IRON File Systems

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

Computer Sciences Department, University of Wisconsin, Madison

{vijayan,laksh,nitina,haryadi,dusseau,remzi}@cs.wisc.edu

ABSTRACT

Commodity file systems trust disks to either work or fail completely, yet modern disks exhibit more complex failure modes. We suggest a new *fail-partial failure model* for disks, which incorporates realistic localized faults such as latent sector errors and block corruption. We then develop and apply a novel *failure-policy fingerprinting* 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 partial disk failures. Finally, we design, implement, and evaluate a prototype IRON file system, Linux ixt3, showing that techniques such as in-disk checksumming, replication, and parity greatly enhance file system robustness while incurring minimal time and space overheads.

Categories and Subject Descriptors:
D.4.3 [**Operating Systems**]: File Systems Management
D.4.5 [**Operating Systems**]: Reliability
General Terms: Design, Experimentation, Reliability
Keywords: IRON file systems, disks, storage, latent sector errors, block corruption, fail-partial failure model, fault tolerance, reliability, internal redundancy

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 [56]; 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* [16, 34, 57], where a block or set of blocks are inaccessible. Worse, blocks sometimes become *silently corrupted* [9, 26, 73]. Finally, disks sometimes exhibit *transient* performance problems [7, 67].

SOSP'05, October 23–26, 2005, Brighton, United Kingdom.

There are many reasons for these complex failures in disks. For example, a buggy disk controller could issue a "misdirected" write [73], placing the correct data on disk but in the wrong location. Interestingly, while these failures exist today, simply waiting for disk technology to improve will not remove these errors: indeed, these errors may *worsen* over time, due to increasing drive complexity [5], immense cost pressures in the storage industry, and the escalated use of less reliable ATA disks – not only in desktop PCs but also in large-scale clusters [23] and storage systems [20, 28].

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 [33, 57], 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 [16]. Similarly, highly-reliable systems (*e.g.*, Tandem NonStop) utilize end-to-end checksums to detect when block corruption occurs [9].

Unfortunately, such technology has not filtered down to the realm of commodity file systems, including Linux file systems such as ext3 [71], ReiserFS [49], and IBM's JFS [11], or Windows file systems such as NTFS [63]. 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 [23].

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 *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. Our approach leverages gray-box knowledge [6, 62] of file system data structures to meticulously exercise file system access paths to disk.

To better characterize failure policy, we develop an *Internal RObustNess (IRON)* taxonomy, which catalogs a broad range of detection and recovery techniques. Hence, 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 three important and substantially different open-source file systems, ext3, ReiserFS, and IBM's JFS, and one closed-source file system, Windows NTFS. Across all platforms, we find 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. 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 of transient failures; most file systems assume a single temporarily-inaccessible block indicates a fatal whole-disk failure. Finally, we show that none of the file systems can recover from partial 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 end-to-end argument [53]: at the top of the storage stack, the file system is fundamentally responsible for reliable management of its data and metadata.

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 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 errors 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.

The contributions of this paper are as follows:

- We define a more realistic failure model for modern disks (the fail-partial model) (§2).
- We formalize the techniques to detect and recover from disk errors under the IRON taxonomy (§3).
- We develop a fingerprinting framework to determine the failure policy of a file system (§4).
- We analyze four popular commodity file systems to discover how they handle disk errors (§5).
- We build a prototype version of an IRON file system (ixt3) and analyze its robustness to disk failure and its performance characteristics (§6).

To bring the paper to a close, we discuss related work (§7), and finally 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 *fail-partial 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.

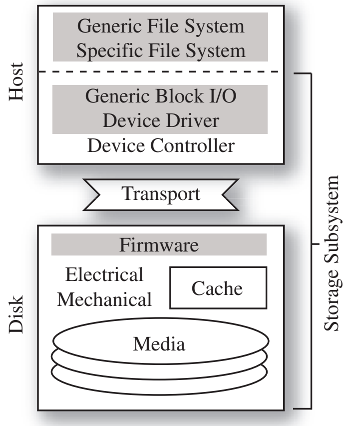

Figure 1: **The Storage Stack.** *We present a schematic of the entire storage stack. At the top is the file system; beneath are the many layers of the storage subsystem. Gray shading implies software or firmware, whereas white (unshaded) is hardware.*

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) and electrical components (*e.g.*, busses). 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 [19]).

