IRON File Systems

Vijayan Prabhakaran, Nitin Agrawal, Lakshmi Bairavasundaram, Haryadi Gunawi
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison

Abstract

Commodity file systems trust disks to either work or fail completely, yet modern disks exhibit more complex failure modes. We suggest a new *fractured failure model* for disks, which incorporates realistic localized faults such as latent sector errors and block corruption. We then develop and apply a novel fault-injection framework, to investigate how commodity file systems react to a range of more realistic disk failures. We classify their failure policies in a new taxonomy that measures their *Internal RObustNess (IRON)*, which includes both failure detection and recovery techniques. We show that commodity file system failure policies are often inconsistent, sometimes buggy, and generally inadequate in their ability to recover from localized disk failures. Finally, we design, implement, and evaluate a prototype IRON file system, ixt3, showing that techniques such as in-disk checksumming and replication greatly enhance file system robustness while incurring minimal time and space overheads.

1 Introduction

Disks fail – but not in the way most commodity file systems expect. For many years, file system and storage system designers have assumed that disks operate in a "fail stop" manner [49]; within this classic model, the disks either are working perfectly, or fail absolutely and in an easily detectable manner.

The fault model presented by modern disk drives, however, is much more complex. For example, modern drives can exhibit *latent sector faults* [28, 13, 50], where a block or set of blocks is inaccessible. Worse, blocks sometimes become *silently corrupted* [5, 21, 63]. Finally, disks sometimes exhibit *transient* performance problems [57].

There are many reasons for these complex failures in disks. For example, a buggy disk controller could issue a "misdirected" write [63], placing the correct data on disk but in the wrong location. Interestingly, while these failures exist in disks today, simply "waiting" for disk technology to improve will not remove these errors: indeed, these errors may *worsen* over time, due to the increasing complexity [4], immense cost pressures in the drive industry, and the escalated use of less reliable ATA drives, not only in desktop PCs but also in large-scale clusters [19] and storage systems [17].

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 *disk scrubbing* [27, 50] to proactively detect and subsequently correct latent sector errors by creating a new copy of inaccessible blocks; some recent storage arrays incorporate extra levels of redundancy to lessen the potential damage of undiscovered latent errors [13]. Similarly, highly-reliable systems (*e.g.*, Tandem NonStop) have long utilized end-to-end checksums to detect when block corruption occurs [5].

Unfortunately, such technology has not filtered down to the realm of commodity file systems, including Linux file systems such as ext3 [61], ReiserFS [42], and IBM's JFS [7], or Windows file systems such as NTFS [53]. Such file systems 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 in many internet services such as Google [19].

In this paper, the first question we pose is: *how do modern commodity file systems react to failures that are common in modern disks?* To answer this query, we aggregate knowledge from the research literature, industry, and field experience to form a new model for disk failure. We label our model the *fractured failure model (FFM)* to emphasize that *pieces* of the disk can fail.

With the model in place, we develop and apply an automated framework to inject more realistic disk faults beneath a file system. Our goal is to unearth the *failure policy* of each system: how it detects and recovers from disk failures. To better characterize failure policy, we develop an *Internal RObustNess (IRON)* taxonomy, which catalogs a broad range of detection and recovery techniques.

Our study focuses on three important and substantially different open-source file systems, ext3, ReiserFS, and IBM's JFS, and one commercial file system, Windows NTFS. Across all platforms, we find a great deal of *inconsistency* in failure policy, often due to the diffusion of failure handling code through the kernel; such inconsistency leads to substantially different detection and re-

covery strategies under similar fault scenarios. We also find 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. Finally, we show that none of the file systems can recover from localized disk failures, due to a lack of *in-disk redundancy*.

This behavior under realistic disk failures leads us to our second question: *how can we change file systems to better handle modern disk failures?* We advocate a single guiding principle for the design of file systems: *don't trust the disk*. 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 age-old end-to-end argument [46]: at the top of the storage stack, the file system is fundamentally responsible for reliable management of its data and meta-data.

In our initial efforts, we focus on one specific "sweet spot" in the IRON taxonomy: an IRON version of ext3 (ixt3) that uses redundancy within a single disk for its meta-data structures. We show that ixt3 incurs little overhead while greatly increasing the robustness of modern file systems to latent sector errors and corruption. By implementing detection and recovery techniques from the IRON taxonomy, a system can implement a well-defined failure policy and subsequently provide increased levels of protection against the broader range of disk failures.

The rest of this paper is structured as follows. First, we present a detailed examination of how disks fail and the fractured failure model (§2). Then, we discuss detection and recovery techniques within our IRON taxonomy (§3), present our fault-injection method (§4), and our analysis of failure policy under such faults (§5). We then propose, implement, and evaluate IRON ext3 (§6), discuss related work (§7), and conclude (§8).

