TOWARDS RELIABLE STORAGE SYSTEMS

by

Haryadi S. Gunawi

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

Doctor of Philosophy
in
Computer Sciences

University of Wisconsin-Madison

2009

Awarded:

The 2009 departmental Best Thesis Award

The 2009 ACM Doctoral Dissertation Award Honorable Mention

Committee in charge:

Prof. Andrea C. Arpaci-Dusseau (Co-chair)
Prof. Remzi H. Arpaci-Dusseau (Co-chair)
Prof. Ben Liblit
Prof. Michael M. Swift
Prof. Kewal Saluja

To the three important women in my life:
my daughter, my wife, and my mother

Acknowledgements

Faculty, friends, and family members have been a great support for me in completing this Ph.D. journey. In this special section, I would like to express my deep gratitude to these individuals.

Andrea and Remzi definitely come first, for without them this Ph.D. would not have been an exceptional journey. This journey started nine years ago when I was still an undergraduate looking for a professor who wanted to advise me in an independent study. I recall knocking some doors without any luck until I knocked Remzi's after which he welcomed me for an independent study. My research life started then, and I was extremely happy to be able to continue to graduate school under Remzi's and Andrea's supervision.

It has been a great privilege to have both of them as my "research parents". As great research parents, they care not only about results, but also crucial skills that a student should develop to be a successful researcher. I am wholeheartedly thankful for the many opportunities they have given me, including teaching a senior Operating System class, connecting to top industries for internships, advising some students, attending special meetings and conferences, and many more. I recall some students asked me: "What is the key to success in graduate school?" I paused and answered "you simply learn from the best" – in my case, I have learned a great deal from Andrea and Remzi.

I also would like to thank my other thesis-committee members, Ben Liblit, Mike Swift, and Kewal Saluja, for their insights, questions, and advice for my research. I would like to thank Mike for his challenging questions during my preliminary exam; the questions have helped me in better preparing my defense. I also really enjoyed working with Ben Liblit and his student, Cindy Rubio-Gonzalez. This wonderful collaboration has resulted in two published papers, one is part of this dissertation.

I am fortunate to get the chance to work on some projects with smart and hard-working colleagues: Nitin Agrawal, Lakshmi Bairavasundaram, Thanh Do, Shweta Krishnan, Zhenxiao Luo, Vijayan Prabhakaran, Abhishek Rajimwale, Sriram Sub-

ramanian, and Yupu Zhang. I also have enjoyed interacting with other members in the group: Leo Arulraj, John Bent, Nate Burnett, Tim Denehy, Todd Jones, Andrew Krioukov, Joe Meehan, Florentina Popovici, Meenali Rungta, Muthian Sivathanu, Swami Sundararaman, Laxman Visampalli, and Yiying Zhang. I feel lucky to have Lakshmi and Abhishek as my officemates as both of them are really neat persons.

I have benefited greatly from interning at EMC Corp. and Microsoft Research. I would like to thank the companies as well as my mentors and managers there: Jiri Schindler (now at NetApp) and Peter Lauterbach at EMC, and Orion Hodson and Galen Hunt at Microsoft Research.

Finally, I would like to thank my family, without whose love and support this Ph.D. would have been meaningless. My wife, Anna, has shared with me the happy and also stressful moments. I am thankful that she has kept me energetic and extremely healthy during my Ph.D. years. My almost 2-year-old daughter, Livia, has always cheered me up when I am tired. My mother is a special mother. She worked very hard to fulfill her commitment that her sons should be able to study abroad, taste high-quality undergraduate education, and continue to graduate school. I dedicate this dissertation to these three important women in my life.

Abstract

TOWARDS RELIABLE STORAGE SYSTEMS
Haryadi S. Gunawi

"There's no way of reporting error ... to userspace. So ignore it."
– A comment in ext3 (inode.c, line 1517)

Users are storing increasingly massive amounts of data. Storage software complexity is growing. The use of cheap and less reliable hardware is increasing. The combination of these trends presents us with a terrific challenge: *How can we promise users that storage systems work robustly in spite of the complex failures that can arise?*

In the first part of this dissertation, we respond to this question with our analysis of three reliability components present in many modern file systems: the file system checker (fsck), failure detection and recovery policies (failure policy), and journaling. We find that these subsystems are deficient in handling partial disk failures: in the fsck analysis, we find that some repairs are buggy (making the repaired file system more corrupted) and some repairs are missing (leaving some corruptions unattended). In the failure policy analysis, we observe a major problem of diffused fault handling, which causes policies to be inconsistent, buggy, and inflexible to change. In the journaling analysis, we uncover that current journaling frameworks cannot recover from checkpoint write failures, and hence write failures are intentionally ignored. The results of our analysis hint that managing failures is hard (as also hinted by the developer's comment), and hence demand for novel solutions towards building more reliable storage systems.

In the second part of this dissertation, we present our solutions to the problems above. First, we re-architect the file system checker by introducing SQCK, a robust file system checker that employs a declarative query language. By writing hundreds

of checks and repairs in a query language (*e.g.*, SQL), the high-level intent of the checker can be specified in a clear and compact manner. We show that SQCK is able to perform the same functionality as the Linux ext2/3 checker with elegant and compact queries.

Second, we present EDP, a static analysis tool that shows how error codes flow through file systems and storage drivers. We observe that low-level errors are sometimes lost as they travel through the many layers of the storage subsystem: out of the 9022 function calls through which the analyzed error codes propagate, we find that 1153 calls (13%) do not correctly save the propagated error codes. Our detailed analysis shows that many violations are not corner-case mistakes; the return codes of some functions are consistently ignored.

Finally, we present I/O shepherding, a new reliability infrastructure for file systems. With I/O shepherding, the reliability policies of a file system are well-defined, easy to understand, and simple to tailor to environment and workload. As part of this framework, we also introduce *chained transactions*, a novel and more powerful transactional model for checkpoint recoveries. We show that I/O shepherding enables simple, powerful, and correctly-implemented reliability policies by implementing an increasingly complex set of policies.

Contents

Chapter 1

Introduction

"Just ignore errors at this point.
There is nothing we can do except to try to keep going."
– A comment in XFS (xfs_vnodeops.c, line 1785)

With the success of low-cost, high-capacity disk drives, all types of users are storing increasingly massive amounts of data. Personal users create digitized forms of music, images, and videos, as well as conventional documents, and it is estimated that almost 800 MB of data is produced per person each year [91]. Organizations, in addition to storing new data, are also keeping old data longer for the purposes of compliance and business intelligence [121]. Scientific users are capturing large amounts of data, as much as 200 GB data per day per project [118], and they estimate that the amount is doubling every year [134].

Given the rising amount of data, access to data is critical. Data unavailability may cost a company more than one million dollars per hour [81, 101]. Data loss can be more catastrophic; a recent survey shows that 7 out of 10 small and medium firms that experience a major data loss go out of business within a year [35]. For larger organizations such as banking, data loss can have much greater consequences; in July 2009, a large bank was fined a record total of £3 millions after losing data on thousands of its customers [98].

Unfortunately, disks fail, and they fail more often than manufacturers expect them to [115]. Furthermore, the manner in which their failures arise is becoming more complex. The simple view that disks either work or fail completely no longer holds. The reality today is that disks not only exhibit whole-disk failure [115] but also *partial* failures. For example, disks can exhibit latent sector errors, where

a disk block or set of blocks are inaccessible [30, 79, 117]. Worse, disk blocks sometimes become silently corrupted [18, 53, 130, 131]. In a recent large scale study of 1.5 million disk drives deployed in the field, Bairavasundaram *et al.* have shown that latent sector errors affect a significant percentage of disk drives (*e.g.*, up to 20% of the drives of a SATA disk model in just two years) [13] and corruptions affect 400,000 blocks over three years [14].

Exacerbating the aforementioned problems, several technology trends and market forces may combine to make storage system failures occur *more* frequently over time. First, disk drives are becoming more dense as more bits are packed into smaller spaces [55]. As density increases, the logical complexity of the drive mechanics and firmware also increases, which can lead to more failures. For example, errors such as bit spillovers on adjacent tracks can corrupt more bits at higher areal densities [10]. It is also known that complex firmware logic can introduce bugs that could corrupt data [143]. In this denser world, reliability becomes more challenging.

Second, the use of cheap and less reliable hardware is increasing. Companies are striving to lower costs by cutting corners in a competitive market place, and thus, they increasingly consolidate on low-cost PCs using ATA drives [46, 54, 128]. These low-cost PC drives tend to be less tested and have less internal machinery to prevent failures from occurring [72]. The result, in the field, is that ATA drives are observably less reliable [6, 11, 13, 14, 135]. Therefore, the prevalent use of these drives implies that disk failures will not be a rarity but rather a commonplace occurrence.

Finally, the amount of software is increasing in storage systems and, as others have noted, software is often the root cause of errors [50]. 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 [40, 133], and hence a likely source of increased errors in the storage stack.

The combination of these trends presents us with a terrific challenge: *How can we promise users that storage systems work robustly in spite of their massive software complexity and all the complex disk failures that can arise?* As hinted by the developers' comments quoted throughout this dissertation, this is a challenging problem; even when the developers are aware of the failures that can arise, they do not know how to implement the correct response. To respond to this challenging question, we believe it is of utmost importance to first *analyze* existing approaches and then *build* new techniques for designing and building robust software on top of increasingly complex and unreliable hardware. We discuss these two parts of this

dissertation in the following two sections.

1.1 Analysis of File System Reliability Components

Since disks fail, it is important to analyze how disk failures impact modern systems. In the first part of this dissertation, we focus on analyzing the impact of *partial disk failures* on *local commodity file systems*.

Partial disk failures such as latent sector errors and corruptions occur due the complex nature of disk drives; trapped particles could cause scratches thereby rendering sectors unreadable [11, 117]; high gaps between the disk head and the medium could cause data to be written poorly [13]; firmware bugs could silently lose, misdirected, or torn disk writes [46, 53, 130, 131]; software bugs in device drivers could corrupt data [28, 40, 133]. As mentioned earlier, recent studies have shown that partial disk failures occur in practice at a higher rate than what was expected before [13, 14].

The impact of partial disk failures could affect a full range of data management systems such as databases, distributed file systems, and RAID arrays. In this dissertation, we focus and limit our scope only to local commodity file systems for the following two reasons. First, these file systems are widely deployed in personal computers of millions of users. Users store valuable data such as financial information, pictures, videos, and other types of documents in their personal laptops or PCs. Once users store their data, they expect it to be persistent forever, and perpetually available. Second, a fundamental change is occurring within high-end storage systems design; while traditionally, high-end storage systems are assembled from highly customized components, today, people want to lower hardware costs and build large scalable storage clusters. Thus, modern high-end storage systems are comprised of commodity PCs running commodity operating systems and file systems [37, 46, 58, 68, 70, 129]. Therefore, analyzing local file systems will bring benefits to other systems built on top of it.

As a first step, to emulate partial disk failures, we developed type-aware fault injection in earlier work [106]. Our approach is to inject faults just beneath the system under test and observe how the system responds. Many standard fault injectors fail disk blocks in a *type-oblivious* manner [25, 120]; that is, a block is failed regardless of how it is being used by the system. In contrast, with type-aware fault injection, we fail blocks of a specific type (*e.g.*, an inode block in a file system or a user data page in a virtual-memory system). Type information is crucial for reverse-engineering the different strategies that a system applies for its different data structures.

With the fault-injection framework in place, we analyze how partial disk failures are handled by three important reliability components present in many modern file systems: the file system checker [65, 93], an important repair utility that fixes file system inconsistency; failure policy [106], the file system detection and recovery techniques for dealing with disk failures; and journaling [52, 64, 140], a mechanism that guarantees write atomicity in the presence of system crashes.

This dissertation reports our analysis of the three components within the Linux ext3 file system [140], which is the default file system for many popular Linux distributions such as Red Hat. However, in some cases, we have extended our analysis to other file systems including ReiserFS [109], IBM's JFS [20], and Windows NTFS [124]. Interestingly, although these file systems have been in active development for more than one decade, we still find flaws, bugs, and design problems in terms of how their reliability components deal with partial disk failures. This highlights that the impact of partial disk failures to file systems has not been well examined in literature and in practice. Below, we discuss our motivations for choosing the three reliability components and also present the summary of our findings.

1.1.1 Analysis of File System Checker

The first component that we evaluate is the offline file system checker (also known as fsck) [61]. Tools such as fsck have existed for many years [93] and are applied to restore a damaged or otherwise inconsistent file system image to a working and usable state. Although many newer file systems have tried to avoid the inclusion of an offline checker in their tool suite [65] (for example, by assuming that journaling always keeps the file system consistent), they inevitably find that a checker must be deployed. For example, SGI's XFS was introduced as a file system with "no need for fsck, ever," but soon found it necessary to deliver such a tool [44]. Thus, a key component to a robust file system is a robust fsck.

Unfortunately, robust checkers are not currently straightforward to design or implement. First, checkers are large and complex beasts; for example, the Linux ext2 checker performs more than 120 data structure repairs in sixteen thousand lines of C code, while the ext2 file system itself is less than ten thousand lines. Checkers are often written in a low-level systems language such as C, which can be difficult to reason about. Checkers also are hard to test, given the huge possible state space of input file systems. Finally, checkers are often run only when a serious problem has occurred; it is well known that rarely-run recovery code tends to be less reliable [26, 107].

Nevertheless, fsck is considered as the last line of defense to fix damaged file

systems; if fsck fails to repair file systems correctly, no other tools can. Therefore, it is important to analyze whether current checkers could properly repair inconsistencies due to partial disk failures (*e.g.*, corruptions). To achieve that, we evaluate e2fsck, the Linux ext2/3 checker. We injected corruptions to ext2 on-disk pointers in order to observe how e2fsck repairs file system inconsistencies.

Our results show many weaknesses of e2fsck that lead to unmountable file systems and data loss [61]. For example, first, e2fsck sometimes performs out-of-order repairs that can corrupt the file system image by overwriting important metadata such as files, directories, and even the superblock. Second, e2fsck does not always use all available information to perform the correct repairs, and hence can lose portions of the directory tree; a dreadful mistake is when e2fsck ignores replicas of important pointers such that a corrupt pointer is considered unfixable, and hence all information behind that pointer (*e.g.*, all files in an inode table) is considered lost. Third, e2fsck does not follow the same policies (*e.g.*, block allocation policy) as the original ext2/3 file system. Finally, e2fsck does not always perform a secure repair, and hence data from one user can leak to another user.

These findings show that fsck is truly complex and when not designed properly, the correctness of such a complex system is hard to achieve. Other than e2fsck, other checkers are unfortunately written in the same way (*i.e.*, hundreds of checks and repairs in thousands of lines of code). Thus, we believe other checkers have the same weaknesses as in e2fsck.

1.1.2 Analysis of Failure Policy

Our second analysis is of *failure policy* [106]. That is, we attempt to unearth how a running file system detects and recovers from a range of partial disk failures (*e.g.*, corruptions, read and write errors). In this analysis, we specifically choose commodity file systems because their failure policies have not been captured in literature, unlike those of high-end storage systems. For example, many high-end storage systems are known to incorporate a background *disk scrubbing* process [78, 117] for detecting latent sector faults. Some also employ extra levels of redundancy to reduce the potential data loss of undetected latent faults [30]. Finally, it is well-known that highly-reliable systems utilize end-to-end checksums to detect block corruptions [18].

As part of a larger group effort [106], we analyze several commodity file systems; the author specifically analyzes Windows NTFS [124] and the rest of the group analyze three other file systems, ext3 [140], ReiserFS [109], and IBM JFS[20]. To unearth the failure policies of these file systems, we use the type-aware fault injection framework to insert block-level read/write faults and corruptions.

Our findings point us to a major problem of *diffused handling*; there are more than a hundred places where the file system tries to handle I/O failures. One reason for this diffusion is that the file system tries to handle each fault where it arises in the code. Because I/Os are generated from many different locations within the file system, the fault-handling policy is also spread throughout the code. As a result, we find that failure policies are *inconsistent*, *buggy*, and *inflexible*: different recovery actions are employed under similar failure scenarios, error-codes are dropped incorrectly leading to serious silent failures, and changing one simple policy requires modifications in many places. This shows that commodity file systems do not have a proper framework for disk failure handling.

1.1.3 Analysis of Journaling

In our final analysis, we look into journaling [60]. Journaling (also known as write-ahead logging [52]) is a mechanism that guarantees write atomicity in the presence of system crashes. This mechanism was first introduced to the file system world more than two decades ago [64]. Since then, journaling has been widely deployed in many modern file systems including ext3 [140], ReiserFS [109], IBM JFS [20], XFS [132] and Windows NTFS [124].

Journaling is accomplished with a sequence of three main operations. First, file system updates are first committed as a transaction in the journal (write-ahead log). Then, the updates in the transaction are checkpointed to their final locations. Finally, after the checkpoint completes, the transaction can be released. Although this sequence of technique works perfectly in anticipating crashes, its correctness has not been evaluated when partial disk failures (write failures in particular) come into the picture. Therefore, to analyze this, we inject write failures at all unique points within the journaling sequence.

We find that journaling file systems suffer from a general *problem of failed intentions* [60], which arises when the intent as written to the journal cannot be realized due to disk failure during checkpointing. More specifically, when a transaction is being checkpointed and a checkpoint write fails, the file system might desire to perform a recovery (*e.g.*, remap the failed write), which could result in a metadata change (*e.g.*, a remap table is modified). In order to properly reflect the recovery on behalf of the checkpointed transaction, the metadata update must be written to the disk *before* the checkpointed transaction is released from the journal. This is done by committing a new transaction that contains the metadata update. However, the current journaling semantic allows the checkpointed transaction to be released before the new transaction is committed. Thus, if a crash occurs after the checkpointed transaction is released and before the new transaction is committed,

then the metadata update performed by the recovery is lost. As a result, the file system will be in an inconsistent state.

In summary, with this major flaw, a simple block remapping during a checkpoint failure cannot be done at the file system level. As a result, many modern file systems that employ journaling (such as ext3, IBM JFS, and ReiserFS) ignore checkpoint failure. Thus, the fact that we cannot recover from a checkpoint failure properly with the current journaling scheme is disastrous.

1.2 Building More Reliable File Systems

We believe our three analyses illustrate a sad reality of today's commodity file systems; when recovery is hard, failures are often ignored (again, as hinted by the developers' comments). Thus, all the issues we raised above call for novel solutions towards building more reliable storage systems. In the second part of this dissertation, we present our approaches to solving the problems we have found. First, we introduce SQCK [59], a robust file system checker that employs a declarative query language. Next, we present EDP [62], a static analysis tool that shows where error-codes are ignored in file systems and storage drivers. Finally, we present I/O shepherding [60], a simple yet powerful way to build robust and centralized failure policies within a file system.

1.2.1 SQCK: A Declarative File System Checker

As mentioned before, robust checkers are not currently straightforward to design or implement; a typical implementation needs to perform hundreds of checks and repairs in tens of thousands of lines of C code. Given this reality, it is perhaps not surprising that file system checkers often corrupt or lose data. Thus, to build a new generation of robust and reliable file system checkers, we believe a new approach is required. The ideal approach should enable the high-level intent of the checker to be specified in a clear and compact manner; further, the description of the intent should be cleanly separated from its low-level implementation and how it is optimized. A high-level specification has multiple benefits: by its very nature it is easier to understand, modify, and maintain.

To realize this new approach, we introduce SQCK [61] (pronounced "squeak"), a novel file system checker. Borrowing heavily from the database community, SQCK employs declarative queries to check and repair a file system image. We find that a declarative query language is an excellent match for the cross-checks that must be made across the different structures of a file system; declarative re-

pairs can be surprisingly elegant and compact, especially compared to the original e2fsck code. Specifically, we find that SQCK can reproduce the functionality of e2fsck in many fewer lines of code; we rewrote the checks and repairs in e2fsck in 150 queries in about 1100 lines of SQL statement (along with some helper code written in C).

As checks and repairs are written in declarative queries, SQCK enables file system developers to plug-in/out checks and repairs in a straightforward fashion. In our evaluation, we show how SQCK can improve upon the traditional checks and repairs. First, SQCK avoids the inconsistent repairs performed by e2fsck by ensuring that its queries are executed in the correct order; specifically, a file system structure is only repaired after the location of that structure has been validated. Second, SQCK can perform more interesting and complete repairs than e2fsck by combining information from multiple sources. For example, SQCK easily performs majority voting over superblock and group descriptor replicas to handle the case where the primary copy is corrupted. Finally, SQCK ensures that its repairs follow the same allocation policies as ext2/3 by laying out new blocks with the appropriate locality.

SQCK achieves this simplicity and completeness with little cost to performance. Our evaluation of the first-generation prototype of SQCK on top of the MySQL DBMS shows that SQCK can handle even large file system partitions with comparable performance to e2fsck. Overall, we believe that the SQCK-style declarative approach will lead to a new generation of simpler, more robust, and more complete file system checking and repair.

