (even hundreds of thousands for commercial file systems) of lines of intricate code involving lots of low-level calls. Instead, we use a unique *Semantic Failure Analysis (SFA)* technique, wherein we use block type information and transactional semantics to fail particular blocks in order to understand how file systems handle different block failures.

We analyze five different journaling file systems (Linux ext3, ReiserFS, JFS, XFS, and Windows NTFS) and from our analysis we draw the following main conclusions: (a) Commodity file systems store little or no redundant information on disk. As a result, they are not able to recover when portions of a disk fail. (b) We find illogical inconsistency in file system failure handling policies. (c) Finally, we find several bugs in failure handling routines, which show that it is hard to handle block failures in a logically consistent manner.

We begin by explaining the failure analysis, which uses semantic block level information (Section 4.1), followed by a description of our analysis methodology (Section 4.2), and finally present the results (Section 4.3).

4.1 Semantic Block-level Information

The concept of using semantic block-level information is quite simple: use file system level semantic knowledge about block types along with low-level block I/O to perform file system performance and failure analysis. This builds on our previous work on "semantically-smart" disk systems [10, 12, 105, 106, 108]. We use a pseudo-device driver to interpose on the file system traffic to disk and apply file system semantics to block I/Os. We discuss how we use the semantic information in detail below.

Semantic Block Analysis: File systems have traditionally been evaluated using one of two approaches; either one applies synthetic or real workloads and measures the resulting file system performance [23, 61, 73, 75, 76] or one collects traces to understand how file systems are used [8, 13, 77, 92, 123, 131]. However, performing each in isolation misses an interesting opportunity: by correlating the observed disk traffic with the running workload and with performance, one can often answer *why* a given workload behaves as it does.

Block-level tracing of disk traffic allows one to analyze a number of interesting properties of the file system and workload. First, at the coarsest granularity, one can record the *quantity* of disk traffic and how it is divided between reads and writes; for example, such information is useful for understanding how file system caching and write buffering affect performance. Second, at a more detailed level, one can track the *block number* of each block that is read or written; by analyzing the block numbers, one can see the extent to which traffic is sequential or random. Finally, one can analyze the *timing* of each block; with timing information, one can understand when the file system initiates a burst of traffic.

By combining block-level analysis with *semantic* information about those blocks, one can infer much more about the behavior of the file system and we call this approach *Semantic Block Analysis* (SBA) [84]. For example, by simply quantifying the journal traffic under various journal sizes and measuring the application bandwidth, one can understand the role played by journal sizes on journal commit policy [84]. The main difference between SBA and more standard block-level tracing is that SBA understands the on-disk format of the file system under test.

Semantic Failure Analysis: In addition to understanding the performance behaviors, block type information can also be used for fault injection purposes. Many standard fault injectors [24, 104] fail disk blocks in a *type oblivious* manner; that is, a block is failed regardless of how it is being used by the file system. However, repeatedly injecting faults into random blocks and waiting to uncover new aspects of the failure policy would be a laborious and time-consuming process, likely yielding little insight.

The key idea that allows us to test a file system in a relatively efficient and thorough manner is Semantic Failure Analysis (SFA). In semantic failure analysis, instead of failing blocks obliviously, we fail blocks of a specific type (*e.g.*, inode block) or at a particular transactional state (*e.g.*, ordered data block). Semantic information is crucial in reverse-engineering failure policy, allowing us to discern the different strategies that a file system applies for its different data structures. The disadvantage of our approach is that the fault injector must be tailored to each file system tested and requires a solid understanding of on-disk structures. However, we believe that the benefits of SFA clearly outweigh these complexities.

4.1.1 Alternative Techniques

One might believe that directly instrumenting a file system to obtain timing information, disk traces, and inject faults would be equivalent or superior to SBA and SFA that use a pseudo-device driver. We believe this is not the case for several reasons. First, to directly instrument the file system, one needs source code for that file system and one must re-instrument new versions as they are released; in contrast, SBA and SFA do not require file system source, and much of the driver code can be reused across file systems and versions (Section 4.2.5). Second, when directly instrumenting the file system, one may accidentally miss some of the conditions for which disk blocks are written; however, the pseudo-device driver is guaranteed to see all disk traffic. Finally, instrumenting existing code may accidentally change the behavior of that code [129]; an efficient driver will likely have no impact on file system behavior (Section 4.2.5).

In summary, semantic block-level information can be used for various purposes: to understand performance problems, fingerprint file system parameters, uncover design flaws or correctness bugs, and perform failure policy analysis using type-aware fault injection. In the following sections, we focus only on the failure policy aspect of the technique.

4.2 Failure Policy Analysis Methodology

Next, we describe the overall methodology we use for analyzing the failure policies 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.

4.2.1 Failure Policy Fingerprinting

Our main objective in *failure-policy fingerprinting* is to determine which detection and recovery techniques each file system uses and the assumptions each makes about how the underlying storage system can fail. By comparing the failure policies across file systems, we can learn not only which file systems are the most robust to disk failures, but also suggest improvements for each. Our analysis will also be helpful for inferring which IRON techniques can be implemented the most effectively.

Our approach is to inject faults just beneath the file system and observe how the file system reacts. We follow slightly different methodology for write and read failure analysis. To inject write failures, we first construct an analytical model of the transaction update process in journaling file systems and inject faults at various points in the model. Specifically, in order to maintain the file system consistency, journaling file systems update their file system structures following certain ordering constraints. That is, file system updates are first written to the journal as part of a transaction, which are followed by a commit write to mark the transaction as complete. Once the transaction is successfully written, the fixed-location structures are modified at a later point in time. We model this ordering using a simple analytical model and inject failures at specific points in the model. In addition to unearth the failure policy, this approach also enables us to evaluate if a file system follows journaling semantics in the presence of errors. However, file systems issue reads without any ordering constraints as they do not change the on-disk state. Therefore, for read failure and block corruption analysis, we simply fail (or corrupt) different file system block type reads.

If the fault policy is entirely consistent within a file system, this could be done quite simply; we could run any workload, fail one of the blocks that is accessed, and conclude that the reaction to this block failure fully demonstrates the failure policy of the system. However, file systems are in practice more complex: they employ different techniques depending upon the operation performed and the type of the faulty block.

Therefore, to extract the failure policy of a system, we must trigger as many interesting cases as possible. Our challenge is to coerce the file system down its different code paths to observe how each path handles failure. This requires that we run workloads exercising all relevant code paths in combination with induced faults on all file system data structures. Although we do not claim to stress every code path, we do strive to execute as many of the interesting internal cases as possible.

Overall, our failure policy analysis consists of three major steps: create the right workload, inject faults, and deduce failure policy. We describe each of these

Workload	Purpose
Singlets: access, chdir, chroot, stat, statfs, lstat, open, utimes, read, readlink, getdirentries, creat, link, mkdir, rename, chown, symlink, write, truncate, rmdir, unlink, mount, chmod, fsync, sync, umount	Exercise the Posix API
Generics:	
Path traversal	Traverse hierarchy
Recovery	Invoke recovery
Log writes	Update journal

Table 4.1: **Workloads.** *The table presents the workloads applied to the file systems under test. The first set of workloads each stresses a single system call, whereas the second group invokes general operations that span many of the calls (e.g., path traversal).*

steps in detail.

4.2.2 Applied Workload

Our workload suite contains two sets of programs that run on UNIX-based file systems (fingerprinting NTFS requires a different set of similar programs). The first set of programs, called *singlets*, each focus upon a single call in the file system API (*e.g.*, `mkdir`). The second set, *generics*, stresses functionality common across the API (*e.g.*, path traversal). Table 4.1 summarizes the test suite.

Certain workload requires an already existing file, directory or a symbolic link as its parameter. For example, the `stat` POSIX call takes a file path as an input, searches the parent directories, and returns information about the specified file. Before running such workloads, we must first create the files and directories that are necessary. In case of injecting read faults, it is necessary to clear the file system buffer cache so that the on-disk copy will be read by the workload.

Each file system under test also introduces special cases that must be stressed. For example, the ext3 inode uses an imbalanced tree with indirect, doubly-indirect, and triply-indirect pointers, to support large files; hence, our workloads ensure that

Ext3 Structures	Purpose
inode	Info about files and directories
directory	List of files in directory
data bitmap	Tracks data blocks per group
inode bitmap	Tracks inodes per group
indirect	Allows for large files to exist
data	Holds user data
super	Contains info about file system
group descriptor	Holds info about each block group
journal super	Describes journal
journal revoke	Tracks blocks that will not be replayed
journal descriptor	Describes contents of transaction
journal commit	Marks the end of a transaction
journal data	Contains blocks that are journaled

Table 4.2: **Ext3 Data Structures.** *The table presents the data structures of interest in ext3 file system. In the table, we list the names of the major structures and their purpose.*

sufficiently large files are created to access these structures. Other file systems have similar peculiarities that we make sure to exercise (*e.g.*, the B+-tree balancing code of ReiserFS). The block types of the file systems we test are listed in Tables 4.2, 4.3, 4.4, 4.5, and 4.6.

4.2.3 Fault Injection

Our second step is to inject faults that emulate a disk adhering to the fail-partial failure model. We use type aware fault injection technique to inject faults in the disk traffic. We discuss the error model and fault injection framework used in the following sections.

Error Model

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

ReiserFS Structures	**Purpose**
leaf node	Contains items of various kinds
stat item	Info about files and directories
directory item	List of files in directory
direct item	Holds small files or tail of file
indirect item	Allows for large files to exist
data bitmap	Tracks data blocks
data	Holds user data
super	Contains info about tree and file system
journal header	Describes journal
journal descriptor	Describes contents of transaction
journal commit	Marks end of transaction
journal data	Contains blocks that are journaled
root/internal node	Used for tree traversal

Table 4.3: **ReiserFS Data Structures.** *The table presents the ReiserFS data structures of interest and their purpose.*

JFS Structures	**Purpose**
inode	Info about files and directories
directory	List of files in directory
block alloc map	Tracks data blocks per group
inode alloc map	Tracks inodes per group
internal	Allows for large files to exist
data	Holds user data
super	Contains info about file system
journal super	Describes journal
journal data	Contains records of transactions
aggregate inode	Contains info about disk partition
bmap descriptor	Describes block allocation map
imap control	Summary info about imaps

Table 4.4: **JFS Data Structures.** *The table presents the JFS data structures of interest and their purpose.*

XFS Structures	Purpose
super	Contains info about file system
journal record header	Marks a journal record
start transaction	Records the beginning of a transaction
journal commit record	Marks the end of a transaction
unmount record	Marks the journal as clean
data	Holds user data

Table 4.5: **XFS Data Structures.** *The table presents the XFS data structures of interest and their purpose.*

NTFS Structures	Purpose
MFT record	Info about files/directories
directory	List of files in directory
volume bitmap	Tracks free logical clusters
MFT bitmap	Tracks unused MFT records
logfile	The transaction log file
data	Holds user data
boot file	Contains info about NTFS volume

Table 4.6: **NTFS Data Structures.** *The table presents the NTFS data structures of interest and their purpose.*

The errors we inject into the block write stream have three different attributes, similar to the classification of faults injected into the Linux kernel by Gu *et al.* [49]. The fault specification consists of the following attributes:

Failure Type: This specifies whether a read or write must be failed. If it is a read error, one can specify either a latent sector fault or block corruption. Additional information such as whether the system must be crashed before or after certain block failure can also be specified.

Block Type: This attribute specifies the file system and block type to be failed. The request to be failed can be a dynamically-typed one (like a directory block) or a statically typed one (like a super block). Specific parameters can also be passed such as an inode number of an inode to be corrupted or a particular block number to be failed.

Transience: 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).

Fault Injection Framework

Our testing framework is shown in Figure 4.1(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. We use a similar fault injection framework for NTFS in Windows, where a filter driver is used to interpose on the file system traffic.

A flow diagram of the benchmarking process is shown in Figure 4.1(b). We now describe the entire process in detail.

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. It is placed in the Linux kernel and associated with a real disk. The file system of interest is mounted on top of the pseudo-device and the driver simply interposes upon all I/O requests to the real underlying disk. As the driver passes the traffic to and from the disk, it also efficiently tracks each request and response by storing a small record in a fixed-sized circular buffer.

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 struc-

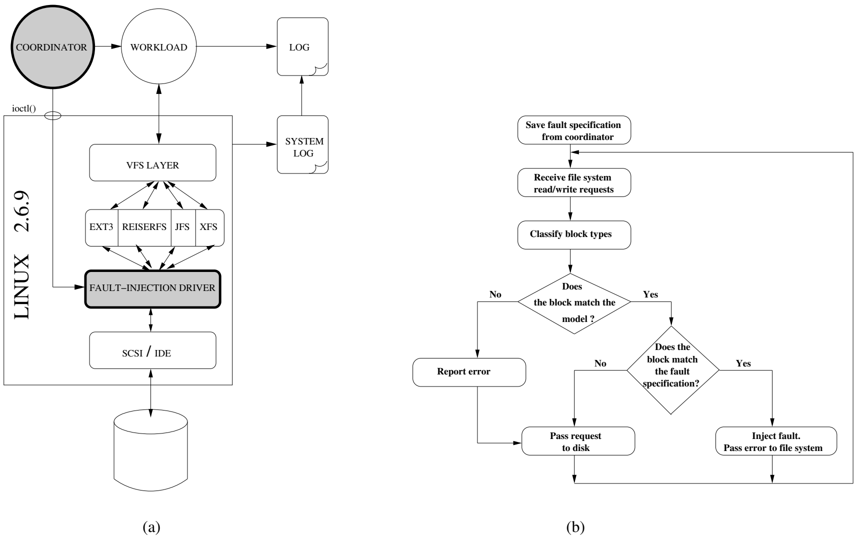


Figure 4.1: **Benchmarking Framework and Algorithm Flow.** *Figure (a) shows the benchmarking framework we use to measure the fault tolerance of journaling file systems to I/O failures. The two main components in the figure are the user level process that issues the fault and the fault-injection driver that classifies blocks and injects faults. Figure (b) shows a simplified flowchart of our benchmarking algorithm that is implemented by the fault-injection driver.*

ture this write represents). For example, one must interpret the contents of the journal to infer the type of journal block (*e.g.*, a descriptor or commit block) and one must interpret the journal descriptor block to know which data blocks are journaled. Therefore, it is necessary that we interpret the semantics online so that block failures can be injected correctly. We explain the complexity and overhead of this classification in Sections 4.2.5 and 4.2.5.

Second, the driver must "model" what the journaling file system above is doing. Specifically, for write failures, such a model represents the correct sequence of states that a transaction must go through in committing to disk. For read failures, it is sufficient to correctly interpret the file system block types. By inserting failures for specific block types and transactional states, 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. This layer injects both block faults (on reads or writes) and block corruption (on reads). To emulate a block fault, we simply return the appropriate error code and do not issue the operation to the underlying disk. To emulate corruption, we change bits within the block before returning the data; in some cases we inject random noise, whereas in other cases we use a block similar to the expected one but with one or more corrupted fields. The software layer also models both transient and sticky faults.

By injecting failures just below the file system, we emulate faults that could be caused by any of the layers in the storage subsystem. Therefore, unlike approaches that emulate faulty disks using additional hardware [24], we can imitate faults introduced by buggy device drivers and controllers. A drawback of our approach is that it does not discern how lower layers handle disk faults; for example, some SCSI drivers retry commands after a failure [90]. However, given that we are characterizing how file systems react to faults, we believe this is the correct layer for fault injection.

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. The driver and the coordinator communicate through the Linux `ioctl` interface to exchange information such as the fault specification and disk traffic observed by the driver.

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 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 mounted in one of the three journaling modes. For write failure analysis, the journaling mode information is passed to the driver, which then selects the appropriate analytical model internally. After specifying the journaling mode, the files and directories needed for the testing are created in the clean file system. Depending on the type of the block to fail, the coordinator constructs a fault specification (which contains the attributes described in §4.2.3) 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. For write failure analysis, based on the journaling mode and the blocks written by the file system, the driver moves internally from one state to another in its journaling model. When the expected block is read or written by the file system, the driver injects the fault by failing or corrupting that block I/O. The driver also records any file system I/O 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.

4.2.4 Failure Policy Inference

After running a workload and injecting a fault, the final step is to determine how the file system behaved. To determine how a fault affected the file system, we compare the results of running with and without the fault. We perform this comparison across all observable outputs from the system: the error codes and data returned by the file system API, the contents of the system log, and the low-level I/O traces recorded by the fault-injection layer. Currently, this is the most human-intensive part of the process, as it requires manual inspection of the visible outputs.

In certain cases, if we identify an anomaly in the failure policy, we check the source code to verify specific conclusions; however, given its complexity, it is not feasible to infer failure policy only through code inspection.

We collect large volumes of results in terms of traces and error logs for each fault injection experiment we run. Due to the sheer volume of experimental data, it is difficult to present all results for the reader's inspection. We represent file system failure policies using a unique representation called *failure policy graphs*, which is similar to the one shown in Figure 4.2.

In Figure 4.2, we plot the different workloads on x-axis and the file system data structures on y-axis. If applicable, each $<$row, column$>$ entry presents the IRON detection or recovery technique used by a file system. If not applicable (*i.e.*, if the workload does not generate the particular block type traffic), a gray box is used. The symbols are superimposed when multiple mechanisms are employed by a file system.

Next, we explain how the entries in Figure 4.2 must be interpreted by walking through an example. Specifically, consider the entry for workload "W1" and "D5" data structure. It has three symbols superimposed: retry (R_{Retry}), error propagation ($R_{Propagate}$) and finally, a file system stop (R_{Stop}). This means that whenever an I/O to "D5" fails during workload "W1", the file system first retries and if that fails, stops and propagates the error to the application.

We use failure policy graphs not only to present our analysis results but also throughout this thesis to represent the failure policies of different versions of IRON file systems we build.

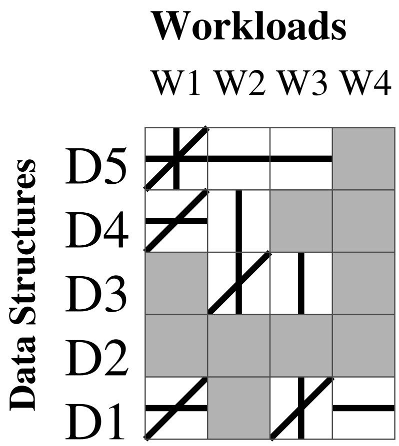
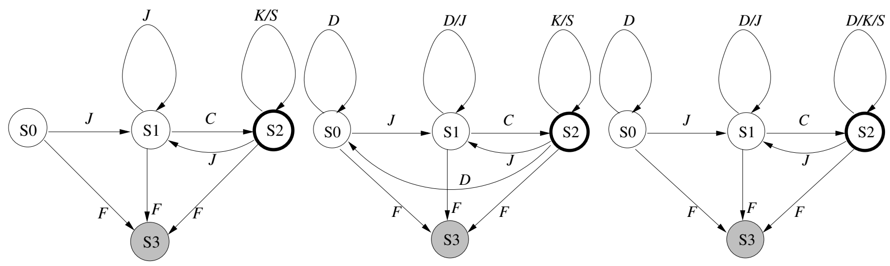


Figure 4.2: **Example Failure Policy Graph.** *The figure presents a sample failure policy graph. A gray box indicates that the workload is not applicable for the block type. If multiple mechanisms are observed, the symbols are superimposed.*

Key for Recovery	
$\bigcirc$	R_{Zero}
$/$	R_{Retry}
$-$	$R_{Propagate}$
$\backslash$	$R_{Redundancy}$
$\vert$	R_{Stop}



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

Figure 4.3: **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.*

4.2.5 Journaling Models

We now describe how we model journaling file systems for write failures. As explained in Section 3.1, 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 4.3 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 represent 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 4.3(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 4.3(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 4.3(c) is obtained by taking this regular expression and adding the state $S3$ to it.

Code	Detailed Analysis			Preliminary Analysis	
	Ext3	ReiserFS	JFS	XFS	NTFS
SBA Generic	1940	1940	1940	1940	448
SBA FS Specific	463	358	509	450	179
SBA Total	2403	2298	2449	2390	627

Table 4.7: **Code Size of Fault Injection Drivers.** *The number of C statements (counted as the number of semicolons) needed to perform a thorough failure policy analysis for ext3, ReiserFS, and JFS and a preliminary analysis for XFS and NTFS. Our XFS and NTFS analysis are preliminary because we do not run all the workloads and fail all the data structures in these file systems.*

Complexity of Fault Injection Driver

The fault injection driver is customized to the file system under test. One concern is the amount of information that must be embedded within the driver for each file system. Given that the focus is on understanding all the block types, our drivers are embedded with enough information to interpret the placement and contents of journal, metadata, and data blocks. We now analyze the complexity of the driver for four Linux journaling file systems, ext3, ReiserFS, JFS and XFS, and Windows NTFS.

Journaling file systems have both a journal, where transactions are temporarily recorded, and fixed-location data structures, where data permanently reside. Our SBA driver distinguishes between the traffic sent to the journal and to the fixed-location data structures. This traffic is simple to distinguish in ReiserFS, JFS, XFS, and NTFS because the journal is a set of contiguous blocks, separate from the rest of the file system. However, to be backward compatible with ext2, ext3 can treat the journal as a regular file. Thus, to determine which blocks belong to the journal, fault injection driver uses its knowledge of inodes and indirect blocks; given that the journal does not change location after it has been created, this classification remains efficient at run-time. The driver is also able to classify the different types of journal blocks such as the descriptor block, journal data block, and commit block.

To perform useful analysis of journaling file systems, the driver does not have to understand any of the in-core data structures of the file systems. Moreover, the driver does not know anything about the policies or parameters of the file system.

Table 4.7 reports the number of C statements required to implement the fault injection driver. These numbers show that most of the code in the driver (*i.e.*, 1940 statements for Linux and 448 statements for Windows) is for general infrastructure;

only between approximately 350 and 500 statements are needed to support different journaling file systems.

Overhead of Fault Injection

In this section, we describe the overheads of classifying the file system traffic into different block types as incurred by the driver. The processing and memory overheads due to the driver are minimal. For every I/O request, the following operations are performed by the driver:

• A block number comparison to find out certain file system block types. First, block numbers can be used to identify whether a block belongs to the journal or fixed-location. Journaling file systems typically allocate the journal as a contiguous set of blocks on disk in order to improve the sequential write performance. Therefore, a simple boundary check can classify journal traffic from fixed-location traffic. However, since journal is created as a file in ext3, the driver compares the I/O block number with the block numbers of the journal file to distinguish journal traffic from others. This comparison is performed efficiently by using hash tables within the driver. Second, the block number comparison can also give information about certain statically assigned block types. For example, in ext3, inode block locations are statically assigned during file system creation. By comparing the block number with the static inode table on disk, the driver can find if a request is issued to an inode block.

• Verifying the contents of the block to understand other file system block types. For example in ext3, the driver checks for certain magic number on the journal blocks to distinguish journal metadata from journal data. The driver also uses the contents to identify other block types such as directories whose locations are dynamically allocated at run time by the file system.