2 Disk Failure

There are many reasons that the file system may see errors in the storage system below. In this section, we first discuss common causes of disk failure. We then present a new, more realistic *fractured failure model* for disks and discuss various aspects of this model.

2.1 The Storage Subsystem

Figure 1 presents a typical layered storage subsystem below the file system. 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 drive itself; beyond the magnetic storage media, there are mechanical (*e.g.*, the motor and arm assembly) and electrical components (*e.g.*, power, buses, and so forth). A particularly important component of the drive is its firmware – the

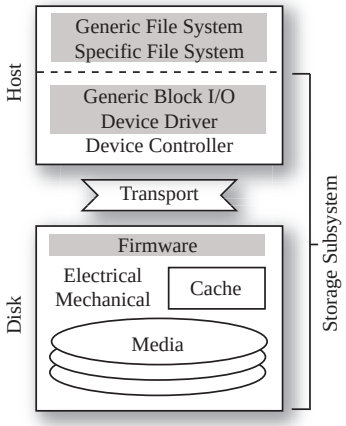

Figure 1: **The Storage Stack.** *We present a schematic of the entire storage stack. At the top is the file system; beneath are the many layers of the "storage subsystem". Gray shading implies a software or firmware component, whereas white (unshaded) is hardware.*

code embedded within the drive to control most of its higher-level functions, including caching, disk scheduling, and error handling. This firmware code is substantial and complex, *e.g.*, a modern drive from Seagate contains roughly 400,000 lines of low-level code [16].

Connecting the drive to the main host is the transport layer. In low-end systems, the transport medium is often a bus (*e.g.*, IDE/ATA or SCSI), whereas networks are common in higher-end systems (*e.g.*, FibreChannel).

At the top of this "storage stack" is the host. At a low level in the host is hardware: a device controller that is used to communicate with the device. Above this layer is software: a low-level device driver to communicate with the hardware. Block-level software is layered on top of this, to provide a generic interface to clients and implement various optimizations (*e.g.*, request reordering).

Note that the file system that operates above the storage subsystem is often split into two pieces: a generic component that is common to all file systems, and a specific component that maps generic operations onto the data structures of the particular file system. A generic interface (*e.g.*, Vnode/VFS [30]) is positioned between the two.

2.2 Why Do Disks Fail?

To motivate our failure model, we first describe how errors in the layers of the storage stack can cause failures.

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 [50]. 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 even sometimes include various filters to remove particles [4]. Media errors most often lead to permanent failure or corruption of individual disk blocks.

Mechanical: "Wear and tear" eventually leads to the failure of moving parts. The drive motor can spin irregularly or fail completely. Erratic arm movements can lead to head crashes and media flaws. Inaccurate arm movement can position the disk poorly upon a write, making blocks inaccessible or corrupted for subsequent reads.

Electrical: A power spike or surge can damage in-drive circuits and hence lead to drive failure [58]. 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 [31] or have "memory leaks" that lead to errors [58]. 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 [63]; phantom writes can be caused by a buggy or even misconfigured cache (*i.e.*, write-back caching is enabled). In summary, errors in drive firmware often lead to sticky or transient block corruption.

Transport: The transport connecting the drive and host can also be problematic. For example, a study of a large disk farm [57] 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 [62]. 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 [21]. Others have also observed IDE protocol version problems that yield corrupt data [19]. Thus, problems with the bus controller 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 [12, 18, 56]. While some of these bugs will likely crash the operating systems, others can issue disk requests with bad parameters, data, or both.

2.3 The Fractured 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 in the storage subsystem 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 errors" [28, 27].

• **Block corruption:** The data within individual blocks is altered. This failure is perhaps the most insidious error manifestation because it is silent – the storage subsystem simply returns "bad" data upon a read.

We term this model the *fractured failure model (FFM)*, to emphasize that pieces of the storage subsystem can fail. We now discuss some other key elements of FFM, including the transience, locality, and frequency of failures.

2.3.1 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 likely indicates that subsequent requests will continue to fail. In contrast, a transport or higher-level software issue could at first cause a block failure or corruption; however, the operation could subsequently succeed if retried.

2.3.2 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 indicate that some forms of block failure do indeed exhibit spatial locality [28]. For example, a scratched surface 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.

2.3.3 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" [23]. 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 [57], 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 [3]. 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 errors indicate that such errors occur more

commonly than absolute disk failure [28], and more recent research estimates that such errors may occur five times more often than absolute disk failures [50].

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 [25]. 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. 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.

2.3.4 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.