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).

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).

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 [36]) 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 [57]. 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 [5]. 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 [68]. 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 [37] or have memory leaks that lead to intermittent failures [68]. Other firmware problems can lead to poor drive performance [54]. 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 [73]. 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 [67] 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 [72]. 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 [26]. Others have also observed IDE protocol version problems that yield corrupt data [23]. 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 [15, 22, 66]. 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.

2.3 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 errors" [33, 34].

• **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.

2.3.1 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 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.

2.3.2 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 suggest that some forms of block failure do indeed exhibit spatial locality [34]. 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" [29]. 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 [67], 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 [4]. 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 indicates that such errors occur more commonly than absolute disk failure [34], and more recent research estimates that such errors may occur five times more often than absolute disk failures [57].

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 [31]. 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 [24] 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.

2.3.4 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 [5].

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 [5]. Low-cost "PC class" drives tend to be tested less and have less internal machinery to prevent failures from occurring [31]. The result, in the field, is that ATA drives are observably less reliable [67]; however, cost pressures serve to increase their usage, even in server environments [23].

Finally, the amount of software is increasing in storage systems and, as others have noted, software is often the root cause of errors [25]. 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 [22, 66] – 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) partial 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 [3, 74].

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.

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.**

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.**

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.

• $Zero$: 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 §5, 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 [48]. 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` [41]). 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 [9] and has recently been applied to improve security as well [43, 64]. 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 [9, 73]; 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 [12], parity [42, 45] and other error-correction codes [38], 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 [45].

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 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 errors [34]. 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.

• $Zero$: 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.* [51], 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), 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 [45], 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 [38]) 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 [18].

3.4 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 [53]; 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 §6). 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" [55] 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 [5]; no interface exists to control these different low-level strategies in current systems.

3.5 Doesn't RAID Make Storage Reliable?

Another question that must be answered is: can't we simply use RAID techniques [45] 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 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 [74], 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 errors [16].

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.

4. FAILURE POLICY FINGERPRINTING

We now describe our methodology to uncover the *failure policy* of file systems. Our main objective with *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. 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 all interesting cases. 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. We now describe how we create workloads, inject faults, and deduce failure policy.

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 (leaving this as an avenue for future work), we do strive to execute as many of the interesting internal cases as possible.

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 3 summarizes the test suite.

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 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).

4.2 Type-Aware Fault Injection

Our second step is to inject faults that emulate a disk adhering to the fail-partial failure model. Many standard fault injectors [13, 59] 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 *type-aware fault injection*, which builds on our previous work with "semantically-smart" disk systems [8, 60, 61, 62]. With type-aware fault injection, instead of failing blocks obliviously, we fail blocks of a specific type (*e.g.*, an inode). Type 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 type-aware 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 type-awareness clearly outweigh these complexities. The block types of the file systems we test are listed in Table 4.

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 3: **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).*

Our mechanism for injecting faults is to use a software layer directly beneath the file system (*i.e.*, a pseudo-device driver). This layer injects both block failures (on reads or writes) and block corruption (on reads). To emulate a block failure, 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 [13], 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 [50]. 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 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 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. Currently, this is the most human-intensive part of the process, as it requires manual inspection of the visible outputs.

4.4 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.

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

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

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

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: **File System Data Structures.** *The table presents the data structures of interest across the four file systems under test: ext3, ReiserFS, JFS, and NTFS. In each table, we list the names of the major structures and their purpose. Note that our knowledge of NTFS data structures is incomplete, as it is a closed-source system.*

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 illogical 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 [40]. 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 [71]; 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 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.

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 and hence are not useful.

5.2 ReiserFS

ReiserFS [49] is comprised of vastly different data structures than ext3. Virtually all metadata and data are placed in a balanced tree, similar to a database index. A key advantage of tree structuring is scalability [65], allowing many files to coexist in a directory.