The fault injection driver also stores a small trace of records for each I/O with details like read or write, block number, block type, time of issue and completion, and whether the fault is injected for that block. Each trace record takes about 50 bytes and is stored in an internal circular buffer. After fault injection experiments, the coordinator collects the block-level trace for further analysis.

We run the fault injection experiments on a relatively small file system size (about 250 MB). Note that even with a small file system, it is possible to exercise certain data structures such as the triple indirect block in ext3, which is created only for large file sizes. We do this by creating a file with lots of "holes" and only few actual blocks in it. Therefore, recreating the file system for each new testing does not take much time and each fault injection experiment takes only a few seconds. However, when a system crash is induced to analyze the file system

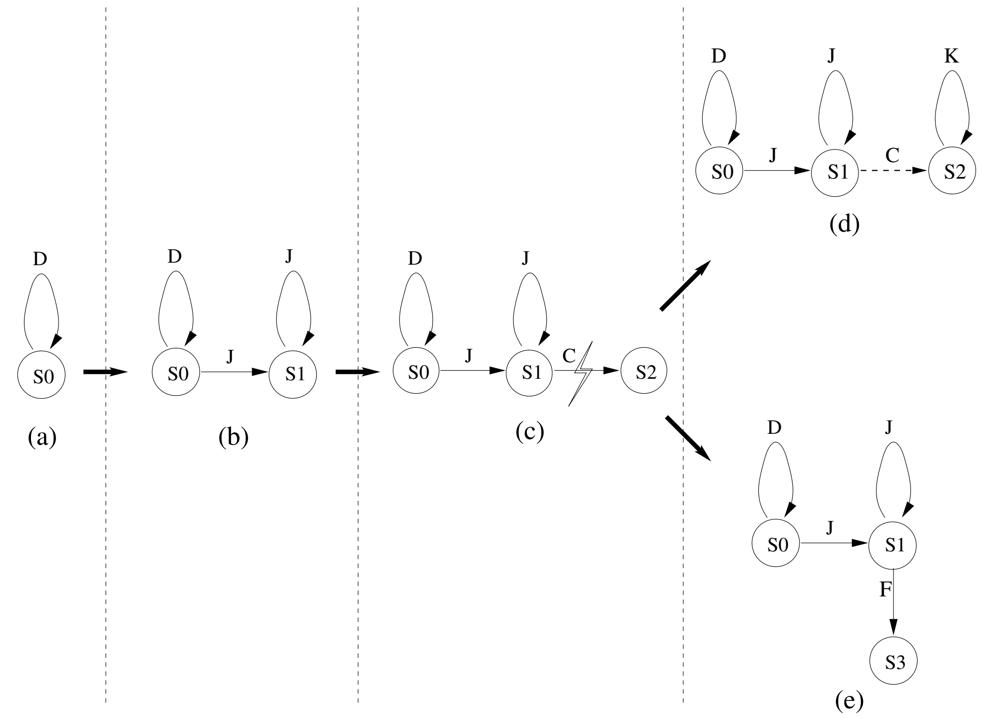


Figure 4.4: **Fault Injection Example.** *This figure shows the sequence of steps followed by the fault-injection driver to track transaction writes in ordered journaling mode and fail the commit block write.*

recovery properties, the fault injection can take up to a few minutes if the system needs to be rebooted. Since a system crash is caused only on few experiments, it did not add much overhead to the analysis.

4.2.6 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 4.4 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 4.4a). After the data writes, the journal blocks are logged (Figure 4.4b). The commit block is written after all the data and journal writes are complete and it is failed (Figure 4.4c). The file system can be oblivious to this commit block failure and continue to checkpoint the journaled blocks (Figure 4.4d). Or, the file system can recognize this failure and take steps to prevent file system corruption by moving to state $S3$ (Figure 4.4e). 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.

4.2.7 Summary

We have developed a three-step fingerprinting methodology to determine file system failure policy. We believe our approach strikes a good balance: it is straightforward to run and yet exercises the file system under test quite thoroughly. Our workload suite contains roughly 30 programs, each file system has on the order of 10 to 20 different block types, and each block can be failed on a read or a write or have its data corrupted. For each file system, this amounts to roughly 400 relevant tests.

4.3 Failure Policy Analysis Results

We now present the results of our failure policy analysis for five commodity file systems: ext3, ReiserFS (version 3), IBM's JFS, and XFS on Linux and NTFS on Windows. For each file system, we discuss the general failure policy we uncovered along with bugs and illogical inconsistencies; where appropriate and available, we also look at source code to better explain the problems we discover.

For each file system that we studied in depth, we present failure policy graphs of our results, showing for each workload/block type pair how a given detection or recovery technique is used. Figures 4.5, 4.6, and 4.7 present a (complex) graphical depiction of our results. Totally, there are six graphs in each figure, three each for detection and recovery policies. Among the three graphs, one graph each is plotted for read failures, write failures, and block corruption results. Under certain columns, we collapse multiple workloads into a single entry if the disk traffic and failure handling by the file system is same for all those workloads. We now provide a qualitative summary of the results that are presented within the figure.

4.3.1 Linux ext3

Key for Detection		Key for Recovery	
$\bigcirc$	D_{Zero}	$\bigcirc$	R_{Zero}
$-$	$D_{ErrorCode}$	$/$	R_{Retry}
$\vert$	D_{Sanity}	$-$	$R_{Propagate}$
		$\backslash$	$R_{Redundancy}$
		$\vert$	R_{Stop}

Table 4.8: **Keys for Detection and Recovery.** *The table presents the keys we use to represent the detection and recovery policies in file systems.*

Detection

To detect read failures, ext3 primarily uses error codes ($D_{ErrorCode}$). However, when a write fails, ext3 does not record the error code (D_{Zero}); hence, write errors are often ignored, potentially leading to serious file system problems (*e.g.*, when checkpointing a transaction to its final location). Ext3 also performs a fair amount of sanity checking (D_{Sanity}). For example, ext3 explicitly performs type checks for certain blocks such as the superblock and many of its journal blocks. However, little type checking is done for many important blocks, such as directories, bitmap blocks, and indirect blocks. Ext3 also performs numerous other sanity checks (*e.g.*, when the file-size field of an inode contains an overly-large value, `open` detects this and reports an error).

Recovery

For most detected errors, ext3 propagates the error to the user ($R_{Propagate}$). For read failures, ext3 also often aborts the journal (R_{Stop}); aborting the journal usually leads to a read-only remount of the file system, preventing future updates without explicit administrator interaction. Ext3 also uses retry (R_{Retry}), although sparingly; when a prefetch read fails, ext3 retries only the originally requested block.

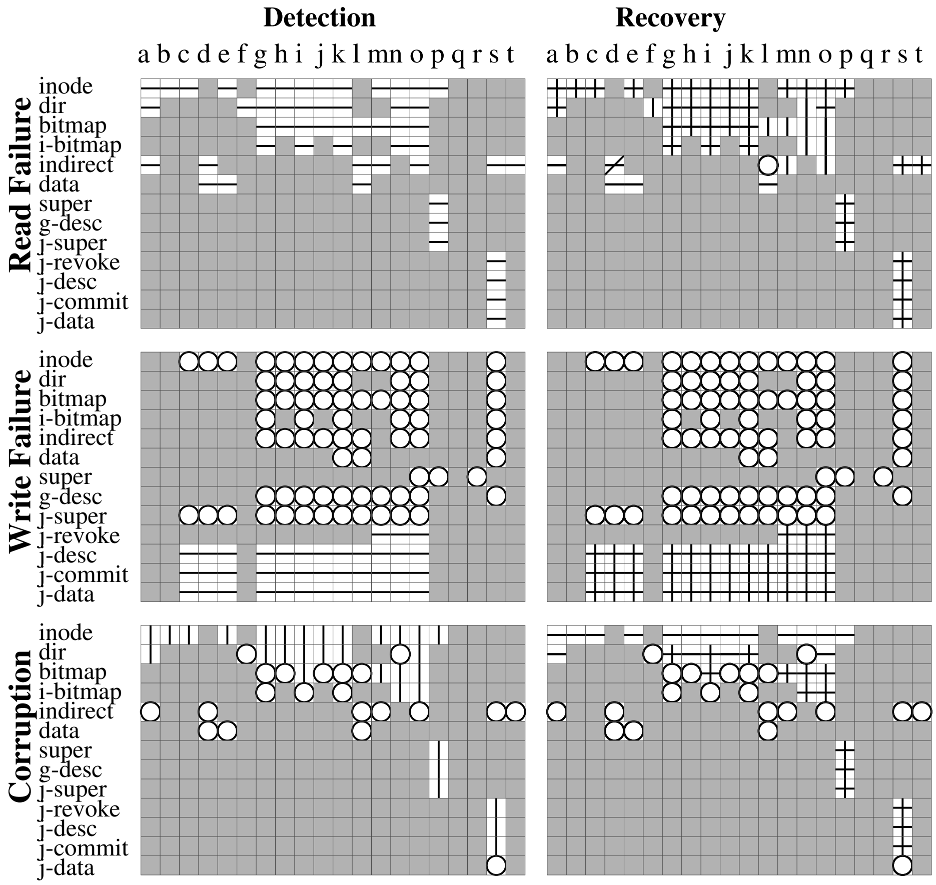


Figure 4.5: **Ext3 Failure Policies.** *The failure policy graphs plot detection and recovery policies of ext3 for read, write, and corruption faults injected for each block type across a range of workloads. The workloads are* **a:** *path traversal* **b:** *access,chdir,chroot,stat,statfs,lstat,open* **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:** *fysnc,sync* **r:** *umount* **s:** *FS recovery* **t:** *log write operations. A gray box indicates that the workload is not applicable for the block type. If multiple mechanisms are observed, the symbols are superimposed. The keys for detection and recovery are presented in Table 4.8.*

Bugs and Inconsistencies

We found a number of bugs and inconsistencies in the ext3 failure policy. First, errors are not always propagated to the user (*e.g.*, `truncate` and `rmdir` fail silently). Second, ext3 does not always perform sanity checking; for example, `unlink` does not check the `linkscount` field before modifying it and therefore a corrupted value can lead to a system crash. Third, although ext3 has redundant copies of the superblock ($R_{Redundancy}$), these copies are never updated after file system creation and hence are not useful. Finally, there are important cases when ext3 violates the journaling semantics, committing or checkpointing invalid transactions. We discuss each such cases below.

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.

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.

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.

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.

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.

4.3.2 ReiserFS

Detection

Our analysis reveals that ReiserFS pays close attention to error codes across reads and writes ($D_{ErrorCode}$). ReiserFS also performs a great deal of internal sanity checking (D_{Sanity}). For example, all internal and leaf nodes in the balanced tree have a block header containing information about the level of the block in the tree, the number of items, and the available free space; the super block and journal metadata blocks have "magic numbers" which identify them as valid; the journal descriptor and commit blocks also have additional information; finally, inodes and directory blocks have known formats. ReiserFS checks whether each of these blocks has the expected values in the appropriate fields. However, not all blocks are checked this carefully. For example, bitmaps and data blocks do not have associated type information and hence are never type-checked.

Recovery

The most prominent aspect of the recovery policy of ReiserFS is its tendency to `panic` the system upon detection of virtually any write failure (R_{Stop}). When ReiserFS calls `panic`, the file system crashes, usually leading to a reboot and recovery sequence. By doing so, ReiserFS attempts to ensure that its on-disk structures are not corrupted. ReiserFS recovers from read and write failures differently. For most read failures, ReiserFS propagates the error to the user ($R_{Propagate}$) and sometimes performs a single retry (R_{Retry}) (*e.g.*, when a data block read fails, or when an indirect block read fails during `unlink`, `truncate`, and `write` operations). However, ReiserFS never retries upon a write failure.

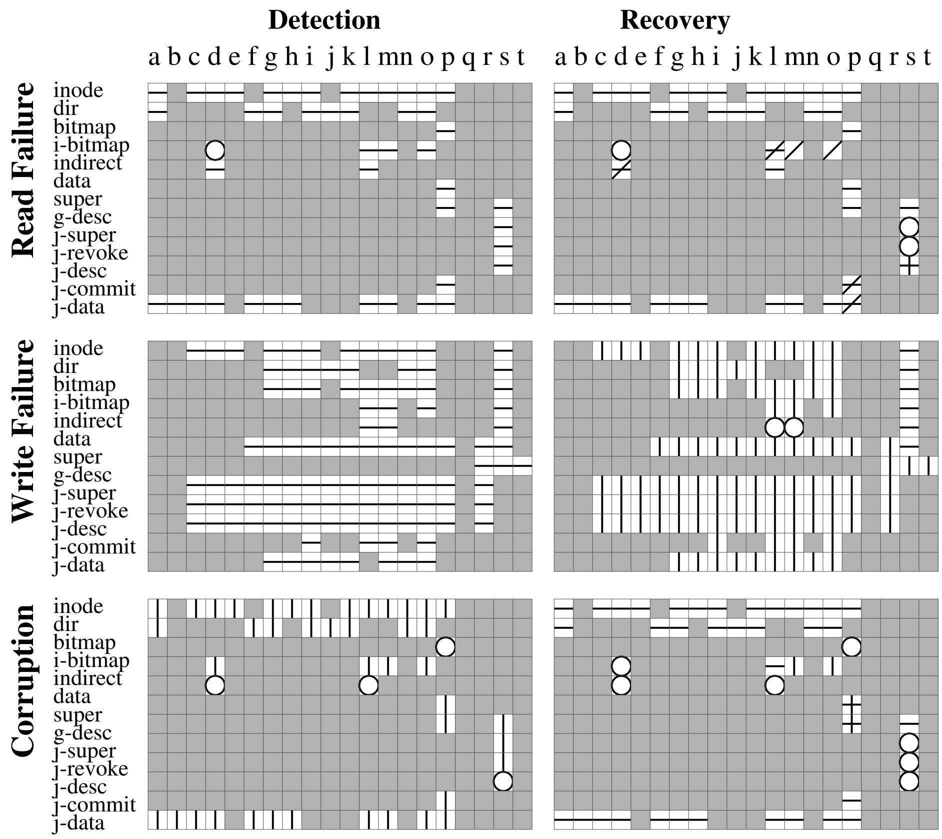


Figure 4.6: **ReiserFS Failure Policy.** *The failure policy graphs plot detection and recovery policies of ReiserFS for read, write, and corruption faults injected for each block type across different workloads. The workloads are varied across the columns of the figure, and the different block types of the ReiserFS file system are varied across the rows. 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:** *fysnc,sync* **r:** *umount* **s:** *FS recovery* **t:** *log write operations. A gray box indicates that the workload is not applicable for the block type. For multiple mechanisms, the symbols are superimposed. The keys for detection and recovery are presented in Table 4.8.*

Bugs and Inconsistencies:

ReiserFS also exhibits inconsistencies and bugs. First, while dealing with indirect blocks, ReiserFS detects but ignores a read failure; hence, on a `truncate` or `unlink`, it updates the bitmaps and super block incorrectly, leaking space. Second, ReiserFS sometimes calls `panic` on failing a sanity check, instead of simply returning an error code. Third, there is no sanity or type checking to detect corrupt journal data; therefore, replaying a corrupted journal block can make the file system unusable (*e.g.*, the block is written as the super block). Finally, ReiserFS also violates journaling semantics during certain write failures. We explain the details below.

Committing, Checkpointing, and Replaying Failed Transactions: On certain write failures, ReiserFS does not crash but continues to commit and checkpoint 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. ReiserFS also replays failed transaction on ordered data write errors.

4.3.3 IBM JFS

Detection

Error codes ($D_{ErrorCode}$) are used to detect read failures, but, like ext3, most write errors are ignored (D_{Zero}), with the exception of journal superblock writes. JFS employs only minimal type checking; the superblock and journal superblock have magic and version numbers that are checked. Other sanity checks (D_{Sanity}) are used for different block types. For example, internal tree blocks, directory blocks, and inode blocks contain the number of entries (pointers) in the block; JFS checks to make sure this number is less than the maximum possible for each block type. As another example, an equality check on a field is performed for block allocation maps to verify that the block is not corrupted.

Recovery

The recovery strategies of JFS vary dramatically depending on the block type. For example, when an error occurs during a journal superblock write, JFS crashes

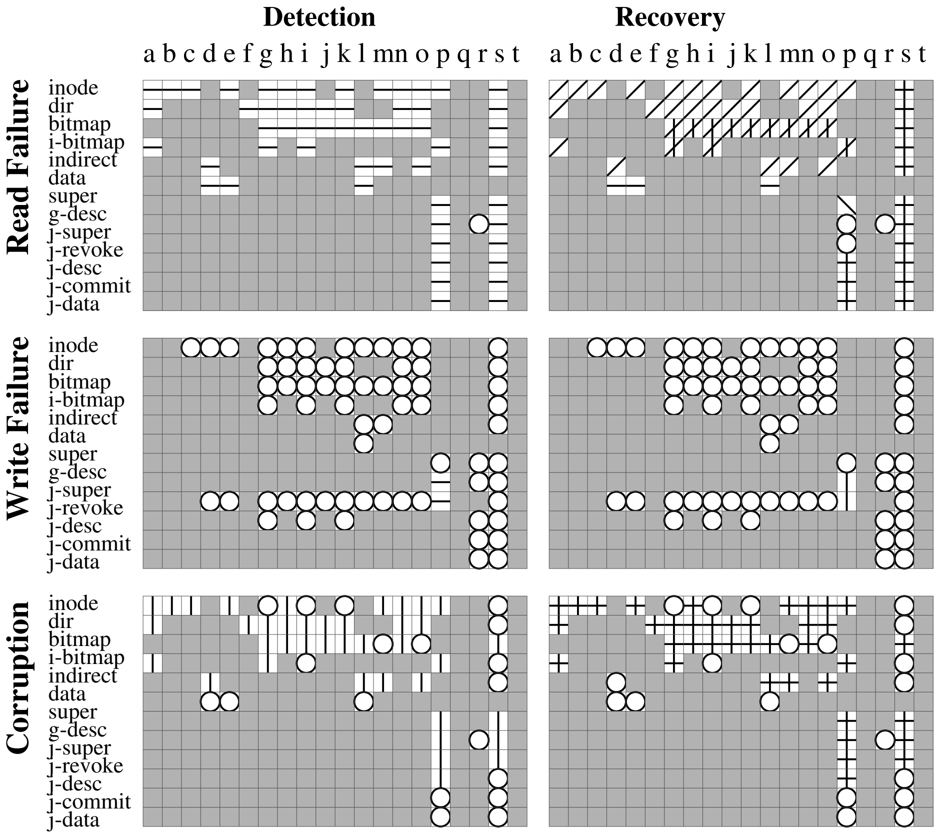


Figure 4.7: **JFS Failure Policy.** *The tables plot detection and recovery policies of JFS for read, write, and corruption faults. The workloads are varied across the columns, and the different block types are varied across the rows. The workloads are* **a:** *path traversal* **b:** *access,chdir,chroot,stat,statfs,lstat,open* **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:** *fysnc,sync* **r:** *umount* **s:** *FS recovery* **t:** *log write operations. A gray box indicates that the workload is not applicable for the block type. For multiple mechanisms, the symbols are superimposed. The keys for detection and recovery are presented in Table 4.8.*

the system (R_{Stop}); however, other write errors are ignored entirely (R_{Zero}). On a block read failure to the primary superblock, JFS accesses the alternate copy ($R_{Redundancy}$) to complete the mount operation; however, a corrupt primary results in a mount failure (R_{Stop}). Explicit crashes (R_{Stop}) are used when a block allocation map or inode allocation map read fails. Error codes for all metadata reads are handled by generic file system code called by JFS; this generic code attempts to recover from read errors by retrying the read a single time (R_{Retry}). Finally, the reaction for a failed sanity check is to propagate the error ($R_{Propagate}$) and remount the file system as read-only (R_{Stop}); during journal replay, a sanity-check failure causes the replay to abort (R_{Stop}).

Bugs and Inconsistencies

We also found problems with the JFS failure policy. First, while JFS has some built-in redundancy, it does not always use it as one would expect; for example, JFS does not use its secondary copies of aggregate inode tables (special inodes used to describe the file system) when an error code is returned for an aggregate inode read. Second, a blank page is sometimes returned to the user (R_{Guess}), although we believe this is not by design (*i.e.*, it is a bug); for example, this occurs when a read to an internal tree block does not pass its sanity check. Third, some bugs limit the utility of JFS recovery; for example, although generic code detects read errors and retries, a bug in the JFS implementation leads to ignoring the error and corrupting the file system. Finally, we present the cases of journaling semantics violations during write failures in JFS.

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.

Committing, Checkpointing, and Replaying Failed Transactions: We find that all the five journaling file systems commit and checkpoint 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.

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.

4.3.4 XFS

We perform a preliminary analysis of failure policy for write failures in XFS. We fail the writes of the following block types in XFS: ordered data, journal blocks containing both the metadata and commit records, journal blocks with unmount records, and checkpoint blocks.

Detection

XFS detects most write failures using the error code ($D_{ErrorCode}$), including the asynchronous checkpoint write errors, which are ignored by ext3 and JFS. XFS does not always handle write failures correctly. We find a flaw that we found in ext3, ReiserFS, and JFS as well. That is, when an ordered data block write fails, XFS continues to log the failed transaction to the journal resulting in data corruption.

Recovery

XFS takes different recovery techniques under different block failures. It retries checkpoint write (which are asynchronous) failures repeatedly without any bounds (R_{Retry}). If the block error is a transient one, the repeated retry successfully writes the block. On permanent errors, the failed block write is repeated infinitely. If one unmounts the system to rectify the write error, XFS crashes (R_{Stop}). On journal write failures, XFS reacts immediately by stopping the file system (R_{Stop}).

4.3.5 Windows NTFS

NTFS [2, 109] is the only non-UNIX file system in our study. Because our analysis requires detailed knowledge of on-disk structures, we do not yet have a complete analysis as in Figures 4.5, 4.6, and 4.7.

We find that NTFS uses error codes ($D_{ErrorCode}$) to detect both block read and write failures. Similar to ext3 and JFS, when a data write fails, NTFS records the error code but does not use it (D_{Zero}), which can corrupt the file system.

NTFS performs strong sanity checking (D_{Sanity}) on metadata blocks; the file system becomes unmountable if any of its metadata blocks (except the journal) are corrupted. NTFS surprisingly does not always perform sanity checking; for example, a corrupted block pointer can point to important system structures and hence corrupt them when the block pointed to is updated.

In most cases, NTFS propagates errors ($R_{Propagate}$). NTFS aggressively uses retry (R_{Retry}) when operations fail (*e.g.*, up to seven times under read failures).