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 [4]. 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 [4]. Low-cost "PC class" drives tend to be tested less and have less internal machinery to prevent failures from occurring [25]. The result, in the field, is that ATA drives are observably less reliable [57]; however, cost pressures serve to increase their usage, even in server environments [19]. Finally, the amount of software is increasing in storage systems and, as others have noted, software is often the root cause of errors [20]. 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 [18, 56] – hence a likely source of increased errors in the storage stack.

3 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) fractured faults and *recover* (Level R) from them. Tables 1 and 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 [2, 10, 47, 64].

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 1: **The Levels of the IRON Detection Taxonomy.**

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. We believe that with the IRON taxonomy, one will be able to describe the failure policy of a file system, much as one can already describe a cache replacement or a file layout policy.

3.1 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.

• $ErrorCode$: The most basic detection that an IRON file system can perform is to check the return codes provided by the disk and lower levels of the storage system. Given that disks currently only report that a block is unavailable, this level cannot detect corruption.

• $Sanity$: With sanity checks, the file system verifies that its on-disk 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 [41]. 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 [35]). Even journaling file systems can benefit from periodic integrity checks; journaling does not prevent a buggy system from corrupting on-disk structures.

• $Redundancy$: 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 [5] and has recently been applied to improve security as well [37, 54]. Checksums are useful for a number of reasons. First, they assist in detecting classic "bit

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

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 [5, 63]; 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 [8], parity [36, 39] and other error-correction codes [32], 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 [39].

3.2 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 an IRON file system should always contain some form of lazy detection and should consider additional eager methods.

For example, *disk scrubbing* is a classic eager technique used by RAID systems to scan a disk and thereby discover latent sector errors [28]. 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.

3.3 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 errors and block corruptions.

• $Propagate$: The simplest recovery mechanism is to propagate errors up through the file system; the file system informs the application that an error occurred and assumes the 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 .login file), 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.* [44], 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. 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. 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, the journal, and so forth), 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 [39], 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 [32]) could also be used, forming a rich space for 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 [15].

3.4 Why The File System?

One natural question to ask is: why should the file system implement detection and recovery instead of the disk? After all, modern disks do contain a number of mechanisms for detecting and recovering from errors.

In our view, the primary reason for detection and recovery within the file system is found in the classic end-to-end argument [46]; 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 meta-data, perhaps leaving failure detection and correction of user data to applications (indeed, this is a specific solution that we explore). 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" [48] 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 remaps blocks, it can move an entire logically-related unit (*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 [4]; no interface exists to control these different low-level strategies in current systems.

Finally, it should be noted that while most of this discussion focuses on the case of a single disk, IRON techniques may be useful even when multiple drives are used in a RAID-like manner. Specifically, in-disk parity or replication can be another useful technique that storage arrays employ to combat the increasingly common sector fault and corruption errors they encounter.

4 Failure Policy: Methodology

We now describe our methodology to uncover the *failure policy* of file systems. Our main objective 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 basic approach is to inject faults just beneath the file system and then observe how the file system reacts to those faults. If the fault policy is entirely consistent within a file system, then this can be done very simply; for example, we can run any workload, fail one of the blocks that is read or written, and then conclude that the way the file system reacts to this block failure is the way it reacts to all block failures. However, the file system might have different failure policies depending upon the operation performed and the type of the faulty block.

Therefore, to explore the complete failure policy of a file system, we must trigger all interesting cases. Our challenge is to coerce the file system down its different internal code paths in order to observe how each path handles failure. This requires that we run workloads exercising all relevant code paths in combination with faults on all blocks containing different types of file system meta-data. We now describe how we create workloads, inject faults, and infer the reaction of the file system.

4.1 Applied Workload

Our goal when applying workloads is to exercise the file system as thoroughly as possible. Although we do not claim to stress every code path, we do strive to execute many of the interesting internal cases.

Our workload suite contains some general programs that are common for all file systems. For example, the suite contains a separate program for each function of the file system API (*e.g.*, mkdir, truncate, *etc.*), a set of programs to stress journal recovery, and programs to exercise common functionality such as path traversal.

In addition to the general functionality, each file system also has a number of special cases that must be stressed. For example, the Linux ext3 inode uses a standard im-

balanced tree with indirect, doubly-indirect, and triply-indirect pointers to track file blocks; hence, our workload ensures that sufficiently large files are created to handle these different cases. Other file systems have similar peculiarities that must be exercised (*e.g.*, the B+-tree balancing code of ReiserFS).

4.2 Type-Aware Fault Injection

Our second step is to inject faults that emulate a disk adhering to the fractured failure model. Many standard fault injectors [9, 52] 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 on the failure handling policy.