1.2.2 EDP: A Static Analysis Tool for Tracing Error-Codes Propagation

Our failure-policy analysis has shown that file systems are especially unreliable when the underlying disk system does not behave as expected [106]. Specifically, many modern commodity file systems have serious bugs and inconsistencies in how they handle errors from the storage system. However, the question remains unanswered as to why these fault-handling bugs are present.

Therefore, we investigate what we believe is one of the root causes of deficient fault handling: *incorrect error code propagation*. To be properly handled, a low-level error code (*e.g.*, an "I/O error" returned from a device driver) must be correctly propagated to the appropriate code in the file system. Further, if the file system is unable to recover from the fault, it should pass the error up to the application, again requiring correct error propagation. Without correct error propagation, any comprehensive failure policy is useless: recovery mechanisms and policies cannot

be invoked if the error is not propagated.

To analyze how error codes are propagated in file and storage system code, we have developed a static source-code analysis technique. Our technique, named *Error Detection and Propagation (EDP)* analysis [62], shows how error codes flow through the file system and storage drivers. EDP performs a dataflow analysis by constructing a function-call graph showing how error codes propagate through return values and function parameters.

We have applied EDP analysis to all file systems and three major storage device drivers (SCSI, IDE, and Software RAID) implemented in Linux 2.6. We find that *error handling is occasionally correct*. Specifically, we see that low-level errors are sometimes lost as they travel through the many layers of the storage subsystem: out of the 9022 function calls through which the analyzed error codes propagate, we find that 1153 calls (13%) do not correctly save the propagated error codes.

Our detailed analysis enables us to make a number of conclusions. First, we find that the more complex the file system (in terms of both lines of code and number of function calls with error codes), the more likely it is to incorrectly propagate errors; thus, these more complex file systems are more likely to suffer from silent failures. Second, we show how inter-module calls play a major part in causing incorrect error propagation. Third, we observe that I/O write operations are more likely to neglect error codes than I/O read operations. Finally, we find that many violations are not corner-case mistakes: the return codes of some functions are consistently ignored, which makes us suspect that the omissions are intentional. The last two observations hint that dealing with failures in the current infrastructure is hard, and hence failures are often ignored; in fact, the quoted developers' comments are found near where error codes are dropped. Therefore, our next approach is to revisit the need for a new reliability infrastructure.

1.2.3 I/O Shepherding: A New Reliability Infrastructure

Our final contribution stems from two needs: the need for proper storage fault handling and for flexible policies within the file system. First, current approaches bury fault handling features deep within the file system code, making both the intent and the realization of the approach to reliability difficult to understand or evolve. As a consequence, storage fault handling is buggy and inconsistent. This shows that reliability is a second-class citizen in commodity file systems.

Second, even if a perfectly working fault management system could be built, there is little consensus on the set of detection and recovery mechanisms that it should deploy, especially because file systems have classically been deployed in diverse environments. For example, a file system that runs on a desktop machine

with a single SATA drive is often the same file system that runs atop a hardware RAID consisting of high-end SCSI drives. Furthermore, file systems typically run underneath a wide variety of application workloads with differing needs. For example, desktop workloads may wish for high data reliability with reasonable performance, while web-server workloads with database support may desire the highest performance possible combined with modest reliability. As file systems are used in diverse settings, the best fault management strategy is likely a function of the environment (*e.g.*, how reliable the disks are) and the workload (*e.g.*, how much performance overhead can be tolerated, or how fault-tolerant the applications are). Unfortunately, systems today have a single approach built in, allowing little flexibility when deployed.

To fulfill the two needs above, we present the design, implementation, and evaluation of *I/O shepherding* [60], a new reliability infrastructure for file systems. I/O shepherding provides a simple yet powerful way to build robust reliability policies within a file system, and does so by adhering to a single underlying design principle: *reliability should be a first-class file system concern*. As a result, the reliability policies of a file system are well-defined, easy to understand, powerful, and simple to tailor to environment and workload.

The I/O shepherd achieves these ends by interposing on each I/O that the file system issues. The shepherd then takes responsibility for the "care and feeding" of the request, specifically by executing a *reliability policy* for the given block. Simple policies will do simple things, such as issue the request to the storage system and return the resulting data and error code (success or failure) to the file system above. However, the true power of shepherding lies in the rich set of policies that one can construct; I/O shepherding makes the creation of policies simple by providing a library of primitives that can be readily assembled into a fully-formed reliability policy.

A major challenge in implementing I/O shepherding is proper systems integration; it requires changes to the file system consistency management routines, layout engine, disk scheduler, and buffer cache, as well as the addition of thread support. Of these changes, the most important interaction between the shepherd and the rest of the file system is in the consistency management subsystem (*i.e.*, the journaling subsystem). Policies developed in the shepherd often add new on-disk state (*e.g.*, checksums, or replicas) and thus must also update these structures atomically. However, as mentioned earlier, we have found that journaling file systems suffer from a general problem of failed intentions (Section 1.1.3). To solve this major flaw, the shepherd incorporates *chained transactions*, a novel and more powerful transactional model that allows policies to handle unexpected faults dur-

ing checkpointing and still consistently update on-disk structures. The shepherd provides this support transparently to all reliability policies, as the required actions are encapsulated in various systems primitives.

We demonstrate how I/O shepherding enables simple but powerful policies by implementing an increasingly complex set of policies including sophisticated retry mechanisms, strong sanity checking, the addition of checksums to detect data corruption, and mirrors or parity protection to recover from lost blocks or disks. All these policies are written in a few lines of code; even complex policies can be implemented in less than 100 lines of code. As an implication, policies can be correctly implemented, and hence behave as desired under disk faults. We thus conclude that I/O shepherding is a powerful framework for building robust and efficient reliability features within file systems.

1.3 Summary of Contributions / Overview

Below is the summary of our contributions (and also how the rest of this dissertation is organized).

- **Background:** Chapter 2 provides a background on the storage stack, the partial disk failures that occur within, the ext2/3 file system data structures, and the type-aware fault-injection technique we have developed to analyze file systems.

- **Problems:** We begin presenting the first contribution of this dissertation in Chapter 3, which reports the impact of partial disk failures to three reliability components present in many modern file systems: the file system checker [61] (Section 3.1), failure policy [106] (Section 3.2) and journaling [60] (Section 3.3).

- **Solutions:** The next three chapters collectively present the second major contribution of this dissertation, which is a set of solutions to the problems we have found. Chapter 4 presents SQCK [61], a declarative file system checker. Chapter 5 presents EDP [62, 113], a static analysis tool for error-code detection and propagation. Finally, Chapter 6 presents I/O shepherding [60], a new reliability infrastructure for file systems.

- **Related Work:** Chapter 7 summarizes research efforts in analyzing system robustness and building more robust file systems. We also discuss other specific approaches related to our three solutions.

- **Conclusions and Future Work:** Chapter 8 concludes this dissertation, first summarizing our work and highlighting the lessons learned, and then discussing various avenues for future work that arise from our research, including extending our techniques and analyses to other data management systems such as databases [125]. Thus, we believe our contributions are generally applicable to many data management systems beyond file systems.

Chapter 2

Background

"We ... encounter a disk error."
– A comment in XFS (xfs_utils.c, line 182)

This chapter provides a background on various aspects integral to this dissertation. First, Section 2.1 explains the components of a storage stack, which is a complex layered collection of electrical, mechanical, firmware and software components. Second, Section 2.2 discusses failures that occur in the storage stack, describes specific partial disk failures that are addressed in this dissertation, and present statistics on the frequency of partial disk failures. Third, Section 2.3 gives an introduction on file system data structures as disk failures affect these structures. We will specifically describe the data structures of a popular file system, the Linux ext3 [141]; ext3 is the default file system in several distributions of Linux (*e.g.*, Red Hat). Although we heavily focus on ext3 file system in this dissertation, we believe our analyses and solutions are applicable to other file systems as well. Finally, Section 2.4 presents our fault-injection methodology developed in previous work [106]. This fault-injection methodology is crucial for two purposes: analyzing the robustness of file systems in dealing with disk failures and evaluating the robustness of our solutions.

2.1 The Storage Stack

Figure 2.1 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.

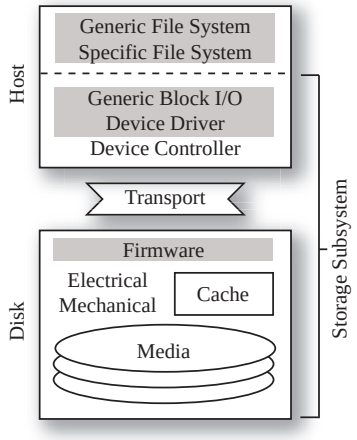


Figure 2.1: **The storage stack.** *We present a schematic of the entire storage stack. At the top is the file system; beneath are the many layers of the storage subsystem. Gray shading implies software or firmware, whereas white (unshaded) is hardware.*

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.*, buses). 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 [36]).

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 [84]) is positioned between the two.

2.2 Disk Failures

This section provides a background on disk failures, with a focus on *partial disk failures*. Partial disk failures implies that disks do not always fail in *whole-failure* mode where the disk is either working completely or not usable at all. Realistically, some parts of the disk can fail while some other parts are still working.

To understand partial disk failures, we first discuss the different sources of failures in the storage stack and then describe specific types of partial disk failures that we model. In reality, any element of the storage stack could cause a failure that appears as a "disk failure." We refer to such failures in other subsystem components as disk failures as well; most systems today cannot distinguish between failures that occur at different levels of the stack. Finally, we present some statistics on the frequency of partial disk failures.

2.2.1 Sources of Failures

This section presents different causes of partial failures in the storage subsystem. Almost all layers of the storage stack contribute to these partial failures.

Media: There are two primary problems that occur in the magnetic medium. First, the medium may have imperfections. These imperfections could either cause the medium to be poorly magnetized during writes, or could cause a "head crash", where the drive head contacts the surface momentarily. Second, a medium scratch could occur when a particle is trapped between the drive head and the media [117]. Such dangers are well-known to drive manufacturers, and hence today's 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 [11]. Media errors most often lead to permanent failure 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 caused by rotational vibration can misposition the drive head during writes, leaving blocks inaccessible or corrupted upon subsequent reads. "High-fly" writes, in which the gap between the disk head and the medium is too high, could cause data to be poorly written, thereby causing an ECC error when the sector is eventually read.

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 [88] or have memory leaks that lead to intermittent failures [137]. Other firmware problems can lead to poor drive performance [114]. 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 [143]. 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 [135] 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 [142]. 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 [53]. Others have also observed IDE protocol version problems that yield corrupt data [46]. 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 [28, 40, 133]. 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.

File system: Finally, at the very top of the storage stack, the file system itself may contain bugs that lead to silent data corruption. Recent research has identified various bugs various file system components including the journaling infrastructure, file-system mount code, and in failure-handling code [106, 145, 146, 147].

2.2.2 Types of Partial Disk Failures

In order to emulate the failures mentioned above, we need a more realistic model of disk failures. From the perspective of the file system, disk failures manifest as block-level failures; the disk interface abstracts the disk as a linear array of equal sized blocks each identified by a logical block number (LBN). Thus, in our model, failures manifest themselves in two specific ways:

- **Block failure:** One or more blocks are not accessible; often referred to as *latent sector faults* [78, 79]. As an implication, a read or a write to the block will fail.
- **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 model*, to emphasize that pieces of the storage subsystem can fail. We now discuss two key elements of the fail-partial model: the transience and locality of failures.

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.

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 [79]. For example, a scratched surface or thermal asperity can render a number of contiguous blocks inaccessible. However, all failures do not exhibit locality; for example, a corruption due to a misdirected write may impact only a single block.

2.2.3 Frequency of Failures

Until recently, there was very little data on how often partial disk failures arose in modern storage systems. Although there was much anecdotal information [18, 131, 143], and a host of protection techniques that systems employ to handle such corruptions [86], there was little hard data.

Recently, Bairavasundaram *et al.* performed the first large-scale study on partial disk failures [13]. they analyzed data collected from production systems over 32 months across 1.53 million disks (both nearline and enterprise class). The find that a total of 3.45% of 1.53 million disks developed latent sector errors over a period of 32 months. The also find that latent sector errors affect a significant percentage of a disk drive model (*e.g.*, up to 20% of the drives of a SATA disk model).

In a subsequent study, Bairavasundaram *et al.* also demonstrated that corruption does indeed occur across a broad range of modern drives [14]. In that study of 1.5 million disk drives deployed in the field, the authors found more than 400,000 blocks have checksum mismatches over three years. They also found that nearline disks develop checksum mismatches an order of magnitude more often than enterprise class disk drives. Furthermore, checksum mismatches within the same disk show high spatial and temporal locality, and checksum mismatches across different disks in the same storage system are not independent. This shows that corruption takes place, and systems must be prepared to handle it.

2.3 Ext3 Data Structures

To analyze how file systems deal with disk failures, we want to emulate the block-level failures mentioned in the previous section and observe how file systems react to the failures. In emulating block-level failures, we do not use a random fault-injection approach. Rather, we use specific file system knowledge (*i.e.*, file system data structures) to enhance our fault-injection methodology. In this section, we first give an introduction on the data structures of a popular file system, Linux ext3 [140, 141]. In the next section, we show how we utilize this knowledge to enhance our fault-injection methodology.

Ext3 is built as an extension to the ext2 file system [27]; ext2 and ext3 use the same data structures. The only difference is ext3 employs journaling (or write-ahead logging) to perform write-atomicity [140].

Figure 2.2 depicts the ext2/3 on-disk layout. In this organization (which is loosely based on FFS [92]), the disk is split into a number of *block groups*; within each block group are bitmaps, an inode table, and data blocks. Each block group also contains a redundant copy of crucial filesystem control informations such as the superblock and the group descriptors.

The *superblock* contains important layout information such as inodes count, blocks count, and how the block groups are laid out. Without the information in the superblock, the file system cannot be mounted properly.

A *group descriptor* describes a block group. It contains information such as the

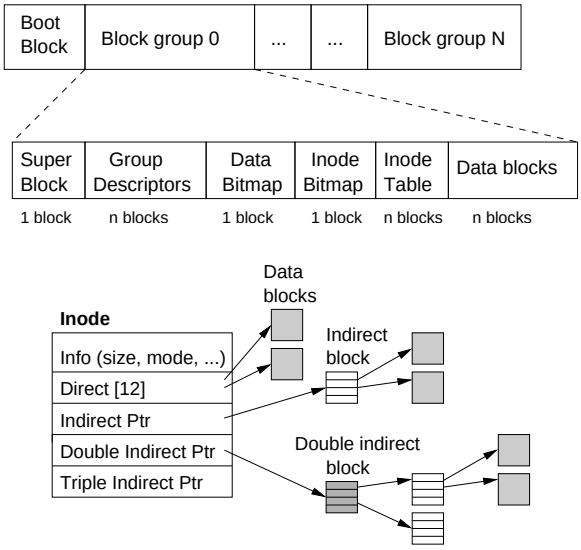


Figure 2.2: **Ext2/3 Layout.** *The upper picture shows the layout of an ext2/3 file system. The disk address space is broken down into a series of block groups (akin to FFS cylinder groups), each of which is described by a group descriptor and has bitmaps to track allocations and regions for inodes and data blocks. The lower figure shows the organization of an inode. An ext2/3 inode has twelve direct pointers to data blocks. If the file is large, indirect pointers are used.*

location of the inode table, block bitmap, and inode bitmap for the corresponding group. In addition, it also keeps track of allocation information such as the number of free blocks, free inodes, and used directories in the group.

An *inode table* consists of an array of *inodes*, and it can span multiple blocks. An inode can represent a user file, a directory, or other special files (*e.g.*, symbolic link). An inode mainly contains file attributes (*e.g.*, size, access control list) and pointers to its data blocks. An inode has 12 direct pointers to its data blocks. If its data needs more blocks, the inode will use its indirect pointer that points to an *indirect block* which contains pointers to data blocks. If the indirect block is not enough, the inode will use a *double indirect block* which contains pointers to indirect blocks. At most, an inode can use a triple indirect block which contains pointers to double indirect blocks.

A *data block* can contain user's data or directory entries. If an inode represents a user file, its data blocks contain user's data. If an inode represents a directory, its data blocks contain directory entries. Directory entries are managed as linked

lists of variable length entries. Each directory entry contains the inode number, the entry length, the file name and its length.

2.4 Type-Aware Fault Injection

In this section, we describe a fault-injection technique that we developed in previous work [106] to uncover file system behaviors in responding to disk failures. We have used this methodology as a basic framework in uncovering many reliability problems in file systems (Chapter 3) and evaluating our solutions to the problems (Chapter 4 and Chapter 6). This section provides a brief outline of the methodology; more details are described in the corresponding chapters.

To emulate block-level failures, our approach is to inject faults just beneath the system under test and observe how the system responds. If the responses are entirely consistent within a system, this could be done quite simply; we could run any workload, fail one of the blocks that is accessed, and conclude that the response to this block failure fully demonstrates the reaction of the system. However, systems are in practice more complex: they employ different techniques depending upon the operation performed and the type of the faulty block. For example, upon a write failure of a data block, the file system can simply propagate the failure to the application, while upon the superblock, the file system might want to retry the write or remap the superblock. Therefore, we must trigger all these interesting cases. Our challenge is to coerce the system down its different code paths to observe how each path handles failure. This requires that we run workloads exercising as many code paths as possible in combination with induced faults on all data structures.

Type Awareness: Many standard fault injectors [25, 120] fail disk blocks in a *type oblivious* manner; that is, a block is failed regardless of how it is being used by the system. However, repeatedly injecting faults into random blocks and waiting to uncover new aspects of the system's reactions would be a laborious and time-consuming process, likely yielding little insight. The key idea that allows us to test a system in a relatively efficient and thorough manner is *type-aware fault injection*. With type-aware fault injection, we fail blocks of a specific type (*e.g.*, an inode block in a file system). Type information is crucial in reverse-engineering the system's responds, allowing us to discern the different strategies that the system applies for its different data structures. The disadvantage of our type-aware approach is that the fault injector must be tailored to each system. However, we believe that the benefits of type-awareness clearly outweigh these complexities.

Context Awareness: Our goal in fault injection is to exercise the system as thoroughly as possible, following as many internal code paths as possible. We be-

lieve that different code paths using the same data structures may not respond to failure in a consistent manner. Therefore, we use a suite of workloads that stress the system in different ways. These workloads are fine-grained; each workload performs a very specific action, often corresponding to a single system call (*e.g.*, `open` of a file). Each system under test also introduces special cases that must be stressed. For example, in the case of the ext3 file system, the 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.

Mechanism: Our mechanism for injecting faults is to use a software layer directly beneath the system (*e.g.*, a pseudo-device driver in Linux). This layer injects both block read and write errors, and can also corrupt contents of disk blocks. By injecting failures just below the 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 [25], 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 [110]. However, given that we are characterizing how a specific file system responds to partial disk failures, we believe this is the correct layer for fault injection.

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

Chapter 3

Analysis of File System Reliability Components

> *"Error, skip block and hope for the best."*
> – A comment in ext3 (namei.c, line 880)

File systems have an important task: managing data *reliably*. Unfortunately, in a world of imperfect software and hardware, many problems arise that lead to data loss. In order to manage data reliably, file systems have to cope with many kinds of problems such as disk failures, crashes, file system bugs, and many more. To deal with these problems, file systems are typically equipped with many reliability components, each handling a certain kind of problem. In this chapter, we look into three reliability components present in many modern file systems and show in what ways each of the components is unreliable.

More specifically, Section 3.1 first describes the *file system checker (fsck)*, an important repair utility that fixes file system inconsistency (*e.g.*, due to bugs or disk corruption). In this section we unearth many weaknesses of a popular checker (*e.g.*, some repairs are incorrect, making the repaired file system more corrupted). Section 3.2 then looks into *failure policy*, the file system component responsible for dealing with disk failures. Our findings show that the failure policies of several commodity file systems are broken (*e.g.*, some disk failures are ignored, error-codes are dropped incorrectly, and many other problems). Finally, Section 3.3 describes *journaling*, a mechanism that guarantees write atomicity in the presence of system crashes. We have found that the current journaling framework does not work correctly when partial disk failures come into the picture.

To build a new generation of robust file systems, solutions to these problems are needed. Thus, Section 3.4 preliminarily introduces our solutions which are presented more extensively in following chapters.

3.1 Analysis of File System Checker

A file system checker, also known as fsck, is historically used to repair file system inconsistency caused by system crashes. When a file system update takes place, a set of blocks is written to the disk. Unfortunately, if the system crashes in the middle of the sequence of writes, the file system is left in an inconsistent state. To repair the inconsistency, earlier systems such as FFS [92] and ext2 [27] scan the entire file system and perform integrity checks using fsck [93] before mounting the file system again. This scan is a time-consuming process and can take several hours for large file systems.

To alleviate the long scan, modern file systems employ write-ahead logging or journaling [52]. By forcing journal updates to disk before updating complex file system structures, this write-ahead logging technique enables efficient crash recovery; a simple scan of the journal and a redo of any incomplete committed operations bring the file system to a consistent state.