Level	ext3	ReiserFS	JFS
D_{Zero}	$\surd\surd$	$\surd$	$\surd\surd\surd$
$D_{ErrorCode}$	$\surd\surd\surd\surd$	$\surd\surd\surd\surd$	$\surd\surd$
D_{Sanity}	$\surd\surd\surd$	$\surd\surd\surd\surd$	$\surd\surd\surd$
$D_{Redundancy}$			
R_{Zero}	$\surd\surd$	$\surd$	$\surd\surd$
$R_{Propagate}$	$\surd\surd\surd$	$\surd\surd$	$\surd\surd$
R_{Stop}	$\surd\surd$	$\surd\surd\surd$	$\surd\surd$
R_{Guess}			$\surd$
R_{Retry}		$\surd$	$\surd\surd$
R_{Repair}			
R_{Remap}			
$R_{Redundancy}$			$\surd$

Table 4.9: **IRON Techniques Summary.** *The table depicts a summary of the IRON techniques used by the file systems under test. More check marks ($\surd$) indicate a higher relative frequency of usage of the given technique.*

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

Table 4.10: **File System Bugs.** *This table gives a summary of the bugs we identified in ext3, ReiserFS, IBM JFS, and XFS.*

With writes, the number of retries varies (*e.g.*, three times for data blocks, two times for MFT blocks).

4.3.6 File System Summary

We now present a qualitative summary of each of the file systems we tested. Table 4.9 presents a summary of the techniques that each file system employs (excluding XFS and NTFS) and Table 4.10 lists the bugs we found in our file system analysis.

• **Ext3: Overall simplicity.** Ext3 implements a simple and mostly reliable failure policy, matching the design philosophy found in the ext family of file systems. It checks error codes, uses a modest level of sanity checking, and recovers by propagating errors and aborting operations. The main problem with ext3 is its failure handling for write errors, which are ignored and cause serious problems including possible file system corruption.

• **ReiserFS: First, do no harm.** ReiserFS is the most concerned about disk failure. This concern is particularly evident upon write failures, which often induce a `panic`; ReiserFS takes this action to ensure that the file system is not corrupted. ReiserFS also uses a great deal of sanity and type checking. These behaviors combine to form a Hippocratic failure policy: first, do no harm.

• **JFS: The kitchen sink.** JFS is the least consistent and most diverse in its failure detection and recovery techniques. For detection, JFS sometimes uses sanity, sometimes checks error codes, and sometimes does nothing at all. For recovery, JFS sometimes uses available redundancy, sometimes crashes the system, and sometimes retries operations, depending on the block type that fails, the error detection and the API that was called.

• **XFS: Simple and well-defined.** From our preliminary analysis, we find that XFS has a simple and well-defined failure policy to handle write failures. It checks error codes and on synchronous write failures, XFS stops the file system and propagates errors. On asynchronous write failures, the failed write is retried persistently.

• **NTFS: Persistence is a virtue.** Compared to several Linux file systems, NTFS is more persistent, retrying failed requests many times before giving up. It also seems to propagate errors to the user quite reliably. However, more thorough testing of NTFS is needed in order to broaden these conclusions (a part of our ongoing work).

4.3.7 Technique Summary

Finally, we present a broad analysis of the techniques applied by all of the file systems to detect and recover from disk failures. We concentrate upon techniques that are underused, overused, or used in an inappropriate manner.

• **Detection and Recovery: Illogical inconsistency is common.** We found a high degree of *illogical inconsistency* in failure policy across all file systems (observable in the patterns in Figures 4.5, 4.6, and 4.7). For example, ReiserFS performs a great deal of sanity checking; however, in one important case it does not (journal replay), and the result is that a single corrupted block in the journal can corrupt the entire file system. JFS is the most illogically inconsistent, employing different techniques in scenarios that are quite similar.

We note that inconsistency in and of itself is not problematic [37]; for example, it would be *logically* inconsistent (and a good idea, perhaps) for a file system to provide a higher level of redundancy to data structures it deems more important, such as the root directory [106]. What we are criticizing are inconsistencies that are undesirable (and likely unintentional); for example, JFS will attempt to read the alternate superblock if a read failure occurs when reading the primary superblock, but it does not attempt to read the alternate if it deems the primary corrupted.

In our estimation, the root cause of illogical inconsistency is *failure policy diffusion*; the code that implements the failure policy is spread throughout the kernel. Indeed, the diffusion is encouraged by some architectural features of modern file systems, such as the split between generic and specific file systems. Further, we have observed some cases where different developers implement different portions of the code and hence implement different failure policies (*e.g.*, one of the few cases in which ReiserFS does *not* panic on write failure arises due to this); perhaps this inconsistency is indicative of the lack of attention paid to failure policy. We pay attention to this problem and design a centralized failure handler for file systems, which we discuss in Chapter 6.

• **Detection and Recovery: Bugs are common.** We also found numerous bugs across the file systems we tested, some of which are serious, and many of which are not found by other sophisticated techniques [129]. We believe this is generally indicative of the difficulty of implementing a correct failure policy; it certainly hints that more effort needs to be put into testing and debugging of such code. One suggestion in the literature that could be helpful would be to periodically inject faults in normal operation as part of a "fire drill" [80]. Our method reveals that testing needs to be broad and cover as many code paths as possible; for example, only by testing the indirect-block handling of ReiserFS did we observe certain classes of fault mishandling.

• **Detection: Error codes are sometimes ignored.** Amazingly (to us), error codes were sometimes clearly ignored by the file system. This was most common in JFS, but found occasionally in the other file systems. Perhaps a testing framework such as ours should be a part of the file system developer's toolkit; with such tools, this class of error is easily discovered.

• **Detection: Sanity checking is of limited utility.** Many of the file systems use sanity checking to ensure that the metadata they are about to use meets the expectations of the code. However, modern disk failure modes such as misdirected and phantom writes lead to cases where the file system could receive a properly formatted (but incorrect) block; the bad block thus passes sanity checks, is used, and can corrupt the file system. Indeed, all file systems we tested exhibit this behavior. Hence, we believe stronger tests (such as checksums) should be used.

• **Recovery: Stop is useful – if used correctly.** Most file systems employed some form of R_{Stop} in order to limit damage to the file system when some types of errors arose; ReiserFS is the best example of this, as it calls `panic` on virtually any write error to prevent corruption of its structures. However, one has to be careful with such techniques. For example, upon a write failure, ext3 tries to abort the transaction, but does not correctly squelch all writes to the file system, leading to corruption. Perhaps this indicates that fine-grained rebooting is difficult to apply in practice [25].

• **Recovery: Stop should not be overused.** One downside to halting file system activity in reaction to failure is the inconvenience it causes: recovery takes time and often requires administrative involvement to fix. However, all of the file systems used some form of R_{Stop} when something as innocuous as a read failure occurred; instead of simply returning an error to the requesting process, the entire system stops. Such draconian reactions to possibly temporary failures should be avoided.

• **Recovery: Retry is underutilized.** Most of the file systems assume that failures are not transient, or that lower layers of the system handle such failures, and hence do not retry requests at a later time. The systems that employ retry generally assume read retry is useful, but write retry is not; however, transient faults due to device drivers or transport issues are equally likely to occur on reads and writes. Hence, retry should be applied more uniformly. NTFS is the lone file system that embraces retry; it is willing to issue a much higher number of requests when a block failure is observed.

• **Recovery: Automatic repair is rare.** Automatic repair is used rarely by the file systems; instead, after using an R_{Stop} technique, most of the file systems require manual intervention to attempt to fix the observed problem (*i.e.*, running fsck).

• **Detection and Recovery: Redundancy is not used.** Finally, and perhaps most importantly, while virtually all file systems include some machinery to detect disk failures, none of them apply *redundancy* to enable recovery from such failures. The lone exception is the minimal amount of superblock redundancy found in JFS; even this redundancy is used inconsistently. Also, JFS places the copies in close proximity, making them vulnerable to spatially-local errors. As it is the least explored and potentially most useful in handling the failures common in drives today, we next investigate the inclusion of various forms of redundancy into the failure policy of a file system.

4.4 Conclusion

Commodity file systems use a variety of detection and recovery techniques to handle disk sector errors. In this chapter, we explain the process of semantic failure analysis, wherein we apply our knowledge about file system block types and transactional semantics to unearth the failure policies. From our analysis, we find that commodity file systems are built mostly with the assumption that disk fails in fail-stop manner and therefore, store little or no redundant information on disk. We also find that the failure policies enacted by a file system are often ad hoc, inconsistent, and buggy. We conclude that it is time we rearchitect the file systems with more redundant information to handle partial disk errors and a framework that can support well-defined failure policies.

Chapter 5

Building Low-level Redundancy Machinery

As we mention in the Introduction (Section 1.2), there are two challenges in building a robust file system. First, we must design low-level redundancy machinery to detect and recover from partial disk errors. Second, we need a new failure handling framework to unify the different low-level machineries with various partial disk errors and failure policies. In this chapter, we focus on the first problem, and design and implement new redundancy techniques within a single disk drive (the second problem is addressed in the next chapter). Note that the challenge of building cost effective low-level machinery is especially important in the context of commodity file systems because currently they store little or no redundant information on disk.

A well-known technique in building reliable systems is to use redundant components and specialized hardware mechanisms, not only to improve reliability but also to keep the performance from degrading [16]. However, commodity systems such as desktops and laptops may not be able to afford redundant components such as extra disk drives or special hardware support like NVRAM (both of which are usually available in high-end systems) as they will result in an increased system cost, system size, and heat dissipation. Therefore, we specifically focus on storing redundant information within a single disk drive with no additional hardware support.

Redundant information can be stored and used in various ways, where each technique has its own merits and demerits. We explore three types of redundant information in this thesis: checksums, parity, and replicas due to their wide spread use in high-end robust file and storage systems [16, 20, 21, 28, 71, 79, 96, 110]. We first describe and qualitatively evaluate the robustness, performance, and space overheads of various redundancy options in file systems (Section 5.1). Then, we describe the implementation and evaluation of a robust file system, IRON ext3, wherein we broadly explore the applications of replication, parity, and checksums to file system data and metadata (Section 5.2). In IRON ext3, we evaluate the redundancy mechanisms separately and in various combinations, and find that they incur minimal time and space overhead that are tolerable in desktop settings. Our evaluation also shows that parity is cost-effective both in terms of space and performance overhead and therefore, we study parity in detail in the following sections. In order to evaluate the impact of locality of failures in a parity-based redundancy technique, we develop a probabilistic model that characterizes spatial correlation in disk errors (Section 5.3.1). Finally, we design two parity update techniques to update the redundant information without NVRAM support (Section 5.3.2). Our evaluation shows that better data and parity layout along with new parity update techniques can greatly improve the robustness of a file system while incurring modest performance overhead.

5.1 In-Disk Redundancy

A file or storage system can store redundant copies of data on disk both for detection and recovery purposes. For example, a data block can be replicated and both the copies can be read to detect block corruption. In addition to replicas, redundant information may also include certain processed copies of data such as checksums, parity, or more involved error correction codes [64, 69]. In this thesis, we study in detail three types of redundant information: checksums, parity, and replicas.

Our philosophy in building robust systems is that the end-point must be able to detect and recover from failures, irrespective of whether the intermediate layers can handle errors; that is, our approach is an instance of the end-to-end argument [94]. Following this principle, we design and implement all the redundant information at the file system layer rather than at a block device layer such as the software RAID. Since we focus on journaling file systems, we integrate our techniques with transactional semantics in order to provide consistency between primary and secondary copies. There are several important issues in incorporating redundancy into file systems: the effectiveness of a redundancy technique in handling disk sector errors, its performance, and space overheads. We discuss these issues in detail.

Effectiveness: The effectiveness of redundancy techniques vary. For example, a parity block computed over a set of N blocks can recover from at most one failure among the N blocks, while a set of N replicas can recover more errors. Similarly, a cryptographic checksum such as SHA-1 on actual data is likely to have fewer collisions with corrupted data than simple checksums such as CRC.

Another factor that affects the effectiveness is the location of the redundant information with respect to its primary copy, since sectors on a disk might be subject to spatially local errors. Specifically, we can consider two block layout schemes. First, one can treat block errors as independent events and place the redundant information either within the same block (in case of checksums) or on the next block. We represent this scheme in our discussion by subscripts "$Within$" and "$d = 0$", where d represents the distance between the data and its redundant information. The other scheme takes into account spatial locality in disk errors and separates them by t tracks. We denote this by "$d = t$". To achieve greater reliability, a redundant copy must be stored farther from the primary copy so that they both are not subject to correlated failures.

For a redundant copy to be useful for detection and recovery, it must be updated carefully with respect to its primary copy. If the primary and secondary copies of a block are not updated atomically, then after a file system crash multiple data copies might be inconsistent with each other, and therefore, the redundant information might not be useful to recover from latent sector faults that might occur during journal recovery. Hardware support such as NVRAM can be useful in updating redundant information consistently and atomically, a lack of which in low-end systems means we need new techniques for updating the redundant data. For example, writing both primary and secondary copies into the journal as part of a transaction before updating their fixed location copies can be used to achieve atomicity although with high performance overheads. In our discussion, we denote all techniques that update both the data and its corresponding redundant information atomically with a subscript "$Atomic$", while those techniques that do not have this property are represented by "$Non\text{-}atomic$".

Performance Overheads: Adding more redundant information can decrease the performance of a system, and there are several reasons why this may happen. First, file systems have to pay the cost of updating the redundant copies whenever the primary copy is updated. For example, in a journaling file system where all the metadata blocks (including the journal) are replicated, every metadata write incurs four write costs: two writes to the primary and secondary journal, and two writes to the primary and secondary fixed-location copies. Although some systems relax the synchronous update of redundant copies in order to mitigate the performance overheads [96], they can endanger the reliability of the system. Second, performance may drop due to additional disk reads that are issued to redundant information such as checksums in order to verify the validity of the data. Third, file systems might follow certain write ordering constraints when updating the primary and secondary copies to achieve atomicity, which in turn can cause additional rotational latencies and disk seeks. Fourth, a portion of the host main memory might be used to hold redundant information such as checksums, which can affect the performance as it reduces the amount of available memory to the overall system. Finally, it is computationally expensive to produce certain types of redundant information such as the Reed-Solomon or Tornado error correction codes [69], and such processing may use significant amount of CPU power available in a system.

Space Overheads: Not surprisingly, redundant information can incur space overheads as well and it varies according to the redundancy technique used. For example, although stronger in detecting corruptions, a cryptographic checksum such as SHA-1 occupies more bytes than a simple CRC-type checksum. Similarly, parity blocks, replicas, and error correction codes each occupy different amount of space depending on how many copies are maintained.

Table 5.1 presents a qualitative evaluation of effectiveness, performance, and space overheads of various redundancy techniques. Different redundancy techniques are listed under each row and the effectiveness under a single block error, spatially local errors that spread up to t disk tracks, and spatially local errors that spread beyond t disk tracks are listed under different columns along with the performance and space cost. We also study whether a technique can recover from a failure both during normal file system operation and journal recovery. A $\surd$ mark means that a particular technique is guaranteed to detect or recover from the failure whereas a $\times$ mark represents that no guarantee can be given for failure detection or recovery.

We now explain the table entries in detail. Broadly, checksums can be stored in 3 classes of locations: within a sector, at a neighboring block, or at a block separated by t tracks. If checksum is stored within a sector, its update can be guaranteed to be atomic due to the basic write guarantees offered by a disk drive. If separated from the data, checksum update can be either atomic or not. Although, Checksum$_{Within}$ is atomic, it is not guaranteed to detect phantom write or misdirected write errors. While in general, the space and time costs are tolerable for checksums, atomic techniques might require extra writes to the journal, which can increase the performance overheads.

In case of parity, it can either be calculated for a set of contiguous N blocks or, if one wants to tolerate spatially correlated failures, it can be calculated for blocks that are separated by say, t tracks. Parity update can be atomic as well; however, logging both parity and its corresponding blocks into the journal can decrease the performance significantly. Finally, replicas incur high space and performance overheads. One can use journal to ensure atomic updates although with severe

	Effectiveness						Overheads	
	Single block		**Spatial** (<= t **tracks)**		**Spatial** (> t **tracks)**		**Time**	**Space**
Techniques	Normal	FS Recovery	Normal	FS Recovery	Normal	FS Recovery		
Checksum$_{Within}$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	low	low
Checksum$_{d=0,Non-atomic}$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	low	low
Checksum$_{d=0,Atomic}$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	high	low
Checksum$_{d=t,Non-atomic}$	$\surd$	$\times$	$\surd$	$\times$	$\times$	$\times$	low	low
Checksum$_{d=t,Atomic}$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	high	low
Parity$_{d=0,Non-atomic}$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	low	low
Parity$_{d=0,Atomic}$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	high	low
Parity$_{d=t,Non-atomic}$	$\surd$	$\times$	$\surd$	$\times$	$\times$	$\times$	low	high
Parity$_{d=t,Atomic}$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	high	high
Replica$_{d=0,Non-atomic}$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	high	high
Replica$_{d=0,Atomic}$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	high	high
Replica$_{d=t,Non-atomic}$	$\surd$	$\times$	$\surd$	$\times$	$\times$	$\times$	high	high
Replica$_{d=t,Atomic}$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	high	high

Table 5.1: **Qualitative Evaluation of Redundancy Techniques.** *The table presents a qualitative evaluation of the effectiveness, performance, and space overheads of different redundancy techniques to detect and recover from sector errors or corruption. A* $\surd$ *mark means that a particular technique is guaranteed to detect or recover from the failure whereas a* $\times$ *mark represents that no guarantee can be given for failure detection or recovery.*

performance penalties.

In conclusion, we can observe from the table that the more effective a redundancy technique is in handling various errors, higher its performance and/or space overhead becomes. In the following sections, we implement various redundancy techniques from the table and evaluate their robustness and cost quantitatively.

5.2 ixt3: A Prototype IRON File System

We now describe our implementation and evaluation of *IRON ext3 (ixt3)*. Our design goal is to detect and recover from most errors with an affordable time and space cost. Within ixt3, we implement a family of recovery techniques that most commodity file systems do not currently provide. We give more importance to file system metadata and therefore protect them using replicas, while we use a simple parity-based redundancy scheme for data blocks that can only recover from a single block error per file. By using a single parity block per file, we also strive to minimize the performance and space cost that can be significant if data blocks were replicated. To increase its robustness, ixt3 applies checksums to both metadata and data blocks. In our design, we also take care to handle spatially local failures by separating the replicas and checksums from their corresponding disk blocks. Ixt3 is not backward compatible with ext3 as we modify some of its internal structures such as the `inode`, which stores information about file block pointers.

In this section, we first describe our implementation, and then demonstrate that it is robust to a broad class of partial disk failures. Then, we investigate the time and space costs of ixt3, showing that the time costs are often quite small and otherwise modest, and the space costs are also quite reasonable. In our performance measurements, we activate and deactivate each of the IRON features independently, so as to better understand the cost of each approach.

5.2.1 Implementation

We now describe the ixt3 implementation. We explain how we add checksumming, metadata replication, user parity, and a new performance enhancement known as transactional checksums into the existing ext3 file system framework.

Checksumming: To implement checksumming within ixt3, we borrow techniques from other recent research in checksumming in file systems [79, 110]. Specifically, we compute checksums for each file system block and treat checksums as metadata by placing them first into the journal, and then checkpoint them to their final location, which is at the end of the file system. The checksums are separated from their corresponding blocks by a distance of about 2500 blocks (approximately 15 tracks).

We ensure atomicity for metadata checksums (*i.e.*, Checksum$_{d=t,Atomic}$) by logging both the metadata and their checksums into the journal before updating their final fixed-location copies. However, data checksum updates are not guaranteed to be atomic (Checksum$_{d=t,Non-atomic}$). By incorporating checksumming into existing transactional machinery, ixt3 cleanly integrates into the ext3 framework. Checksums are very small and can be cached for read verification. In our current implementation, we use SHA-1 to compute the checksums. By implementing a strong cryptographic checksum such as SHA-1, which involves more computational overhead than a simple checksum like CRC, we explore the extent to which checksums can impact the performance of a file system.

Metadata Replication: We use replicas to protect file system metadata, including the ext3 journal, which is treated as a metadata. We apply a similar approach in adding metadata replication to ixt3. All metadata blocks are written to a separate *replica log*; they are later checkpointed to a fixed location in a block group distant from the original metadata. The replica log is at the beginning of the file system and the corresponding blocks on the primary and replica logs are separated by at least 350 tracks. We again use transactions to ensure that either both copies reach disk consistently, or that neither do. In summary, we implement Replica$_{d=t,Atomic}$ to protect metadata blocks.

Parity: We implement a simple parity-based redundancy scheme for data blocks. One parity block is allocated for each file. This simple design enables one to recover from at most one data block failure in each file (we explore a more involved design later in Sections 5.3.1 and 5.3.2). We modify the inode structure of ext3 to associate a file's parity block with its data blocks. Specifically, ext3 inode has 15 block pointers, out of which 12 are direct, 1 single indirect, 1 double indirect, and 1 triple indirect block pointers. We use one direct pointer for pointing to the parity block of the file. Parity blocks are allocated when files are created. When a file is modified, its parity block is read and updated with respect to the new contents. To improve the performance of file creates, we preallocate parity blocks when we create the file system and assign them to files when they are created. In the current simple design, we preallocate about 250 blocks in the first block group when the file system is created, and once they are used, we follow the default block allocation scheme in ext3.

In ordered journaling mode, if a system crashes in between a data block write and the corresponding transaction commit, then during journal recovery, the file system must recompute the parity and make it consistent with the modified data block contents. However, in ext3 ordered journaling mode, no additional information is stored in the journal about the locations of data blocks that are being modified.[1] To address this problem, Denehy *et al.* introduce a new journaling mode called *declared journaling* mode, where they store the block numbers of data blocks that are being modified as part of a transaction commit into the journal [30]. To support parity for data blocks, we add declared mode to ixt3. In summary, we implement the Parity$_{d=0,Non\text{-}atomic}$ design for parity blocks in ixt3.

Transactional Checksums: We also explore a new idea for leveraging checksums in a journaling file system; specifically, checksums can be used to relax ordering constraints and thus to improve performance. In particular, when updating its journal, standard ext3 ensures that all previous journal data reaches disk before the commit block; to enforce this ordering, standard ext3 induces an extra wait before writing the commit block, and thus incurs extra rotational delay. To avoid this wait, ixt3 implements what we call a *transactional checksum*, which is a checksum over the contents of a transaction. By placing this checksum in the journal commit block, ixt3 can safely issue all blocks of the transaction concurrently. If a crash occurs during the commit, the recovery procedure can reliably detect the crash and not replay the transaction, because the checksum over the journal data will not match the checksum in the commit block. We use SHA-1 for transactional checksums also. Note that a transactional checksum provides the same crash semantics as in the original ext3 and thus can be used without other IRON extensions.

Cleaning Overheads: Note that "cleaning overhead", which can be a large problem in pure log-structured file systems [93, 101], is not a major performance issue for journaling file systems, even with ixt3-style checksumming and replication. Journaling file systems already incorporate cleaning into their on-line maintenance costs; for example, ext3 first writes all metadata to the journal and then cleans the journal by checkpointing the data to a final fixed location. Hence, the additional cleaning performed by ixt3 increases total traffic only by a small amount.

5.2.2 Evaluation

We now evaluate our prototype implementation of ixt3. We focus on three major axes of assessment: robustness of ixt3 (with checksums, parity, and replicas) to modern disk failures, and both the time and space overhead of the additional redundancy mechanisms employed by ixt3.