The key idea that allows us to test a file system in a relatively efficient and thorough manner is *type-aware fault injection*, in which a block of a specific type (*e.g.*, inode, directory data, or indirect block) is explicitly failed. Type information is crucial in reverse-engineering the failure policy, allowing us to discern the different strategies that a file system applies for its different data structures. The disadvantage of our type-aware approach is that the fault injector must be tailored to each file system tested and requires a solid understanding of the on-disk structures. However, we believe that the benefits of type-awareness clearly outweigh these complexities.

Our mechanism for injecting faults is to use a software layer directly beneath the file system (*i.e.*, a pseudo-device driver). This software layer injects both block failures and block corruption. To emulate a block failure, we simply return the appropriate error code and do not issue the operation to the underlying disk. To emulate block 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 some number of corrupted fields. The software layer also models both transient and sticky faults and operates on either read or write operations.

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 [9], we can capture faults introduced by buggy device drivers and controllers. A drawback of our approach is that it does not model how lower-layers of the system handle disk faults; for example, some SCSI drivers retry certain commands after a failure [43]. However, given that we are characterizing how file systems react to faults, we believe this is the correct layer for fault injection.

4.3 Failure Policy Inference

After running a workload and injecting a fault, the final step is to determine how the file system behaved. To infer 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 errors 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.

In summary, we believe that we have constructed a fairly comprehensive set of workloads and possible disk faults to exercise file system code. Our workload suite contains about 30 programs, each file system has between 12 and 14 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 about 400 tests which generate relevant block traffic.

5 Failure Policy: Results

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

Due to the sheer volume of experimental data, it is difficult to present all results for the reader's inspection. For each file system that we studied in depth, we present a graphical depiction of our results, showing for each workload/blocktype pair how a given detection or recovery technique is used. Figure 2 presents a (complex) graphical depiction of our results – see the caption for interpretation details. We now provide a qualitative summary of the results that are presented within the figure.

5.1 Linux ext3

Linux ext3 is the most similar to many classic UNIX file systems such as the Berkeley Fast File system [34]. Ext3 divides the disk into a set of block groups; within each are statically-reserved spaces for bitmaps, inodes, and data blocks. The major addition in ext3 over ext2 is journaling [61]; hence, ext3 includes a new set of on-disk structures to manage its write-ahead log.

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 the fixed-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