Although journaling alleviates the need for fsck upon system crashes, fsck is still widely used today. The reason is that despite the best efforts of the file and storage system community, file system images become corrupt and require repair. In particular, problems with many different parts of the file and storage system stack can corrupt a file system image: disk media, mechanical components, drive firmware, the transport layer, bus controller, and OS drivers [13, 14, 46, 53, 106, 135]. Since file systems do not usually contain the machinery to fix corruptions themselves [15, 106], there is still a broad need for robust file system checkers.

Since fsck is a crucial repair utility, it should be robust in handling all possible corruption scenarios. More specifically, the repair process should have the following goals:

- **Consistent:** The explicit purpose and goal of fsck is to always create a consistent file system. All possible corruptions should be repaired such that the file system is usable.

- **Information-complete:** A file system usually contains some explicit redundancies (*e.g.*, multiple copies of superblock) and some implicit redundancies (*e.g.*, double-linked pointers connecting parent and child directories). Fsck should use this redundant structural information to perform consistency

checks. Thus, we define a repair to be information-complete if it reconstructs the file system to match the original file system to the greatest extent possible given the information available on disk. The notion of an information-complete repair is needed because a repair can easily create a consistent, but useless file system by simply removing all of the contents.

- **Policy-consistent:** A repair to be policy consistent must follow the same policies as the original file system; for example, if the ext3 file system allocates data blocks in the same group as its corresponding inode, then its checker should as well.

- **Secure:** Since a file system could be used by multiple users, a repair should not leak information from one user to another. For example, when a data block is accidentally shared by two users, a user's file should not be repaired to contain data from the other user's file.

To evaluate the robustness of file system checkers, this section presents our evaluation of a popular file system checker, e2fsck (the Linux ext2/3 checker). Section 3.1.1 gives a brief overview of the checks and repairs performed by e2fsck. Section 3.1.2 presents the methodology. Finally, our evaluation in Section 3.1.3 shows that e2fsck has many weaknesses that do not satisfy the goals mentioned above.

3.1.1 Ext2 Fsck Overview

Given both its popularity and our ability to access its source code, we focus on the file system checker for ext2/3, e2fsck. The purpose of the e2fsck utility is to check and repair the data structures of an ext2/3 file system on disk; in the ideal case, the repaired file system is readable, writable, and contains all of the directories, files, and data of the original file system.

E2fsck is a non-trivial piece of code: it contains more than 16 thousands lines of C code and can identify and return 269 different error codes. Its checks and repairs are performed in six different phases [27] as listed in Table 3.1.

In pass 0, e2fsck checks the consistency of the superblock. The fields in the superblock are crucial for obtaining the group layouts. Since ext2/3 file system replicates the superblock across block groups, all copies of the superblock can be cross-checked.

In pass 1, e2fsck iterates over all of the inodes and performs checks over each inode in isolation; these checks do not require any cross-checks to other file system structures. Examples of such checks include making sure the file mode is valid and

#	Checks Performed
28	**Phase 0: Check consistency in the superblock**
23	*Field check:* Check all superblock fields (*e.g.*, fs size, inode count, groups count, mount/write time)
3	*Range check:* Ensure pointers to block bitmap, inode bitmap, inode table are in the group
2	*Special feature:* Check resize inode feature
35	**Phase 1: Scan and check inodes and block pointers**
9	*Bad block:* Check fields of bad-block inode; ensure superblocks and group descriptors in healthy blocks
18	*Inode structure:* Check fields (*e.g.*, mode, time, size) of different inodes (*e.g.*, root, reserved, boot load)
1	*Range check:* Ensure direct and indirect pointers point within the file system
7	*Conflicts:* Ensure no conflict among block pointers (*e.g.*, two inodes should not share a block)
38	**Phase 2: Scan and check all directory entries**
16	*Directory:* Check each has '.' and '..' entry, '.' points to itself, does not have missing block, fields of dir inode consistent (*e.g.*, acl, fragment size)
9	*Dir Entry:* Check each entry has correct name length, each points to an in-range inode, record length valid, filename contains legal characters
5	*Pathnames:* Each entry points to used inode, does not point to self, does not point to inode in bad block, does not point to root, dir has only one parent
8	*Special inodes:* Check device inodes and symlinks
6	**Phase 3: Ensure all directories are connected to the file system tree**
3	*lost+found*: Ensure lost+found directory is valid and ready to be populated
3	*Reattach*: Reattach orphan directory to lost+found
3	**Phase 4: Fix reference counts and reattach zero-linked file to lost+found**
11	**Phase 5: Check block and inode bitmaps against on-disk bitmaps**
121	**Total**

Table 3.1: **Repairs performed by e2fsck.** *The table summarizes the 121 repairs performed by the e2fsck.*

that all of the data block pointers point to valid block numbers. If e2fsck notices data blocks that are claimed by more than one inode, it resolves the conflict by

cloning the shared blocks by default. In this pass, the checker also keeps track of blocks and inodes that are marked as being used.

In pass 2, e2fsck checks directory entries in isolation; since directory entries do not span disk blocks, each directory block can be checked individually. The directory blocks are checked to make sure that the directory entries are valid and contain references to inode numbers that are in use. For the first directory block in each directory inode, e2fsck verifies that the "." and ".." entries exist, and that the "." entry points to the current directory. Pass 2 also records each sub-directory inode that is pointed to by multiple parent directories.

In pass 3, the directory connectivity is checked; all directories should be accessible from the root inode. At this time, the ".." entry for each directory is also checked. Any directories not reachable from the root are attached to the `/lost+found` directory.

In pass 4, e2fsck checks the reference counts for all inodes by comparing the stored link counts on the disk and the computed link counts from the earlier passes.

Finally, in pass 5, e2fsck checks the validity of the file system summary information such as the block and inode bitmaps. It compares the block and inode bitmaps which were constructed during the previous passes against the actual bitmaps stored on the disk.

3.1.2 Methodology

To begin to understand the complex runtime behavior of e2fsck, we explore how e2fsck repairs a single on-disk corruption. Given that it is infeasible to exhaustively corrupt every data structure field to every possible value, we limit our scope to corrupting on-disk pointers.

A pointer-corruption study is especially difficult because it is nearly impossible to corrupt every pointer on disk to every possible value in a reasonable amount of time. Often, the solution has been to use random values [120]. This approach suffers from two problems: (a) a large number of corruption experiments might be needed to trigger the interesting scenarios, and (b) use of random values makes it more difficult to understand underlying causes of observed behavior.

To address both problems, we use *type-aware pointer corruption* (TAC), which is an extension of type-aware fault injection described in Section 2.4. Type-awareness reduces the exploration space for corruption experiments by assuming that system behavior depends only on two types: (i) the type of pointer that has been corrupted, and (ii) the type of block that it points to after corruption. Examples are (i) corrupting File A's data pointer is the same as corrupting File B's data pointer, and (ii) corrupting a pointer to refer to inode-block P is the same as corrupting it to refer

Block pointer types:	Boot, superblock, group descriptor, block bitmap, inode bitmap, inode table, single indirect, double indirect, triple indirect, directory, used data, free data, out of range
Index pointer types:	Directory inode, file inode, free inode, out of range inode

Table 3.2: **Block-pointer and index-pointer types in ext2.** *The table shows different types of block pointers and index pointers in ext2.*

to inode-block Q (if all inodes in P and Q are for user files). This approach is motivated by the fact that code paths that exercise the same types of pointers are the same, and disk blocks of the same type of data structure contain similar contents. Thus, TAC greatly reduces the experimental space while still covering almost all of the interesting cases. Also, by its very design, this approach attaches file system semantics to each experiment, which can be used to understand the results.

Our TAC model reflects the state of a file system on functioning hardware that experienced a corruption event in the past:

- Exactly one pointer is corrupted for each experiment. The rest of the data is not corrupted. Also, other faults like crashes or sector errors are not injected.
- We emulate pointer corruptions that are persistent. The corruption is persistent because simply re-reading the pointer from disk will not recover the correct value.

For ext2, we define two classes of pointers: block and index pointers. First, a block pointer contains a physical block number; for example, data block pointers in inodes contain the block numbers of corresponding data blocks. Second, an index pointer contains an index into a table; for example, an inode index picks an entry in the inode table within a block group. Table 3.2 lists the types of block and index pointers in ext2. We use these types to change the value of a pointer. For example, we could corrupt the data-block pointer of an inode to point to the primary superblock, an inode table, or an out-of-range block.

To examine the results, we use the the goals introduced earlier in this chapter, *i.e.*, repairs should be consistent, information-complete, policy-consistent, and secure. We use the debugfs utility [1] to compare the original file system and the repaired file system, and check manually if any of the goals have been violated. For example, to check if e2fsck performs consistent repairs, we check if the repaired

file system can be mounted properly. To compare how much the repaired file system matches the original file system, we manually analyze the differences of their data structures, especially those that are affected by the injected corruptions. This manual process is manageable and not time-consuming since the file system that we corrupt is relatively small.

3.1.3 Results

In this section, we first describe our visual representation of the results and then distill the results into higher-level observations. Table 3.3 shows the results of injecting block-pointer corruptions. Each row presents the results of corrupting one pointer (*e.g.*, indirect pointer) and is divided into 13 columns, each corresponding to different block types that we have introduced in Table 3.2. Note that the number of columns (*i.e.*, all block types) is higher than the number of rows (*i.e.*, corruptable block pointers). This is because ext2 does not explicitly store all types of block-pointers on the disk. For example, ext2 does not store group-descriptor pointers; it uses fixed group size stored in the superblock to locate the group descriptor block for each group.

Each cell (row X and column Y) represents an experiment where a block pointer X is corrupted to a Y-value. Each cell is marked with one or more symbols representing our observations when the pointer for its row is corrupted with the column value. For cells where the row and column have the same type, the pointer is corrupted to some other block that has the same type. For example, a pointer to a data block can be corrupted to point to some other data block that belongs to a different user. A cell with a dot (**.**) represents that e2fsck performs the correct repair.

Table 3.4 shows the results of injecting index-pointer corruptions. The format of this table is similar to Table 3.3.

From our fault-injection experiments, we find that e2fsck fails along the four different axes that we setup earlier: e2fsck sometimes performs *inconsistent* (marked with **C**), *information-incomplete* (**I**), *policy-inconsistent* (**P**), and *insecure* (**S**) repairs. We now describe the specific behaviors of e2fsck that lead to these problems.

Inconsistent Repair (C): *Clears "Indirect Blocks" Incorrectly.* Fundamentally, e2fsck checks and repairs certain pointers in an incorrect order; as a result, e2fsck can itself corrupt arbitrary data on disk, even the superblock. Specifically, e2fsck clears block pointers that fall out of range of the file system inside indirect blocks without first checking that the pointer to the indirect block itself is correct. Thus, if an indirect pointer was corrupt, e2fsck may clear the block that the indirect pointer incorrectly refers to. This clearing

		1. Boot	2. Superblock	3. Group descriptor	4. Block bitmap	5. Inode bitmap	6. Inode table	7. Single indirect	8. Double indirect	9. Triple indirect	10.Directory	11.Used data	12.Free data	13.Out of range
1.	Block bitmap	.	.	.	.	.	.	.	.	.	.	.	.	.
2.	Inode bitmap	.	.	.	.	.	.	.	.	.	.	.	.	.
3.	Inode table	**I**	**I**	**I**	**I**	**I**	.	**I**	**I**	**I**	**I**	**I**	**I**	**I**
4.	Single indirect	.	**C**	**C**	.	.	**C**	.	.	.	**C**	**C**	**C**	.
5.	Double indirect	.	**C**	**C**	**C**	**C**	**C**	**C**	.	.	**C**	**C**	**C**	.
6.	Triple indirect	.	**C**	**C**	**C**	**C**	**C**	**C**	**C**	.	**C**	**C**	**C**	.
7.	Directory	.	.	.	.	.	.	**PS**	**PS**	**PS**	**PS**	**PS**	.	.
8.	Data	.	.	.	.	.	.	**S**	**S**	**S**	**PS**	**S**	.	.

Symbols: **C**: Inconsistent repair **I**: Information-incomplete repair
P: Policy-inconsistent repair **S**: Insecure-repair **Dot (.)**: Correct repair

Table 3.3: **Results of block-pointer corruptions.** *This figure shows how e2fsck responds to block-pointer corruptions. Each row characterizes the behavior for the given pointer. Each cell in a row is marked with the behavior observed for the given pointer when it is corrupted with the value of that column.*

can lead to arbitrary corruptions of file, directory, and meta-data in the file system; most notably, if the file system contains only a single superblock, the file system can even be unmountable after running e2fsck (*e.g.*, row #4 and column #2 of Table 3.3).

Information Incomplete (I): *False Parenthood.* e2fsck does not always use all of the information available to it regarding directories. One example is the case where an inode index within a directory is corrupted to point to a different valid directory inode (row #1 and column #1 of Table 3.4). This situation is illustrated in Figure 3.1. If a directory entry is corrupted to point to another target directory (parts a and b), the e2fsck repair might move the target directory to the wrong parent (part c).

We emphasize that enough information is available in an ext2 file system for e2fsck to make the correct repair: each directory contains an entry for

		1. Directory inode	2. File inode	3. Free inode	4. Out of range inode
1.	Directory inode	**I**	.	.	.
2.	File inode	**I**	.	.	.

Table 3.4: **Results of index-pointer corruptions.** *This figure shows how e2fsck responds to index-pointer corruptions. Each row characterizes the behavior for the given pointer. Each cell in a row is marked with the behavior observed for the given pointer when it is corrupted with the value of that column.*

its parent (denoted "..."). To perform an information complete repair, e2fsck could simply observe this entry to keep the target directory with its correct parent and to reattach the lost directory to its parent instead of moving it to lost+found. In general, the directory hierarchy in ext2 contains much more information than is being used currently in e2fsck.

Information Incomplete (I): *Ignores Replicas of Inode Table Pointers.* Ext2 contains replicas for important meta-data, such as pointers to the inode tables; however, e2fsck does not always use this redundant information. For example, when an inode-table pointer becomes corrupted and points to other blocks (*e.g.*, block bitmap) inside the same block group (*e.g.*, row #3 and column #4 of Table 3.3), e2fsck assumes the pointer is correct; e2fsck then finds that the "inodes" are not valid. For consistency, e2fsck removes the corresponding directories and files from the directory tree; if this group contains the root directory, the file system is trivially consistent with no directories. Again, enough information is available for e2fsck to make the correct repair: each inode-table pointer is replicated across block groups; e2fsck should check that all block groups agree on these important values.

Policy Inconsistent (P): *Different Layout.* e2fsck does not allocate blocks on disk with the same layout policy as ext2; as a result, e2fsck can fragment files and directories, degrading the future performance of file system operations. For example, when e2fsck detects that the same data block is pointed to by both

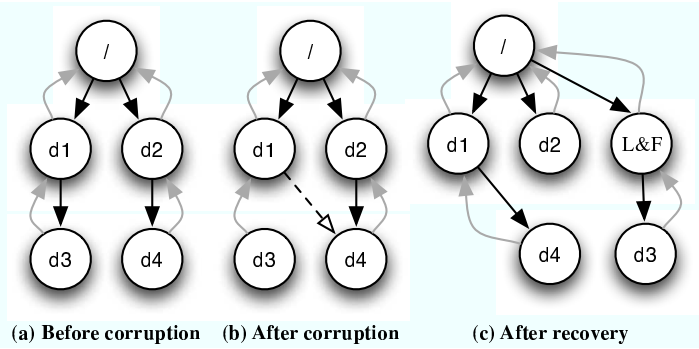


Figure 3.1: **The false parenthood problem.** *This figure shows the problem in the recovery done by e2fsck for corruption in directories. Each node is a directory in the file system. For clear understanding, we use dotted backpointers to show the parent for each directory as present in the '..' entry for that directory. Part (a) of the figure shows the initial file system structure. Part (b) shows the file system structure after corruption. We inject this fault where the entry for dir3 in dir1 is corrupted to point to the inode of dir4. After recovery by e2fsck, the dir1 claims dir4 and the original parent child link between dir2 and dir4 is deleted. This results in totally different structure of the file system after recovery as shown in part (c). For convenience we show the lost+found (L&F) directory only in the final structure.*

a directory and a file (row #8 and column #10 of Table 3.3), e2fsck clones the block by allocating a new block for the file and retaining the old block for the directory. To perform a policy-consistent repair, e2fsck should allow the closer inode to retain the original data block.

Insecure Repair (S): *Copies Data Freely.* Whenever e2fsck discovers that two pointers refer to the same block (row #8 and column #11 of Table 3.3), e2fsck clones the block. However, this policy has the potential to leak private information. For example, if a data block is shared by two inodes, one in the `/home/userA` directory and one in the `/root` directory, we might want to remove the pointer from `userA` and keep the one from the root.

3.1.4 Summary: The Need for a New Fsck Framework

We have found that e2fsck has a number of problems in how it performs repairs; we note that these problems are not simple implementation bugs, but are fundamental

design flaws. In particular, it is difficult for e2fsck to combine the many pieces of information available (*e.g.*, replicas of pointers and parent directory entries) and to ensure that all checks and repairs are done in the correct order.

We believe one of the reasons why these problems exist is because e2fsck is a complex piece of imperfect code written in more than ten thousands lines of low-level C code, which is hard to reason about. Other than e2fsck, other checkers are unfortunately written in the same way; the ReiserFS [5] checker performs 156 cross-checks and the corresponding repairs in 11 thousands lines of C code, and the XFS [132] checker performs 344 in 22 thousands lines. To the best of our knowledge, we have not found any available tools that the file system developers use to verify the correctness of these checkers. Thus, although so far we have only analyzed one checker, we believe these other checkers might have the same weaknesses as in e2fsck.

To build a new generation of robust and reliable file system checkers, we believe a new approach is required. Chapter 4 presents our new approach, SQCK, in which the high-level intent of a checker can be specified in a clear and compact manner; further, the description of the intent is cleanly separated from its low-level implementation and how it is optimized.

3.2 Analysis of Failure Policy

In this section, we turn our attention to how running file systems deal with disk failures. As mentioned in Section 2.2, storage systems fail due to a large number of reasons such as latent sector faults, silent block corruption, and performance glitches. 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 [78, 117] to proactively detect and subsequently correct latent sector faults 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 faults [30]. Finally, highly-reliable systems (*e.g.*, Tandem NonStop) utilize end-to-end checksums to detect when block corruption occurs [18].

The above said failure characteristics (latent sector faults and block corruption) and our knowledge about reliable high-end systems raise the question: *how do commodity file systems handle disk failures?* To answer this question, our main objective is to determine which detection and recovery techniques each file system uses and the assumptions each makes about how the underlying storage system can fail. The detection and recovery mechanisms employed by a file system define its

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

failure policy. 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.

To describe the failure policy of a file system, we begin by presenting the IRON taxonomy [106] of failure-handling policies (Section 3.2.1). This taxonomy serves as an overview of the different techniques that may be used by the file system to handle partial disk failures. Section 3.2.2 then describes our methodology details. Finally, in Section 3.2.3, we present the results of our analysis of four commodity file systems (Linux ext3, ReiserFS, JFS, XFS, and Windows NTFS).

3.2.1 IRON Taxonomy

We now describe the IRON taxonomy of failure-handling strategies that we developed in previous work [106]. IRON stands for "Internal RObustNess"; it focuses on failure-handling strategies to be used, not across disks as is common in RAID systems, but within a single disk. We have found from experience that this taxonomy can be used to sufficiently describe the failure-handling strategies of various file systems.

To cope with the failures in modern disks, file systems include machinery to both *detect* (Level D) partial faults and *recover* (Level R) from them. Tables 3.5 and 3.6 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 [8, 144].

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_{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 3.6: **The Levels of the IRON Recovery Taxonomy.**

Levels of Detection

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

$Zero$: The simplest detection strategy is none at all; the file system assumes the disk works and does not check return codes. As we will see in Section 3.2.3, this approach is surprisingly common (although often it is applied unintentionally).

$ErrorCode$: A more pragmatic detection strategy that a file system can implement is to check return codes provided by the lower levels of the storage system. For example, if there are low-level I/O failures, `EIO` error-codes are often returned.

$Sanity$: With sanity checks, the file system verifies that its data structures are consistent by verifying individual fields (*e.g.*, that pointers are within valid ranges) or verifying 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 [108]. By checking whether a block has the correct type information, a file system can guard against some forms of block corruption.

$Redundancy$: 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 [18] and has recently been applied to improve security as well [100, 126]. 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 [18, 143]; 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 [22], parity [99, 103] and other error-correction codes [90], 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 [103].