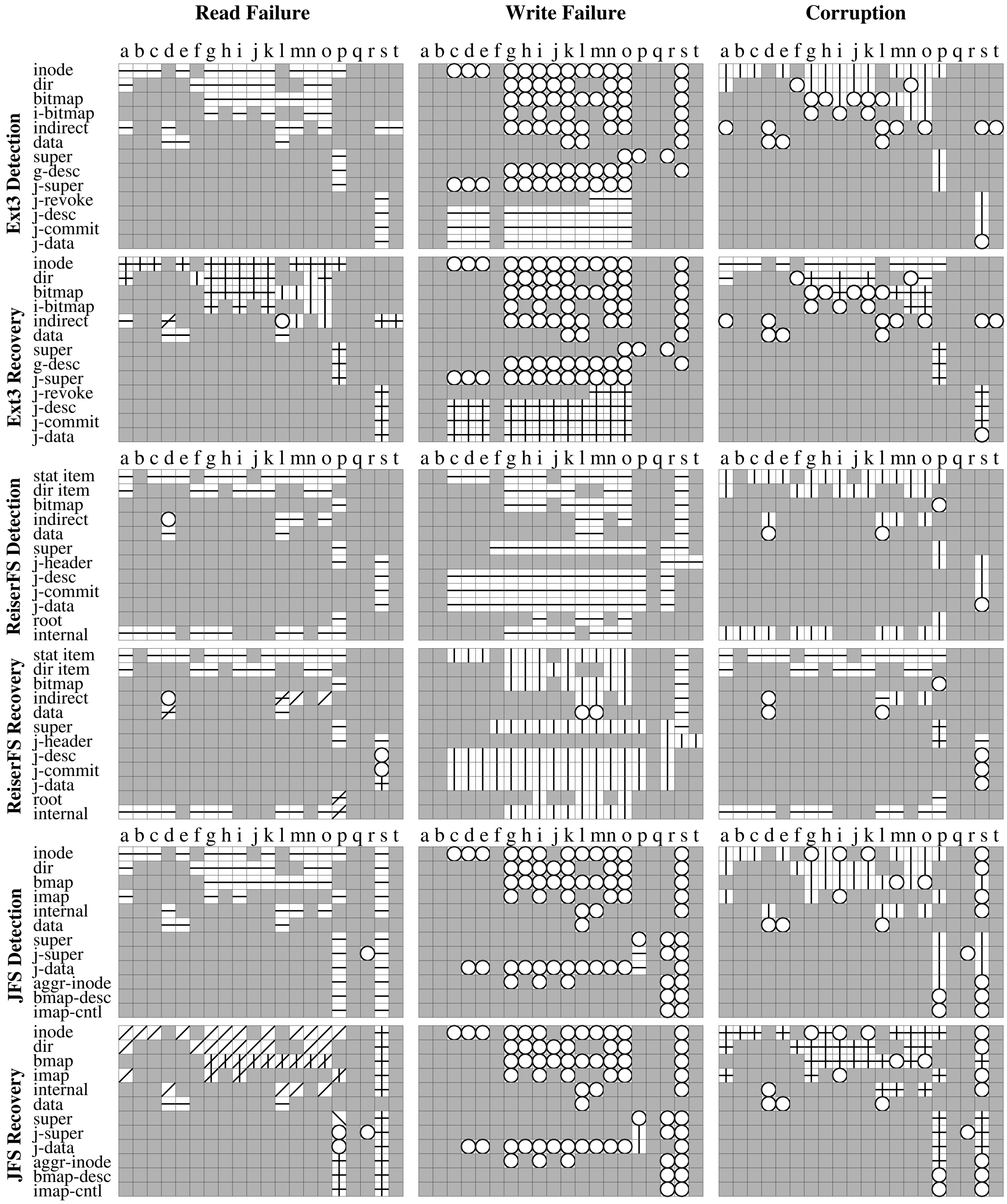

Figure 2: **File System Failure Policies.** *The tables indicate both detection and recovery policies of ext3, ReiserFS, and JFS for read, write, and corruption faults injected for each block type across a range of workloads. The workloads are* **a:** *path traversal* **b:** *access,chdir,chroot,stat,statfs,lstat,open*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

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

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 (*e.g.*, the block is written as the super block).

5.3 IBM JFS

JFS [11] uses modern techniques to manage data, block allocation and journaling, with scalable tree structures to manage files, directories, and block allocation. Unlike ext3 and ReiserFS, JFS uses record-level journaling to reduce journal traffic.

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 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.

5.4 Windows NTFS

NTFS [2, 63] 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 Figure 2.