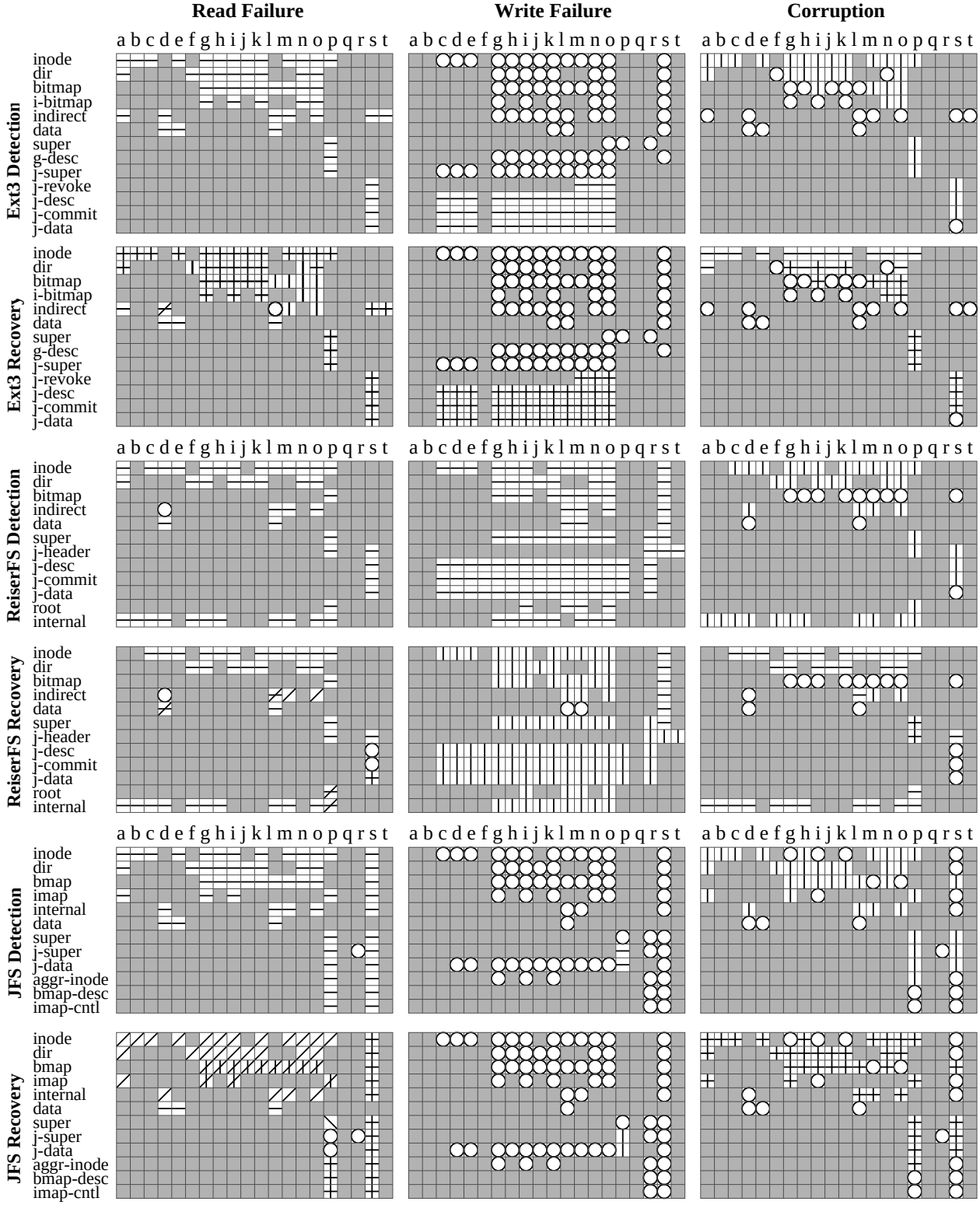

Figure 2: **File System Failure Policies.** *The tables indicate both detection and recovery policies of ext3, ReiserFS and JFS for read, write and corruption faults injected for each block type across a range of workloads. The workloads are* **a:** *path traversal* **b:** {*access,chdir,chroot,stat,statfs,lstat,open*} **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. When multiple recovery mechanisms are observed, the symbols are superimposed in the table.*

Key for Detection		Key for Recovery	
○	D_{Zero}	○	R_{Zero}
–	$D_{ErrorCode}$	/	R_{Retry}
|	D_{Sanity}	–	$R_{Propagate}$
		\	$R_{Redundancy}$
		|	R_{Stop}

type checking is done for many important blocks, such as directories, bitmap blocks, and indirect blocks. Ext3 also performs numerous other sanity checks in type-specific ways; *e.g.*, when the file size field in an inode contains a very 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}); when a prefetch read fails, ext3 retries only the originally requested block.

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, there are important cases when ext3 does not immediately abort the journal on failure (*i.e.*, does not implement R_{Stop}). For example, when a journal write fails, ext3 still writes the rest of the transaction, including the commit block, to the journal; thus, if the journal is later used for recovery, the file system can easily become corrupted. Third, 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. Finally, although ext3 has redundant copies of the superblock ($R_{Redundancy}$), these copies are never updated after file system creation.

5.2 ReiserFS

ReiserFS [42] is quite different in its internal structure than ext3. Virtually all meta-data and data are placed in a balanced tree, similar to a database index. The major advantage of tree-like structuring is scalability [55], allowing large numbers of files to co-reside in a directory.

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.

Bugs and Inconsistencies: ReiserFS also exhibits inconsistencies and bugs. For example, when an ordered data block write fails, ReiserFS journals and commits the transaction without handling the error (R_{Zero} instead of the expected R_{Stop}), which can lead to corrupted data blocks since the metadata blocks now point to invalid data contents. Second, 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. Third, ReiserFS sometimes calls `panic` on failing a sanity check, instead of simply returning an error code. Finally, there is no sanity or type checking to detect corrupt journal data; therefore, replaying a corrupted journal block can make the file system unusable, if for example, the block is written as the super block or a bitmap block.

5.3 IBM JFS

JFS uses modern techniques to manage data, block allocation and journaling, with B+ trees to manage files and directories in addition to using a tree structure for block allocation maps. Unlike ext3 and ReiserFS, JFS uses record-level journaling to reduce journal traffic.

Detection: Error codes are used to detect read failures ($D_{ErrorCode}$), but most write errors are ignored (D_{Zero})(like ext3), 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 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 meta-data 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.

5.4 Windows NTFS

NTFS [53] is the only commercial file system in our study. Because our analysis requires detailed knowledge of on-disk structures, our NTFS analysis requires more effort to first reverse-engineer such information; hence, we have not yet run the full set of tests.

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 meta-data blocks; the file system becomes unmountable if any of its meta-data blocks (except the journal) are corrupted. NTFS surprisingly does not always perform sanity checking, *e.g.*, 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 the error to the user ($R_{Propagate}$). NTFS aggressively uses retry (R_{Retry}) when operations fail. For example, NTFS retries up to seven times under read failures. In the case of write failures, the number of retries varies (*e.g.*, three times for data blocks, two times for MFT blocks).