[1] The same problem exists in other journaling file systems such as ReiserFS, JFS, XFS, and NTFS.

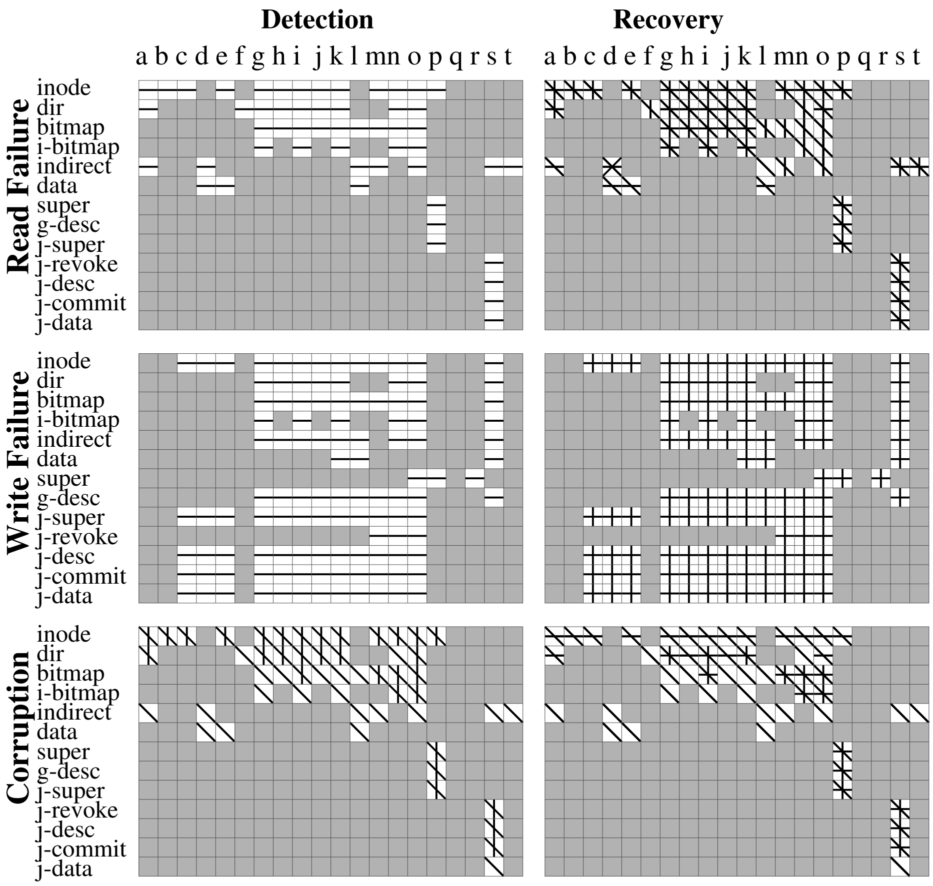


Figure 5.1: **Ixt3 Failure Policy.** *The tables plot both detection and recovery policies of ixt3 for read, write, and corruption faults injected for each block type across a range of workloads. The workloads are varied across the columns of the figure, and the different block types of the ixt3 file system are varied across the rows. The workloads are grouped in the same manner as in Figure 4.5. The keys for detection and recovery are presented in Table 4.8.*

Robustness: To test the robustness of ixt3, we harness our fault injection framework, running the same partial-failure experiments on ixt3. The results are shown in Figure 5.1.

Ixt3 detects read failures in the same way as ext3, by using the error codes from the lower level ($D_{ErrorCode}$). When a metadata block read fails, ixt3 reads the corresponding replica copy ($R_{Redundancy}$). If the replica read also fails, it behaves like ext3 by propagating the error ($R_{Propagate}$) and stopping the file system activity (R_{Stop}). When a data block read fails, the parity block and the other data blocks of the file are read to compute the failed data block's contents ($R_{Redundancy}$).

Ixt3 detects write failures using error codes as well ($D_{ErrorCode}$). It then aborts the journal and mounts the file system as read-only to stop any writes from going to the disk (R_{Stop}).

When a data or metadata block is read, the checksum of its contents is computed and is compared with the corresponding checksum of the block ($D_{Redundancy}$). If the checksums do not match, a read error is generated ($R_{Propagate}$). On read errors, the contents of the failed block are read either from the replica or computed using the parity block ($R_{Redundancy}$).

In the process of building ixt3, we also fixed numerous bugs within ext3. By doing so, we avoided some cases where ext3 would commit failed transactions to disk and potentially corrupt the file system [85].

Overall, by employing checksumming to detect corruption, and replication and parity to recover lost blocks, ixt3 provides robust file service in spite of partial disk failures. More quantitatively, ixt3 detects and recovers from over 200 possible different partial-error scenarios that we induced. The result is a logical and well-defined failure policy.

Time Overhead: We now assess the performance overhead of ixt3. We isolate the overhead of each IRON mechanism by enabling checksumming for metadata (M_c) and data (D_c), metadata replication (M_r), parity for user data (D_p), and transactional checksumming (T_c) separately and in all combinations.

We use four standard file system benchmarks: SSH-Build, which unpacks and compiles the SSH source distribution; a web server benchmark, which responds to a set of static HTTP GET requests; PostMark [61], which emulates file system traffic of an email server; and TPC-B [117], which runs a series of debit-credit transactions against a simple database. We run each experiment five or more times and present the average results.

The SSH-Build time measures the time to unpack, configure, and build the SSH source tree (the tar'd source is 11 MB in size); the Web server benchmark transfers 25 MB of data using http requests; with PostMark, we run 1500 transactions with

#	M_c	M_r	D_c	D_p	T_c	SSH	Web	Post	TPCB
0		*(Baseline: ext3)*				1.00	1.00	1.00	1.00
1	M_c					1.00	1.00	1.01	1.00
2		M_r				1.00	1.00	**1.18**	**1.19**
3			D_c			1.00	1.00	**1.13**	1.00
4				D_p		1.02	1.00	1.07	1.03
5					T_c	1.00	1.00	1.01	[0.80]
6	M_c	M_r				1.01	1.00	**1.19**	**1.20**
7	M_c		D_c			1.02	1.00	**1.11**	1.00
8	M_c			D_p		1.01	1.00	**1.10**	1.03
9	M_c				T_c	1.00	1.00	1.05	[0.81]
10		M_r	D_c			1.02	1.00	**1.26**	**1.20**
11		M_r		D_p		1.02	1.00	**1.20**	**1.39**
12		M_r			T_c	1.00	1.00	**1.15**	1.00
13			D_c	D_p		1.03	1.00	**1.13**	1.04
14			D_c		T_c	1.01	1.01	**1.15**	[0.81]
15				D_p	T_c	1.01	1.00	1.06	[0.84]
16	M_c	M_r	D_c			1.02	1.00	**1.28**	**1.19**
17	M_c	M_r		D_p		1.02	1.01	**1.30**	**1.42**
18	M_c	M_r			T_c	1.01	1.00	**1.19**	1.01
19	M_c		D_c	D_p		1.03	1.00	**1.20**	1.03
20	M_c		D_c		T_c	1.02	1.00	1.06	[0.81]
21	M_c			D_p	T_c	1.01	1.00	1.03	[0.85]
22		M_r	D_c	D_p		1.03	1.00	**1.35**	**1.42**
23		M_r	D_c		T_c	1.02	1.00	**1.26**	1.01
24		M_r		D_p	T_c	1.02	1.00	**1.21**	**1.19**
25			D_c	D_p	T_c	1.02	1.01	**1.18**	[0.85]
26	M_c	M_r	D_c	D_p		1.03	1.00	**1.37**	**1.42**
27	M_c	M_r	D_c		T_c	1.04	1.00	**1.24**	1.01
28	M_c	M_r		D_p	T_c	1.02	1.00	**1.25**	**1.19**
29	M_c		D_c	D_p	T_c	1.03	1.00	**1.18**	[0.87]
30		M_r	D_c	D_p	T_c	1.05	1.00	**1.30**	**1.20**
31	M_c	M_r	D_c	D_p	T_c	1.06	1.00	**1.32**	**1.21**

Table 5.2: **Overheads of ixt3 Variants.** *Results from running different variants of ixt3 under the SSH-Build (SSH), Web Server (Web), PostMark (Post), and TPC-B (TPCB) benchmarks are presented.*

file sizes ranging from 4 KB to 1 MB, with 10 subdirectories and 1500 files; with TPC-B, we run 1000 randomly generated debit-credit transactions. These benchmarks exhibit a broad set of behaviors. Specifically, SSH-Build is a good (albeit simple) model of the typical action of a developer or administrator; the web server is read intensive with concurrency; PostMark is metadata intensive, with many file creations and deletions; TPC-B induces a great deal of synchronous update traffic to the file system.

Table 5.2 reports the relative performance of the variants of ixt3 for the four workloads, as compared to stock Linux ext3. Along the rows, we vary which redundancy technique is implemented, in all possible combinations: M_c implies that metadata checksumming is enabled; D_c that data checksumming is enabled; M_r that replication of metadata is turned on; D_p that parity for data blocks is enabled; T_c that transactional checksums are in use. All results are normalized to the performance of standard Linux ext3; for the interested reader, running times for standard ext3 on SSH-Build, Web, PostMark, and TPC-B are 117.78, 53.05, 150.80, and 58.13 seconds, respectively. Slowdowns greater than 10% are marked in **bold**, whereas speedups relative to base ext3 are marked in [brackets]. All testing is done on the Linux 2.6.9 kernel on a 2.4 GHz Intel P4 with 1 GB of memory and a Western Digital WDC WD1200BB-00DAA0 disk.

We now explain the reasons behind some of the performance overheads. Checksumming data and metadata does not add high overhead for most workloads (*i.e.*, less than 5% in most cases). However, data checksums for PostMark increase the performance cost by about 13%. This is due to the fact that our PostMark benchmark writes over 1.4 GB of data with checksums being calculated for every data block. Since checksums are considered as metadata, when data checksums are coupled with metadata replication, the overhead on PostMark raises to about 26%. When applied individually, metadata replication reduces the performance for metadata intensive workloads such as PostMark and TPC-B, while causing little overheads for SSH-Build and Web server benchmarks. Parity for data blocks does not add much cost individually but when combined with metadata replication, it raised the overhead to as much as 39% for TPC-B. The reason for this behavior is due to the interaction between the declared mode [30] and metadata replication. In declared mode, extra synchronous writes are issued to the journal, which in addition to causing rotational latencies also incur disk seeks between the primary and secondary journal. However, when coupled with transactional checksums, this overhead drops to about 20%.

From these numbers, we draw three main conclusions. First, for both SSH-Build and the web server workload, there is little time overhead, even with all

IRON techniques enabled. Hence, if SSH-Build is indicative of typical activity, using checksumming, replication, and even parity incurs little cost. Similarly, from the web server benchmark, we can conclude that read-intensive workloads do not suffer from the addition of IRON techniques.

Second, for metadata intensive workloads such as PostMark and TPC-B, the overhead is more noticeable – up to 37% for PostMark and 42% for TPC-B (row 26). Since these workloads are quite metadata intensive, these results represent the worst-case performance that we expect. We also can observe that our implementation of metadata replication (row 2) incurs a substantial cost on its own, as does data checksumming (row 3). User parity and metadata checksums, in contrast, incur very little cost (rows 1 and 4). Given our relatively untuned implementation of ixt3, we believe that all of these results demonstrate that even in the worst case, the costs of robustness are not prohibitive.

Finally, the performance of the synchronous TPC-B workload demonstrates the possible benefits of the transactional checksum. In the base case, this technique improves standard ext3 performance by 20% (row 5), and in combination with checksumming, replication, and parity, reduces overall overhead from roughly 42% (row 26) to 21% (row 31). Hence, even when not used for additional robustness, checksums can be applied to improve the *performance* of journaling file systems.

#	Files (count)	Dir (MB)	Indir (MB)	Size (MB)	Checksum (MB)	Parity (MB)	Replicas (MB)
1	22182	5.19	9.98	1612.61	7.87 (0.5%)	086.64 (05.37%)	129.17 (8.01%)
2	12688	4.53	6.74	1885.45	9.20 (0.5%)	049.56 (02.62%)	125.27 (6.64%)
3	62222	34.56	22.26	4771.69	23.30 (0.5%)	243.05 (05.09%)	170.82 (3.58%)
4	11547	5.57	6.35	1379.23	6.73 (0.5%)	045.10 (03.27%)	125.93 (9.13%)
5	13821	5.52	10.61	1756.90	8.58 (0.5%)	053.98 (03.07%)	130.14 (7.40%)
6	126697	22.86	32.43	4043.66	19.74 (0.5%)	494.91 (12.23%)	169.29 (4.18%)
7	10207	4.28	0.54	1396.82	6.82 (0.5%)	039.87 (02.85%)	118.82 (8.50%)
8	68985	17.17	14.29	1896.05	9.26 (0.5%)	269.47 (14.21%)	145.46 (7.67%)
9	70314	26.21	17.01	1646.76	8.04 (0.5%)	274.66 (16.67%)	157.23 (9.54%)

Table 5.3: **Space Overhead.** *The above table lists the space overhead due to redundant information across nine different home directories. Inode blocks are allocated statically in ext3 and it adds about 64 MB of replica overhead per file system. Checksums for each block cause a constant space overhead of about 0.5% for each file system.*

Space Overhead: To evaluate space overhead, we measured about nine local file systems (mostly home directories of graduate students at UW Computer Sciences Department) and computed the increase in space required if all metadata was replicated, room for checksums was included, and an extra block for parity was allocated. Table 5.3 presents a detailed list of space overhead numbers. Each row lists the space cost per file system. For each file, one parity block is allocated. Directory and indirect blocks are dynamic metadata blocks, which changes across file systems, while inode blocks and bitmap blocks are statically allocated when the file system is created. Checksums cause only a constant, small space overhead of about 0.5%. Overall, we found that the space overhead of checksumming and metadata replication is small, in the 3% to 10% range. We found that parity-block overhead for all user files is a bit more substantial, in the range of 3% to 17% depending on the volume analyzed.

5.2.3 Summary

We investigate a family of redundancy techniques, and find that ixt3 greatly increases the robustness of the file system under partial failures while incurring modest time and space overheads. We also see that parity blocks can strike a balance in terms of robustness, performance, and space overheads, especially for user data, which occupies a significant part of a local file system. We believe that ixt3 represents just a single point in a large space of possible IRON file systems. In the following sections, we explore other designs, focusing specifically on parity blocks.

5.3 Redundant Parity Blocks

In the previous section, we studied in-breadth the cost and effectiveness of various redundant information. However, in some cases, our design decisions were *simple* such as using a single parity block per file. In this section, we explore more in-depth the issues in building in-disk redundancy on commodity hardware, focusing primarily on parity as the redundant information due to its simplicity, minimal space overhead, ease of computation, and widespread use in file and storage systems [21, 28, 96].

First, in order to explore a data layout for parity blocks that is robust against spatially correlated failures, we need a failure model. In this regard, we propose using Markov Random Fields [68] to express the spatial dependencies across disk blocks and develop a simple probabilistic model for data placement assuming spatially local failures. Second, we need new techniques to update the parity blocks and data blocks atomically, since writing both to the journal can significantly increase the performance cost. We discuss two parity update techniques, a traditional *overwrite* and a radical *no-overwrite* technique, which differ in the way they interact with the file system journaling when ordering their block reads and writes. The overwrite technique does not guarantee atomic update of data and parity (Parity$_{Non\text{-}atomic}$) but no-overwrite technique provides this atomicity (Parity$_{Atomic}$). Finally, we evaluate the performance overhead of overwrite and no-overwrite techniques.

5.3.1 Spatial Locality in Sector Failures

To handle block failures within the file system, one must have a model of how such block failures occur. One important characteristic such failures exhibit is spatial locality (Section 2.4). Specifically, media errors due to particle scratches, thermal asperity [43], or bad disk head can render a set of contiguous blocks unusable [59].

File system developers are aware of this problem and have considered fault-isolated data placements for certain important blocks. For example, in the original design of Berkeley Fast File System, redundant super blocks are carefully placed such that no single failure of a disk track, cylinder or surface endangers all the copies [71]. However, generic block layout algorithms in modern file systems are focused on improving performance and are oblivious to spatially local disk errors. For example, IBM JFS stores both the primary and secondary copies of its super block close together (separated by seven blocks). Such design decisions can make both the primary and secondary blocks unusable if there are spatially local errors.

In this section, we propose a probabilistic disk failure model that characterizes disk block errors with spatial locality and use this model to layout blocks such that the probability of two or more errors affecting related blocks is low. Our techniques are developed specifically for journaling file systems because modern file systems such as Windows NTFS, Linux ext3, ReiserFS, IBM JFS, and XFS are journaling file systems, and none of these file systems currently provide parity-based redundancy. We evaluate the validity of the probabilistic model by comparing it with results from fault injection experiments and the probability of multiple failures as predicted by the failure model matches closely with the results from the fault injection experiments.

Probabilistic Failure Model

Let us define a *redundancy_set(k,f)* to be a set of k disk blocks, viz., B_0, B_1, ..., B_{k-1} that can tolerate a maximum of f failures. That is, if more than f blocks in the set of k blocks fail, the redundancy set cannot be recovered. The k disk blocks can contain file system data or metadata. In the following discussion, we consider a *redundancy_set(k,1)* that can tolerate only up to 1 failure, and cannot recover from multiple failures. [2] We define two events:

F_1: A block, B_i, in the redundancy set fails.
F_2: At least one more block in the redundancy set fails.

When event F_2 happens, the file system cannot recover the failed blocks from the redundancy set. $P(F_2|F_1)$ gives the probability that a second failure happens given that a first failure has already occurred. Using conditional probability,

$$P(F_2|F_1) = \frac{P(F_2 \cap F_1)}{P(F_1)}.$$

The objective of the data layout algorithm is to select blocks such that the probability of more than one block failure occurring in the same redundancy set is below a certain threshold $\mathbb{T}$. That is, we want

$$P(F_2 \cap F_1) < \mathbb{T} \quad (5.1)$$

$$\Rightarrow \{P(F_2|F_1) \times P(F_1)\} < \mathbb{T} \quad (5.2)$$

We calculate the probability of more than one block failure under two different cases. First, we consider block errors as independent events and derive the probability for 2 or more failures in a redundancy set. Then, we incorporate spatial locality into the model and solve for a more generic case. We denote $P(F_2 \cap F_1)$ by $P(\mathbb{E}_r)$, where $\mathbb{E}_r$ denotes the event of two or more failures happening in a redundancy set r containing k blocks.

Case 1: We can treat block failures as independent events and assume that they are uniformly spread on the disk. That is, if there are n blocks on the disk and some block fails, then the probability that a block B fails is $\frac{1}{n}$. If the block failures are independent, then $P(\mathbb{E}_r)$ is given by:

$$P(\mathbb{E}_r) = 1 - P(\text{Exactly one failure}) - P(\text{No failures})$$

$$\Rightarrow P(\mathbb{E}_r) = 1 - [{}^k\mathrm{C}_1(\frac{1}{n})(1 - \frac{1}{n})^{k-1}] - [(1 - \frac{1}{n})^k] \quad (5.3)$$

where ${}^k\mathrm{C}_j$ gives the number of combinations of j blocks out of k blocks and $(1 - \frac{1}{n})$ is the probability with which a block does not fail.

Case 2: In reality, latent sector faults are not independent and can have spatial locality. Therefore, a more realistic fault model takes into account that block

[2] Note that we interchangeably use the term "multiple failures" to refer to 2 or more failures.

failures may be correlated and the probability of failure can differ according to the location of a block with respect to other faulty blocks on the disk.

We propose that the spatial dependencies among a set of sites can be expressed using Markov Random Fields (MRF). MRF gives the probability distribution at a site i, specified conditionally on the sites in the *neighborhood* of i. The neighborhood of a site i is defined by some function. For example, all the sites within a radius of R from site i could be defined as neighbors of i.

We give a brief introduction to MRF and then discuss how we can express the dependencies between the disk block failures as an MRF (a more detailed treatment of MRF can be found elsewhere [68]). Let F be a finite collection of random variables *i.e.*, $F = \{V_1, ..., V_n\}$. Each random variable V_i corresponds to a site i and takes a value v_i from a set L_i. Let N_i be the set of neighbors of i. F is an MRF if and only if the random variable V_i depends only on its neighbors. More formally,

$$P(V_i = v_i | V_j = v_j, \text{ for } j \neq i) \quad = \quad P(V_i = v_i | V_j = v_j \text{ for } j \in N_i) \qquad (5.4)$$

The above condition brings out the local characteristics of F, which states that a site i depends only on its neighbors N_i. Note that for an MRF, it is always possible to take a sufficiently large N_i such that the above condition holds.

If the probability that a disk block fails depends only on its neighbors then the spatial locality of disk block failures can be expressed as a Markov Random Field. For example, if a block is affected only by the failures of other blocks on the same track, then blocks on the same track can be defined as neighbors of each other. The most general definition would consider all the blocks on the disk as neighbors (in fact, this is the definition we adopt for our evaluation). Depending on the definition of the neighborhood of a block, the layout algorithm can construct a redundancy set such that two or more errors happening in the same redundancy set is minimal.

In the following discussion, we treat the state of each block as a random variable, and it can take one of the two values $\{$*valid, failed*$\}$. We treat the set of disk block states as an MRF and derive the probability for two or more failures as follows. Consider two blocks B_i and B_j in the redundancy set. Assume that B_i has failed. If B_j is not a neighbor of B_i, then it is not affected by B_i's failure. However, if B_j is a neighbor of B_i, then the probability that B_j fails depends upon the distance of B_j from B_i (due to spatial locality). Let d be the distance of block B_j from the failed block B_i, and $P(d)$ represent the probability of failure at a distance d. The distribution $P(d)$ can be chosen to reflect the extent to which block errors are spatial. For example, one can choose an exponential decay distribution for $P(d)$ if the probability of failure drops exponentially as the distance d increases. Among all the k blocks of the redundancy set, let the block B_i have m_i blocks as neighbors.

For the sake of simplicity, let all the m_i blocks be at an exact distance of d_i from the failed block B_i. The probability of two or more failures in a redundancy set given that block B_i has failed is given by:

$$P(\text{At least one more block out of } m_i \text{ fails} \mid B_i \text{ has failed}) \times P(\text{Block } B_i \text{ has failed})$$
$$\Rightarrow [1 - P(\text{None of the } m_i \text{ blocks fail} \mid B_i \text{ has failed})] \times P(\text{Block } B_i \text{ has failed})$$
$$\Rightarrow [1 - (1 - P(d_i))^{m_i}] \times \frac{1}{n} \tag{5.5}$$

where $(1 - P(d_i))^{m_i}$ is the probability with which all the m_i blocks do not fail. In practice, the m_i blocks must be *at least* at distance d_i and then Equation 5.5 gives the upper bound on the probability of multiple failures. Our objective is to find the probability of multiple failures for all such B_i in the redundancy set r. This is given by,

$$P(\mathbb{E}_r) = \sum_{i=1}^{k}[1 - (1 - P(d_i))^{m_i}] \times \frac{1}{n} \tag{5.6}$$

Equations 5.3 and 5.6 give the probability of 2 or more failures in a redundancy set r with k blocks. A disk with n blocks has $\frac{n}{k}$ redundancy sets. To find the probability of multiple failures happening over the entire disk, one must consider all the $\frac{n}{k}$ redundancy sets while calculating the probability using either Equation 5.3 or 5.6. Let $q = \frac{n}{k}$. We derive the probability of multiple failures within at least one redundancy set (represented by $P(\mathbb{M})$) over the entire disk as follows:

$$P(\mathbb{M}) = 1 - P(\text{No multiple failures in all redundancy sets})$$
$$= 1 - \prod_{i=1}^{q}[1 - P(\mathbb{E}_i)] \tag{5.7}$$

where $\prod_{i=1}^{q}[1 - P(\mathbb{E}_i)]$ gives the probability of no multiple failures happening in all the redundancy sets.