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

5.5 File System Summary

We now present a qualitative summary of each of the file systems we tested. Table 5 presents a summary of the techniques that each file system employs (excluding NTFS).

- **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.

Level	ext3	Reiser	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 5: **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.*

• **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

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 Figure 2). 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 [21]; 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 [61]. 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.

• **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 [75]. 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" [44]. 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 [14].

• **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.

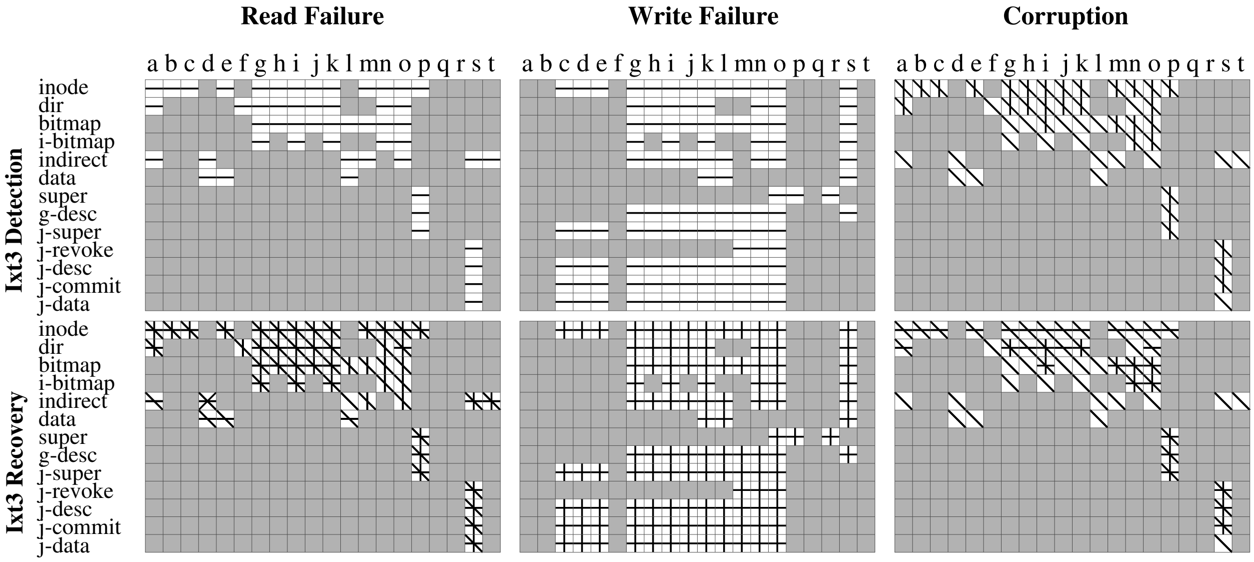

Figure 3: **Ixt3 Failure Policy.** *The tables plot both detection and recovery policies of ixt3 for read, write, and corruption faults injected for each block type across a range of workloads. The workloads are varied across the columns of the figure, and the different block types of the ixt3 file system are varied across the rows. The workloads are grouped in the same manner as in Figure 2.*

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

6. AN IRON FILE SYSTEM

We now describe our implementation and evaluation of *IRON ext3 (ixt3)*. Within ixt3, we implement a family of recovery techniques that most commodity file systems do not currently provide. To increase its robustness, ixt3 applies checksums to both metadata and data blocks, uses pure replication for metadata, and employs parity-based redundancy to protect user data.

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.

6.1 Implementation

We now briefly 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 [64, 43]. Specifically, we place checksums first into the journal, and then checkpoint these checksums to their final location, distant from the blocks they checksum. 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.

Metadata Replication: 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. We again use transactions to ensure that either both copies reach disk consistently, or that neither do.

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 modify the inode structure of ext3 to associate a file's parity block with its data blocks. 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 and assign them to files when they are created.

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. 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 [52, 58], 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.

6.2 Evaluation

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

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 3.

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 [47].

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 [35], which emulates file system traffic of an email server; and TPC-B [69], 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.

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 6 reports the relative performance of the variants of ixt3 for the four workloads, as compared to stock Linux ext3. 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.

#	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 6: **Overheads of ixt3 File System 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. 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 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 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.*

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 parity, 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.

Space Overhead: To evaluate space overhead, we measured a number of local file systems 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. 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.