5.5 File System Summary

• **Ext3: Overall simplicity.** Ext3 implements a simple and mostly reliable failure policy, matching the general 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.

• **NTFS: Persistence is a virtue.** Compared to the Linux file systems, NTFS is the most 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).

5.6 Technique Summary

• **Detection and Recovery: Inconsistency is common.** We found a high degree of inconsistency (observable in the patterns in Figure 2) in failure policy across all file systems. 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 least consistent across all file systems, employing different techniques across block types and routines.

In our estimation, the root cause of 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 mechanisms need to be put into place to encourage consistency.

• **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 [65]. 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 help-

ful would be to periodically inject faults in normal operation as part of a "fire drill" [38]. 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 meta-data 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 [11].

• **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, and hence do not retry the request at a later time (this could be stated in an alternate fashion: they assume that retry has been done at a lower level in the system). 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). We believe that more effort should be placed into developing automatic repair strategies.

• **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 close, endangering them to spatial locality of 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 redundancy into the failure policy of a file system.

6 An IRON File System

We now describe our implementation and evaluation of *IRON ext3 (ixt3)*. Within ixt3, we implement recovery techniques that most single-disk file systems do not currently provide: checksumming (to detect corruption) and in-disk replication (to recover from block failure or corruption). We apply these mechanisms to the *meta-data* of the file system; by doing so, ixt3 can detect and recover from block failures and corruptions to its inodes, directories, and other important file system structures. Thus, in our taxonomy, ixt3 employs $D_{ErrorCode}$ and $D_{Redundancy}$ (checksumming) to detect block failure and $R_{Redundancy}$ (replication) to recover from any detected loss or corruption.

6.1 Implementation

To implement checksumming within ixt3, we borrow techniques from other recent research in checksumming in file systems [54, 37]. Specifically, we place meta-data checksums first into the journal (with the meta-data blocks that are covered by the checksums), and then checkpoint these checksums to their final location, distant from the meta-data blocks. Checksums are very small and can be cached for read verification. In our current implementation, we use SHA-1 to compute the checksums. By incorporating checksumming into existing transactional machinery, ixt3 cleanly integrates into the ext3 framework.

We apply a similar approach in adding meta-data replication to ixt3. All meta-data blocks are written to a separate *replica log*; they are later checkpointed to a fixed location in a block group distant from the original meta-data. We again use transactions to ensure that either both copies reach disk consistently, or that neither do.

Note that "cleaning overhead", which can be a large

problem in pure log-structured file systems [45, 51], is not a major performance issue for journaling file systems, even with ixt3-style replication. Journaling file systems already incorporate "cleaning" into their on-line maintenance costs; for example, ext3 first writes all meta-data to the journal and then "cleans" the journal by checkpointing the data to a final fixed location. Hence, the additional cleaning of meta-data checksums and the replica log increases total traffic only by a small amount.

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. Note that a transactional checksum provides the same crash semantics as in the original ext3 and thus can be used even without other IRON extensions.

6.2 Evaluation

We now evaluate our prototype implementation of ixt3. We focus on three major axes of assessment: robustness to modern disk failures, and both the time and space overhead of the additional redundancy mechanisms.

Robustness: To test the robustness of ixt3, we harness our fault injection framework. Under most failure and corruption tests (not shown), ixt3 successfully detects errors and recovers all metadata. Compared to ext3 in Figure 2, ixt3 eliminates 184 problems out of the 191 observed (all except data block corruption). The result is a consistent and well-defined failure policy.

Time Overhead: We now assess the performance overhead of the IRON mechanisms used within ixt3. We isolate the overhead of each mechanism by enabling meta-data checksumming, meta-data replication, and transactional checksumming 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 [29], which emulates file system traffic of an email server; and TPC-B [59], which runs a series of debit-credit transactions against a simple database. These benchmarks exhibit a broad set of behaviors. Specifically, SSH-Build is a good (albeit simple) model of a "typical" action of a developer or administrator; the web server is read intensive with concurrency; PostMark is meta-data intensive, with many file creations and deletions; TPC-B induces a great deal of synchronous update traffic to the file system.