Using the failure model

In this section, we discuss how the failure model can be employed by a file system to construct redundancy sets and lay out blocks. First, we use Equation 5.7 to compute the probability of multiple failures and then describe how this probability can be used by file systems to construct redundancy sets.

In practice, the type of distribution to choose for $P(d)$ must be derived by analyzing the sector errors over a large set of failed disks. Although no such distribu-

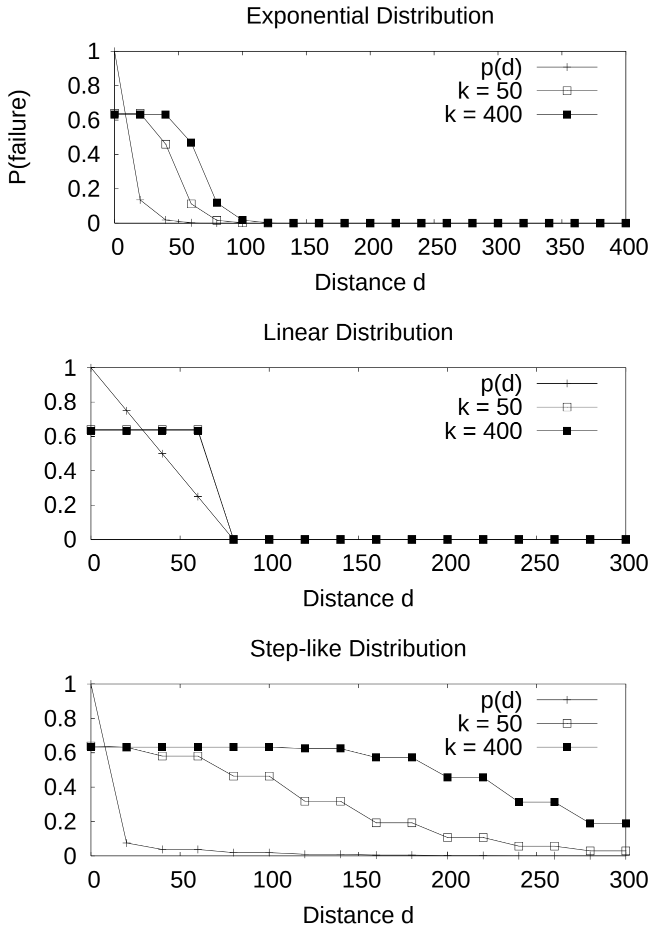


Figure 5.2: **Multiple Failures Under Different Probability Distributions.**

tion is available today (due to the proprietary nature of the disk industry), they can be made available in the future by disk manufacturers or users of large disk farms. Since there are no numbers available to choose for $P(d)$, we use different distribution functions for $P(d)$ to compute the probability of multiple errors happening on at least one redundancy set over the entire disk.

In Figure 5.2, we plot the probability of multiple failures under three different distributions for $P(d)$: one, an exponential decay curve, where the probability of failure drops exponentially as the distance of a block increases from an already failed block (left-most graph in Figure 5.2); two, a distribution with a linear drop in probability (middle graph in Figure 5.2); finally, a step-like distribution that has the same probability over a certain set of contiguous tracks, and then drops further (right-most graph in Figure 5.2). Each graph in Figure 5.2 has 3 curves in it. The curve denoted by $P(d)$ plots the failure probability of a block at a distance d from an already failed block. The two other curves plot the probability of multiple failures happening over the entire disk for different values of k (size of the redundancy set).

We discuss three main points of Figure 5.2. First, instead of plotting the values of $P(\mathbb{E}_r)$ (Equation 5.6), which gives the probability of multiple failures for a *particular* redundancy set r, we plot the probability of multiple failures happening within at least one redundancy set over the *entire* disk of size 50 GB (Equation 5.7). The reason is $P(\mathbb{E}_r)$ does not capture how a large disk with several thousands of redundancy sets can have a higher probability of multiple failures. Second, we plot the probability of multiple failures for two different values of k viz., $k = 50$ and $k = 400$. As k (the size of the redundancy set) increases, the probability of two or more failures also increases. Finally, the graphs plot the failure distribution over a simple two dimensional surface. However, disks can have multiple surfaces, and therefore, the failure distribution $P(d)$ could be chosen to express the spatial locality across multiple surfaces, and across multiple tracks on a single surface.

A file system can compute the probability of multiple failures using the failure model and layout the related blocks accordingly. For example, assuming that the distance d represents track separation, if $P(d)$ follows an exponential decay curve similar to the left most graph of Figure 5.2, file systems can construct redundancy sets such that the blocks of the set are separated at least by a distance of 100 tracks in order to keep the probability of multiple failures low. It is important to note that logical file system entities like files and directories themselves are allocated contiguously irrespective of whether the distance d is set to 100 or 0. Setting d to 100 means that the redundancy set blocks are separated by at least 100 tracks. Figure 5.3 depicts one particular case of such a layout where a file is allocated contiguously on a single track, while the corresponding blocks on every 100th track belong to the same redundancy set. A file system can also use Equation 5.7 to derive the size of the redundancy set based on its fault tolerance requirements.

Failure model limitations

Our model has the following limitations. First, we do not account for temporal locality in block access patterns. If two blocks are accessed together, they might be affected by a bad disk head. Second, the block layout algorithm is based on

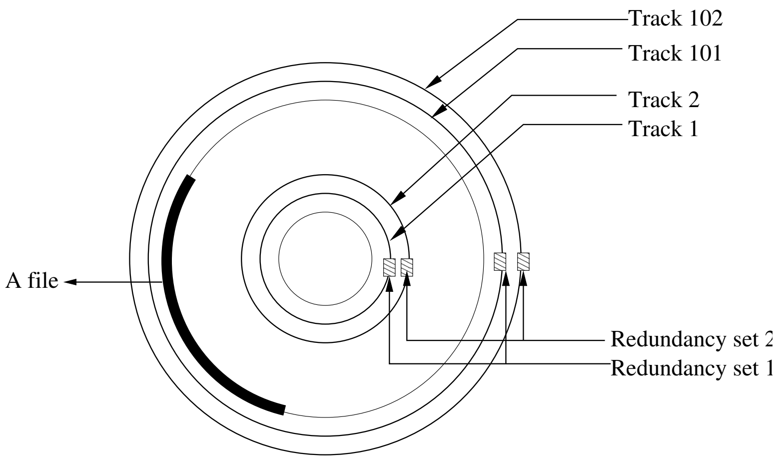


Figure 5.3: **Block Layout Using the Failure Model.** *The above figure shows an example of how redundancy sets are constructed from various disk tracks. Note that the physical contiguity of logically contiguous blocks are not affected. For example, all the blocks in a file can still be allocated continuously.*

the assumption that file systems can observe the disk geometry and control block placements. Logical block to physical disk address mappings can be obtained for SCSI disks using low level SCSI commands. However, for low end ATA drives such features may not be available. For such low end drives, the disk features can be extracted by running micro benchmarks similar to the one developed by Talagala *et al.* [114]. Finally, lower layers on the storage stack can reassign blocks to different locations on the disk breaking the layout pattern suggested by the file system [111]. For example, disk drives themselves can do bad block remapping which can affect the file system's assumption about where disk blocks are located. However, in modern systems, such remapping is rare and the locations of the remapped blocks can be obtained through low-level disk commands.

Model Evaluation

In this section, we describe the evaluation of the probabilistic failure model. We evaluate the probability model by comparing the results from equation 5.7 with the results from a set of fault injection experiments on a simulated disk.

The fault injection experiment works as follows. First, we construct parity sets on a disk with n blocks. Then, a block B out of the n disk blocks is selected at

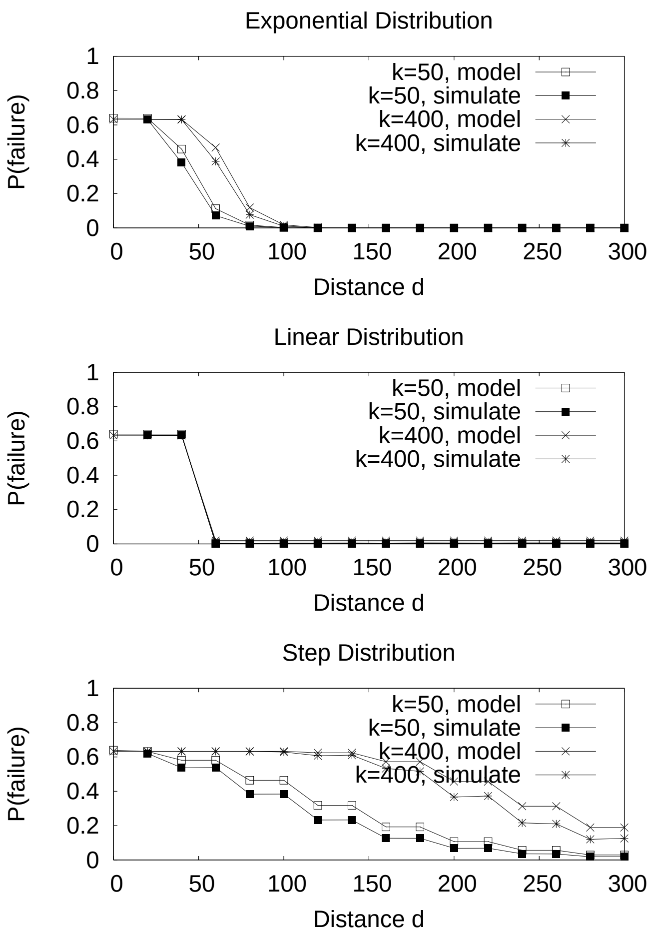


Figure 5.4: **Comparing the Probability of Multiple Failures Using Failure Model and Simulation.**

random and failed. Following the first failure, the other blocks from the same parity set as B are considered for the second failure. Depending on the distance of a block from the failed block B, we compute the chances of the second failure and inject the second fault probabilistically. We run the experiment for 10,000 iterations on a simulated disk of 50 GB. The probability of the second (or higher number of failures) is computed as the number of times a second or a higher number of faults were injected over 10,000 runs. We repeat the experiment for different parity set sizes and block distances. In Figure 5.4, we plot the probability of 2 or more failures from the equation 5.7 and the results of the fault injection tests. From Figure 5.4, we can see that the curves from the model and experiments match closely.

5.3.2 Redundant Data Update Techniques

As we pointed out in Section 5.3, commodity file systems run on cheap disk drives without additional hardware support such as NVRAM. NVRAM support can be useful when redundant information must be updated *atomically*, that is, either both the primary and secondary copies are updated or none of them are modified. In this section, we describe two techniques to update data and its corresponding parity. First, we explain a traditional overwrite technique, which updates the data blocks *inplace* and we employ this technique while implementing parity in ixt3. Although simpler to implement, we show that this method does not guarantee atomic update of data and parity (Parity$_{Non\text{-}atomic}$). Second, we develop a radical no-overwrite approach, which writes the data to unallocated blocks and switches block pointers to point to the new blocks. No-overwrite technique can atomically update the data and parity (Parity$_{Atomic}$), and in principle, this is similar to the Write Anywhere File Layout [53] approach that is used to maintain file system integrity. Both the overwrite and no-overwrite techniques are integrated with the transactional semantics in journaling file systems.

We implement overwrite and no-overwrite techniques along with a block layout proposed by the probabilistic model from the previous section. Our performance evaluation shows that separating related blocks widely to avoid spatially local errors has some performance impact and it varies according to the workload. We run microbenchmarks and show that overwrite update incurs more performance overhead compared to no-overwrite technique. By batching parity updates, no-overwrite technique reduces costly disk seeks and thereby reduces performance overhead. However, no-overwrite technique can fragment a file and therefore requires periodic cleaning to re-layout the file blocks contiguously.

First, we briefly recapitulate the journaling functionality, followed by an explanation of how parity update schema interact with journaling modes. Then, we describe the overwrite and no-overwrite techniques, and finally, evaluate the techniques.

Journaling file systems

As we explain in Section 3.1, journaling file systems offer three different journaling modes: data journaling, ordered journaling, and writeback journaling. Each mode differs from the other by the type of blocks they write to the journal and the order in which the blocks are written. We focus on ordered journaling mode, which is the default mode in the modern file systems such as Linux ext3, ReiserFS, IBM JFS, XFS, and Windows NTFS due to its low performance overhead compared to the data journaling mode and better consistency semantics compared to the writeback mode. In fact, JFS, XFS, and NTFS do not support any other journaling modes.

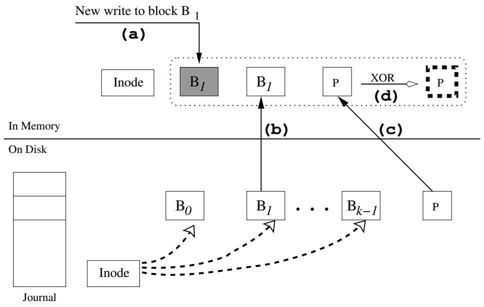
In ordered journaling mode, only the metadata blocks are written to the journal. The data blocks are written directly to their fixed locations. But, the data writes are ordered such that they complete before the metadata blocks are committed to the journal. For example, when an application writes to a block, first the data block on the disk is updated. Next, the inode block and other metadata blocks are committed to the journal. Once the transaction is committed, at a later point in time, the inode and other metadata blocks are written to their final fixed locations on disk and this process is called *checkpointing*.

Parity Scheme in Journaling File Systems

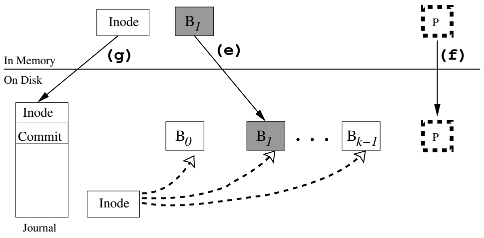
A file system can implement a parity-based redundancy scheme. Parity is calculated on a set of blocks and stored on a separate disk location. Parity is calculated for both file system data and metadata. The location of the parity block for a set of disk blocks is stored on a mapping and this mapping itself is protected by redundant copies. In the following discussion, we refer to a redundancy set that uses parity as the redundant information as a parity set.

In a journaling file system, parity-based redundancy works along with the journaling layer. That is, whenever a block is written, the parity block that protects the block must also be updated in tandem during ordered writes. It is important that the data and parity are updated atomically, or else the contents of the parity block will be inconsistent with respect to the other blocks in the parity set.

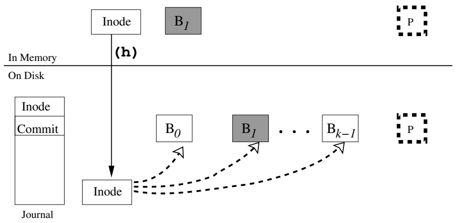
There are three phases in the data and parity update procedure: one, a *parity update* phase where the old copies of the data and parity blocks are read from the disk and the new contents of the parity block are computed using XOR operations; two, a *transaction commit* phase where the data and parity blocks are updated through ordered writes and the metadata blocks are committed to the journal; and finally, a *checkpoint* phase where the metadata blocks that were logged to the journal are written to their final fixed locations on the disk. The overwrite and no-overwrite techniques differ in the order in which these phases are executed and where the data is written to the disk. Specifically, overwrite and no-overwrite techniques implement Parity$_{Non\text{-}atomic}$ and Parity$_{Atomic}$ respectively from Table 5.1.



(i) Overwrite Phase I: Parity Update



(ii) Overwrite Phase II: Transaction Commit



(iii) Overwrite Phase III: Transaction Checkpoint

Figure 5.5: **Overwrite Technique for Parity Update.** *The figure shows the sequence of steps followed by the file system to update the parity block in overwrite technique.*

Overwrite

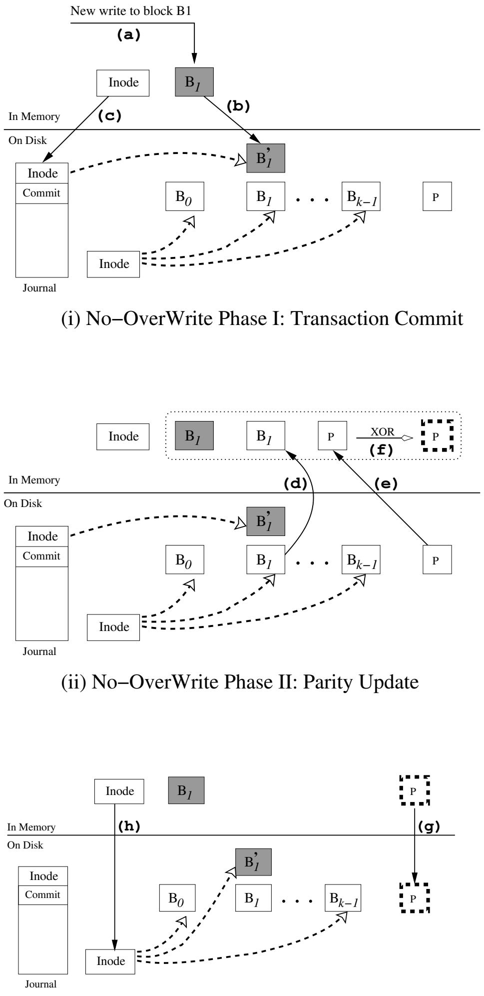
Figures 5.5(i) to 5.5(iii) depict the three phases of overwrite update. The first phase is the parity update phase. When a new write is issued, the old copies of the data block and the parity block are read from the disk (steps (a), (b), and (c)). Using the old copies and the new write, the new parity contents are calculated (step (d)). The second phase is the transaction commit phase in which the new data and parity blocks are written to their fixed locations on disk (steps (e) and (f)). Once the ordered writes are over, the metadata blocks including the inode block are committed to the journal (step (g)). The last phase is the checkpointing phase during which the journaled blocks are written to their fixed locations on disk (step (h)). At the end of the third phase, the parity set is consistent and if one of the blocks in the parity set fails, it can be recovered using the rest of the blocks.

However, the overwrite update algorithm does not provide atomic update of data and parity, and therefore, cannot recover from block errors during journal recovery. This can be explained by considering the following case: if the file system crashes after step (e) and before the completion of step (g) in Figure 5.5(ii), then during journal recovery, all the disk blocks in the parity set must be read to recompute parity. This is due to the lack of atomicity, which results in an inconsistent parity block after the crash. However, while reading the parity set blocks, if the file system encounters a latent sector fault or block corruption, then the parity block cannot be used to recover the failed block because the parity set is in an inconsistent state. The root cause of the problem is in the overwrite update; by modifying the old contents of the data block, the parity set is made inconsistent until the parity block is also updated with its new contents. Note that even if the parity block is in a consistent state during the crash (say, the crash happened after step (f) and before step (g)), the file system cannot provide guarantees about the consistency of the parity block, and therefore it has to read all k blocks to compute the parity. In general, the same problem would occur if the file system crashed at any point in time before the completion of journal commit (step (g)) in the transaction commit phase (Figure 5.5(ii)).

No-Overwrite

In this section, we describe the no-overwrite technique to update the parity block. In this schema, the data block is not overwritten; rather, it is written to a new location on disk. The three phases of this technique are shown in Figures 5.6(i) to 5.6(iii).

The first phase is the transaction commit (in contrast to the parity update in the overwrite technique). When a write to a block is issued, instead of overwriting the old contents, a new block is allocated and the data is written to the new location (steps (a) and (b)). This constitutes the ordered write in the ordered journaling mode. Note that the new block allocated must be in the same parity set as the old block. That is, in Figure 5.6, blocks B_1 and $B_1^{'}$ must be from the same parity group. After the ordered writes, the metadata blocks including the inode are committed to



(i) No−OverWrite Phase I: Transaction Commit

(ii) No−OverWrite Phase II: Parity Update

(iii) No−OverWrite Phase III: Transaction Checkpoint

Figure 5.6: **No-overwrite Technique for Parity Update.** *The figure shows the sequence of steps followed by the file system to update the parity block in no-overwrite technique.*

the journal (step (c)). The second phase, which is the parity update phase, happens before checkpointing. During this phase, the old versions of the data and parity blocks are read from the disk, and the new parity contents are calculated (steps (d), (e), and (f)). The final phase is the checkpoint phase, where the parity block is written to its original location on disk (step (g)), followed by the checkpoint of the metadata blocks to their fixed locations (step (h)). Note that in step (h), the block pointer in the inode has changed to refer to the new location of the data block.

Since the data blocks are not overwritten, this schema provides the necessary atomicity and can recover from block failures even during journal recovery. Considering the same example we saw before, even if the file system crashes after step (b) in Figure 5.6(i), the parity set remains in a consistent state. Therefore, any block error in the parity set can be recovered by reading the other blocks in the parity set. A file system crash happening during any step in Figures 5.6(i) to 5.6(iii) followed by a single block error during journal recovery can be successfully recovered because at any point the parity block is either consistent with the old contents of block B_1 or with the new contents of block B_1'. Since the parity block is always in a consistent state, block errors can be recovered successfully.

The disadvantage of this approach is that it fragments a file into several blocks due to a new block allocation on every overwrite. Periodic cleaning of fragmented blocks to reallocate them contiguously can be performed to improve the file contiguity on the disk.

5.3.3 Performance Analysis

We now evaluate the performance of overwrite and no-overwrite approaches, and compare it to default ext3. We run a set of microbenchmarks and macrobenchmarks similar to those used by Stein *et al.* to evaluate their checksum implementation [110]. First, we present the implementation and system configuration details, then discuss the microbenchmark results, and finally explain the macrobenchmark results.