6.3 Summary

We have investigated a family of redundancy techniques, and found that ixt3 greatly increases the robustness of the file system under partial failures while incurring modest time and space overheads. However, much work is left; new designs and implementation techniques should be explored to better understand the benefits and costs of the IRON approach.

7. RELATED WORK

Our effort builds upon related work from two bodies of literature. Our file system analysis (§4) is related to efforts that inject faults or otherwise test the robustness of systems to failure. Our prototype IRON file system (§6) draws on recent efforts in building software that is more robust to hardware failure. We discuss each in turn.

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 [10, 17, 27, 39, 59, 70]. For example, FIAT simulates the occurrence of hardware errors by altering the contents of memory or registers [10]; similarly, FINE can be used to inject software faults into an operating system [39].

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 [13]. 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 "failure policy", but target the file system (not RAID), hence requiring a more complex type-aware approach.

Recent work by Yang *et al.* [75] 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 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.

Our work builds upon our earlier work in failure injection underneath file systems [47]. In that work, we developed an approach to test how file systems handle write failures during journal updates. Our current work extends this to look at all data types under read, write, and corruption failures.

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 [23]. 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) [32]. 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 [53]).

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 errors [16]. Further, virtually all Network Appliance products use checksumming to detect block corruption [30]. Similarly, systems such as the Tandem NonStop kernel [9] include end-to-end checksums, to handle problems such as misdirected writes [9].

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 [40]. 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 [76]. 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.* [43] and Stein *et al.* [64] 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 [73]. 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 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.

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, 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 [46].

Because disks should 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 a 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 the next generation of robust and reliable IRON file systems.

Acknowledgments

We would like to extend particular thanks to Steve Kleiman of Network Appliance and Dave Anderson and Jim Dykes of Seagate for their insights into how disks really work and fail. We would also like to thank Liuba Shrira (our shepherd), Dave DeWitt, Mark Hill, Jiri Schindler, Mike Swift, the anonymous reviewers, and the members of ADSL for their excellent suggestions and comments. We thank Himani Apte and Meenali Rungta for their invaluable work on implementing parity within ext3. Finally, we thank the Computer Systems Lab (CSL) for providing a terrific computing environment for systems research. This work has been sponsored by NSF CCR-0092840, CCR-0133456, NGS-0103670, ITR-0325267, IBM, Network Appliance, and EMC.

9. REFERENCES

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

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

[3] G. A. Alvarez, W. A. Burkhard, and F. 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] D. Anderson. "Drive manufacturers typically don't talk about disk failures". Personal Communication from Dave Anderson of Seagate, 2005.

[5] D. Anderson, J. Dykes, and E. 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.

[6] A. C. Arpaci-Dusseau and R. 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.

[7] R. H. Arpaci-Dusseau and A. 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.

[8] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. 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.

[9] 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.

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

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

[12] D. Bitton and J. 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.

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

[14] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. 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.

[15] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. 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.

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

[17] J. DeVale and P. 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.

[18] J. R. Douceur and W. 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.

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

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

[21] R. W. 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.*

[22] 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 *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[23] S. Ghemawat, H. Gobioff, and S.-T. 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.

[24] G. A. Gibson, D. Rochberg, J. Zelenka, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, and E. 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.

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

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

[27] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. 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.

[28] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. 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.

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

[30] D. Hitz, J. Lau, and M. 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.

[31] 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.

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

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

[34] 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.

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

[36] S. 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.

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

[38] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] S. Patil, A. Kashyap, G. Sivathanu, and E. 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.

[44] 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.

[45] D. Patterson, G. Gibson, and R. 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.

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

[47] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2005)*, Yokohama, Japan, June 2005.

[48] 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.

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

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

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

[52] 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.

[53] 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.

[54] J. 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.

[55] S. W. Schlosser and G. 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.

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

[57] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. 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.

[58] M. Seltzer, K. Bostic, M. K. McKusick, and C. 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.

[59] D. Siewiorek, J. Hudak, B. Suh, and 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.

[60] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. 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.

[61] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. 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.

[62] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. 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.

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

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

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

[66] M. M. Swift, B. N. Bershad, and H. 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.

[67] 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.

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

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

[70] 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.

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

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

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

[74] 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.

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

[76] 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.