Workload	CS	R	CS & R	TC	TC & CS	TC & R	All
SSH-Build	1.00	1.00	1.00	1.00	1.00	1.00	1.01
Web	1.00	1.00	1.00	1.00	1.00	1.00	1.00
PostMark	1.00	1.18	1.22	1.00	1.00	1.18	1.22
TPC-B	1.00	1.30	1.45	0.81	0.80	1.13	1.27

Table 3: **The Costs of Redundancy.** *Results from running different variants of ixt3 under the SSH-Build, Web Server, PostMark, and TPC-B benchmarks are presented. The SSH-Build time measures the time to unpack, configure, and build the SSH source tree (the tar'd source is 11 MBin size); we run a web server on top of ixt3 and transfer 25 MB of data using http requests; we run 1500 PostMark transactions with 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. Along the columns, we vary which redundancy technique is implemented, in all possible combinations; "CS" implies meta-data checksumming is enabled, "R" that replication of meta-data is turned on, and "TC" 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 118.5, 52.6, 153.0, and 58.8 seconds, respectively. All testing is done on the Linux 2.6.9 kernel on a 2.4 GHz Intel P4 with 1 GB of memory. The disk is a Western Digital WDC WD1200BB-00DAA0.*

Table 3 reports the relative performance of the variants of ixt3 for the four workloads, as compared to stock Linux ext3. From these numbers, we draw four conclusions. First, for workloads similar to SSH-Build, there is virtually no time overhead with higher levels of redundancy. Hence, if SSH-Build is indicative of the "typical" activity, using IRON for meta-data robustness incurs little cost.

Second, for the web server benchmark, we again see no observable degradation. Hence, for highly read and CPU intensive applications such as the web server, the additional cost due to checksumming is not substantial.

Third, the synchronous workload of TPC-B demonstrates the possible benefit of a transactional checksum. In the base case, this technique improves standard ext3 performance by 20%, and in combination with meta-data checksumming and replication reduces overall overhead from roughly 45% to 27%. Hence, even when not used for additional robustness, checksums can be applied to improve the *performance* of journaling file systems.

Finally, for meta-data intensive workloads such as PostMark and TPC-B, the overhead is more noticeable – 22% for PostMark and 27% for TPC-B. Since these workloads are very meta-data intensive, these results represent the worst-case performance that we expect. Given our relatively untuned implementation of ixt3, we believe this demonstrates that even in the worst case, the costs of meta-data robustness are not prohibitive.

Space Overhead: To evaluate space overhead, we measured a number of local file systems and computed the increase in space required if all meta-data was replicated

and room for checksums included. Overall, we found that the space overhead of checksumming and meta-data replication is small, in the 2% to 5% range.

6.3 Summary

Our implementation of ixt3 represents a middle-ground in the space of IRON file systems. By applying IRON techniques solely to its meta-data, ixt3 lowers the cost of redundancy, both in terms of space and time overheads. In doing so, ixt3 leaves detection and recovery of *user* data to the applications themselves, which can implement application-appropriate strategies. However, we believe that ixt3 represents just a single point in a large space of possible IRON file systems. Many different designs should be explored in order to better understand the benefits and costs of the IRON approach.

7 Related Work

Fault Injection and Robustness Testing: The fault-tolerance community has worked for many years on techniques for injecting faults into a system to determine its robustness [6, 14, 22, 33, 52, 60]. For example, FIAT simulates the occurrence of hardware errors by altering the contents of memory or registers [6]; similarly, FINE can be used to inject software faults into an operating system [33]. More recent work develops techniques to test the Linux kernel behavior under errors [22].

One major difference with most of this previous work 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.

Our work is similar to Brown and Patterson's work on RAID failure analysis [9]. Therein the authors suggest that hidden policies of RAID systems are worth understanding, and demonstrate (via fault injection) that three different software RAID systems have qualitatively different failure-handling and recovery policies. We also wish to discover such "failure policy", but target the file system instead of the RAID, hence requiring a more complex type-aware approach.

Finally, recent work by Yang *et al.* [65] uses model-checking to find a host of file system bugs. Their techniques are well-suited to finding certain classes of bugs, whereas our approach is aimed at the discovery of file system failure policy. Interestingly, our approach also uncovers some serious file system bugs that Yang *et al.* do not. One reason for this may be that our more focused testing is better under scale; whereas model-checking must be limited to small file systems to reduce run-time, our approach can be applied to large file systems.

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 [19]. 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 *meta-data* block becomes corrupted or 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) [26]. 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 [46]).

The fractured failure model for disks is likely 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 errors [13]. Further, virtually all Network Appliance products use checksumming to detect block corruption [24]. Similarly, systems such as the Tandem NonStop kernel [5] include end-to-end checksums, to handle problems such as misdirected writes [5].

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 [34]. 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 [66]. 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 meta-data). 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.* [37] and Stein *et al.* [54] suggest, implement, and eval-

uate 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 [63]. 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.

8 Conclusions

Commodity operating systems have grown to assume the presence of fairly reliable hardware. The result, in the case of file systems, is that most commodity file systems do not include machinery to handle the types of faults one can expect from modern disk drives.

We believe it is time to reexamine how file systems handle failure. One excellent model is already available to us within the operating system kernel: the networking subsystem. Indeed, as 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 [40].