Implementation: We implement the overwrite and no-overwrite techniques in Linux ext3. In both the techniques, reads are directly issued to the corresponding disk blocks (*i.e.*, reads do not incur any additional processing overhead). However, file system writes may incur extra disk reads and writes. In overwrite mode, for ordered and checkpoint data writes, old copies of the blocks are read (if they are not already present in a local cache), parity is computed, and written to disk. In no-overwrite mode, a new block is allocated for an ordered write and the parity computation occurs during checkpointing. For the evaluation purposes, we run the experiments on 4 GB partition and separate the blocks in the same parity set by at least 5 tracks (about 800 KB). That is, this design can tolerate spatially local failures that are no longer than 800 KB. In order to reduce the seek overhead, we group the writes in a queue and issue them as a whole. In all the experiments, we cache the parity blocks, which consumed about 1 MB of memory. Our test system

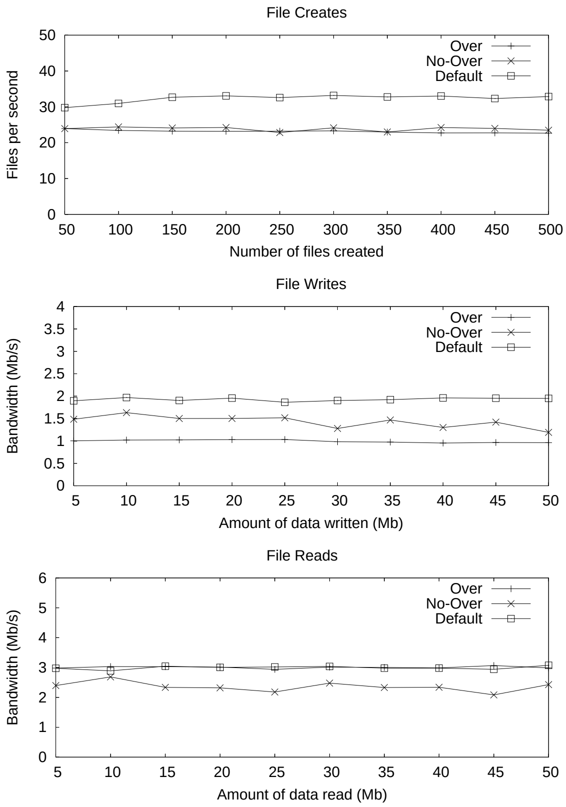


Figure 5.7: **Performance Under Microbenchmarks.** *The three graphs plot the performance of overwrite and no-overwrite techniques with respect to ext3. Performance under file creates, file writes, and file reads are plotted in the top, middle, and bottom graphs respectively.*

consists of a Linux 2.6.9 kernel on a 2.4 GHz Intel P4 with 1 GB of memory and a Western Digital WDC WD1200BB-00DAA0 disk.

Microbenchmarks: The microbenchmarks consist of three phases. First, files are created synchronously with 1KB data each. We create between 50 to 500 files, and not more than 50 files per directory in order to limit the pathname lookup time. Second, a 100 KB of data is written to each file synchronously. Totally, we write between 5 MB to 50 MB to all files. Finally, the files are read. All the phases issue plenty of random I/Os to small files spread across multiple directories and some sequential I/Os within each file. Between each phase, the file system is unmounted to clear the cache contents. Although artificial, this helps us separate the performance impact in each phase.

Figure 5.7 presents the results of microbenchmarks and we make the following observations. First, in File Creates, overwrite and no-overwrite perform worse than default ext3 and this is due to the additional disk traffic incurred by reads to old copies of blocks and writes to parity blocks. Second, in File Writes, no-overwrite performs better than overwrite approach. Examining the disk traffic further, we find that this is due to the higher disk seek incurred in the overwrite approach. If writes are issued over multiple transactions, overwrite technique updates the parity on every transaction by reading the old copies of the data and parity blocks (Figure 5.5(i)) whereas the no-overwrite scheme batches all the parity updates together just before checkpointing (Figure 5.6(ii)). This parity update batching reduces the disk seeks and improves the write performance. Finally in File Reads, in contrast to writes, no-overwrite scheme performs worse than overwrite approach (which has no performance degradation with respect to default ext3). Since no-overwrite allocates a new block for every ordered write, over time the data gets scattered around the disk. Therefore, while reading back the data written, we pay additional cost by seeking to the scattered data locations. With periodic de-fragmentation, this overhead can be reduced.

Techniques	**SSH**	**Web**	**PostMark**	**TPC-B**
Overwrite	1.09	1.00	**1.19**	**1.46**
No-Overwrite	**1.11**	1.04	**1.26**	**1.50**

Table 5.4: **Performance Overheads of Overwrite and No-Overwrite.** *The table lists the relative running overheads of four different benchmarks in overwrite and no-overwrite techniques when compared to Linux ext3. Overheads greater than 10% are marked in* **bold**.

Macrobenchmarks: We run a set of four macrobenchmarks, similar to the ones we used to evaluate the performance of ixt3 (Section 5.2.2): a benchmark representing desktop type workloads (SSH-Build), a read intensive workload (web server), a metadata intensive benchmark (PostMark [61]), and finally, a workload with lots of synchronous writes (TPC-B [117]).

The relative overheads of four benchmarks in overwrite and no-overwrite techniques with respect to default ext3 are presented in Table 5.4. From the table, we can derive the following conclusions. First, both overwrite and no-overwrite incur additional overhead when compared to default ext3, and the magnitude of the overhead depends on the workload. For write intensive workloads with no idle time like PostMark or TPC-B, the overhead varies from 19% to 50%. Note that we run these benchmarks on a cold cache with no copies of disk blocks present in memory and this results in extra overhead due to reads to old disk block copies. However, on an actual system some blocks may be present in the cache, and we expect that this can reduce some overhead. For SSH, which is a desktop-like workload, the overhead is modest (9% for overwrite and 11% for no-overwrite). Second, there are no additional overheads for the web server workload in overwrite technique but no-overwrite incurs some overhead (about 4%) due to scattered file blocks (we see a similar behavior in the File Reads microbenchmark as well). Finally, although no-overwrite technique performs better when compared to overwrite technique on simple write intensive microbenchmarks, its performance degrades when the benchmark involves both reads and writes.

In summary, the no-overwrite technique incurs less overhead compared to the overwrite technique, primarily due to the batching of parity updates. However, the no-overwrite technique has the disadvantage of reallocating the file blocks to different locations on disk and thereby fragmenting the file. Periodic cleaning with file re-layouts is necessary to maintain the physical contiguity of logically contiguous file blocks.

5.4 Conclusion

Although redundant information has been used for years in building robust file and storage systems, their use within a single disk has not been thoroughly studied before. Low-end systems offer several challenges including the lack of redundant disk drives and NVRAM support. In this chapter, we explore the effectiveness and cost of different redundancy techniques that use checksum, parity, and replica to detect and recover from errors. From our experiences in building ixt3, we conclude that while the performance cost of redundant information can be severe for high I/O intensive workloads, the overheads are modest for typical desktop applications. We also find that the parity approach strikes a balance in terms of effectiveness, performance, and space overheads. Since disk sectors can be subject to spatially correlated failures, parity blocks must be laid out carefully. We derive a simple probabilistic equation that can be used by a file system to construct parity sets such that the chances of multiple errors affecting the same set is low. Finally, we develop two parity update techniques. While the traditional overwrite approach modifies data blocks in place, the radical no-overwrite approach writes data to new locations and changes block pointers. Although no-overwrite offers the atomicity in updating redundant information, it can fragment a file and therefore requires periodic cleaning.

Chapter 6

Unifying Failure Handling with Low-level Machinery

In this chapter, we address the second challenge in building robust file systems; that is, how to incorporate low-level redundancy machinery with the myriads of high-level failure handling policies. We restructure commodity file systems with a Centralized Failure Handler that unifies all the I/O failure handling at a single point. First, we describe the design and implementation details and then explain the evaluation of the system.

6.1 Centralized Failure Handler

Our failure policy analysis reveals that file system failure handling is broken with failure policies that are inconsistent, inflexible, coarse-grained, and erroneous in error propagation.

File systems contain complex interleaving of I/O calls that are initiated from various locations in the file system code. For example, disk reads and writes are issued by system calls, background daemons, journaling layer, and buffer cache manager. In addition, there are functions that notify the I/O completion, which are special routines that are different from the functions that issue the I/O. This results in diffusion of I/O specific routines throughout the file system code.

Diffusion of I/O calls leads to diffusion of I/O failure handling. That is, the code that detects I/O failures and performs recovery (such as retry or stopping the file system) is spread over different places. This leads to several problems:

Inconsistent Policies: File systems contain *illogically inconsistent* failure handling policies, where different failure handling techniques are used even under similar failure scenarios *unintentionally*. For example, in Figure 4.5 that is presented in Section 4.3.1, we can observe that for similar indirect block read errors, four different recovery actions are taken by ext3: zero (R_{Zero}), propagate ($R_{Propagate}$), stop (R_{Stop}), and retry (R_{Retry}). We confirm by verifying the source code that this inconsistent failure handling is due to *ad hoc* application of recovery techniques.

Tangled Policies and Mechanisms: Due to the diffusion of I/O calls and their corresponding failure handling, it becomes harder to separate failure policies (*e.g.*, "detect block corruption") from their implementation (*e.g.*, "do a specific sanity checking"). As a result of this tangled policy and mechanism, neither can be modified without affecting the other, resulting in an inflexible failure handling system.

Coarse Grained Policies: Modern file systems do not support fine-grained per-block-type failure policies. Under different block type failures, file systems often use the same recovery technique such as stopping the entire file system. In the current file system framework, if different I/O failures must be handled with different failure policies, the detection and recovery code must change at all the places where an I/O is being issued. This can increase the diffusion of failure handling and in turn, worsen the above mentioned problems.

We propose a Centralized Failure Handler (CFH) for file systems as a way to mitigate the aforementioned problems and support well-defined failure policies. The CFH architecture decouples failure handling mechanisms from policies, and offers a configurable framework where a file system can specify per-block-type failure policies.

There are many challenges in building a CFH. First, the CFH needs semantic information about I/O calls; that is, what block type each I/O represents. Broadly, there are two ways to obtain this information. One, we can apply reverse engineering techniques with file system level gray-box knowledge [9, 83] to the I/O stream and find the block types. Although similar techniques have been developed in the past to do this to a certain extent [10, 12, 105, 106, 108], they are complex and cannot guarantee correct block type detection due to file system asynchrony. The other approach is to modify the file system source code to explicitly pass block type information along with each I/O call, and we adopt this approach. Second, even if one has access to source code, it is not always possible to get the correct block type due to generic file system components. For example, if an I/O is issued by a file system via the generic buffer cache layer, block type information is lost. We modify the common journaling layer and buffer cache manager to overcome this limitation. Finally, failure policies are harder to specify without the correct framework due to hundreds of combinations of block types, detection, and recovery actions that are possible. We have developed a policy table that abstracts away the complex details of mechanisms and offers a simple interface to specify even intricate failure policies.

We design, implement, and evaluate a prototype CFH for Linux ext3, where we show that CFH can detect and recover from failures in a consistent manner as specified by the file system. Our evaluation brings out the flexibility of the CFH, where a few lines of change in the configuration modifies the failure policy of ext3 to behave like ReiserFS or NTFS.

6.1.1 Design and Implementation

The main goal of CFH is to provide a single locale within the system for a file system policy to be specified and appropriately handled. Thus, all requests must be routed through the CFH and I/O failures must be reported to the CFH first.

Key to this vision is the *policy table*, which specifies the failure-handling policy of the system, thus enabling CFH to cleanly separate policy from mechanism. The policy table should be easy to read (thus ensuring that the policy that is enacted is the policy that is desired) and easy to modify (thus ensuring that new policies can easily be put into place as needed).

Also critical to the design of CFH is the availability of the correct *semantic information*. For example, when a specific request passes through the CFH, information about its block type and other relevant information about the request should be accessible within the CFH. Such information is crucial in enabling fine-grained and flexible policies to be implemented.

We now describe our approach to centralized failure handling. We start with the basic framework, then discuss issues in the design of the policy table and acquiring semantic information, and finally explain a few implementation details.

Basic Framework

Figure 6.1 shows the framework of the CFH. I/O calls are issued from different components of a file system such as the core file system, journaling layer, and buffer cache manager. Each of the reads and writes are passed to the CFH, which then issues the request to the lower level device drivers of the disk.

Each I/O call, besides from containing the standard information such as the block number and request size, also contains semantic information about the I/O. For example, semantic information of each I/O includes the block type, inode num-

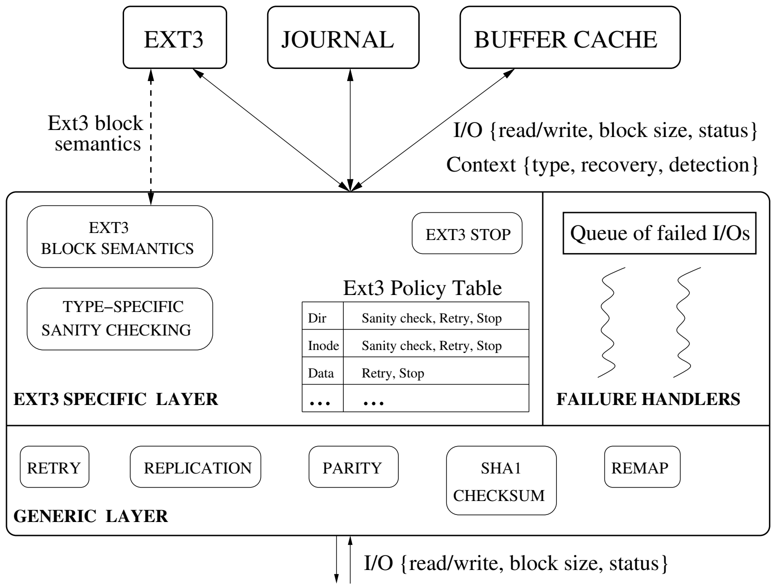


Figure 6.1: **The CFH Framework.** *The figure shows components of a centralized failure handler. It has three main subsystems: a file system specific layer, a generic layer, and threads for failure handling. The file system specific layer also maintains a failure policy table. The generic layer contains various mechanisms such as replication, parity, and checksums that can be commonly used by various file systems.*

ber, pointers to specific detection and recovery routines, and so on. Such information is crucial in enabling fine-grained and flexible policies at the CFH.

The CFH basic framework consists of three main components: a file system specific layer, a generic layer, and additional threads to handle failures. We describe each in turn.

File system specific layer: The file system specific layer receives requests that are issued from a particular file system or from other layers on behalf of the file system, and it understands the semantics of the I/O requests. For example, the ext3 specific component understands different block types of ext3.

A file system can use generic detection and recovery routines such as the checksum or retry module or file system specific block type sanity checking modules or its own special recovery routines. In our current prototype implementation, the policies are specified by the file system developer. However, extending the interface to other users such as system administrators is simple. When a file system mounts, it registers its specific detection and recovery routines. For example, ext3 could register a function pointer to `ext3_inode_sanity_check()` with the

CFH during its mount, which will be called every time an inode block is read or written by ext3. Note that using this framework, a file system can even perform sanity checking before its writes are issued to the disk, which can help find certain software-induced or memory-related corruptions before the data reaches the disk.

Generic layer: The CFH consists of a generic layer that implements several mechanisms such as retry, replication, parity, remap, and different types of checksums, all of which are available to the policy implementor. The functionality provided by the generic layer can be used commonly across multiple file systems and the generic mechanisms can be easily configured to suit the needs of the file system. For example, the number of retries can be easily set for ext3 and they can be different for reads, writes, and other block types. This design aspect of the CFH provides a clean separation of mechanisms from policy.

Failure handlers: CFH contains additional threads to handle I/O failures. An I/O completion is notified under interrupt context and if there is a failure, the recovery actions cannot be invoked under interrupt context because they are time critical and performing complex recovery actions under them might crash the entire system.

When an error is detected, the I/O block along with its context information is added to a queue and the failure handler thread is woken up. The failure handler thread removes the failed I/O and enforces the recovery policy specified by the file system. For example, if the file system wants to retry a failed I/O, the I/O is reissued. One can implement and apply any of the techniques under the IRON taxonomy (Tables 2.1 and 2.2) using the failure handlers.

Failure handling threads are also useful in monitoring asynchronous I/O calls. Typically, asynchronous I/O failures are not detected by a file system and no recovery action is taken. In CFH, the failure handlers capture all I/O failures including asynchronous ones and handle them appropriately.

The Policy Table

CFH maintains a policy table, which is initialized by the file system during its mount and can be modified while it is running. The policy table specifies the kind of detection and recovery actions to be taken for each block type failure. It provides a single and easily understandable location within which failure-handling policy is specified.

Table 6.1 shows the fields on each record in the policy table. One record is used for each block type, enabling the CFH to implement fine-grained policy per block type of the file system. We explain the policy table fields in detail. Block type information is specified by `Type` and for all blocks confirming to this type,

Fields	What they represent
Type	Block type information
Validate_IO	Whether to verify the block contents.
Retry_max	Maximum number of retries
Remap	Whether this block must be remapped
Isolate	Number of blocks to isolate around failed block
Inode	File or directory this block belongs to
(*Sanity)	Pointer to block type specific sanity checking module
(*Recover)	Pointer to recovery module
FS_Arg	Any file system specific argument (*e.g.*, in-core super block) passed to the recovery function

Table 6.1: **CFH Policy Table.** *The table presents various fields in a CFH policy table record. The fields can be configured separately for each I/O and offer great flexibility in supporting fine-grained failure policies.*

the policy specified by the record is applied. `Validate_IO` is a flag that specifies whether a block content must be validated are not, and if so, whether to use checksums or sanity checking for validation. Possible values for this field include DONT_VALIDATE, SANITY_CHECK, and CHECKSUM.

We implement certain generic recovery routines such as retrying failed requests and remapping writes to a different location. If a file system wants to use those recovery techniques, it can use the `Retry_max` to specify maximum number of times the failed I/O must be retried and `Remap` to point to the block to which a write failure must be remapped. Blocks around a failed block may have higher chances of failing and therefore, a file system can use `Isolate` to specify the number of blocks that must be remapped around a failed block. In addition to offering a per-block-type failure policy, one can even fine tune the policy to per file or directory. For example, one can apply stronger failure handling approaches to root directory than rest of the directories and this can be achieved by specifying the inode number of the file or directory in `Inode` field. `(*Sanity)` and `(*Recover)` are function pointers specific to a file system. For example, in inode failure policy record, a file system can set `(*Sanity)` to an inode sanity checking routine and `(*Recover)` to a file system stop function. Certain file system recovery actions such as stopping the entire file system require access to the in-core super block of the file system and the policy table record also provides a field (`FS_Arg`) to store it.

It must be noted that this is just one possible design for the policy table fields. While we have taken a first approach here, more complex designs are possible. For example, a file system might want to use its own remap routine in order to remap an entire logically contiguous unit. In general, a file system can either implement its own specialized routines for every IRON detection and recovery techniques, or use all the techniques from the CFH generic layer. To support such highly flexible designs, the policy table fields must be changed. However, we believe that those changes are not significant and can be implemented as an extension of our prototype design.

Acquiring Semantic Information

One of the design issues in building a CFH is understanding the semantics of each I/O. The file system specific component of a CFH must understand the context of an I/O such as the block type to invoke specific failure policies on I/O failure.

Understanding the I/O semantics is straightforward for reads and writes that are issued directly from the file system. We modified ext3 to pass the I/O context information along with an I/O call.

However, I/O calls can be issued from common file system layers such as the generic buffer cache manager. There are two issues in modifying the functions in buffer cache layer to use CFH. First, since the buffer cache is commonly used by other parts of the system, file systems that do not use CFH will be affected. We solve this problem by altering the buffer cache layer such that only if a file system uses CFH, its I/O calls are redirected. Second, since the I/Os are issued from a generic layer, the file system specific semantics is lost in these I/O calls. We handle this problem by adding calls in ext3 that pass the block type information to CFH for those I/Os that are issued via the generic layer.

When a file system is recovering after a crash, file system blocks that are written in the journal as part of a successfully committed transaction are read and written to their fixed locations. It is important that we know the type of the recovered blocks so that the failure policies can be applied even during recovery. We cannot configure the file system to pass the block type information during journal recovery because the file system itself is not active until the recovery completes. We solve this problem by storing the block type information along with the blocks in the journal, which can be read during recovery to apply the appropriate failure policy.

Function	Ext3 block type
`readdir`	Directory
`rmdir`	Directory, Data bitmap, Inode bitmap
`unlink`	Data bitmap, Inode bitmap, Indirect block
`truncate`	Data bitmap, Indirect block
`write`	Data bitmap, Indirect block

Table 6.2: **Fixing Error Propagation.** *The table presents a list of functions that fail to propagate error to the application on various block type failures in ext3. We fix all these functions to send an "I/O Error" code to the application upon failure.*

Implementation Details

We have implemented a centralized failure handler for Linux 2.6.9 kernel and modified Linux ext3 to use it. We also modify the generic buffer cache and journaling layer to issued disk reads and writes to CFH if the requests are issued on behalf of ext3. We call the resulting system as ext3$_C$.

We verified that all the I/O calls are redirected to the CFH by flagging I/O requests within the CFH and ensuring that all requests that are about to be passed to disk have such a flag. Although not comprehensive, this simple technique gave us confidence that the CFH interposed on all relevant I/Os.

In order to build a robust system, error codes must be propagated reliably across different layers of the system until the error is handled appropriately. CFH is designed such that when an I/O failure cannot be recovered, it propagates the failure to the file system and also logs error message in the system log. However, file systems ignore failures and often fail to propagate the error to the application [86]. From our analysis, we find that ext3 fails to propagate error to the application under several POSIX calls (for example, directory block failure in `rmdir` is not notified to an application). Table 6.2 lists a set of functions that ignore error propagation in ext3 and we fix all those functions in ext3$_C$.

6.1.2 Evaluation

We evaluate ext3$_C$ as follows. First, we show the flexibility and consistency of ext3$_C$ by making it mimic ReiserFS-like and NTFS-like failure policies. By changing a few lines of code in the policy table, ext3$_C$ acts entirely differently in response to a disk failure. Second, we evaluate how ext3$_C$ can be used to implement fine-grained policies for different block types. Specifically, we develop and evaluate a policy that acts in a more paranoid fashion about its own metadata, while simply propagating failures encountered when accessing user data. Finally, we evaluate the performance overhead of routing I/O requests via CFH by running several macrobenchmarks.

Flexibility and Consistency in ext3$_C$

Linux ext3$_C$ is highly flexible and can be easily configured to mimic different failure policies just by modifying a few lines in its configuration file. A configuration command in ext3$_C$ is as simple as:

```
ext3c_dir_block     = {
     .Type          = ext3c_DIR_BLOCK,
     .Validate_IO   = SANITY_CHECK,
     .Retry_max     = 0,
     .Sanity        = ext3c_dir_sanity_check,
     .Recover       = ext3c_stop,
};
```

The above code defines the failure policy for directory blocks in ext3$_C$. It informs CFH to perform sanity checking using `ext3c_dir_sanity_check` routine and if there is an I/O failure, asks CFH to stop the system by calling `ext3c_stop`.