Levels of Recovery

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

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

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

$Stop$: One way to recover from a disk failure is to stop the current file system activity. This action can be taken at many different levels of granularity. At the coarsest level, one can crash the entire machine. One positive feature is that this recovery mechanism turns all *detected* disk failures into fail-stop failures and likely preserves file system integrity. However, crashing assumes the problem is transient; if the faulty block contains repeatedly-accessed data (*e.g.*, a script run during initialization), the system may repeatedly reboot, attempt to access the unavailable data, and crash again. At an intermediate

level, one can kill only the process that triggered the disk fault and subsequently mount the file system in a read-only mode. This approach is advantageous in that it does not take down the entire system and thus allows other processes to continue. At the finest level, a journaling file system can abort only the current transaction. This approach is likely to lead to the most available system, but may be more complex to implement.

Retry: A simple response to failure is to retry the failed operation and recent work shows that file systems do recover from most number of disk errors by simply retrying [51]. Retry can appropriately handle transient errors, but wastes time retrying if the failure is indeed permanent.

Repair: If a 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.

Remap: 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 [103], 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 [90]) could also be used, a subject worthy of future exploration.

3.2.2 Methodology

This subsection describes our methodology to uncover the failure-policies of several commodity file systems that we analyzed. As described in Section 2.4, our approach is to inject faults just beneath the file system and observe how the file system reacts. Overall, our failure policy analysis consists of three major steps: create the right workload, inject faults, and deduce failure policy. We describe each of these steps in detail.

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 3.7: **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).*

Applied Workload

Our workload suite contains two sets of programs that run on UNIX-based file systems (fingerprinting NTFS requires a different set of similar programs). The first set of programs, called *singlets*, each focus upon a single call in the file system API (*e.g.*, `mkdir`). The second set, *generics*, stresses functionality common across the API (*e.g.*, path traversal). Table 3.7 summarizes the test suite.

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

Each file system under test also introduces special cases that must be stressed. For example, the ext3 inode uses an imbalanced tree with indirect, doubly-indirect, and triply-indirect pointers, to support large files; hence, our workloads ensure that sufficiently large files are created to access these structures. Other file systems have similar peculiarities that we make sure to exercise (*e.g.*, the B+-tree balancing code of ReiserFS). The block types of the file systems we test are listed in Tables 3.8

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

Table 3.8: **Ext3 Data Structures.** *The table presents the data structures of interest in ext3 file system. In the table, we list the names of the major structures and their purpose.*

Fault Injection

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

The errors we inject into the block write stream have three different attributes, similar to the classification of faults injected into the Linux kernel by Gu *et al.* [56]. The fault specification consists of the following attributes:

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

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

Transience: This determines whether the fault that is injected is a transient error

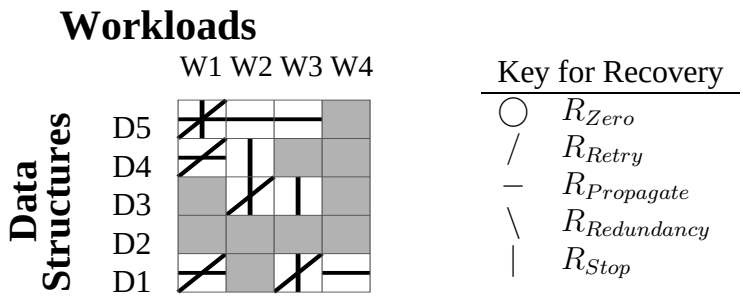


Figure 3.2: **Example Failure Policy Graph.** *The figure presents a sample failure policy graph. A gray box indicates that the workload is not applicable for the block type. If multiple mechanisms are observed, the symbols are superimposed.*

(*i.e.*, fails for the next N requests, but then succeeds) or a permanent one (*i.e.*, fails upon all subsequent requests).

Failure Policy Inference

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

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

We collect large volumes of results in terms of traces and error logs for each fault injection experiment we run. Due to the sheer volume of experimental data, it is difficult to present all results for the reader's inspection. We represent file system failure policies using a unique representation called *failure policy graphs*, which is similar to the one shown in Figure 3.2.

In Figure 3.2, we plot the different workloads on x-axis and the file system data structures on y-axis. If applicable, each <row, column> entry presents the IRON detection or recovery technique used by a file system. If not applicable (*i.e.*, if the workload does not generate the particular block type traffic), a gray box is used. The symbols are superimposed when multiple mechanisms are employed by a file system.

Next, we explain how the entries in Figure 3.2 must be interpreted by walking through an example. Specifically, consider the entry for workload "W1" and "D5" data structure. It has three symbols superimposed: retry (R_{Retry}), error propagation ($R_{Propagate}$) and finally, a file system stop (R_{Stop}). This means that whenever an I/O to "D5" fails during workload "W1", the file system first retries and if that fails, stops and propagates the error to the application.

We use failure policy graphs not only to present our analysis results but also throughout this thesis to represent the failure policies we craft in our solution (Chapter 6).

3.2.3 Results

We have performed a failure-policy analysis for four commodity file systems: ext3 [140], ReiserFS (version 3) [109], and IBM's JFS [20] on Linux and NTFS [124] on Windows; we have analyzed the impact of read errors, write errors, and corruption of entire disk blocks in these file systems. This analysis was done by four people [106]: Vijayan Prabhakaran (who analyzed ext3), Lakshmi Bairavasundaram (IBM JFS), Nitin Agrawal (ReiserFS), and Haryadi Gunawi (Windows NTFS). In this subsection, we primarily present the problems that Prabhakaran found in ext3 because we use this file system to evaluate our solution to the problems in Chapter 6. Thus, this subsection serves as an important background for Chapter 6. At the end of this subsection, we summarize the findings of the entire study [106].

Linux ext3

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

Table 3.9: **Keys for Detection and Recovery.** *The table presents the keys we use to represent the detection and recovery policies in file systems.*

Figure 3.3 shows Prabhakaran's findings for ext3 [106]. The figure presents the detection and reaction techniques used by ext3 to handle read, write, and corruption

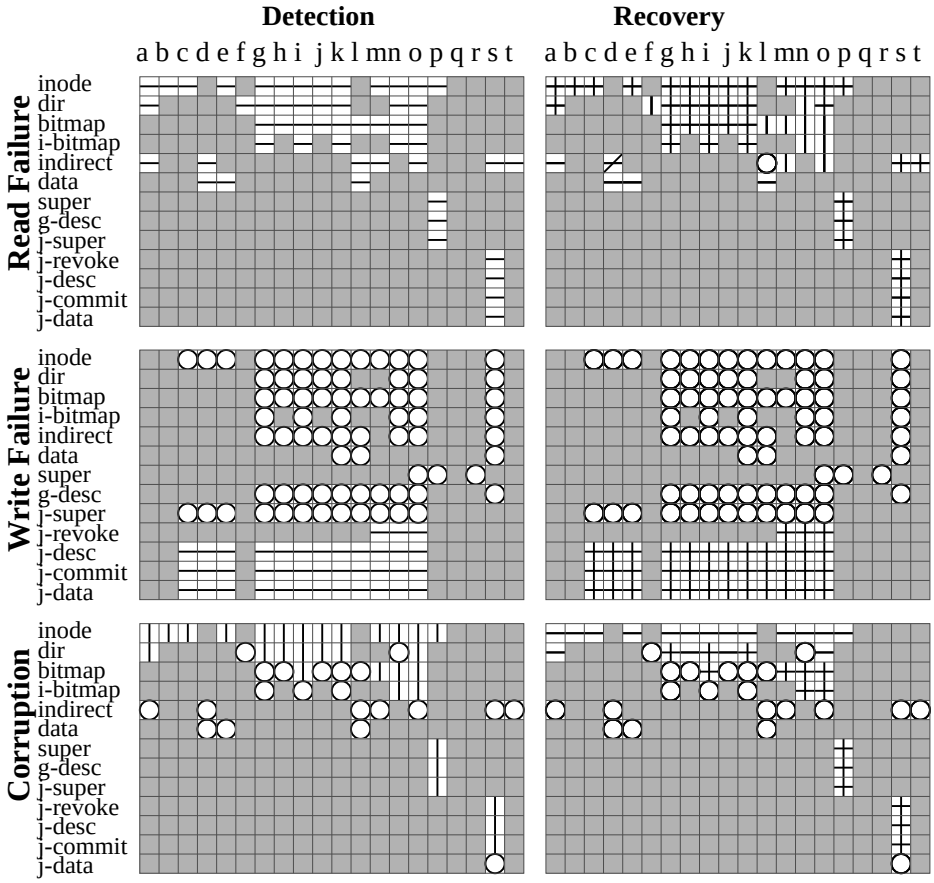


Figure 3.3: **Ext3 Failure Policies.** *The failure policy graphs plot detection and recovery policies of ext3 for read, write, and corruption faults injected for each block type across a range of workloads. The workloads are* **a:** *path traversal* **b:** *access,chdir,chroot,stat,statfs,lstat,open* **c:** *chmod,chown,utimes* **d:** *read* **e:** *readlink* **f:** *getdirentries* **g:** *creat* **h:** *link* **i:** *mkdir* **j:** *rename* **k:** *symlink* **l:** *write* **m:** *truncate* **n:** *rmdir* **o:** *unlink* **p:** *mount* **q:** *fysnc,sync* **r:** *umount* **s:** *FS recovery* **t:** *log write operations. A gray box indicates that the workload is not applicable for the block type. If multiple mechanisms are observed, the symbols are superimposed. The keys for detection and recovery are presented in Table 3.9. These ext3 failure policies were analyzed by Prabhakaran [106].*

failures. Each row in the set of figures corresponds to a data structure. Each column corresponds to a specific workload. The symbols in each cell correspond to how ext3 responds when the data structure for that row fails when accessed as a result of the workload for that column. Note that symbols corresponding to different policies may be superimposed.

Detection: Prabhakaran observed that, 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: Prabhakaran also found a number of bugs and inconsistencies in the ext3 failure policy. First, errors are not always propagated to the user (*e.g.*, `truncate` and `rmdir` fail silently). Second, ext3 does not always perform sanity checking; for example, `unlink` does not check the `linkscount` field before modifying it and therefore a corrupted value can lead to a system crash. Third, although ext3 has redundant copies of the superblock ($R_{Redundancy}$), these copies are never updated after file system creation and hence are not useful.

File System Summary

We now present a qualitative summary of each of the file systems we tested. Table 3.10 presents a summary of the techniques that each file system employs (excluding NTFS); because our analysis requires detailed knowledge of on-disk struc-

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

Table 3.10: **IRON Techniques Summary.** *The table depicts a summary of the IRON techniques used by the file systems under test. More check marks (√) indicate a higher relative frequency of usage of the given technique.*

tures, and not all NTFS structures are publicly documented, we could not completely analyze all NTFS failure policies.

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 reacts by reporting 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 failures. 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 reaction techniques. For detection, JFS sometimes uses sanity, sometimes checks error codes, and sometimes does nothing at all. For reaction, JFS sometimes uses available redundancy, sometimes crashes the system, and sometimes retries operations, depending on the block type that fails, the error detection and the API that was called.

NTFS: Persistence is a virtue. Compared to the Linux file systems, NTFS is the most persistent, retrying failed requests many times before giving up. It also seems to report errors to the user quite reliably.

Overall, we find that different file systems use different sets of policies to detect and react to partial disk failures. For example, JFS was only Linux file system that used some redundancy to recover (in the case of the superblock). Even in using the same policies, the degree to which a policy is used changes from one file system to another. For example, while all file systems employ retries to some extent, NTFS retries a failed operation many more times than the other file systems.

Technique Summary

Finally, we present three high-level observations of the techniques applied by all of the file systems to detect and recover from disk failures.

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

We note that inconsistency in and of itself is not problematic [38]; 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 [123]. What we are criticizing are inconsistencies that are undesirable (and likely unintentional); for example, as shown in Figure 3.3, when reading an indirect block fails, ext3 sometimes propagates the error to the user ($R_{Propagate}$), retries the operation (R_{Retry}), remounts the file system read-only (R_{Stop}), and ignores the failure (R_{Zero}), depending on where in the code the fault is detected.

After a closer source code inspection, we found that the root cause of illogical inconsistency is *failure policy diffusion*; the code that implements the failure policy is spread throughout the kernel. Figure 3.3 illustrates the scattered policy code in ext3. There are more than a hundred of places where the file system tries to handle I/O failures. One reason for this diffusion is that the file system tries to handle each fault where it arises in the code. Because I/Os

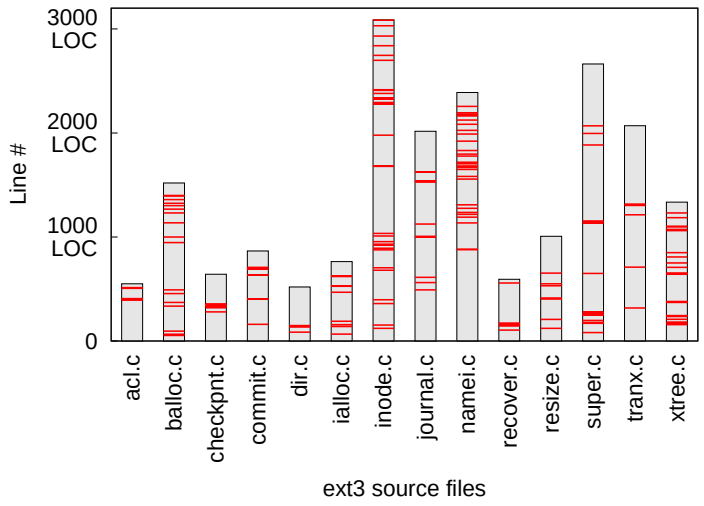


Figure 3.4: **Diffused policy code.** *The figure shows the scattered policy code in ext3 source code. The x-axis lists all the source files for Linux ext3 file system. The y-axis represents the lines of code. Each vertical bar represents how big is the file (in terms of LOC). The small horizontal lines appearing inside the bars represent the location of ext3's failure policies.*

are generated from many different locations within the file system, the fault-handling policy is also spread throughout the code; as other researchers have shown, diffused handling leads to policies that are likely to be inconsistent, buggy, and inflexible [83]. We pay attention to this problem and design a centralized failure handler for file systems, which we discuss in Chapter 6.

Detection and Recovery: Bugs are common. We also found numerous bugs across the file systems we tested, some of which are serious, and many of which are not found by other sophisticated techniques [147]. 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" [102]. 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. As an example, the ext3 code below shows a serious silent failure arises during file system recovery because an error code is silently dropped. `sync_blockdev` (line 2) is responsible in flushing the dirty buffer pages. It does the job by calling two other functions (line 4 and 5). When a low-level I/O failure occurs, these two functions will propagate `EIO` error codes via return values (line 4 and 5). `sync_blockdev` then correctly propagates the `EIO` error codes to the caller (line 7). Unfortunately, `journal_recover` neglects the error code propagated by `sync_blockdev` (line 12), leading to a silent failure during journal recovery.

```
 1 // fs/buffer.c
 2 int sync_blockdev() {
 3     ...
 4     ret = fm_fdatawrite(); /* PROPAGATE EIO */
 5     err = fm_fdatawait();  /* PROPAGATE EIO */
 6     if(!ret) ret = err;
 7     return ret;            /* RETURN EIO */
 8 }
 9 // jbd/recovery.c
10 int journal_recover() {
11     ...
12     sync_blockdev(); /* IGNORE EIO */
13     ...
14 }
```

The example above clearly shows that correct error propagation is an important aspect of a robust file system. To be properly handled, any fault must be correctly propagated to the code within the file system that is responsible for handling such fault. Further, if the file system is unable to recover from the fault, it may desire to simply pass the error up to the application, again requiring correct error propagation. Thus, an infrastructure to analyze how errors propagate should be a part of the file system developer's toolkit; with such tools, this class of error is easily discovered. In fact, in Section 5, we show how a static analysis tool that we have developed can find hundreds of errors in file systems and storage drivers.

3.2.4 Summary: The Need for a New Fault-Management Framework

The results of our failure policy analysis point us to two lessons: First, different file systems use different sets of policies to detect and react to partial disk failures.

This shows that there is no single best policy. However, given a file system, we are limited to the reliability policies that the file system provides; it is hard to modify its policies. Second, diffused handling causes policies to be inconsistent, buggy, and inflexible to change. In addition to that, as fault handlers are buried deep in the code, they generally do not have access to all of the contextual information about the failed request, thus can only implement a limited set of responses. Therefore, what we need is a new fault-management framework where we could deploy different sets of failure policies in a centralized fashion. With such locality, the hope is to make the policies more flexible, less buggy, and much easier to manage. These needs drive our I/O shepherding approach, presented in Chapter 6.

Furthermore, we need to solve the problem of incorrect error propagation or otherwise it could lead to serious problems (*e.g.*, misleading error-codes, wrong recovery actions, or even frustation for human debugging). Thus, building an infrastructure that unearths all instances of this problem and enables deeper root-cause analysis is an essential component in building a robust file system. One approach is via a static analysis. In Chapter 5, we present our technique, named *Error Detection and Propagation (EDP) analysis*, which shows how error codes flow through file systems and storage drivers.

3.3 Analysis of Journaling

In addition to disk failures, another failure that file systems need to handle is system crashes. When a file system update takes place, a set of blocks is written to the disk. If the system crashes in the middle of the sequence of writes, the file system is left in an inconsistent state. The idea of journaling is to ensure the atomicity of the writes despite the presence of crashes, specifically by recording some extra information on the disk in the form of a write-ahead log or a journal [52]. By forcing journal updates to disk before updating complex file system structures, this write-ahead logging technique enables efficient crash recovery; a simple scan of the journal and a redo of any incomplete committed operations bring the file system to a consistent state.

In this section, we first give an introduction to how journaling works (Section 3.3.1). Although the journaling approach works perfectly in anticipating crashes, we have found that journaling file systems suffer from a general *problem of failed intentions* (Section 3.3.2) when disk failures come in to the picture. Section 3.3.3 summarizes the significance of this problem as many modern file systems employ journaling and disk failures happen in practice.

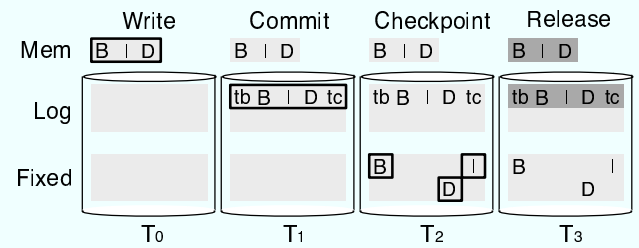


Figure 3.5: **Journaling Data Flow..** *The figure shows the series of actions that take place in data journaling mode. Both in-memory (top) and on-disk (bottom) states are shown.* **D** *is a data block,* **I** *an inode,* **B** *a bitmap,* **tb** *the beginning of a transaction, and* **tc** *the commit block. Darker gray shading indicates that blocks have been released after checkpointing.*

3.3.1 Journaling Basics

To describe the basics of journaling, we borrow the terminology of ext3 [140]. Journaling exists in three different modes (data, ordered, and write-back), each of which provides different levels of consistency. In data journaling, all traffic to disk is first committed to a log (*i.e.*, the journal) and then checkpointed to its final on-disk location. The other journaling modes journal only metadata, enabling consistent update for file systems structures but not for user data. In this section, we only illustrate how data journaling works.

Figure 3.5 illustrates the sequence of operations in data journaling when an application appends a data block **D** to a file. At time T_0, the data block **D** and bitmap **B** are updated in memory and a pointer to **D** is added to the inode **I**; all three blocks (**D, I, B**) now sit in memory and are dirty. At time T_1, the three blocks **D, I,** and **B** are wrapped into a *transaction* and the journaling layer *commits* the transaction (which includes markers for the start **tb** and end **tc** of the transaction) to the journal; the blocks are now marked clean but are still pinned in memory. After the commit step is complete, at T_2, the blocks are *checkpointed* to their final fixed locations. Finally, at T_3, the transaction is *released* and all three blocks are unpinned and can be flushed from memory and the log space reused. Note that multiple transactions may be in the checkpointing step concurrently (*i.e.*, committed but not yet released). If the system crashes, the file system will recover by replaying transactions in the journal that are committed but not yet released.

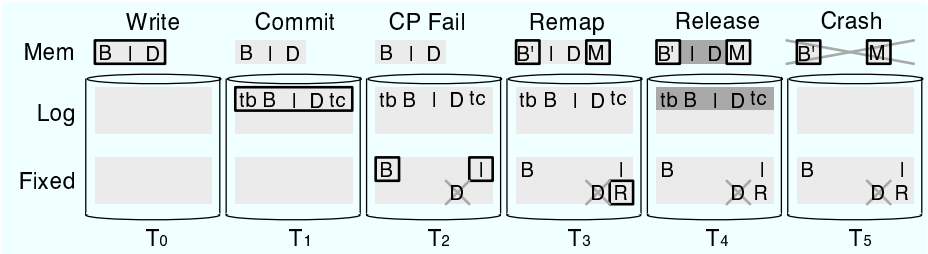


Figure 3.6: **Failed intentions.** *The figure illustrates how current journaling framework cannot deal with the problem of failed intentions.*

3.3.2 Failed Intentions

Although journaling works correctly for crashes, disk failures happen. In this case, we can assume that a checkpoint failure to block **D** has occurred. A simple way to react to this fault is to perform a retry. However, if the failure is permanent, we might want to perform a more sophisticated recovery approach such as remapping. To illustrate the problem of failed intentions, let's assume that the file system maintains a remapping table **M** to track bad block remapping. This remapping procedure should be very simple: allocate a new block (*e.g.*, **R**), update the bitmap block **B'** for this new allocation, write the content of **D** to the new location **R**, and update the remap table **M**, specifying that **D** has been remapped to **R**. However, in the current journaling scheme, these recovery actions cannot be performed consistently in the presence of crashes, as illustrated in Figure 3.6.