As disks can now be viewed as less than fully reliable, such mistrust must be woven into the storage system framework as well. Many challenges remain: Which failures should disks expose to the layers above? How should the file system software architecture be redesigned to enable more consistent and well-defined failure policy? What kind of controls should be exposed to applications and users? What low-overhead detection and recovery techniques can IRON file systems employ? Answers to these questions should lead to a better understanding of how to effectively implement robust file systems.

References

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

[2] G. A. Alvarez, W. A. Burkhard, and F. Cristian. Tolerating Multiple Failures in RAID Architectures with Optimal Storage and Uniform Declustering. In *ISCA '97*, pages 62–72, Denver, CO, May 1997.

[3] D. Anderson. Personal Communication, 2005.

[4] D. Anderson, J. Dykes, and E. Riedel. More Than an Interface: SCSI vs. ATA. In *FAST '03*, San Francisco, CA, April 2003.

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

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

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

[8] D. Bitton and J. Gray. Disk shadowing. In *VLDB 14*, pages 331–338, Los Angeles, CA, August 1988.

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

[10] W. Burkhard and J. Menon. Disk Array Storage System Reliability. In *FTCS-23*, pages 432–441, Toulouse, France, June 1993.

[11] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *OSDI '04*, pages 31–44, San Francisco, CA, December 2004.

[12] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *SOSP '01*, pages 73–88, Banff, Canada, October 2001.

[13] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *FAST '04*, pages 1–14, San Francisco, CA, April 2004.

[14] J. DeVale and P. Koopman. Performance Evaluation of Exception Handling in I/O Libraries. In *DSN-2001*, Goteborg, Sweden, June 2001.

[15] J. R. Doucer and W. J. Bolosky. A Large-Scale Study of File-System Contents. In *SIGMETRICS '99*, pages 59–69, Atlanta, GA, May 1999.

[16] J. E. Dykes. 'A modern disk has roughly 400,000 lines of code within it'. Personal Communication, 2005.

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

[18] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01*, pages 57–72, Banff, Canada, October 2001.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP '03*, pages 29–43, Bolton Landing, NY, October 2003.

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

[21] R. Green. EIDE Controller Flaws Version 24. http://mindprod.com/eideflaw.html, February 2005.

[22] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of Linux Kernel Behavior Under Error. In *DSN-2003*, pages 459–468, San Francisco, CA, June 2003.

[23] V. Henson. A Brief History of UNIX File Systems. http://infohost.nmt.edu/∼val/fs_slides.pdf, 2004.

[24] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *USENIX Winter '94*, San Francisco, CA, January 1994.

[25] G. F. Hughes and J. 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.

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

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

[28] H. 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.

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

[30] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Summer '86*, pages 238–247, Atlanta, GA, June 1986.

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

[32] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann. Practical Loss-Resilient Codes. In *STOC '97*, pages 150–159, El Paso, TX, May 1997.

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

[34] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

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

[36] A. 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.

[37] S. Patil, A. Kashyap, G. Sivathanu, and E. Zadok. I^3FS: An In-kernel Integrity Checker and Intrusion detection File System. In *LISA '04*, Atlanta, GA, November 2004.

[38] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report CSD-02-1175, U.C. Berkeley, March 2002.

[39] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, pages 109–116, Chicago, Illinois, June 1988.

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

[41] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C.Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

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

[43] P. M. Ridge and G. Field. *The Book of SCSI 2/E*. No Starch, June 2000.

[44] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI '04*, San Francisco, CA, December 2004.

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

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

[47] S. Savage and J. Wilkes. AFRAID — A Frequently Redundant Array of Independent Disks. In *USENIX 1996*, pages 27–39, San Diego, CA, January 1996.

[48] S. W. Schlosser and G. R. Ganger. MEMS-based storage devices and standard disk interfaces: A square peg in a round hole? In *FAST '04*, pages 87–100, San Francisco, CA, April 2004.

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

[50] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *MASCOTS '04*, Volendam, Netherlands, October 2004.

[51] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *USENIX Winter '93*, pages 307–326, San Diego, CA, January 1993.

[52] D. Siewiorek, J. Hudak, B. Suh, and Z. Segal. Development of a Benchmark to Measure System Robustness. In *FTCS-23*, Toulouse, France, June 1993.

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

[54] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying File System Protection. In *USENIX '01*, Boston, MA, June 2001.

[55] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *USENIX 1996*, San Diego, CA, January 1996.

[56] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP '03*, Bolton Landing, NY, October 2003.

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

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

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

[60] 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.

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

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

[63] G. Weinberg. The Solaris Dynamic File System. http://members.visi.net/∼thedave/sun/DynFS.pdf, 2004.

[64] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[65] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.

[66] 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 *OSDI '00*, San Diego, CA, October 2000.