The above configuration mimics ReiserFS-like failure policy. Previous work has shown that ReiserFS is paranoid about disk failures and therefore stops the file system on most block I/O errors [86]. We show the flexibility of CFH by changing its failure policy from ReiserFS to that of NTFS by altering only a few lines. NTFS assumes failures are more transient, and therefore performs persistent retries [45, 86]. We make ext3$_C$ behave like NTFS on directory failures by setting `Retry_max` to a non-zero value and `Recover` to an error propagating function,

We use a testing framework similar to that of our previous analysis (Section 5.2) to perform a thorough failure policy analysis of ext3$_C$. We run different workloads and fail specific file system block types. Tables in Figure 6.2 present ext3$_C$ failure policy under read failures. Each column represents one or a set of workloads and each row represents a data structure in ext3. If a data structure is read on a workload, then the corresponding entry gives the failure policy enforced by CFH. We can see that by just changing 2 lines for each of the 13 data structures, ext3$_C$ can be made to mimic a different failure policy for all block type failures. Although the results are not shown, we ensure that ext3$_C$ enforces similar policy on write failures too.

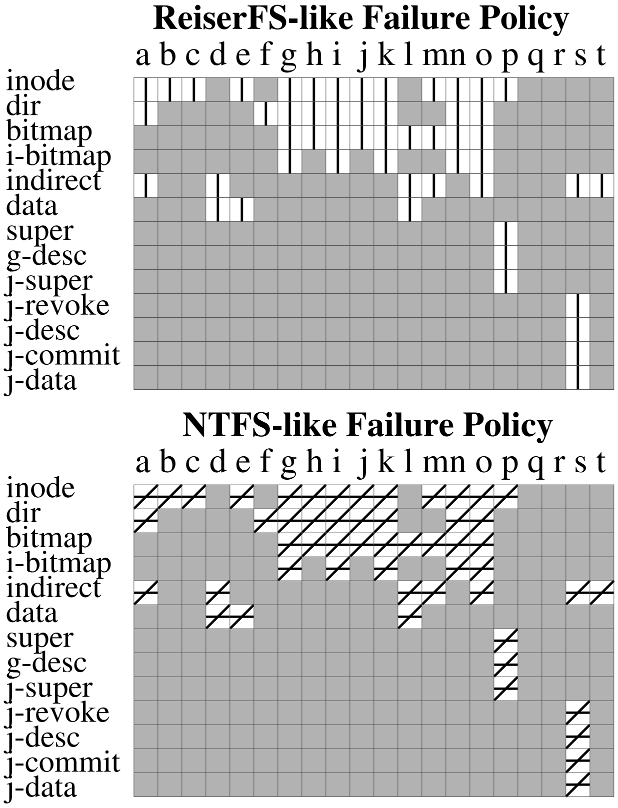


Figure 6.2: **Consistent Failure Policies in ext3$_c$.** *The table indicates recovery policies of ext3$_C$ for read faults injected for each block type across a range of workloads. The workloads are* **a:** *path traversal* **b:** *access,chdir,chroot,stat,statfs,lstat,open* **c:** *chmod,chown,utimes* **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:** *fysnc,sync* **r:** *umount* **s:** *FS recovery* **t:** *log write operations. A gray box indicates that the workload is not applicable for the block type. If multiple mechanisms are observed, the symbols are superimposed. Key for recovery: A "/", "−", and "|" represent retry, error propagation, and file system stop respectively.*

In addition to being flexible, ext3$_C$ can be highly consistent in applying its policies and this can be noticed from Figure 6.2. Two design aspects of CFH enables this consistent application of policies: first, we separate failure policies from mechanisms; second, all the failure policies are enforced at one central place. The result is a file system with a well-defined failure handling system.

Fine-grained Failure Policies

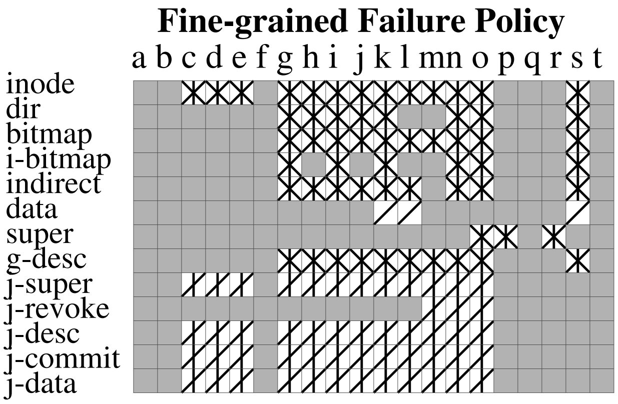


Figure 6.3: **Fine-grained Failure Policies in ext3$_C$.** *The table indicates fine-grained policies of cxt3 for write failures. The workloads and data structures are the same as in Figure 6.2. Key for recovery failure policy: A "/", "\", and "|" represent retry, remap, and file system stop respectively. Error is propagated on all the failures either through error code or by logging error messages.*

File systems can configure CFH to implement different policies for different block types depending upon their importance to the system. For example, a file system can be more paranoid about metadata block failures and handle them differently from data block failures. We show how CFH can be used to implement fine-grained policies by configuring ext3$_C$ with the following policy: file system metadata write errors are retried and remapped, and journal write failures are only retried. In both the above cases, ext3$_C$ stops the system when it cannot recover from the write error. When data write fails, ext3$_C$ only retries but does not remap or stop, but rather just propagates the error, enabling the application to take the appropriate action. We evaluate this failure policy for write failures of all block types and Figure 6.3 presents the failure policy table. From the table, we can see that under all

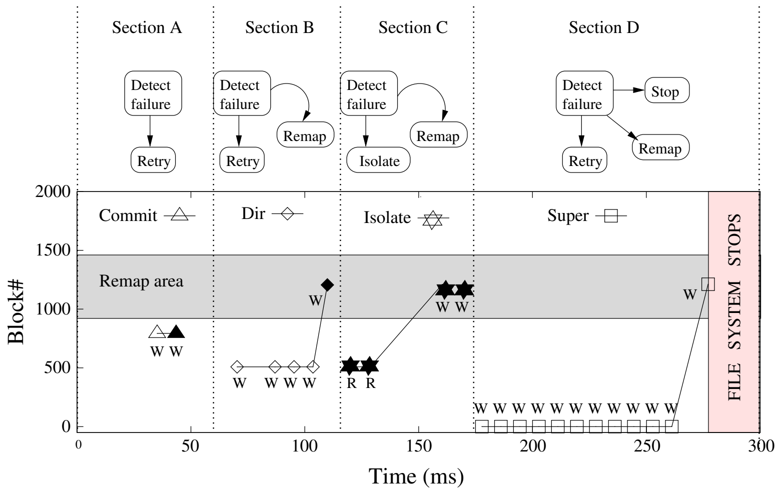


Figure 6.4: **Fine-grained Failure Handling.** *The above figure shows how different block type failures are handled differently by CFH. On X-axis, I/O completion time is plotted and on Y-axis, block numbers are shown. Failed I/Os are shown with legends that are not filled, while successful I/Os are shaded in black. Writes are marked with a "W" and reads with a "R". We show three different block type failures: a commit block write failure, which is a transient error, a directory block and a super block write errors, which are permanent. CFH retries the commit write failure and successfully recovers (Section A). CFH is configured to retry 3 times after directory write failure and when all the retries have failed, it remaps the directory block to a different location on disk (Section B). CFH also marks few blocks around the failed dir block for isolation, reads and remaps them (Section C). Finally, when the super block write fails, it is retried 10 times and remapped. When the remap also fails, CFH stops the file system (Section D).*

workloads, all metadata write failures are handled more thoroughly than data block failures.

Figure 6.4 shows a specific example of the above policy. When a commit block write fails in a transient manner, it is recovered by a retry. However, when a directory block write fails continuously, CFH remaps it to a new location. We implement an isolation policy in CFH that reads blocks around a failed block, writes them to a new location, and marks the region as unusable to prevent future errors. Finally, when the super block write fails, CFH first retries 10 times, then tries to recover by remapping, and finally stops the file system when the remap also fails. Even though we show a simple policy here, a file system can implement more fine-grained policies such as treating root directory I/O failures differently from other directory failures.

Performance Overhead

Benchmark	Ext3	ext3$_C$
SSH	118.45	118.56 (0.00%)
Web	98.85	99.00 (0.15%)
PostMark	150.50	151.25 (0.49%)
TPC-B	52.48	52.54 (0.11%)

Table 6.3: **Performance Overheads in CFH.** *The table lists the running time (in seconds) for four different benchmarks in ext3 and ext3*$_C$.

Finally, we evaluate ext3$_C$ performance to understand if additional processing caused by passing every I/O through CFH adds any performance overhead. We run four different file system benchmarks: SSH-Build, which unpacks and compiles the SSH source code, a Web server benchmark reads file in ext3$_C$ and sends over the network, the standard PostMark benchmark [61], which emulates a mail server workload, and a TPC-B-like benchmark, which runs debit-credit transactions [117]. The results are presented in Table 6.3. We run over 15 iterations for each of these benchmarks on ext3 and ext3$_C$ and find no noticeable difference in their performance.

6.1.3 Conclusion

We design and implement a centralized failure handler for file systems that forwards I/O requests and responses between a file system and disk. CFH implements generic and specific error handling routines that can be used by a file system to handle disk failures. By unifying all the failure handling at a single point and separating policies from mechanisms, CFH enables fine-grained and consistent failure handling in file systems.

While CFH offers greater flexibility in failure handling, the implementation efforts are non-trivial. Overall, we modified about 600 lines of existing code in ext3, the buffer cache, and the journaling layer and added about 3100 lines of code to build the CFH (including its generic detection and recovery mechanisms such as checksums and remap). Extending CFH to other file systems is less expensive once the basic CFH framework is in place. In fact, certain file systems such as

ReiserFS do not use the generic buffer cache and therefore, it is relatively easier to embed the block-type semantic information in their I/O calls. We also find that it is easier to maintain CFH even if individual file systems go through different design changes. Since CFH contains a clear separation between file system specific layer and generic layer, whenever a file system data structure is modified, routines that handle that data structure alone must change without affecting the rest of the failure handler parts.

Chapter 7

Related Work

Our effort builds upon related work from two bodies of literature. Our file system analysis (Section 4) is related to efforts that inject faults or otherwise test the robustness of systems to failure. Our work on building robust file systems including the prototype ixt3 (Section 5.2), probabilistic model for data layouts (Section 5.3.1), parity update techniques (Section 5.3.2), and centralized failure handler (Chapter 6) draw on efforts in building software that is more robust to hardware failure. We discuss each in turn.

7.1 Robustness Analysis

Several different approaches are taken to analyze and measure the robustness of a system. We discuss the related approaches here. First, faults can be injected and the resulting system behavior can be monitored. Second, more formal techniques such as model checking can be used to find out bugs in system code. Finally, failure characteristics of a system can be measured by stress testing or collecting failure data from production systems.

7.1.1 Fault Injection

Fault injection has been used for several decades as a mechanism to measure the robustness of systems [54]. We use software to simulate the effects of hardware failures and inject faults by dynamically determining the file system block types. Several previous work has used similar software-implemented fault injection (SWIFI) techniques to evaluate the dependability of computer systems.

Generic Fault Injection: FIAT (Fault Injection-based Automated Testing) 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 [17]. FIAT injects memory bit errors into a task image and the physical location of the fault is gathered from the compiler and loader information automatically. In our fault injection experiments, the driver automatically determines the fault location (*i.e.*, the block to fail) by inferring the block type information. FERRARI (Fault and ERRor Automatic Real-time Injector) is another tool that emulates permanent and transient hardware faults in software and uses software traps to inject faults [58]. Its four modules roughly correspond to the functionalities of coordinator and fault injection driver of our system. The initializer and activator module in FERRARI prepares the system for fault injection; the user information module obtains parameters such as fault and error types from the user; the fault injector injects faults during the program execution; and finally, data collection and analysis module collects logs of all the results. The coordinator and driver perform all the above activities in our system.

Other systems that use SWIFI approach include FTAPE (Fault Tolerance And Performance Evaluator), which is a tool that performs dynamic workload measurements and injects faults by automatically determining the time and location that will maximize fault propagation [118]. FTAPE uses stress-bases injection techniques to inject faults during high workload activities, which ensure higher fault propagation. FTAPE uses a framework similar to ours to inject disk system faults, where a driver is used to emulate I/O errors. An important difference is that in their system a fault is initiated by the workload stress, while in our framework a fault is initiated when a particular block type is written to disk. Fault injection experiments have also been used to study fault propagation in the system. FINE (Fault Injection and moNitoring Environment) 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 [70]. Although we do not explicitly trace the fault propagation in our system, the disk errors that we inject propagate from its origin through a generic device driver layer until it reaches the file system, where it can cause several behaviors including system crashes. Several work has targeted the kernel for reliability analysis. In a more recent work, fault injection techniques are used to test the Linux kernel behavior under the presence of errors in the kernel instruction code [49]. They test four kernel subsystems including the architecture dependent code, virtual file system, memory management, and the core kernel. Our testing focuses only on file systems and while they inject faults in instruction streams, we inject disk I/O failures.

File and Storage System Testing: Fault injection has been used specifically to study the reliability of storage systems as well. 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 [24]. 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 [55, 62]. These studies have developed detailed simulation models of RAID storage arrays and network clusters and used them to obtain the dependability measures. In contrast to their simulation-based fault injection, we perform a prototype-based fault injection for our failure analysis.

Several 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 [104]. Their benchmark consists of two parts, each exercising a different part in the file management system. The first part focuses on data structures in file management such as corrupting a file pointer or a file system flag. The second part measures the effectiveness of input error detection mechanisms of the functions in the C standard input/output library (STDIO). Similarly, Koopman *et al.* use the Ballista testing suite to find robustness problems in Safe/Fast IO (SFIO) library [31]. They test 36 functions in the SFIO API and show that although SFIO robustness is far greater than STDIO, it still had a fair number of robustness failures in functions like read and write. In contrast to both the above work, which test the robustness of file system API, we measure the robustness of file system to disk write failures.

One major difference between the related work in fault injection and ours is that our approach focuses on how file systems handle the broad class of modern disk failure modes; we know of no previous work that does so. Our approach also assumes implicit knowledge of file-system block types; by doing so, we ensure that we test many different paths of the file system code. Much of the previous work inserts faults in a "blind" fashion and hence is less likely to uncover the problems we have found.

7.1.2 Formal Methods

Static source code analysis is another popular approach to test the robustness of software systems. Model checking techniques can be applied to the file system code to find bugs and design flaws. Although model checking is more comprehensive than fault injection in its error detection, it requires more work in terms of modeling complex file system operations. In a recent work, Yang *et al.* use model checking comprehensively to find bugs in three different file systems: ext3, ReiserFS and JFS [129]. 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 faults.

More recently, Yang *et al.* develop a method to automatically find bugs in file system code that sanity checks on-disk data structure values [128]. They use a symbolic execution system, called EXE, which instruments a program and runs on symbolic input that is initially free to have any value. Based on the conditional expressions on data values, EXE generates constraints on the input values and generates cases for the "true" path and "false" path of the conditional expressions. This approach, as other formal methods, requires access to source code whereas we do not need source code for our analysis to work.

7.1.3 Other Techniques

Systems can be stress tested and monitored to understand their failure characteristics. Gray *et al.* measure the disk error rates in SATA drives by moving several PB of data [45]. They run programs in office-like and data-center-like setups that write and read data from large files and compare the checksum of the data looking for uncorrectable read errors. They measure about 30 uncorrectable bit errors as seen by the file system and 4 errors at the application level. They conclude that uncorrectable read errors is not a dominant system-fault source and suggest that "Mean Time To Data Loss" (MTTDL) would be a better metric.

Maintainers of large disk farms log and collect failure data from production systems. Some results are published after anonymizing the drive manufacturer information [34, 35, 103, 102]. For example, effect of firmware changes on failure mechanisms is studied over a family of drives and it is concluded that certain fixes applied to improve performance can increase the failure rates [103].

7.2 Building Robust File Systems

System researchers were aware of disk failures even several decades before and considered reliable file systems as one of important pieces of operating system design. To quote Needham *et al.* "*integrity of [file] system is both an important requirement and ... requires special treatment ... in dealing with errors*" [74]. In the following sections, we explain the related work in the realm of designing and building robust file systems.

7.2.1 IRON File Systems

Our work on IRON file systems was partially inspired by work within Google. Therein, Acharya suggests that when using cheap hardware, one should "be paranoid" and assume it will fail often and in unpredictable ways [1]. However, Google (perhaps with good reason) treats this as an application-level problem, and therefore builds checksumming on top of the file system; disk-level redundancy is kept across drives (on different machines) but not within a drive [40]. We extend this approach by incorporating such techniques into the file system, where all applications can benefit from them. Note that our techniques are complimentary to application-level approaches; for example, if a file system *metadata* block becomes inaccessible, user-level checksums and replicas do not enable recovery of the now-corrupted volume.

Another related approach is the "driver hardening" effort within Linux. As stated therein: "A 'hardened' driver extends beyond the realm of 'well-written' to include 'professional paranoia' features to detect hardware and software problems" (page 5) [57]. However, while such drivers would generally improve system reliability, we believe that most faults should be handled by the file system (*i.e.*, the end-to-end argument [94]).

The fail-partial failure model for disks is better understood by the high-end storage and high-availability systems communities. For example, Network Appliance introduced "Row-Diagonal" parity, which can tolerate two disk faults and can continue to operate, in order to ensure recovery despite the presence of latent sector faults [28]. Further, virtually all Network Appliance products use checksumming to detect block corruption [53]. Similarly, systems such as the Tandem NonStop kernel [16] include end-to-end checksums, to handle problems such as misdirected writes [16].

Interestingly, redundancy has been used *within* a single disk in a few instances. For example, FFS uses internal replication in a limited fashion, specifically by making copies of the superblock across different platters of the drive [71]. As we noted earlier, some commodity file systems have similar provisions.

Yu *et al.* suggest making replicas within a disk in a RAID array to reduce rotational latency [130]. Hence, although not the primary intention, such copies could be used for recovery. However, within a storage array, it would be difficult to apply said techniques in a selective manner (*e.g.*, for metadata). Yu *et al.*'s work also indicates that replication can be useful for improving *both* performance and fault-tolerance, something that future investigation of IRON strategies should consider.

Checksumming is also becoming more commonplace to improve system security. For example, both Patil *et al.* [79] and Stein *et al.* [110] suggest, implement, and evaluate methods for incorporating checksums into file systems. Both systems aim to make the corruption of file system data by an attacker more difficult.

Finally, the Dynamic File System from Sun is a good example of a file system that uses IRON techniques [126]. DFS uses checksums to detect block corruption and employs redundancy across multiple drives to ensure recoverability. In contrast, we emphasize the utility of replication within a drive, and suggest and evaluate techniques for implementing such redundancy. Further, we show how to embellish an existing commodity file system, whereas DFS is written from scratch, perhaps limiting its impact.

7.2.2 Disk Failure Modeling

Failure models for disks, specifically in the context of RAID storage arrays have been studied before. Gibson develops an analytical model of the reliability of redundant disk arrays [41]. He discusses four different models ranging from a simple one that considers independent disk failures, to more complex models that include spare disks and dependency among disk unit failures. The models are validated using software simulation. In similar work, Kari develops reliability models which includes both sector faults and disk unit faults. However, he treats sector failures as independent events and does not account for the spatial locality in sector errors. [59]. In comparison with the above work, our failure model is developed for a single disk and uses Markov Random Field to express the local characteristics of disk errors. While Markov Random Field is extensively used in the field of Computer Vision for interpreting spatially correlated features such as image pixels [68], we use it to express the locality on disk sector failures.

Reliability of long term digital archives such as photos, emails, and web site archives has received attention from the research community. Baker *et al.* model the reliability of long-term replicated storage systems [14]. They consider correlated failures that might occur due to spatial locality, assuming that the correlated failures are exponentially distributed. Their model is similar to our probabilistic model, where both model a system that tolerate at most one failure and consider spatial locality.

7.2.3 Parity-based Redundancy

Parity is widely used as a mechanism to store redundant information on disk and has been the topic of several research work. The seminal work on RAID storage arrays includes a discussion on how RAID-5 can be constructed using parity blocks [81]. While it can tolerate a single disk failure, more recent work have focused on using parity blocks to with stand double disk failures. For example, EVENODD [21] and RDP [28] are techniques that spread parity among multiple disks such that the data can be recovered after a double disk failure. In a more recent work, Denehy *et al.* develop techniques to handle the parity update consistency issues on software RAID systems [30]. They introduce a new mode of operation in journaling file system called declare mode that provides information about outstanding disk writes after crash. They augment the software RAID layer to improve the speed of resynchronization of parity with other blocks with the help of the file system. While all the previous work have focused on adding parity at a layer beneath the file system, our work is primarily built around implementing redundancy within the file system.

7.2.4 Centralized Failure Handling

System designers have considered the separation of policies from their mechanisms as a basic design principle for several decades. The classic work on the Hydra operating system built mechanisms into the kernel to perform operations like scheduling, paging, and protection, and let the user-level control them with different policies [66].

The Congestion Manager (CM) architecture [15] is a similar work that advocates centralized mechanism for implementing variety of policies. CM is motivated by problematic behavior exhibited by applications whose flows compete with each other for resources and do not share network information with each other. CM addresses these problems by inserting a module above IP which maintains network statistics across flows and orchestrates data transmissions with a new hybrid congestion control algorithm.

Chapter 8

Conclusions and Future Work

> "One therefore has the problem of being able to deal ... with ... bizarre occurrences such as a disc channel which *writes to the wrong place* on the disc without any error indication or, perhaps even worse, one which *says that it has written but in fact has not*."
>
> "Theory and Practice in Operating System Design"
> R.M.Needham and D.F.Hartley
> SOSP 1969

Commodity operating systems have grown to assume the presence of mostly reliable hardware. The result, in the case of file systems, is that most commodity file systems do not include the requisite machinery to handle the types of partial faults one can reasonably expect from modern disk drives. In this thesis, we develop and employ a new technique, *semantic failure analysis* (SFA), that uses *block type* information and *transaction semantics* to test and understand failure handling in journaling file systems. Then, we built robust versions of Linux ext3 that use various redundant information and a centralized failure handler to provide fine-grained and well-defined failure policies.

In this chapter, we summarize this dissertation by recapitulating our failure policy analysis and experiences in building IRON file systems (Section 8.1). We then list a set of lessons we learned from this dissertation (Section 8.2). Finally, we present the future directions where this thesis can possibly be extended (Section 8.3).

8.1 Summary

Storage stacks on modern computers present complex failure modes such as latent sector faults, data corruption, and transient errors. The first part of this thesis focuses on understanding the failure policies of file system when confronted with such faults. We choose to focus on local file systems due to their ubiquitous presence and new challenges they present.

Commodity file systems are large, complex pieces of software with intricate interactions with other parts of the system such as buffer cache manager, I/O schedulers, journaling layer, and low level drivers. Given this complexity, we need new techniques and approaches that can be used by system developers, and even users of file systems, to test and understand how commodity file systems handle disk failures. In this regard, we develop a new technique called Semantic Failure Analysis, which applies file system specific knowledge, such as block types and transaction semantics to low-level block I/O stream and fails specific file system I/O. In contrast to traditional fault injection, which fails blocks in a "blind" fashion, semantic failure analysis is fast and can be used to identify failure policies, bugs, and inconsistencies relatively easily.