Figure 3.6 illustrates that after the write to a data block **D** fails (T_2) the policy wants to remap block **D** to **R** (T_3), which implies that the bitmap and the remap table are modified (**B'** and **M**). Since it is too late to modify the transaction that has been committed, these modifications only happen in the memory. However, from the perspective of the journaling layer, the checkpointing of this transaction containing **B**, **I**, and **D** have finished, and thus, the transaction can be released (T_4). If a crash occurs (T_5) after the transaction is released (T_4), all metadata changes introduced by the recovery actions will be discarded and the disk will be in an inconsistent state. Specifically, the data block **D** is lost since the modified remap table **M** that has the reference to **R** has been discarded. As a consequence, future access to **I**'s data block will not be remapped to **R**. The general point here is that the current journaling scheme cannot perform any checkpoint failure recoveries that result in metadata changes.

3.3.3 Summary: The Need for a New Journaling Scheme

Journaling is deployed in many modern file systems including ext3 [140], ReiserFS [109], IBM JFS [20], XFS [132] and Windows NTFS [124]. The fact that we cannot recover from a checkpoint failure properly with the current journaling scheme is disastrous. In fact, Prabhakaran *et al.* have shown that ext3, IBM JFS, and ReiserFS ignore checkpoint failure [105] (although they did not point out the reason). In this section, we have uncovered a major flaw that shows why these journaling file systems are unable to react to checkpoint failure. Given the significance of this journaling flaw, in Chapter 6, we present our solution to the problem: *chained transactions*.

3.4 Conclusion

In this chapter, we have shown that many file system reliability components are actually not reliable, in particular when dealing with partial disk failures. To build a new generation of robust and reliable file systems, we believe a new approach to each of the problems is required.

First, in Section 3.1, we have found that file system checkers are not always robust in how they perform repairs. We mainly believe that when checkers are written in the low-level C language, their logic is hard to reason about, and hence can be buggy. Thus, to build a new generation of robust and reliable file system checkers, in Chapter 4, we introduce *SQCK*, a novel file system checker that employs *declarative* queries where the high-level intent of the checker can be specified in a clear and compact manner.

Second, in Section 3.2, we have shown that today's commodity file systems do not have a good reliability framework for dealing with disk failures. Their failure policies are diffused, leading to inconsistent and buggy policies. To solve this problem, in Chapter 6, we present *I/O Shepherding*, a simple yet powerful way to build robust and centralized failure policies within a file system.

Third, in the same section, we also unearthed the problem of incorrect error-code propagation where error-codes are accidentally dropped in the middle of their propagation, leading to serious silent failures. In Chapter 5, we present *Error Detection and Propagation (EDP)*, a static analysis tool that shows where error-codes are dropped in file systems and storage drivers.

Finally, in Section 3.3, we have shown why many journaling file systems cannot recover from checkpoint failure properly. Thus, in Section 6.4.1, as part of the Shepherding framework, we also introduce the concept of *chained-transactions*, a

novel and more powerful transactional model that allows policies to handle unexpected faults during checkpointing and still consistently update on-disk structures.

Chapter 4

SQCK: A Declarative File System Checker

> *"FIXME: In the future, inodes which are still in use should be handled specially. Right now we just (do a simple repair), instead of (the right repair). This won't catch (a particular corrupt scenario), but it's better than nothing. The right answer is that there shouldn't be any bugs in (this corruption) handling. :-)"*
> – A comment in e2fsck (pass1.c, line 778)

Despite the best efforts of the file and storage system community, file system images become corrupt and require repair. In particular, problems with many different parts of the file and storage system stack can corrupt a file system image: disk media, mechanical components, drive firmware, the transport layer, bus controller, and OS drivers [13, 14, 46, 53, 106, 135]. Since file systems do not usually contain the machinery to fix corruptions themselves [15, 106], there is a broad need for robust file system checkers.

Unfortunately, our analysis of file system checkers in Section 3.1 has shown that robust checkers are not straightforward to design or implement. Checkers are typically large and complex; for example, the Linux ext2 checker contains more than ten thousand lines of low-level C code which can be difficult to reason about. Due to this complexity, it is not surprising that we found many weaknesses in the Linux ext2/3 checker.

To build a new generation of robust and reliable file system checkers, we believe a new approach is required. The ideal approach should enable the high-level intent of the checker to be specified in a clear and compact manner; further, the description of the intent should be cleanly separated from its low-level implementation and how it is optimized. A high-level specification has multiple benefits: by its very nature it is easier to understand, modify, and maintain.

In this section, we introduce SQCK (pronounced "squeak"), a novel file system checker. Borrowing heavily from the database community, SQCK employs declarative queries to check and repair a file system image. We find that a declarative query language is an excellent match for the cross-checks that must be made across the different structures of a file system.

The rest of this chapter is organized as follows. We first describe our goals (Section 4.1) and then motivate why declarative query language is suitable for fulfilling the goals (Section 4.2). Next, we present the overall architecture of SQCK, including how declarative checks and declarative repairs are performed (Section 4.3). We discuss implementation challenges in Section 4.4 and finally evaluate SQCK in Section 4.5.

4.1 Goals

We believe that a file system checker should be correct, flexible, and have reasonable performance; we believe a declarative language will enable us to meet these goals for the following reasons.

Correctness: The primary responsibility of a file system checker is to produce a consistent file system image. A declarative language allows one to check and repair hundreds of corruption scenarios in a clean and compact fashion; we believe the ability to produce correct repairs is improved due to the simplicity of the queries and the separation of the specification from the implementation. A secondary goal is to produce repairs that leverage all of the on-disk information to retain as much as possible of the file system. We believe declarative languages allow one to easily combine the disparate information that resides throughout the file system.

Flexibility: Given a single corruption, there are many reasonable repairs that could be performed. For example, if two inodes share the same data block, there are many ways to repair this inconsistency: a "cheap" repair could simply remove one of the pointers [32], the inode with the earliest modification time could release the block [93], the block could be cloned (e2fsck's way),

or the operator could decide. The simplicity of a declarative language encourages one to explore this policy space and even provide different modes of repair (*e.g.*, fast but partial repair, or slow but full/smart repairs).

Performance: While the performance of a file system checker is not a primary concern, it must not be prohibitively slow; specifically, the checker must be able to handle the amount of data on modern disks and storage systems. Thus, our goal is to create a checker that is competitive in speed to a traditional checker.

4.2 Declarative Query Language

Implementing a file system checker that satisfies all the goals above (especially the correctness part) has proven to be difficult. Thus, an alternative is needed. One alternative is to view a file system checking as ensuring that the content satisfies a specification. With this view, some researchers have attempted to auto-generate fsck code by writing a specification in an object modeling language. Specifically, Demsky and Rinard's work repairs inconsistencies automatically given specified constraints [32]. Their automated repair finds the cheapest way to repair the system such that it satisfies the constraints again. For example, if two inodes share the same data block, the cheapest repair could simply remove one of the pointers; however, this may not be the desired result. In fact, there are many ways to solve the problem: the inode with the earliest modification time could release the block [93], the block could be cloned (e2fsck's way), or the operator could decide. In our terminology introduced in Section 3.1, the modeling approaches ensure that the repairs are consistent, but not necessarily information-complete or policy-consistent.

When reinventing fsck, we need a language that can declaratively express both the checks and the repairs. Like others who have applied declarative languages to domains such as system configuration [34] and network overlays [89], we believe the solution is to use a declarative language. We specifically choose a declarative *query* language, SQL [2], as our choice. Thus, we name our declarative file system checker as SQCK (SQl-based fsCK).

The first advantage of using SQL is to easily achieve the performance goal; the database community has tuned this language and its engine for years. Thus, rather than tuning a new declarative or modeling language, we can directly use the built-in optimization of an off-the-self SQL engine. However, besides performance, achieving correctness is also highly important. We believe that by using a query language such as SQL, we can express hundreds of checks and repairs in a more correct fashion. Below, we give several short examples that illustrate why that is.

```
first_block =  sb->s_first_data_block;
last_block = first_block + blocks_per_group;

for (i = 0, gd=fs->group_desc;
     i < fs->group_desc_count;
     i++, gd++) {
  if (i == fs->group_desc_count - 1)
    last_block = sb->s_blocks_count;
  // the core logic of range-checking
  if ((gd->bg_block_bitmap < first_block) ||
      (gd->bg_block_bitmap >= last_block)) {
      px.blk = gd->bg_block_bitmap;
      if (fix_problem(PR_0_BB_NOT_GROUP, ...))
        gd->bg_block_bitmap = 0;
  }
  ...
}
```

Figure 4.1: **Range-checking in C code.** *The C fragment above shows the e2fsck's implementation of the "check block bitmap not in group." The core logic of the range-checking, marked in italic, is buried in implementation details.*

First, range-checking is very common in file system checkers. One example in e2fsck is verifying that the block bitmap pointer for a group points to a block located within that group. The logic of this check is a simple range-checking. However, the actual implementation of this check, shown in Figure 4.1, illustrates that a low-level C implementation tends to make a simple check hard to understand and debug. The core logic of the range-checking, marked in italic, is buried in implementation details such as for-loop, data traversal, and many others. In contrast, with a query language, we can write the check declaratively as shown in Figure 4.2. The query simply performs a SELECT from the group descriptor table to find any bitmaps that are not within the desired range for the group. The query also shows that, in SQCK, all checks and repairs are performed by running queries on database tables. Thus, before running the queries, SQCK must preload the database tables with the on-disk structures. More details on this design are explained in the next section.

Second, what is also common is cross-checking fields across different structures. A simple example is verifying that all pointers refer to blocks within the file system; this check involves verifying that every pointer is within the range specified in the primary superblock. As illustrated in Figure 4.3, this check can be easily

```
SELECT *
FROM   GroupDescTable G
WHERE  G.blkBitmap NOT BETWEEN G.start AND G.end
```

Figure 4.2: **Range-checking in SQL.** *The SQL fragment above shows the SQCK's implementation of the "check block bitmap not in group."*

```
SELECT  B.*
FROM    BlockPointer B, SuperblockTable S
WHERE   B.blknum NOT BETWEEN S.firstBlk AND S.lastBlk
```

Figure 4.3: **Cross-checking in SQL.** *Cross-checking can be easily done in SQL by joining the structures (*FROM *clause) and specifying the condition to be found (*WHERE *clause).*

done in SQL by *joining* the two structures (the FROM clause) and specifying the condition (the WHERE clause). As we will see in the next section, there are many more complex instances of cross-checking that can be easily expressed in query language.

Third, for each inconsistency found, a checker must repair the inconsistency, which is done by updating related structures. In the simplest cases, a repair must simply adjust a few fields within a table (a file system structure). In these cases, a repair can be performed with an UPDATE query in SQL. In more complex cases, repairs may need to update more than one table. In these cases, SQCK easily combines a series of SQL queries with C code. More detailed examples will be shown in Section 4.3.3. In short, repairs can be naturally expressed in SQL as this query language has been built from day one to support massive amount of updates.

Fourth, a checker usually runs hundreds of checks and repairs. With a query language, we can write each check or repair as a query (as shown in the examples above). As a result, hundreds of checks and repairs can be composed by gluing all the queries together. Furthermore, the repairs must be ordered correctly; our evaluation of e2fsck in Section 3.1.3 has shown that misordered repairs can leave the file system in an inconsistent state. We believe that ensuring the correct ordering is easier done in SQCK rather than in C code. This is because, in SQCK, the logic of each query can be understood in isolation, while in C code, the repairs are typically cluttered. In Section 4.3.4, we show how we can ensure the correct ordering of repairs in SQCK.

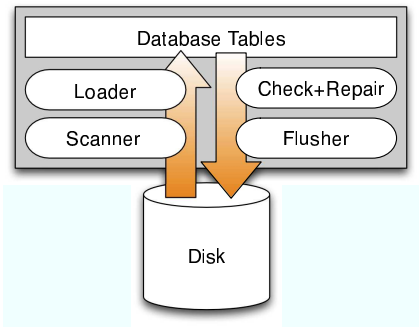


Figure 4.4: **SQCK Architecture.** *The diagram depicts the basic SQCK architecture. The left part of the design, the loader and scanner, and the right part of the design, the checker and flusher are decoupled, allowing us to optimize each component in isolation.*

Finally, there might be many ways to repair an inconsistency; in the beginning of this section, we listed more than one approach to repair a data block shared by two inodes. In SQCK, since a repair is basically a query (or a set of queries), we can plug-in and plug-out different queries in a straightforward fashion. We believe this is hard to do in C code. In the evaluation section (Section 4.5), we show the flexibility of building different versions of fsck in SQCK.

In summary, the above examples have illustrated the ease of using declarative query language to build checkers. In the next two sections we describe our specific design and implementation of SQCK.

4.3 Architecture

In this section, we provide a detailed overview of SQCK architecture. SQCK contains five primary components, as shown in Figure 4.4. The *scanner* reads the relevant portions of the file system from the disk, while the *loader* loads the corresponding information into the database *tables*. The *checker* is then responsible for running the declarative queries that both check and repair the file system structures. The *flusher* completes the loop by writing out the changes to disk. We postpone our description of the scanner, loader, and flusher until Section 4.4. In this section, we explain the tables and the checker.

Tables	Fields
Superblock Table	*blkNum*, *copyNum*, *dirty*, firstBlk, lastBlk, blockSize, ...
GroupDesc Table	*blkNum*, *gdNum*, *copyNum*, *dirty*, *start*, *end*, blkBitmap, inoBitmap, iTable, ...
Inode Table	*ino*, *blkNum*, *used*, *dirty*, mode, linksCount, blocksCount, size, ...
DirEntry Table	*blkNum*, *entryNum*, *dirty*, ino, entryIno, recLen, nameLen, name
Extent Table	start, end, *pBlk*, *pByte*, *type*, *startLogical*, *endLogical*, *ino*, *dirty*, ...

Table 4.1: **SQCK Tables.** *Italic fields represent information we generate since they are not stored on the disk.*

4.3.1 Database Tables

It is important to construct the database tables such that the SQCK checker can perform efficient queries that cover the same repairs as e2fsck. Conceptually, SQCK contains a table for each of the different metadata types in the file system: superblocks, group descriptors, inodes, directories, and block pointers [27]. Together, the tables store all of the information about the file system image that was originally on disk. However, with this on-disk information alone, the SQCK checks and repairs are neither simple nor efficient; therefore, SQCK stores extra, easily calculated information in the tables. Table 4.1 shows the five database tables utilized by SQCK. We describe briefly the important fields in each table.

Superblock: Since the superblock is replicated, we load each replica into a row of the table; this table allows SQCK to easily check the consistency across superblocks. As expected, each row contains the information available from the superblocks on disk. To be able to reflect repairs back to the disk in the flusher, we also introduce *copyNum* and *blkNum* fields that specify where a replica lives on the disk and a *dirty* field.

GroupDescTable: Each group descriptor and its replicas are loaded into separate rows of this table; as expected, we store here the on-disk information such as the pointers to the block bitmap, inode bitmap, and inode table. SQCK also adds the *start* and *end* block of each group; this addition allows SQCK to easily check whether pointers fall within the desired range of the block

group.

InodeTable: Each row of the table corresponds to a different allocated inode, with appropriate fields for the on-disk information such as mode, links, and size. The *used* field tracks which inodes are part of the final directory tree so that SQCK can calculate the final inode bitmap.

DirEntryTable: Each row of the table corresponds to a different directory entry. SQCK performs many cross-checks on this table to verify the directory tree structure.

ExtentTable: The conceptual idea of this table is to record all of the pointers to data blocks, so that SQCK can ensure that no two pointers refer to the same block. In our initial implementation, we loaded each direct pointer as its own row; however, this is intractable for a large file system because the table grows too large and the loader takes too long. Therefore, we switched our table design to represent extents of contiguous direct blocks; specifically, each extent specifies the start and end block. Additionally, each row records the location of the original pointer and the *type* of the pointer (*e.g.*, direct, single, double, or triple indirect).

4.3.2 Declarative Checks

A declarative query language is an excellent match for the checks and repairs that must be performed by a file system checker. To give some intuition as to why this is true, we categorize the different checks that must be made and show how a prototypical check from each category can be specified with SQL [2].

The original e2fsck performs a total of 121 interesting repairs. We have categorized all of these repairs into four categories, depending upon how many file system structures the repair must simultaneously peruse. As shown in Table 4.2, a repair can touch a single instance of a single structure type, one instance of one type with another of a different type, multiple instances of the same type, or multiple instances from multiple types.

There are 63 fsck repairs that involve fields of a single structure in isolation. A simple example of this type of repair is ensuring that the deletion time of a used inode is zero. Another example is verifying that the block bitmap for a group is located within that group. We have shown how this check can be expressed simply and efficiently using SQL in Figure 4.2. The query simply performs a SELECT from the group descriptor table to find any bitmaps that are not within the desired range for the group. Thus, range-checking queries are easily specified.

	Single instance	**Multiple instances**
Intra structure	*Category #1* 63 checks	*Category #3* 11 checks
Inter structures	*Category #2* 12 checks	*Category #4* 35 checks

Table 4.2: **Taxonomy of fsck cross-checking.** *We distinguish four types of cross-check. We report the number of checks that fall into each category. In the first category, a cross-check can be made within an instance of a structure. In the second, a cross-check is performed on an instance of a structure and an instance of another different structure. The third category cross-checks multiple instances of a structure. Finally, the last category involves information stored in multiple instances of more than one structures. Each number in the box represents the number of checks that are done by e2fsck in each category.*

```
SELECT  X.*
FROM    ExtentTable X, SuperblockTable S
WHERE   S.copyNum =  1                  AND
        X.type    =  INDIRECT_POINTER   AND
       (X.start   <  S.firstBlk         OR
        X.end     >= S.lastBlk)
```

Figure 4.5: **Check illegal indirect block.** *An illegal indirect block is one that points to outside the file system range*

The second category includes checks between one instance of a structure and an instance of another different structure; fsck runs 12 checks of this type. A simple example is verifying that all pointers refer to blocks within the file system; this check involves verifying that every pointer is within the range specified in the primary superblock. Unlike the previous example, this example must examine values in different structures and subsequently different tables. Figure 4.5 shows how to check that no indirect block points outside the file system. Specifically, the query returns all extents (`X.start..X.end`) corresponding to indirect pointers that fall outside the file system range specified in the primary superblock (`S.firstBlk..S.lastBlk`). Hence, SQCK can easily join multiple structures to perform the necessary cross-checks.

The third category contains 11 cross-checks of multiple instances of the same

```
SELECT  *
FROM    DirEntryTable P, DirEntryTable C
WHERE   // P says C is his child
        P.entryNum >= 3      AND
        P.entryIno  = C.ino  AND
        // but C says P is not his parent
        C.entryNum  = 2      AND
        C.entryIno <> P.ino
```

Figure 4.6: **Bad dot dot.** *This query finds a directory entry that does not claim the actual parent.*

structure. One example of this type of repair is checking that multiple inodes do not point to the same data block. A second example, shown in Figure 4.6 checks that the ".." entry of a directory points to the actual parent. This check can be done easily in SQL: the query simply joins the directory entry table with itself, selecting cases where the parent directory contains an entry for a child (where `P.entryNum >= 3`), but the child's entry for ".." (`P.entryNum = 2`) is not the parent's inode.

Finally, 35 checks fall into the fourth category in which the cross-checks involve multiple instances of more than one structure. One example is the rule that validates the link count of an inode, since it must traverse all directory entries and count how many times each entry appears. We give two examples of these queries to further convince the reader that even these types of seemingly complicated checks are surprisingly straightforward to express.

The first example checks for conflicting block pointers; in ext2, block pointers are stored in many places and none should refer to the same block. Figure 4.7 shows a query that ensures blocks pointed from an inode do not overlap with file system metadata blocks. The query is a little bit cumbersome because it checks whether an extent overlaps with each piece of file system metadata separately (*i.e.*, superblock copies, group descriptors, inode bitmaps, block bitmaps, and inode tables).

The second example verifies that multiple directory entries do not point to a same directory, corresponding to the false parenthood problem discussed in Section 3.1.3; we show how it can be expressed in SQL in Figure 4.8. Basically, the query selects directory entries that appear more than once in the tree structure. In more detail, the query does not select the "." or ".." entries and selects only directory inodes, as determined by their `mode` field in the inode table. Counting the number of entries satisfying this constraint is straightforward with the ORDER BY and HAVING features of the query language. Note that this query returns the small-

```
SELECT X.*
FROM   ExtentTable X
WHERE EXISTS
 (SELECT *
  FROM   SuperblockTable S
  WHERE
   // extent overlaps superblock copies
   S.blk BETWEEN X.start AND X.end)
 OR EXISTS
 (SELECT *
  FROM   GroupDescTable G, SuperblockTable S
  WHERE
   // or extent overlaps group descriptors
   (X.start BETWEEN G.blk AND G.blkEnd OR
    X.end   BETWEEN G.blk AND G.blkEnd) OR
   // or extent overlaps inode table
   (X.start BETWEEN G.iTbl AND G.iTblEnd OR
    X.end   BETWEEN G.iTbl AND G.iTblEnd) OR
   // or extent overlaps block bitmap
    G.blkBitmap BETWEEN X.start AND X.end OR
   // or extent overlaps inode bitmap
    G.inoBitmap BETWEEN X.start AND X.end)
```

Figure 4.7: **Check block overlaps metadata.** *This query locates inode's extents that overlap with the filesystem metadata. To reduce space, we abbreviate some fields: G.iTblEnd should be G.iTable + S.inodeBlocksPerGroup - 1; G.blkEnd should be G.blk + S.gdBlks - 1.*

est inode number among the parents (`MIN(P.ino)`), which is needed to mimic how e2fsck incorrectly repairs this problem. In particular, e2fsck always assumes the parent with the smallest inode number is the real parent without consulting the ".." entry of the child. We show how we can easily improve this query in Section 4.5.1.

4.3.3 Declarative Repairs

Performing checks of file system state is only part of the problem; after SQCK detects an inconsistency, it must then perform the actual repair. SQCK performs the repair by first modifying its own tables; the flush process then propagates these changes to the disk itself. We have found that repair operations on the tables can be performed in one of two ways.

```
SELECT  P.entryIno, COUNT(*), MIN(P.ino)
FROM    DirEntryTable P, InodeTable I
WHERE   P.entryNum >= 3      AND
        P.entryIno  = I.ino  AND
        I.mode      = DIR
GROUP BY P.entryIno
HAVING (COUNT(P.entryIno) > 1)
```

Figure 4.8: **Check multiple parents.** *This query returns directories that have multiple parents. The parent that has the smallest inode number (*`MIN(P.ino)`*) will be the one that keeps the child directory.*

```
UPDATE  ExtentTable X
        INNER JOIN
        (result from the
        ''check illegal indirect block'' query) AS V
        ON X.ino   = V.ino    AND
           X.type  = V.type   AND
           X.start = V.start  AND
           X.end   = V.end
SET     X.start = 0, X.end = 0, X.dirty = 1
```

Figure 4.9: **Repair illegal indirect block number.** *This query repairs indirect block numbers that fall outside the file system range (returned by the "check illegal indirect block" query in Figure 4.5), by clearing them to zero.*

In the simplest cases, a repair must simply adjust a few fields within a table. These repairs can be performed by embedding the declarative checks presented previously into a larger query that then sets fields within the selected rows. For example, an illegal indirect block pointer (one that points outside the file system range) is fixed by clearing the pointer to zero. Figure 4.9 shows that these pointers can be cleared with a query that sets to zero the illegal extents found by the check query in Figure 4.5. Note that the query also sets the dirty flag so that the flusher will later propagate these changes from the database tables to the on-disk structures.

In more complex cases, repairs may need to update more than one table. In these cases, SQCK combines a series of SQL queries with C code. SQCK currently supports a variety of repair primitives, such as finding free blocks and inodes and adding and deleting extents, directory entries, and inodes. Figure 4.10 shows how a valid directory with a reference count of zero (*i.e.*, a lost directory) is reconnected

```
result = run(findUnconnectedDir.sql);      [9]
while(dir = mysql_fetch_row(result)) {
 run(changeDotDot.sql, dir, lfIno);        [3]
 slot = run(findEntrySlot.sql, lfIno);     [7]
 if (!slot) {
  lfBlk = run(getLocation.sql, lfIno);     [3]
  newBlk = run(allocNewBlock.sql, lfBlk); [25]
  if (run(needIndirect.sql, lfIno))        [5]
   { // alloc indirect (not shown) }
  run(addNewBlock.sql, newBlk, lfIno);     [3]
  run(addInodeSize.sql, lfIno);            [3]
  run(initNewDirBlk.sql, newBlk, lfIno);   [3]
  slot = run(findEntrySlot.sql, lfIno);    [7]
 }
 // now break the slot and prepare        [13]
 // newSlot based on dir.  (not shown)
 run(updateOldSlot.sql, oldSlot);          [3]
 run(insertNewSlot.sql, newSlot);          [3]
 run(incrementLinkCount.sql, lfIno);       [3]
}
```

Figure 4.10: **Complex repair.** *The C pseudo-code above illustrates the complex repair in reattaching unconnected directories to the lost+found directory. The files with the* `.sql` *extension are the SQL files that are executed. The bold numbers in the brackets represent the lines count of each SQL file. The italic number is the lines count of the C code.* `lfIno` *is the inode number of the lost+found directory.*

to the lost+found directory. Briefly, the code behaves as follows. After a query finds the set of unconnected directories, SQCK performs the following operations on each such directory. First, the ".." entry is adjusted to point to lost+found. Next, a directory entry slot is allocated within lost+found, which may require allocating new blocks and increasing the size of the lost+found. After the slot is ready, the entry is filled to correspond to the unconnected directory.

4.3.4 Ordering of Repairs

After all the checks and repairs are written declaratively, they must be ordered correctly. Without a correct ordering, the resulting file system can be more corrupted. For example, as shown in Section 3.1.3, e2fsck wrongly "repairs" direct pointers before checking that the indirect block containing those pointers is valid, leaving the file system in an inconsistent state.

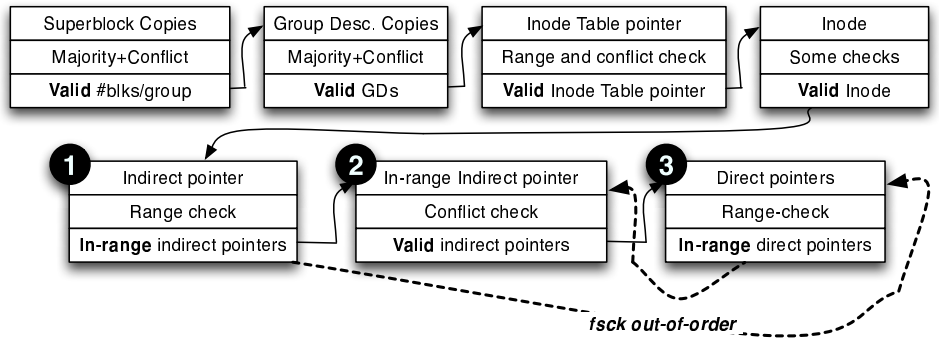


Figure 4.11: **Information dependency graph.** *The figure shows a chain of information dependency. Note that the full graph forms a tree-like graph; to save space, only a partial dependency chain is shown. Each box contains three rows: a new information obtained from the previous box, the check (and the corresponding repair, not shown), and the new state of the information after the check. For example, in box 1, an indirect pointer is obtained from a valid inode. After the range-check, the indirect pointer is marked in-range, but not yet valid. After it passes the conflict-check in box 2, it is finally marked as valid, which implies that we can proceed to box 3 which repairs out-of-range direct blocks contained in this indirect block. Unfortunately, e2fsck does not follow this ordering, as shown by the dashed lines; fsck proceeds repairing the direct pointers from a not yet valid indirect block. When e2fsck later finds out that the indirect block is indeed invalid (e.g., conflicting with other file system metadata), the content of the metadata has been accidentally corrupted in box 3.*

In general, repairs of a complex data structure must be performed in a specific order; specifically, if a piece of information A is obtained from B, then B must be checked and repaired first. To ensure this ordering, we have constructed an *information dependency graph* for the data structures in ext2. A portion of this graph is shown in Figure 4.11. The figure also illustrates that e2fsck does not follow the order specified by the dependency graph. We reorder the relevant queries to ensure that single, double, and triple indirect blocks are all validated in the correct order before repairing the direct pointers themselves. We find that reordering repairs in SQCK is straightforward due to the structure of the queries; we do not believe such reordering is simple in e2fsck.

Currently, the dependency graph must be manually constructed by the file system developer or administrator. Since the repair queries in SQCK are neatly structured, the ordering can then be manually verified against the dependency graph.

Improving Scan Time	
1	Reduce seek time with sorted job queue
Improving Load Time	
2	Make the table content compact
3	Only load checked information
4	Use threads to exploit idle time
Improving Check Time	
5	Write queries that leverage indices
6	Leverage fs domain specific knowledge
7	Use bitmaps to reduce search space

Table 4.3: **Optimization Principles.** *The table lists the optimizations that we have performed such that performance of SQCK is competitive with e2fsck.*

More ideally, a static tool could be built on top of SQCK to verify the ordering automatically. Specifically, each query could be tagged with a unique name that describes the check/repair performed, then a parser could automatically construct the ordering from the code, and finally a verifier could compare the constructed ordering against the specified ordering. This highlights that a structured fsck can be easily verified than a cluttered one.

4.4 Implementation

We now describe our implementation of the SQCK phases for scanning the file system image from disk, loading the database tables, checking and repairing the structures, and finally flushing the repairs to disk. Our current implementation of SQCK runs on top of a MySQL database and targets the ext2 file system in Linux 2.6.12. When describing our implementation, we focus on the optimizations we found were necessary for achieving respectable performance; Table 4.3 summarizes these optimizations across the phases.

4.4.1 Scanning and Loading

In our current implementation, SQCK combines its scanning and loading phases. Conceptually, SQCK maintains a queue of the structures that must be read from disk, processed, and loaded into the tables. As structures are processed, SQCK follows their pointers to determine the next structures. For example, the queue is initialized from the primary superblock; after the superblock, the locations of the

group descriptor copies are known; subsequently, the inode tables are processed, which leads to individual inodes and their data blocks.

SQCK implements a number of performance optimizations for scanning and loading. First, to reduce the scan time, SQCK sorts the requests in the queue based on their on-disk locations; sorting the requests minimizes disk head positioning time, especially for file systems that are fragmented. We note that although e2fsck performs a partial optimization of this sort (*i.e.*, directory blocks are sorted before read from the disk [27]), e2fsck is not able to perform the same optimization (*e.g.*, indirect blocks still have to be traversed logically) because it heavily intermixes scanning with checking [65]. SQCK is able to optimize scanning because reading from disk is completely decoupled from checking; hence, SQCK does not need to follow structures in a logical manner.

The primary reason we decouple scanning from checking is because we want to make the common case fast; if corruption is a rare case than our approach improves the overall fsck time. However, there is a tradeoff: if corruption is huge, extra work is needed to invalidate the garbage loaded into the database. Our design is not limited only to that approach; if desired, SQCK can be redesigned by intermixing some phases of scanning and checking according to the structural logical hierarchy. For example, when loading and checking indirect blocks, triple indirect blocks will be loaded and repaired in the database, then only valid double indirect blocks will be loaded to the database, and so on.

Second, SQCK improves load time (and check time) by reducing the size of the initial database tables. Our initial implementation loaded the ext2 structures to match their on-disk format; specifically, SQCK loaded each on-disk pointer as a direct pointer. However, we found that this approach made checking even 100 GB file systems unattractive. Therefore, our next optimization modified the tables to instead use extents to represent pointers referring to contiguous blocks.

Third, SQCK reduces loading time by only loading allocated meta-data. Given that most file systems are half-full [7], a great deal of the inodes are not actually used. To reduce the size of the tables, SQCK does not load the unused inodes into the database tables (though it of course still scans them from disk). However, e2fsck performs one check on unused inodes that SQCK must be able to replicate: e2fsck verifies that each inode with a link count of zero also has a deletion time of zero. To handle this repair, SQCK performs this one check during processing. If SQCK finds a non-conforming inode, that inode is loaded into the table on the fly; to mark that the inode has been repaired, its *used* field is cleared and the *dirty* field is set. We note that this optimization is consistent with the direction in which future file systems are going: ext4 explicitly marks unallocated sections of the inode table

to help e2fsck run more efficiently [3].

Fourth, the scanner-loader in SQCK is multi-threaded. Each thread within the pool is able to independently grab a structure from the queue, read the data from disk, process it, and load the information into the corresponding table. Multiple threads allow SQCK to overlap reading requests from disk with loading the tables. As we will see in our evaluation, this optimization is especially important for large partitions.

4.4.2 Checker

After all metadata has been uploaded into the database tables, SQCK initiates the checking phase, which runs the queries as discussed in the previous section. One important note is that since the checker runs only after the loader, corrupt data can be loaded into the tables. Hence, SQCK provides primitives to invalidate a structure along with the information that originates from it. For example, if the block number that points to an inode table is corrupt, the wrong inodes and the wrong data pointers will be loaded into the table. Later, when the checker discovers that the inode table pointer is corrupt, it simply calls the SQCK primitives to invalidate the corresponding inodes, extents, and directory entries.

The checker has been optimized for performance in three main ways. First, we have found that SQCK must contain appropriate indices for each table; without indices a full scan must be done for each check and joining multiple tables requires a very long time. Thus, each table contains indices over the fields that are checked with the comparison operators.

Given the indices, some queries must be rewritten to leverage them. In our experience, MySQL is not able to always extract the implicit index comparisons in some queries. For example, the check that no directory entry points to an unused inode was originally written as shown in the top half of Figure 4.12. When the rule was rewritten to make the index comparison explicit, as shown in the bottom part of the figure, the query time improved significantly. Thus, making index comparison explicit is an important principle to do fast checking. We rewrote a total of four queries in this manner, reducing the check time for those four queries from 72 seconds down to just 0.09 seconds on a 1 GB partition.

Second, we have found it beneficial to incorporate file system domain knowledge into the queries. One example is the rule that counts how many blocks are being used in a group. Since SQCK uses extents, it must first select the extents in that group. The naive range-checking query could be written as follows: `(G.start <= X.start AND X.end <= G.end)`. However, given that we know valid extents cannot overlap group boundaries (this has been verified in previous queries),

```
// find an entryIno that is in the list of
// unused inodes
SELECT *
FROM   DirEntryTable
WHERE  entryIno IN
         (SELECT ino
          FROM   InodeTable
          WHERE  used = 0)

       --------- vs. ---------

// find an entryIno that exists in the
// InodeTable and the used field is zero
SELECT *
FROM   DirEntryTable
WHERE  EXISTS
         (SELECT *
          FROM   InodeTable AS I
          WHERE  I.ino  = D.entryIno AND
                 I.used = 0)
```

Figure 4.12: **Explicit index comparison.** *We rewrite the code to unearth the index comparison.*

the range-check query can be simplified to `(G.start <= X.start <= G.end)`. This simplified query improves check performance.

The final optimization addresses how to join tables where an index comparison is not possible. For example, the query finding shared blocks across files joins the ExtentTable with itself to find any overlapping extents. We optimize this query by making the search space smaller with bitmaps. For this example, SQCK uses two bitmaps: one for marking used blocks and one for marking shared blocks; the latter bitmap provides a hint as to which extents have overlapping blocks. To find out which part of an extent is actually overlapping, SQCK joins the resulting small table with itself.

4.4.3 Flusher

Finally, SQCK needs to update any repaired structures to the disk. SQCK is able to determine which structures have been modified by selecting those entries where the dirty flag is set. Following the same behavior as e2fsck, SQCK updates the structures in-place on disk (*i.e.*, it does not currently use a separate journal).

To ensure the metadata writes are ordered correctly [45], currently SQCK performs a series of queries ordered by the dependency graph; the graph ensures that blocks are updated before the pointers to those blocks. In the next generation of SQCK, a journaling facility will be added to ensure that a crashed repair process will not modify the old data partially.

4.5 Evaluation

In this section we evaluate SQCK along four axes: flexibility, complexity, robustness, and performance.

4.5.1 Flexibility

The simplicity of implementing checks and repairs in SQCK enables one to construct different versions with different repair policies. At this time, we have created two versions of SQCK. The first one simply emulates e2fsck with both its good and bad polices. The second one fixes what e2fsck does wrong (by using the information dependency graph in Figure 4.11) and adds new functionality that e2fsck does not even attempt (*i.e.*, performs information-complete and policy-consistent repairs).

Our basic version, SQCK$_{fsck}$, emulates the repairs made by e2fsck. From our analysis of e2fsck, we have determined that it performs 153 different repairs, of which 121 are significant and interesting for ext2 (the remaining 32 repairs fix the ext3 journal and other optional features). These 121 repairs have been detailed in Table 3.1 in Section 3.1.1. As shown, e2fsck performs these repairs in six distinct phases, in which reading the file system image from the disk is intermixed with the actual checks and repairs. SQCK$_{fsck}$ implements these 121 repairs each as a separate query within the check and repair process.

Our second version, SQCK$_{improved}$, improves how the file system is checked by utilizing more of the information that resides within the file system image. Table 4.4 lists the new information-complete, policy-consistent, and secure repairs in SQCK$_{improved}$.

The first three repairs utilize the replicas that ext2 keeps of the group descriptor blocks on disk. While e2fsck does examine these replicas if the primary copy is obviously corrupted, e2fsck misses opportunities to use correct replicas when the primary "looks" fine. Thus, SQCK$_{improved}$ always examines all replicas and performs majority voting across them to determine the correct values; this voting is performed for three important fields: the pointer to the data block bitmap, the inode

New Repair	LOC (C)	LOC (SQL)
Majority rule on block bitmap pointers	40	22
Majority rule on inode bitmap pointers	40	22
Majority rule on inode table pointers	40	22
Finding false parents	13	14
Reconstructing missing directories (*)	47	20
Precedence cloning	23	19
Secure cloning (**)	41	8

Table 4.4: **New repairs.** *The table lists all the new repairs we introduce. (*) In addition to the number of lines reported here, this new rule heavily uses the primitives as in Figure 4.10. (**) The number of lines reported for this rule is the additional code to the original cloning repair.*

bitmap, and the inode table. With these fixes, SQCK$_{improved}$ performs information-complete repairs when the pointer to the inode table is corrupted, as desired. These new repairs are straightforward to implement, requiring only 22 lines of SQL and 40 lines of C.

The fourth repair utilizes the extra information kept in directory ".." fields to repair corrupted directories. First, we fix the false parenthood problem exhibited by e2fsck. With SQCK, we replace the incorrect check of e2fsck originally shown in Figure 4.8 with the one in Figure 4.13. This new query elegantly expresses relatively complex behavior: it only returns false directory entries in which the child directory does not claim them as a parent with ".."; thus, this false directory entry is correctly cleared instead of that of the rightful parent.

We can extend this repair slightly to write the fifth repair, which corrects even more complicated corruptions of the directory hierarchy. For example, if a path `/a/b/c/` exists and `b`'s inode is corrupted such that `b` no longer appears to be a directory, e2fsck does not do any reconstruction and simply moves `c` to lost+found. However, SQCK$_{improved}$ completely reconstructs the contents of `b` from the back pointers of its children. The complete rule requires a total of 20 new SQL lines with C code similar to that shown in Figure 4.10.

The sixth repair corrects the allocation policy of e2fsck. Specifically, e2fsck clones data blocks without checking which file is closer to the shared data block. Ideally, the repair should give the existing block to the closest inode and allocate the new clone to the other inode. With SQCK, locality optimizations are easily performed. For example, Figure 4.14 shows how we utilize the ABS and ORDER

```
SELECT  F.*
FROM    DirEntryTable P, DirEntryTable C,
        DirEntryTable F
WHERE   // P says C is his child
        P.entry_num >= 3      AND
        P.entry_ino  = C.ino  AND
        // and C says P is his parent
        C.entry_num  = 2      AND
        C.entry_ino  = P.ino  AND
        // but F, the false parent, says
        // C is also his child. P wins.
        F.ino <> P.ino        AND
        F.entry_num >= 3      AND
        F.entry_ino  = C.ino
```

Figure 4.13: **Finding false parents.** *This query returns the actual false parents. A false parent is a parent that claims to own a child even though the child is already strongly connected to another parent.*

```
SELECT  X.ino, X.start, X.end,
        V.start, V.end,
        (ABS(X.pBlk-V.start)) as distance
FROM    ExtentTable C,
        (A query that returns the start and
         end of a shared extent) AS V
WHERE   X.start <= V.start AND V.end <= X.end
ORDER BY  V.start, distance
```

Figure 4.14: **Locality-aware repair.** *The query above returns shared blocks that are sorted based on the locality distance from the pointers. The inner query (not shown), stored in Table V, returns a the list of duplicate blocks. The* ABS *command helps sorting the result based on locality distance.*

BY SQL commands to calculate the distance between a block and its pointer. The bold text shows that the results are sorted on the start of the shared extents and then on the distance between the shared extent and the blocks that point to the extent (X.pBlk). Given this list, SQCK can easily perform the repair such that the shared extent is kept with its closest pointer.