We analyze five commodity journaling file systems: Linux ext3, ReiserFS, JFS, and XFS and Windows NTFS. We make the following observations. First, commodity file systems are built with the assumption that disk fails in a fail-stop manner and therefore, they store little or no redundant information on disk. As a result, when portions of a disk fail, file systems are not able to recover from them. Second, file systems exhibit illogical inconsistency in their failure handling policies. That is, even under similar disk failure scenarios, different detection and recovery actions are employed. We suspect that this may be the result of the assumptions made by different developers when writing different sections of the code. Finally, failure handling is hard in current file system architecture, and we need new frameworks for supporting well-defined failure policies.

In the second part of this thesis, we focus on improving the robustness of commodity file systems to disk failures. First, we explore the effectiveness and cost of different redundant information like checksum, replica, and parity. By applying redundancy to file system metadata and data, both individually and in various combinations, we study in breadth the cost in terms of time and space. Our results show that for typical desktop-like workloads, it is indeed feasible to use redundant information with small overheads.

Second, we focus in depth on two issues presented by single disk drives on low-end systems: spatially correlated faults and lack of NVRAM support. We develop a probabilistic model that a file system can use to construct redundancy sets such that the chances of multiple faults affecting blocks in the same set is low. We also develop two parity update techniques that integrate with journaling framework to update data and parity blocks consistently. One of the techniques, the no-overwrite approach, can provide atomicity in updating the redundant information and as a result, can handle latent sector faults even during journal recovery. Our evaluation of the techniques point out that the performance overhead of parity update techniques varies between less than 1% to 11% for typical desktop-type workloads and increases as high as 51% for synchronous, write-intensive benchmarks.

Finally, we rearchitect the file system with a Centralized Failure Handler (CFH) to handle all the failures in a consistent manner. The CFH receives all the I/O calls from a file system along with the I/O semantics such as the block type information. By separating policies from mechanisms and handling all I/O failures at one single point, CFH enables uniform, fine-grained, and flexible failure policies. We also show that CFH incurs no additional overhead in forwarding requests between a file system and disk.

To conclude, we believe it is time to reexamine how file systems handle failures. One excellent model is already available to us within the operating system kernel: the networking subsystem. Indeed, because network hardware has long been considered an unreliable hardware medium, the software stacks above them have been designed with well-defined policies to cope with common failure modes [82].

8.2 Lessons Learned

Next, we present a list of lessons we learned while working on this dissertation. We believe that they will be relevant even beyond the realm of this thesis.

- **Importance of semantic information:** Semantic information is crucial in building high performance and dependable systems [29, 107, 108]. In addition, semantic knowledge can be used to perform more informed analysis as well [84]. We realize this and use it for our failure policy analysis, where transactional semantics and block type information are used to unearth file system failure policies.

  However, it is harder to get the correct semantic information under certain cases, even if one has access to the source code. For example, while building the centralized failure handler, file system semantics are lost for all I/Os issued via the generic buffer cache manager and journaling layer. Although we modified ext3, the generic buffer cache layer, and the journaling layer to pass semantics of the file system, this might not be practically feasible as modifying a common layer like buffer cache that is used by several subsystems is generally discouraged. Under such circumstances, we believe that the techniques developed in the context of semantically-smart disk systems [10, 12, 105, 106, 108] to reverse engineer file system information can be immensely useful.

- **Failure as a first class citizen:** Traditionally, systems have been built with performance as the main focus and as a result, various policies have been developed for different applications, where each policy is aimed at maximizing the performance of a specific workload behavior.

  However, as systems grow in complexity, reliability is becoming a primary design goal [65] and we believe that failures must be treated as first class citizens in such systems. For example, different failure policies must be presented to an application and let it choose the policy that suits its needs. Our CFH is a first step towards achieving this goal, where we apply fine-grained policies to file system data and metadata.

- **Need for benchmarks that represent desktop workloads:** In this thesis, we focus on local file systems running on low-end systems. To estimate the cost of redundancy techniques on such systems, we used SSH-Build (Section 5.2.2) as a representative workload. However, there are no benchmarks built systematically to depict the applications running on a desktop or laptop setting. All the current file system benchmarks [23, 61, 73, 75, 76, 117] stress the file system I/O path and are targeted at emulating high end systems (*e.g.*, PostMark represents a mail server workload [61] and TPC-B represents a transactional workload on a database server [117]). A workload suit built to emulate desktop applications can be useful for commodity operating system developers.

- **Ways to describe policies:** Policies are harder to get right without the right framework. This is especially so for failure policies where there are hundreds of combinations that are possible among different block types, detection, and recovery techniques. In this thesis, we show using CFH that given a right interface, it is indeed possible to specify even intricate policies using simple commands.

8.3 Future Work

Next, we present a set of research directions to pursue in the future. All the work that we discuss here stem from our efforts in analyzing and building robust file systems.

8.3.1 Using Checksums to Relax Ordering Constraints

In order to maintain file system integrity, modern file systems implement certain ordering constraints when they update their on-disk structures. For example, write ordering is used by all the journaling modes in journaling file systems, file systems with soft updates [39], Write Anywhere File Layout (WAFL) [53], and log structured file systems [93].

Ordering the file system interactions with disk introduces certain problems. First, it causes performance overhead by introducing a wait between I/O calls. For example, journaling file systems wait for all the transaction writes to finish before writing the commit block to mark the transaction as complete, and this, in spite of being a sequential write on the journal, incurs additional rotational latency at the disk drive. Second, implementing the correct ordering among several different types of disk blocks with multiple, concurrent updates to it can be a nightmare [120].

We propose to use checksums in file systems to relax ordering constraints while providing the same consistency and integrity guarantees. We explore an application of this idea in this thesis, which we call transactional checksums (Section 5.2.1). We show that transactional checksums significantly improve file system performance, especially for workloads with small, synchronous writes.

In the future, we plan to examine how checksums can be used in other file systems such as the ones that use soft updates or WAFL. Also, as a limiting case study, we plan to apply the idea to a FFS like file system such as ext2 and make its crash recovery faster. However, the benefits do not come for free. Checksum computational cost can add some overhead, but with the ever increasing processing power and limited improvement in disk access speeds, we believe this is a profitable trade-off to make.

8.3.2 Understanding Spatial Locality in Disk Faults

One of the attributes of disk failures is that, under certain conditions like a particle scratch or thermal asperity, faults can be spatially correlated. However, currently we do not have field data on how such faults spread (*i.e.*, the distribution of bad blocks) on disk platters. Deriving such data from drives that are discarded as unusable can shed light into block faults and help higher layers improve their data layout patterns.

SCSI drives provide low-level commands to obtain the mapping details between a logical block to its physical location on disk (including details like cylinder, track, and sector). As a first step, one can reconstruct the layout of logical blocks on disk, which can show the bad blocks that are not used by a disk drive. Although this only brings out the bad blocks that are already identified by a disk drive, it can unearth certain internal policies of a disk firmware such as its bad block remapping algorithm, which can possibly be used by a higher layer to improve its layout.

8.3.3 Impact of Richer Interfaces on Reliability

As we pointed out in Section 2.2, storage stacks consist of several layers, each with its own failure modes, detection, and recovery techniques. However, in the current I/O architecture, not many details about an I/O failure are exposed to the end point (*e.g.*, the file system). The only information a file system receives is whether an I/O has succeeded or not. Exposing more information such as which layer caused the error, and what detection and recovery actions were taken by intermediate layers can improve failure handling at the end point as it may possibly use other redundant components (such as a different controller) to recover from the failure.

More information about the failure handling mechanisms of disk drives and new interfaces to control them can improve the overall robustness of the storage stack. For example, interfaces already exists to control the additional information stored on the 8 bytes of a 520 bytes sector. However, there are other low-level information that are not yet available. Since disk drives implement remapping internally, information such as the number and location of blocks reserved for remapping, list of already remapped disk blocks, algorithm used for remapping, and perhaps, an interface to control the block remapping can be useful for a file system to determine its data layout policies. Moreover, if disks can expose the set of detection and recovery actions taken while reading or writing data, the other layers on the storage stack can avoid re-executing similar actions. For example, if a disk notifies that it retried several times before declaring a block as inaccessible, the layer above the disk drive can avoid retrying the request.

Higher level semantic information at the other end of the storage stack (*i.e.*, the disk drives) can also improve reliability. For example, if a drive understands which physical disk blocks constitute a logical file system entity or which physical blocks are "alive", it can improve its internal reliability mechanisms such as bad block remapping. Semantically-smart disk systems [10, 12, 105, 106, 108] explored some uses of file system level knowledge at the disk drives, where they reverse engineered the higher-level semantics. However, with the advent of object-based storage systems [6] some of this information can be available through explicit interfaces.

8.3.4 Using Virtual Machines to Improve Storage Stack Reliability

In this thesis, we show that failure handling in commodity file systems is broken. While we modify file systems to improve their robustness, altering file systems may not be feasible in practice. In the future, we propose to explore the idea of using virtual machines to improve the reliability of the storage stack, wherein the virtual machine detects and recovers from various partial disk faults but presents a simple fail-stop (virtual) storage system to commodity file system. While the same can be achieved with a pseudo-device driver instead of a more powerful virtual machine, we can use the potential of a virtual machine to alter certain file system behavior. For example, file systems like ReiserFS crash the entire system on even simple read errors by calling a kernel-level `panic`. Virtual machines can possibly be used to capture such calls and turn them into less expensive stops such as a read-only mount.

8.3.5 Analyzing Robustness of Other Data Management Systems

While we limit the focus of this thesis to local file systems, several of the issues we discuss here such as the latent sector faults, block corruption, spatial locality in sector faults, and the impact of redundancy techniques are applicable to other data management systems such as distributed file systems and database management systems. These systems are more complex than local file system with many more layers on their storage stack (like the presence of network in distributed systems). Therefore, the first step is to study new partial failure scenarios, if any, introduced in this context. Second, several of these systems can support redundant components such as multiple disks either within a single site or across multiple sites. This raises the issue of finding trade-offs among redundancy within a single disk, among multiple disks in a same site, and finally, across distributed disks. Third, optimizations such as transactional checksums are directly applicable to databases and in fact, one can even explore the applicability of relaxing ordering constraints using checksums in distributed protocols.

Bibliography

[1] Anurag Acharya. Reliability on the Cheap: How I Learned to Stop Worrying and Love Cheap PCs. EASY Workshop '02, October 2002.

[2] Anton Altaparmakov. The Linux-NTFS Project. http://linux-ntfs.sourceforge.net/ntfs, August 2005.

[3] Guillermo A. Alvarez, Walter A. Burkhard, and Flaviu Cristian. Tolerating Multiple Failures in RAID Architectures with Optimal Storage and Uniform Declustering. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, pages 62–72, Denver, Colorado, May 1997.

[4] Dave Anderson. You Don't Know Jack about Disks. ACM Queue, June 2003.

[5] Dave Anderson. "Drive manufacturers typically don't talk about disk failures". Personal Communication from Dave Anderson of Seagate, 2005.

[6] Dave Anderson. OSD Drives. www.snia.org/events/past/developer2005/0507_v1_DBA_SNIA_OSD.pdf, 2005.

[7] Dave Anderson, Jim Dykes, and Erik Riedel. More Than an Interface: SCSI vs. ATA. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[8] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.

[9] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium*

on Operating Systems Principles (SOSP '01), pages 43–56, Banff, Canada, October 2001.

[10] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, and Muthian Sivathanu. Semantically-Smart Disk Systems: Past, Present, and Future. *Sigmetrics Performance Evaluation Review (PER)*, 33(4):29–35, March 2006.

[11] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 33–38, Schloss Elmau, Germany, May 2001.

[12] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, pages 176–187, Munich, Germany, June 2004.

[13] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[14] Mary Baker, Mehul Shah, David S. H. Rosenthal, Mema Roussopoulos, Petros Maniatis, TJ Giuli, and Prashanth Bungale. A fresh look at the reliability of long-term digital storage. In *Proceedings of the 2006 EuroSys Conference*, Leuven, Belgium, April 2006.

[15] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proceedings of SIGCOMM '99*, pages 175–187, Cambridge, Massachusetts, August 1999.

[16] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[17] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Transactions on Computers*, 39(4):1105–1118, April 1990.

[18] Steve Best. JFS Log. How the Journaled File System performs logging. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 163–168, Atlanta, 2000.

[19] Steve Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

[20] Dina Bitton and Jim Gray. Disk shadowing. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB 14)*, pages 331–338, Los Angeles, California, August 1988.

[21] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: an optimal scheme for tolerating double disk failures in RAID architectures. In *proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994.

[22] IEEE Standards Board. 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology, September 1990.

[23] Tim Bray. The Bonnie File System Benchmark. http://www.textuality.com/bonnie/.

[24] Aaron Brown and David 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.

[25] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 31–44, San Francisco, California, December 2004.

[26] 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*, Amsterdam, Holland, 1994.

[27] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[28] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha 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.

[29] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, Monterey, California, June 2002.

[30] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 87–100, San Francisco, California, December 2005.

[31] John DeVale and Philip Koopman. Performance Evaluation of Exception Handling in I/O Libraries. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2001)*, Goteborg, Sweden, June 2001.

[32] John R. Douceur and Willian J. Bolosky. A Large-Scale Study of File-System Contents. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, pages 59–69, Atlanta, Georgia, May 1999.

[33] James Dykes. "A modern disk has roughly 400,000 lines of code". Personal Communication from James Dykes of Seagate, August 2005.

[34] Jon G. Elerath. Specifying Reliability in the Disk Drive Industry: No More MTBF's. In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS)*, pages 194–199, January 2000.

[35] Jon G. Elerath and Sandeep Shah. Server Class Disk Drives: How Reliable Are They? In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS)*, pages 151–156, 2004.

[36] EMC. EMC Centera: Content Addressed Storage System. http://www.emc.com/, 2004.

[37] Ralph Waldo Emerson. Essays and English Traits – IV: Self-Reliance. The Harvard classics, edited by Charles W. Eliot. New York: P.F. Collier and Son, 1909-14, Volume 5, 1841. *A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines.*

[38] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[39] Gregory R. Ganger, Marshall Kirk McKusick, Craig A.N. Soules, and Yale N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems*, 18(2):127–153, May 2000.

[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing (Lake George), New York, October 2003.

[41] Garth A. Gibson. *Redundant disk arrays: reliable, parallel secondary storage*. PhD thesis, University of California at Berkeley, 1991.

[42] Garth A. Gibson, David Rochberg, Jim Zelenka, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, and Erik Riedel. File server scaling with network-attached secure disks. In *Proceedings of the 1997 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '97)*, pages 272–284, Seattle, Washington, June 1997.

[43] Jason Goldberg. "Spatially local failures can be caused by thermal asperity". Personal Communication from Jason Goldberg of Seagate Research, February 2006.

[44] Jim Gray. A Census of Tandem System Availability Between 1985 and 1990. Technical Report 90.1, Tandem Computers, 1990.

[45] Jim Gray and Catharine Van Ingen. Empirical measurements of disk failure rates and error rates. Microsoft Technical Report, December 2005.

[46] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[47] Roedy Green. EIDE Controller Flaws Version 24. http://mindprod.com/jgloss/eideflaw.html, February 2005.

[48] Edward Grochowski. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. *Datatech*, September 1999.

[49] Weining Gu, Z. Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux Kernel Behavior Under Error. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2003)*, pages 459–468, San Francisco, California, June 2003.

[50] Haryadi S. Gunawi, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiri Schindler. Deconstructing Commodity Storage Clusters. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pages 60–73, Madison, Wisconsin, June 2005.

[51] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[52] Val Henson. A Brief History of UNIX File Systems. http://infohost.nmt.edu/~val/fs_slides.pdf, 2004.

[53] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[54] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[55] Yiqing Huang, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Dependability Analysis of a Cache-Based RAID System via Fast Distributed Simulation. In *The 17th IEEE Symposium on Reliable Distributed Systems*, pages 254 – 260, 1998.

[56] Gordon F. Hughes and Joseph F. Murray. Reliability and Security of RAID Storage Systems and D2D Archives Using SATA Disk Drives. *ACM Transactions on Storage*, 1(1):95–107, February 2005.

[57] Intel Corp. and IBM Corp. Device Driver Hardening. http://hardeneddrivers.sourceforge.net/, 2002.

[58] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A tool for the validation of system dependability properties. In *The 22nd International Symposium on Fault Tolerant Computing (FTCS-22)*, pages 336–344, Boston, Massachusetts, 1992.

[59] Hannu H. Kari. *Latent Sector Faults and Reliability of Disk Arrays*. PhD thesis, Helsinki University of Technology, September 1997.

[60] Hannu H. Kari, H. Saikkonen, and F. Lombardi. Detection of Defective Media in Disks. In *The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 49–55, Venice, Italy, October 1993.

[61] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[62] Mohamed Kaniche, L. Romano, Zbigniew Kalbarczyk, Ravishankar 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*, pages 6 – 15, June 1998.

[63] Steve R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '86)*, pages 238–247, Atlanta, Georgia, June 1986.

[64] John Kubiatowicz, David Bindel, Patrick Eaton, Yan Chen, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Westley Weimer, Chris Wells, Hakim Weatherspoon, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 190–201, Cambridge, Massachusetts, November 2000.

[65] James Larus. The Singularity Operating System. Seminar given at the University of Wisconsin, Madison, 2005.

[66] R. Levin, E. Cohen, W. Corwin, F. J. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP '75)*, pages 132–140, University of Texas at Austin, November 1975.

[67] Blake Lewis. Smart Filers and Dumb Disks. NSIC OSD Working Group Meeting, April 1999.

[68] Stan Li. Markov Random Fields and Gibbs Distributions. In *Markov Random Field Modeling in Image Analysis*, chapter 1. Springer-Verlag, 2001.

[69] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical Loss-Resilient Codes. In *Proceedings of the Twenty-ninth Annual ACM symposium on Theory of Computing (STOC '97)*, pages 150–159, El Paso, Texas, May 1997.

[70] Wei lun Kao, Ravishankar K. Iyer, and Dong 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.

[71] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[72] 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.

[73] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[74] R. M. Needham and D. F. Hartley. Theory and practice in operating system design. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 8–12, Princeton, New Jersey, 1969.

[75] William Norcutt. The IOzone Filesystem Benchmark. http://www.iozone.org/.

[76] John K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, CA, June 1990.

[77] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2

BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, Washington, December 1985.

[78] Arvin Park and K. Balasubramanian. Providing fault tolerance in parallel secondary storage systems. Technical Report CS-TR-057-86, Department of Computer Science, Princeton University, November 1986.

[79] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok. I^3FS: An In-kernel Integrity Checker and Intrusion detection File System. In *Proceedings of the 18th Annual Large Installation System Administration Conference (LISA '04)*, Atlanta, Georgia, November 2004.

[80] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report CSD-02-1175, U.C. Berkeley, March 2002.

[81] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[82] Jon Postel. RFC 793: Transmission Control Protocol, September 1981. Available from `ftp://ftp.rfc-editor.org/in-notes/rfc793.txt` as of August, 2003.

[83] Vijayan Prabhakaran. File System Fingerprinting for Semantically-Smart Disk Systems. Master's thesis, University of Wisconsin, Madison, 2003.

[84] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[85] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2005)*, pages 802–811, Yokohama, Japan, June 2005.

[86] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[87] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[88] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C.Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[89] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[90] Peter M. Ridge and Gary Field. *The Book of SCSI 2/E*. No Starch, June 2000.

[91] 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 *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[92] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 41–54, San Diego, California, June 2000.

[93] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[94] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[95] Russel Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, pages 119–130, Berkeley, CA, June 1985.

[96] Stefan Savage and John Wilkes. AFRAID — A Frequently Redundant Array of Independent Disks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, pages 27–39, San Diego, California, January 1996.

[97] Jiri Schindler. "We have experienced a severe performance degradation that was identified as a problem with disk firmware. The disk drives had to be reprogrammed to fix the problem". Personal Communication from J. Schindler of EMC, July 2005.

[98] Steven W. Schlosser and Gregory R. Ganger. MEMS-based storage devices and standard disk interfaces: A square peg in a round hole? In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 87–100, San Francisco, California, April 2004.

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

[100] Thomas J.E. Schwarz, Qin Xin, Ethan L. Miller, Darrell D.E. Long, Andy Hospodor, and Spencer Ng. Disk Scrubbing in Large Archival Storage Systems. In *Proceedings of the 12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Volendam, Netherlands, October 2004.

[101] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '93)*, pages 307–326, San Diego, California, January 1993.

[102] Sandeep Shah and Jon G. Elerath. Disk Drive Vintage and Its Effect on Reliability. In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS)*, pages 163–167, 2000.

[103] Sandeep Shah and Jon G. Elerath. Reliability analysis of disk drive failure mechanisms. In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS)*, pages 226–231, January 2005.

[104] D.P. Siewiorek, J.J. Hudak, B.H. Suh, and Z.Z. Segal. Development of a Benchmark to Measure System Robustness. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, June 1993.

[105] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.

[106] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 15–30, San Francisco, California, April 2004.

[107] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. *ACM Transactions on Storage (TOS)*, 1(2):133–170, May 2005.

[108] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, California, April 2003.

[109] David A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.

[110] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

[111] Lex Stein. Stupid File Systems Are Better. In *The Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, Sante Fe, New Mexico, June 2005.

[112] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[113] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[114] Nisha Talagala, Remzi H. Arpaci-Dusseau, and Dave Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[115] Nisha Talagala and David 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.

[116] The Data Clinic. Hard Disk Failure. http://www.dataclinic.co.uk/hard-disk-failures.htm, 2004.

[117] Transaction Processing Council. TPC Benchmark B Standard Specification, Revision 3.2. Technical Report, 1990.

[118] T. K. Tsai and R. K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *The 8th International Conference On Modeling Techniques and Tools for Computer Performance Evaluation*, pages 26–40, September 1995.

[119] Theodore Ts'o. Email discussion on file system robustness against cheap hardware failure under the subject "reiser4 plugins". http://www.ussg.iu.edu/hypermail/linux/kernel/0506.3/0786.html.

[120] Theodore Ts'o. Email discussion on soft updates with the subject "soft update vs journaling?". http://lkml.org/lkml/2006/1/22/27.

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

[122] Stephen C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/ release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[123] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, December 1999.

[124] Larry Y. Wang, Mike Sullivan, and Jim Chao. Thermal asperities sensitivity to particles: Methodology and test results. *Journal of Tribology*, 123(2):376–379, April 2001.

[125] John Wehman and Peter den Haan. The Enhanced IDE/Fast-ATA FAQ. http://thef-nym.sci.kun.nl/cgi-pieterh/atazip/atafq.html, 1998.

[126] Glenn Weinberg. The Solaris Dynamic File System. http://members.visi.net/~thedave/sun/DynFS.pdf, 2004.

[127] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[128] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *IEEE Security and Privacy*, Berkeley, California, May 2006.

[129] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal 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.

[130] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.

[131] S. Zhou, H. Da Costa, and A.J. Smith. A File System Tracing Package for Berkeley UNIX. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '84)*, pages 407–419, Salt Lake City, Utah, June 1984.