Finally, the seventh repair adds secure cloning. This is done in two ways. First, suppose a corrupt direct pointer incorrectly points to a bitmap block; since the

	C Code		SQL
Component	**LOC**	**; count**	**LOC**
Scanner	2759	1378	–
Loader	609	177	103
Checker+Repair	*2527	1468	910
Primitives	695	348	98
Flusher	114	49	27
Total	6704	3420	1138

Table 4.5: **SQCK$_{improved}$ LOC.** *The table presents the complexity of SQCK$_{improved}$. Scanner includes threads and functions that process the structures. (*) The C code for the checkers and repairs are mostly wrappers that call the SQL files.*

bitmap block is pointed to by more than one group descriptor replica, it is more likely the direct pointer is mistaken than all of the group descriptor replicas; therefore, cloning of that block simply leaks information and does not need to be performed.

Second, suppose a data block is shared by two inodes, one in the `/root` directory and one in the `/home/UserA` directory. In this case, if we want to prevent leaking of information, we might not want to clone the shared block, instead we remove the pointer from the user and keep the one from the root inode. In addition to the existing block conflict check and cloning primitives, this new rule only requires additional two SQL files, for a total of 8 lines to do the path traversal, and 41 lines of C code.

The secure clone repair could be seen as an example where an administrator's decision is more appropriate than an automated one. SQCK does not throw away the need to ask the administrator for the right decision. In such cases, different policies should be present for the administrator to choose from. In SQCK, we can execute different policies easily; each policy is simply mapped to a query or a set of queries.

4.5.2 Complexity

Table 4.5 presents the complexity of SQCK$_{improved}$, the most complete version of SQCK. As the table suggests, SQCK is comprised of C and SQL code. The scanner is the only place where the complexity of the C code still exists. However, the code is generally simple because it scans the file system in a logical hierarchy.

	SQCK	**ext2**	**ReiserFS**	**XFS**
LOC	2527	16472	11281	21773
# Chks	121	121	156	344
Instr. gap	$16 \pm$ **16**	$71 \pm$ **161**	$56 \pm$ **203**	$128 \pm$ **257**
Func. gap	$1 \pm$ **1**	$4 \pm$ **6**	$1 \pm$ **3**	$5 \pm$ **6**
# Chk func.	121	31	32	72
# Chks/chk-func	$1 \pm$ **0.1**	$4 \pm$ **5**	$5 \pm$ **8**	$5 \pm$ **5**

Table 4.6: **Checkers complexity.** *The table shows the logical complexity of $SQCK_{improved}$ (without the new added repairs), ext2, ReiserFS and XFS checker codes (excluding libraries). Standard deviation is shown right next to the $\pm$ sign. "Inst. and Func. gaps" quantify the number of C instructions and functions separating one check from the next check. "# Chk func" shows in how many functions the checks are diffused. Finally, "# Chks/func" averages the number of checks performed in each checker function.*

The checker code looks big, however, it is mostly wrapper functions that call the corresponding queries; most wrappers consist of the same 15 lines of C code. A generic wrapper could be built to reduce the amount of C code.

SQCK so far has been written all at once by one small group. Thus, it is possible that SQCK will become more complex when developed by a bigger group over a longer period of time. However, we believe the core power of SQCK lies within the simple and robust queries; each query consists of 7 lines of code on average. These queries decouple the checks from the C code, enabling us to maintain reliability in an easier way. Compared to e2fsck, which consists of 16 thousand LOC of cluttered checks and repairs and 14 thousand LOC of scan utilities, all written in low-level C code, SQCK can be considered a big step towards simplifying file system checkers.

To show that we are solving a broader significant problem, Table 4.6 attempts to quantify the logical complexity of ext2, ReiserFS, and XFS checker utilities, all written in C. The metrics shown in the table are generated by our parser written using CIL [97]. In fsck-related code, we annotate the location where each check is performed. The parser computes the complexity-metrics as described in the table. For example, we compute how many instructions and function calls separate each neighboring checks. If the numbers are high, the checks are most likely diffused and reasoning about their correctness might be nontrivial, if not impossible. The numbers reported in Table 4.6 exclude fsck libraries (*e.g.*, scanner), hence they only depict the logical complexity of the checker component.

We make two important observations: First, the average number of C instruc-

		1. Boot	2. Superblock	3. Group descriptor	4. Block bitmap	5. Inode bitmap	6. Inode table	7. Single indirect	8. Double indirect	9. Triple indirect	10.Directory	11.Used data	12.Free data	13.Out of range
1.	Block bitmap	.	.	.	.	.	.	.	.	.	.	.	.	.
2.	Inode bitmap	.	.	.	.	.	.	.	.	.	.	.	.	.
3.	Inode table	***I***	***I***	***I***	***I***	***I***	.	***I***	***I***	***I***	***I***	***I***	***I***	***I***
4.	Single indirect	.	***C***	***C***	.	.	***C***	.	.	.	***C***	***C***	***C***	.
5.	Double indirect	.	***C***	***C***	***C***	***C***	***C***	***C***	.	.	***C***	***C***	***C***	.
6.	Triple indirect	.	***C***	***C***	***C***	***C***	***C***	***C***	***C***	.	***C***	***C***	***C***	.
7.	Directory	.	.	.	.	.	.	***PS***	***PS***	***PS***	***PS***	***PS***	.	.
8.	Data	.	.	.	.	.	.	***S***	***S***	***S***	***PS***	***S***	.	.

Symbols: ***C***: *Consistent repair* ***I***: *Information-complete repair*
P: *Policy-consistent repair* ***S***: *Secure-repair* **Dot (.)**: Correct repair

Table 4.7: **Results of block-pointer corruptions.** *This figure shows how SQCK responds to block-pointer corruptions. Each row characterizes the behavior for the given pointer. Each cell in a row is marked with the behavior observed for the given pointer when it is corrupted with the value of that column. In summary, with SQCK, we have removed all the problems we have found in e2fsck, as shown Table 3.3*

tions and functions that separate two checks are high in all fsck utilities, with significant standard deviations; the separation can be as low as 4 or as high as 1700 instructions. Second, checks are greatly diffused in many functions; a function could make a small number of checks while some other could perform as many as 47 checks. In such implementations verifying that all checks are complete and ordered correctly can be cumbersome. On the other hand, SQCK hides the complex logic of the checks in declarative queries, greatly reducing the gap between neighboring checks; the standard deviations shown in the SQCK column illustrate the neat organization we have achieved. In summary, we believe all C-implementations of fsck are likely to suffer from the same problems as e2fsck.

		1. Directory inode	2. File inode	3. Free inode	4. Out of range inode
1.	Directory inode	***I***	.	.	.
2.	File inode	***I***	.	.	.

Table 4.8: **Results of index-pointer corruptions.** *This figure shows how SQCK responds to index-pointer corruptions. Each row characterizes the behavior for the given pointer. Each cell in a row is marked with the behavior observed for the given pointer when it is corrupted with the value of that column. In summary, with SQCK, we have removed all the problems we have found in e2fsck, as shown Table 3.4*

4.5.3 Robustness

To test the robustness of SQCK, we have verified that it passes the same corruption scenarios that we injected for analyzing e2fsck (as described in Section 3.1.3). Tables 4.7 and 4.8 show how SQCK responds to the corruptions we injected. In summary, we have turned all inconsistent, information-incomplete, policy-inconsistent, and insecure repairs into consistent, information-complete, policy-consistent, and secure ones respectively.

We do not claim that our fault injection methodology is complete (*i.e.*, it covers all possible corruption scenarios). However, we believe the power of SQCK lies in the simplicity of fixing buggy and adding new repairs. Thus, if a more powerful testing tool found more buggy repairs, we can simply change the corresponding queries. Or, if some repairs are missing, we can easily add new queries, as we have illustrated in the previous section.

4.5.4 Performance

The experiments in this section were performed on an 2.2 GHz AMD Opteron machine with 1 GB memory and 1 TB WDC WD10EACS disk. We used Linux 2.6.12, e2fsck 1.39, and MySQL 5.0.51a. The tables are mounted on a 512 MB ramdisk.

We test the performance of SQCK and e2fsck on four partitions with different

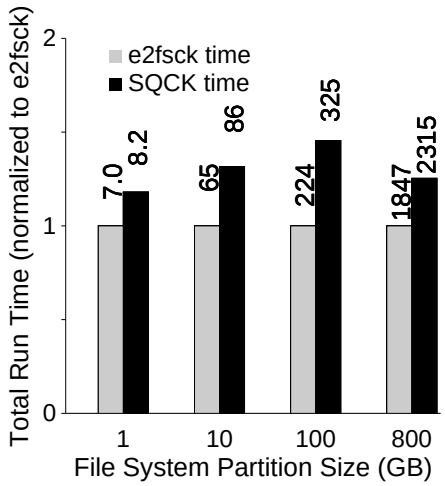


Figure 4.15: **Overall runtime comparison .** *The bar graph shows the comparison of the total runtime of e2fsck and fully optimized SQCK for different file system sizes. The bars are all normalized to e2fsck runtime and the SQCK bar show the relative slowdown. The absolute runtime figures in seconds are shown on top of the bars.*

sizes: 1, 10, 100, and 800 GB. Each of the partition is made half-full [7] by filling it with the root file system image of a machine in our laboratory along with a large number small files from kernel builds and large files from virtual machine images.

Figure 4.15 shows e2fsck compared to our fully optimized SQCK. The fully optimized SQCK incorporates all the principles described in Table 4.3; specifically, it sorts the block scan, loads extents and linked inodes only, uses 16 worker threads, and uses fast queries. In our first generation prototype we managed to keep the running time of SQCK within 1.5 times of e2fsck runtime.

We show in more detail how each of the scan and load optimization principles improve the runtime significantly by turning off one optimization feature at a time. The runtime of each of these unoptimized versions are compared relative to the fully optimized SQCK.

First, the sorted job queue is disabled such that we scan the file system logically. Figure 4.16 shows that for a large file system (*e.g.*, 800 GB), sorting the job queue plays a significant role; scanning the file system logically takes almost 3 times as long as the fully optimized one. Note that in this experiment, we disabled the loading phase to compare only the scan performance. The serial scanning for the 100 GB file system is 8 seconds faster than the fully optimized SQCK because the

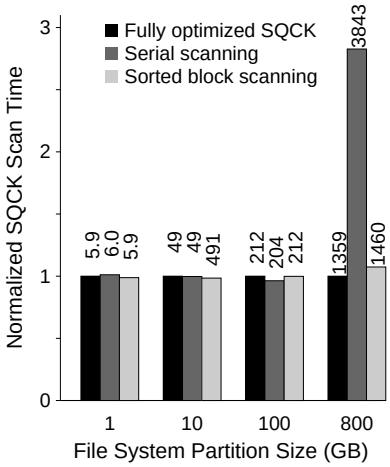


Figure 4.16: **Scan time improvement with sorted queue.** *This bar graph shows the time to scan each file system without loading. In each set, the left-most to right-most bars show the fully optimized SQCK, the logical scan, and the sorted scan with only 1 thread. The values are normalized to the fully optimized SQCK time.*

file system was almost not fragmented at all; the advantage of the sorted scanning is noticeable for fragmented and/or big file systems.

Second, we show the importance of making the initial table compact. Figure 4.17 shows the slowdown of two unoptimized versions: one that loads all inodes, and one that loads direct pointers instead of extents. When loading all inodes, the runtime is increased significantly; for 800 GB file systems, 97 million inodes will be loaded out of which only 900 thousand have non-zero link counts. When loading direct pointers, the runtime increases dramatically. For the 100 GB file system, the DirectPointerTable already consumes 360 MB, while the ExtentTable only consumes 9 MB.

Third, Figure 4.18 shows how multiple threads enable us to significantly overlap scan and load time. When the number of worker threads is reduced to one, the slowdown is almost 1.5 times in all file systems. For large file systems, increasing the number of threads gives a faster runtime; at 800 GB, using 16 worker threads improves the runtime.

In summary, our evaluation of the first generation prototype of SQCK shows that SQCK obtains comparable performance to e2fsck. In the next generation of SQCK, we plan to perform two additional enhancements. First, some checks can be merged so that the table-scan time can be reduced. If the checks find a prob-

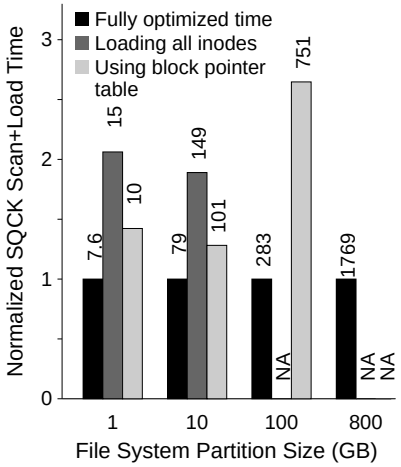


Figure 4.17: **Making the table compact.** *The bar graph shows the slowdown of scan and load time when we load big tables. In each set, the left-most to right-most bars show the fully optimized SQCK, fully optimized SQCK but with loading all inodes, and loading direct pointers instead of extents. The values are normalized to the fully optimized SQCK time. "NAs" imply experiments that do not finish in 3 hours.*

lem, then nested sub-checks will be run to pinpoint the actual problem. Second, we plan to run some checks and repairs concurrently by utilizing the information dependency graph in Figure 4.11. The graph provides the dependency tree that tells which checks and repairs are safe to run in parallel. With a faster overall check time, we hope file system developers will be encouraged to write as many rules as needed.

4.6 Conclusion

We have found that declarative queries can succinctly express the many different types of checks and repairs that fsck performs. Our experience also shows that writing checks and repairs in declarative queries is relatively straightforward; each query is written in a few iterative refinement. A complex check or repair, with a little bit of help from C code, can be broken into several short queries that are easy to understand. On average, each query we have written is 7 lines long, and the longest one is 22 lines. Furthermore, only 24 repairs require help from C code. The functionalities of the corresponding C code are generally simple; C code is only

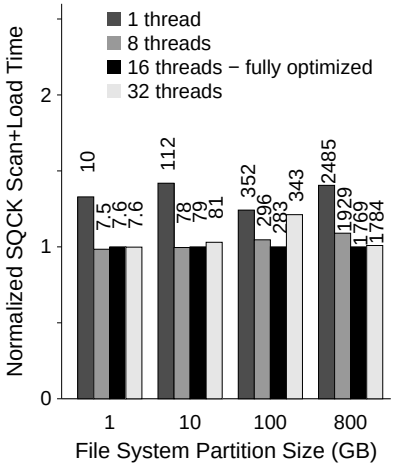


Figure 4.18: **Overlapping scan and load time.** *The bar graphs show the runtime of different runs that use different number of threads. With one thread, the runtime is the worst as we cannot utilize the idle time during scanning. The values shown are normalized to fully optimized SQCK time with 16 threads.*

used to run a set of queries and iterate the query results. Note that this is different than how C code is used for cross-checking in e2fsck, which tends to make a simple check hard to understand and debug.

In conclusion, complexity is the enemy of reliability. Current approaches describe recovery at a very low-level: thousands of lines of C code. Thus, recovery code is complex and hard to get right. We instead advocate a higher-level strategy. By encapsulating the logic of a file system checker in a set of declarative queries, we provide a more concise description of what the checker should do. In doing so, we believe we have taken an important step towards improving the robustness of file system checking.

Nevertheless, SQCK is not the last word in file system checking; it is still possible that developers write bad queries. In this case, applying more formal techniques that find bugs [41] will definitely help and thus evolve the code towards a less-buggy future. What SQCK provides is a nice framework for implementing a checker; if bugs are found, we believe that SQCK-style implementation will be easier to fix than an implementation in C code.

Chapter 5

EDP: A Static Analysis Tool for Error-Code Propagation

"Should we pass any errors back?"
– A comment in CIFS (file.c, line 1869)

The reliability of file systems depends in part on how well they propagate errors. Thus, in this chapter, we investigate the problem of *incorrect error code propagation*. To be properly handled, a low-level error code (*e.g.*, an "I/O error" returned from a device driver) must be correctly propagated to the appropriate code in the file system. Further, if the file system is unable to recover from the fault, it may wish to pass the error up to the application, again requiring correct error propagation.

To analyze how errors are propagated in file and storage system code, we have developed a static source-code analysis technique. Our technique, named *Error Detection and Propagation (EDP)* analysis, shows how error codes flow through the file system and storage drivers. EDP performs a dataflow analysis by constructing a function-call graph showing how error codes propagate through return values and function parameters.

We have applied EDP analysis to all file systems and three major storage device drivers (SCSI, IDE, and Software RAID) implemented in Linux 2.6. We find that *error handling is occasionally correct*. Specifically, we see that low-level errors are sometimes lost as they travel through the many layers of the storage subsystem: out of the 9022 function calls through which the analyzed error codes propagate, we find that 1153 calls (13%) do not correctly save the propagated error codes.

Our detailed analysis enables us to make a number of conclusions. First, we find that the more complex the file system (in terms of both lines of code and number of function calls with error codes), the more likely it is to incorrectly propagate errors; thus, these more complex file systems are more likely to suffer from silent failures. Second, we observe that I/O write operations are more likely to neglect error codes than I/O read operations. Third, we find that many violations are not corner-case mistakes: the return codes of some functions are consistently ignored, which makes us suspect that the omissions are intentional. Finally, we show how inter-module calls play a major part in causing incorrect error propagation, but that chained propagations do not.

The rest of this paper is organized as follows. We first describe our methodology and present our results in Section 5.1 and 5.2 respectively. We then describe our deeper analysis in Section 5.3 in order to understand the root causes of the problem.

5.1 Methodology

To understand the propagation of error codes, we have developed a static analysis technique that we name *Error Detection and Propagation (EDP)*. In this section, we identify the components of Linux 2.6 that we will analyze and describe EDP.

5.1.1 Target Systems

In this paper, we analyze how errors are propagated through the file systems and storage device drivers in Linux 2.6.15.4. We examine all Linux implementations of file systems that are located in 51 directories. These file systems are of different types, including disk-based file systems, network file systems, file system protocols, and many others. Our analysis follows requests through the virtual file system and memory management layers as well. In addition to file systems, we also examine three major storage device drivers (SCSI, IDE, and software RAID), as well as all lower-level drivers. Beyond these subsystems, our tool can be used to analyze other Linux components as well.

5.1.2 EDP Analysis

The basic mechanism of EDP is a dataflow analysis: EDP constructs a function-call graph covering all cases in which error codes propagate through return values or function parameters. To build EDP, we harness the C Intermediate Language (CIL) [97]. CIL performs source-to-source transformation of C programs and thus

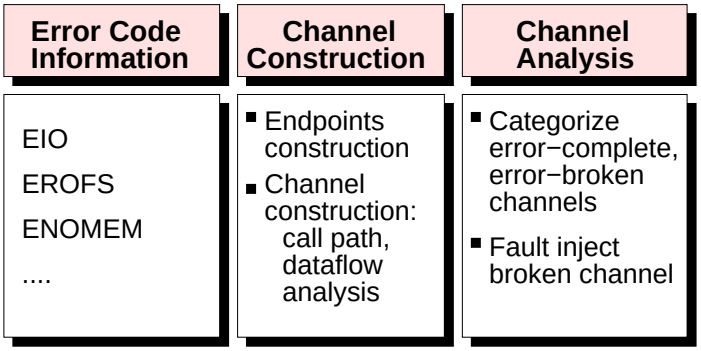


Figure 5.1: **EDP Architecture.** *The diagram shows the framework for Error Detection and Propagation (EDP) analysis of file and storage systems code.*

can be used in the analysis of large complex programs such as the Linux kernel. The EDP analysis is written as a CIL extension in 4000 lines of code in the OCaml language.

The abstraction that we introduce in EDP is that error codes flow along *channels*, where a channel is the set of function calls between where an error code is first generated and where it is terminated (*e.g.*, by being either handled or dropped). As shown in Figure 5.1, EDP contains three major components. The first component identifies the error codes that will be tracked. The second constructs the channels along which the error codes propagate. Finally, the third component analyzes the channels and classifies each as being either complete or broken.

Table 5.1 reports the EDP runtime for different subsystems, running on a machine with 2.4 GHz Intel Pentium 4 CPU and 512 MB of memory. Overall, EDP analysis is fast; analyzing all file systems together in a single run only takes 47 seconds. We now describe the three components of EDP in more detail.

Error Code Information

The first component of EDP identifies the error codes to track. One example is `EIO`, a generic error code that commonly indicates I/O failure and is used extensively throughout the file system; for example, in ext3, `EIO` touches 266 functions and propagates through 467 calls. Besides `EIO`, many kernel subsystems commonly use other error codes as defined in `include/asm-generic/errno.h`. In total, there are hundreds of error codes that are used for different purposes. We report our

Subsystem	**Single (seconds)**	**Full (seconds)**	Subsystem Size (Kloc)
VFS	**4**	–	34
Mem. Mgmt.	**3**	–	20
XFS	**8**	**13**	71
ReiserFS	**3**	**8**	24
ext3	**2**	**7**	12
Apple HFS	**1**	**6**	5
VFAT	**1**	**5**	1
All File Systems Together		**47**	372

Table 5.1: **EDP Performance.** *The table shows the EDP runtime for different subsystems. "Single" runtime represents the time to analyze each subsystem in isolation without interaction with other subsystems (e.g., VFS and MM). "Full" runtime represents the time to analyze a file system along with the virtual file system and the memory management. The last row reports the time to analyze all of the file systems together.*

findings on the propagation of 34 basic error codes that are mostly used across all file systems and storage device drivers. Table 5.2 lists these 34 basic error codes. These error codes can also be found in `include/asm-generic/errno-base.h`.

Channel Construction

The second component of EDP constructs the *channel* in which the specified error codes propagate. A channel can be constructed from function calls and asynchronous wake-up paths; in our current analysis, we focus only on function calls.

We define a channel by its two endpoints: generation and termination. The *generation endpoint* is the function that exposes an error code, either directly through a return value (*e.g.*, the function contains a `return -EIO` statement) or indirectly through a function argument passed by reference. After finding all generation endpoints, EDP marks each function that propagates the error codes; *propagating functions* receive error codes from the functions that they call and then simply propagate them in a return value or function parameter. The *termination endpoint* is the function in which an error code is no longer propagated in the return value or a parameter of the function.

One of the major challenges we address when constructing error channels is handling function pointers. The typical approach for handling function pointers is to implement a points-to analysis [67] that identifies the set of real functions each

Error Codes	Integer Value	Description
EPERM	1	Operation not permitted
ENOENT	2	No such file or directory
ESRCH	3	No such process
EINTR	4	Interrupted system call
EIO	5	I/O error
ENXIO	6	No such device or address
E2BIG	7	Argument list too long
ENOEXEC	8	Exec format error
EBADF	9	Bad file number
ECHILD	10	No child processes
EAGAIN	11	Try again
ENOMEM	12	Out of memory
EACCES	13	Permission denied
EFAULT	14	Bad address
ENOTBLK	15	Block device required
EBUSY	16	Device or resource busy
EEXIST	17	File exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	File table overflow
EMFILE	24	Too many open files
ENOTTY	25	Not a typewriter
ETXTBSY	26	Text file busy
EFBIG	27	File too large
ENOSPC	28	No space left on device
ESPIPE	29	Illegal seek
EROFS	30	Read-only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math argument out of domain of func
ERANGE	34	Math result not representable

Table 5.2: **34 Basic Error Codes.** *The table lists the 34 basic error codes that we analyze. These error codes can also be found in* `include/asm-generic/errno-base.h`.

function pointer might point at; however, field-sensitive points-to analyses can be expensive. Therefore, we customize our points-to analysis to exploit the systematic structure that these pointers exhibit.

First, we keep track of all structures that have function pointers. For example, the VFS read and write interfaces are defined as fields in the `file_ops` structure:

```
struct file_ops {
    int (*read)  ();
    int (*write) ();
};
```

Since each file system needs to define its own `file_ops`, we automatically find all global instances of such structures, look for the function pointer assignments within the instances, and map function-pointer implementations to the function pointer interfaces. For example, ext2 and ext3 define their file operations like this:

```
struct file_ops ext2_f_ops {
    .read  = ext2_read;
    .write = ext2_write;
};
struct file_ops ext3_f_ops {
    .read  = ext3_read;
    .write = ext3_write;
};
```

Given such global structure instances, we add the interface implementations (*e.g.*, `ext2_read`) to the implementation list of the corresponding interfaces (*e.g.*, `file_ops`→`read`). Although this technique connects most of the mappings, a function pointer assignment could still occur in an instruction rather than in a global structure instance. Thus, our tool also visits all functions and finds any assignment that maps an implementation to an interface. For example, if we find an assignment such as `f_op->read = ntfs_read`, then we add `ntfs_read` to the list of `file_ops`→`read` implementations.

In the last phase, we change function pointer calls to direct calls. For example, if VFS makes an interface call such as `(f_op->read)()`, then we automatically rewrite such an assignment to:

```
switch (...) {
    case ext2:  ext2_read();  break;
    case ext3:  ext3_read();  break;
    case ntfs:  ntfs_read();  break;
    ...
}
```

Across all Linux file systems and storage device drivers, there are 191 structural interfaces (*e.g.*, `file_ops`), 904 function pointer fields (*e.g.*, `read`), 5039 implementations (*e.g.*, `ext2_read`), and 2685 function pointer calls (*e.g.*, `(f_op->read)()`). Out of 2865 function pointer calls, we connect all except 564 calls (20%). The unconnected 20% of calls are due to indirect implementation assignment. For example, we cannot map assignment such as `f_op->read = f`, where `f` is either a local variable or a function parameter, and not a function name. While it is feasible to traceback such assignments using stronger and more expensive analysis, we assume that major interfaces linking modules together have already been connected as part of global instances. If all calls are connected, more of the error propagation chain can be analyzed, which means more violations are likely to be found.

Channel Analysis

The third component of EDP distinguishes two kinds of channels: error-complete and error-broken channels. An *error-complete* channel is a channel that minimally checks the occurrence of an error. An error-complete channel thus has this property at its termination endpoint:

$$\exists \;\; if\,(expr)\;\{\;...\;\},\;\; where \\ errorCodeVariable \subseteq expr$$

which states that an error code is considered checked if there exist an `if` condition whose expression contains the variable that stores the error code. For example, the function in the code segment below carries an error-complete channel because the function saves the returned error code (line 2) and checks the error code (line 3):

```
1 void goodTerminationEndpoint() {
2     int err = generationEndpoint();
3     if (err)
4         ...
5 }
6 int generationEndpoint() {
7     return -EIO;
8 }
```

Note that an error could be checked but not handled properly (*e.g.*, no error handling in the `if` condition). Since error handling is usually specific to each file system, and hence there are many instances of it, we decided to be "generous" in the way we define how error is handled (*i.e.*, by just checking it). More violations might be found when we incorporate all instances of error handling.

An *error-broken* channel is the inverse of an error-complete channel. In particular, the error code is either *unsaved*, *unchecked*, or *overwritten*. For example, the function below carries an error-broken channel of unchecked type because the function saves the returned error code (line 2) but it never checks the error before the function exits (line 3):

```
1 void badTerminationEndpoint() {
2     int err = generationEndpoint();
3     return;
4 }
```

An error-broken channel is a serious file system bug because it can lead to a silent failure. In a few cases, we inject faults in error-broken channels to confirm the existence of silent failures. We utilize our block-level fault injection technique (described in Section 2.4) to exercise error-broken channels that relate to disk I/O. In a broken channel, we look for two pieces of information: which workload and which failure led us to that channel. After finding the necessary information, we run the workload, inject the specific block failure, and observe the I/O traces and the returned error codes received in upper layers (*e.g.*, the application layer) to confirm whether a broken channel leads to a silent failure. The reader will note that our fault-injection technique is limited to disk I/O related channels. To exercise all error-broken channels, techniques such as symbolic execution and directed testing [42, 49] that simulate the environment of the component in test would be of great utility.

Limitations

Error propagation has complex characteristics: correct error codes must be returned; each subsystem uses both generic and specific error codes; one error code could be mapped to another; error codes are stored not only in scalar variables but also in structures (*e.g.*, control blocks); and error codes flow not only through function calls but also asynchronously via interrupts and callbacks. In our static analysis, we have not modeled all these characteristics. Nevertheless, by just focusing on the propagation of basic error codes via function call, we have found numerous violations that need to be fixed. A more complete tool that covers the properties above would uncover even more incorrect error handling.

5.2 Results

We have performed EDP analysis on all file systems and storage device drivers in Linux 2.6.15.4. Our analysis studies how 34 basic error codes (listed in Table 5.2) propagate through these subsystems. We examine these basic error codes because they involve thousands of functions and propagate across thousands of calls.

In these results, we distinguish three types of violations that make up an error-broken channel: unsaved, unchecked, and overwritten error codes. An *unsaved error code* (Section 5.2.1) is found when a callee propagates an error code via the return value, but the caller does not save the return value (*i.e.*, it is treated as a void-returning call even though it actually returns an error code). Throughout the paper, we refer to this type of broken channel as a "*bad call*." An *unchecked error code* (Section 5.2.2) is found when a variable that may contain an error code is neither checked nor used in the future; we always refer to this case as an unchecked code. An *overwritten error code* (Section 5.2.3) is found when the container that holds the error code is overwritten with another value before the previous error is checked.

5.2.1 Unsaved Error Codes

First, we report the number of error-broken channels due to a caller simply not saving the returned error code (*i.e.*, the number of bad calls). The simplified HFS code below shows an example of an unsaved error code. The function `find_init` accepts a new uninitialized `find_data` structure (line 2), allocates a memory space for the `search_key` field (line 3), and returns the `ENOMEM` error code when the memory allocation fails (line 5). However, one of its callers, `file_lookup`, does not save the returned error code (line 10) but tries to access the `search_key` field which still points to `NULL` (line 11). Hence, a null-pointer dereference takes place and the system could crash or corrupt data.

```
 1 // hfs/bfind.c
 2 int find_init(find_data *fd) {
 3     fd->search_key = kmalloc(..)
 4     if (!fd->search_key)
 5         return -ENOMEM;
 6     ...
 7 }
 8 // hfs/inode.c
 9 int file_lookup() {
10     find_init(fd); /* NOT-SAVED E.C */
11     fd->search_key->cat = ...; /* BAD!! */
```

```
12      ...
13 }
```

To show how EDP is useful in finding error propagation bugs, we begin by showing a sample of EDP analysis for a simple file system, Apple HFS (Section 5.2.1). Then, we present our findings on all subsystems that we analyze (Section 5.2.1). Finally, we discuss false positives (Section 5.2.1) and serious silent failures caused by unsaved error codes (Section 5.2.1).

EDP on Apple HFS

Figures 5.2 and 5.3 depict the EDP output when analyzing the propagation of the 34 basic error codes in the Apple HFS file system. There are two important elements that EDP produces in order to ease the debugging process. First, EDP generates an error propagation graph (Figure 5.2) that only includes functions and function calls through which the analyzed error codes propagate. From the graph, one can easily catch all bad calls and functions that make the bad calls. Second, EDP provides a table (Figure 5.3) that presents more detailed information for each bad call (*e.g.*, the location where the bad call is made).

Using the information that EDP provides, we found three major error-handling inconsistencies in HFS. First, 11 out of 14 calls to `find_init` drop the returned error codes. As described earlier in this section, this bug could cause the system to crash or corrupt data. Second, 4 out of 5 total calls to the function `__brec_find` are bad calls (as indicated by the four black edges, E, D, N, and Q, found in the lower left of the graph). The task of this function is to find a record in an HFS node that best matches the given key, and return `ENOENT` (no entry) error code if it fails. The only call that saves this error code is made by the wrapper, `brec_find`. Interestingly, all 18 calls to this wrapper propagate the error code properly (as indicated by all gray edges coming into the function).

Finally, 3 out of 4 calls to `free_exts` do not save the returned error code (labeled R, I, and J). This function traverses a list of extents and locates the extents to be freed. If the extents cannot be found, the function returns `EIO`. More interestingly, the developer wrote a comment “panic?” just before the return statement (maybe in the hope that in this failure case the callers will call panic, which will never happen if the error code is dropped). By and large, we found similar inconsistencies in all the subsystems we analyzed.

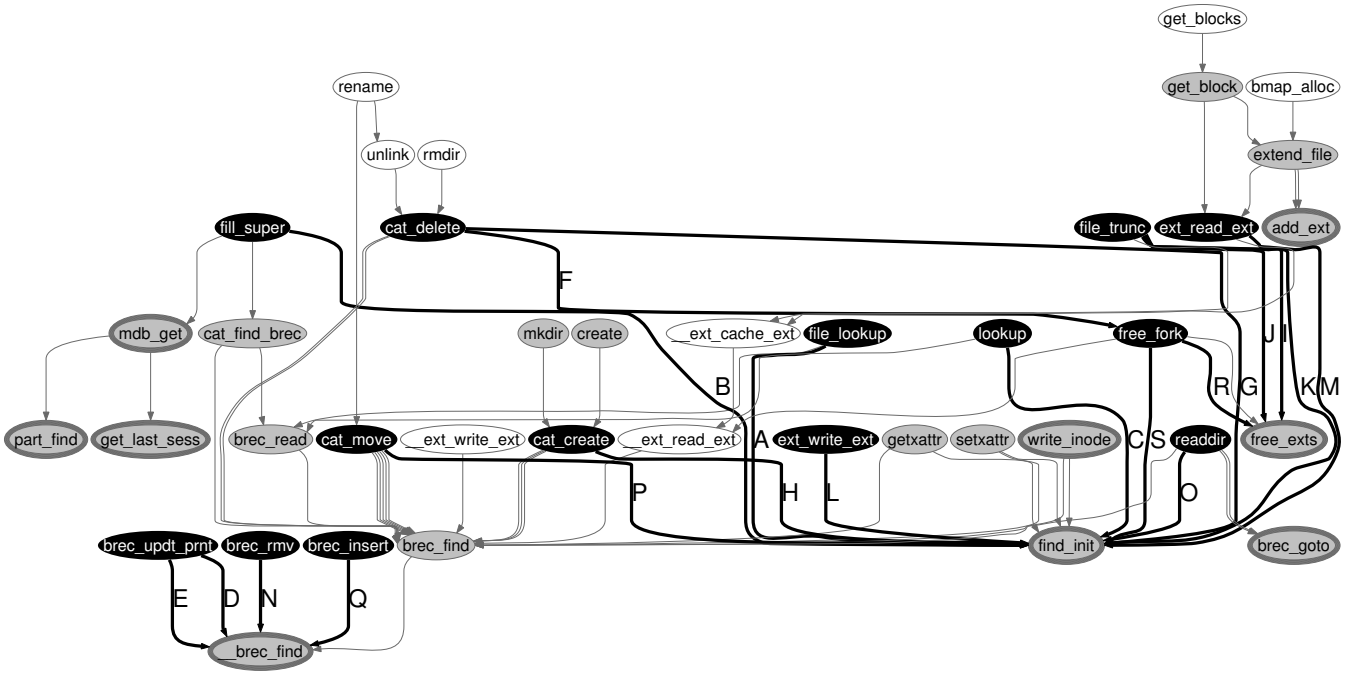
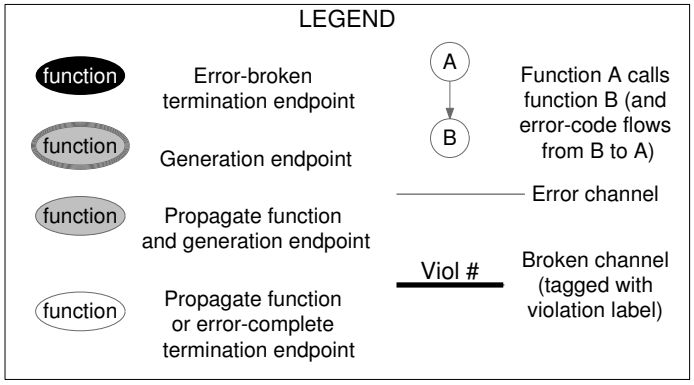


Figure 5.2: **A Sample of EDP Output (The Graph).** *The figure depicts the EDP output for the HFS file system. Some function names have been shortened to improve readability. As summarized in the legend in Figure 5.3, a gray node with a thicker border represents a function that generates an error code. The other gray node represents the same thing, but the function also propagates the error code received from its callee. A white node represents a good function, i.e. it either propagates the error code to its caller or if it does not propagate the error code it minimally checks the error code. A black node represents an error-broken termination endpoint, i.e. it is a function that commits the violation of unsaved error codes. The darker and thicker edge coming out from a black node implies a broken error channel (a bad call); an error code actually flows from its callee, but the caller drops the error code. For ease of debugging, each bad call is labeled with a violation number whose detailed information can be found in the violation table in Figure 5.3. For example, violation #E found in the bottom left corner of the graph is a bad call made by* `brec_updt_prnt` *when calling* `__brec_find`*, which can be located in* `fs/hfs/brec.c` *line 345.*



Viol#	Caller → Callee		Filename	Line#
A	file_lookup	find_init	inode.c	493
B	fill_super	find_init	super.c	385
C	lookup	find_init	dir.c	30
D	brec_updt_prnt	__brec_find	brec.c	405
E	brec_updt_prnt	__brec_find	brec.c	345
F	cat_delete	free_fork	catalog.c	228
G	cat_delete	find_init	catalog.c	213
H	cat_create	find_init	catalog.c	95
I	file_trunc	free_exts	extent.c	507
J	file_trunc	free_exts	extent.c	497
K	file_trunc	find_init	extent.c	494
L	ext_write_ext	find_init	extent.c	135
M	ext_read_ext	find_init	extent.c	188
N	brec_rmv	__brec_find	brec.c	193
O	readdir	find_init	dir.c	68
P	cat_move	find_init	catalog.c	280
Q	brec_insert	__brec_find	brec.c	145
R	free_fork	free_exts	extent.c	307
S	free_fork	find_init	extent.c	301

Figure 5.3: **A Sample of EDP Output (The Table and Legend).** *The top legend describes the graph in Figure 5.2. For ease of debugging, each bad call is labeled with a violation number whose detailed information can be found in the bottom violation table. For example, violation #E found in the bottom left corner of the graph in Figure 5.2 is a bad call made by* `brec_updt_prnt` *when calling* `__brec_find`*, which can be located in* `fs/hfs/brec.c` *line 345.*

EDP on All File Systems and Storage Drivers

Figures 5.4 to 5.9 show EDP outputs for six more file systems whose error-propagation graphs represent an interesting sample. EDP outputs for the rest of the file systems can be downloaded from our web site [57]. A small file system such as HFS+ has simple propagation chains, yet bad calls are still made. More complex error propagation can be seen in ext3, ReiserFS, and IBM JFS; within these file systems, error-codes propagate throughout 180 to 340 function calls. The error propagation in NFS is more structured compared to other file systems. Finally, among all file systems we analyze, XFS has the most complex error propagation chain; almost 1500 function calls propagate error-codes. Note that each graph in the figures was produced by analyzing each file system in isolation (*i.e.*, the graph only shows intra-module but not inter-module calls), yet they already illustrate the complexity of error code propagation in each file system. Manual code inspection would require a tremendous amount of work to find error-propagation bugs.

Next, we analyzed the propagation of error codes across all file systems and storage device drivers as a whole. All inter-module calls were connected by our EDP channel constructor, which connects all function pointer calls; hence, we were able to catch inter-module bad calls in addition to intra-module ones. Tables 5.3, 5.4, and 5.5 summarize our findings. Note that the number of violations reported is higher than the ones reported in the figures because we catch more bugs when we analyze each file system in conjunction with other subsystems (*e.g.*, ext3 with the journaling layer, VFS, and memory management).

Surprisingly, out of 9022 error channels, 1153 (nearly 13%) constitute bad calls. This appears to be a long-standing problem. We ran a partial analysis in Linux 2.4 and found that the magnitude of incomplete error code propagation is essentially the same; we found 61 bad calls in ext3 in Linux 2.4.20 , vs. 80 in 2.6.15. In Section 5.3, we try to dissect the root causes of this problem.

False Positives

It is important to note that while the number of bad calls is high, not all bad calls could cause damage to the system. The primary reason is what we call a *double error code*; some functions expose two or more error codes at the same time, and checking one of the error codes while ignoring the others can still be correct. For example, in the ReiserFS code below, the error code returned from `sync_dirty_buffer` does not have to be saved (line 8) *if and only if* the function performs the check on the second error code (line 9); the buffer must be checked whether it is is up-to-date.

HFS+ [22 bad / 84 calls, 26%]

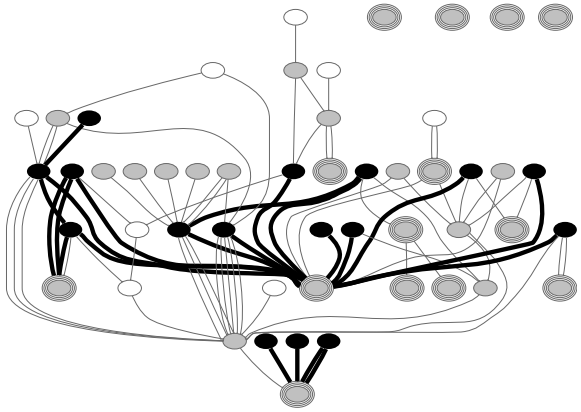


Figure 5.4: **EDP output for HFS+.** *The figures illustrate the prevalent problem of incomplete error-propagation across different types of file systems. Details such as function names and violation numbers have been removed. Gray edges represent calls that propagate error codes. Black edges represent bad calls. The number of edges are reported in [X / Y , Z%] format where X and Y represent the number of black and all (gray and black) edges respectively, and Z represents the fraction of X and Y. For more information, please see the legend in Figure 5.3.*

ext3 [37 bad / 188 calls, 20%]

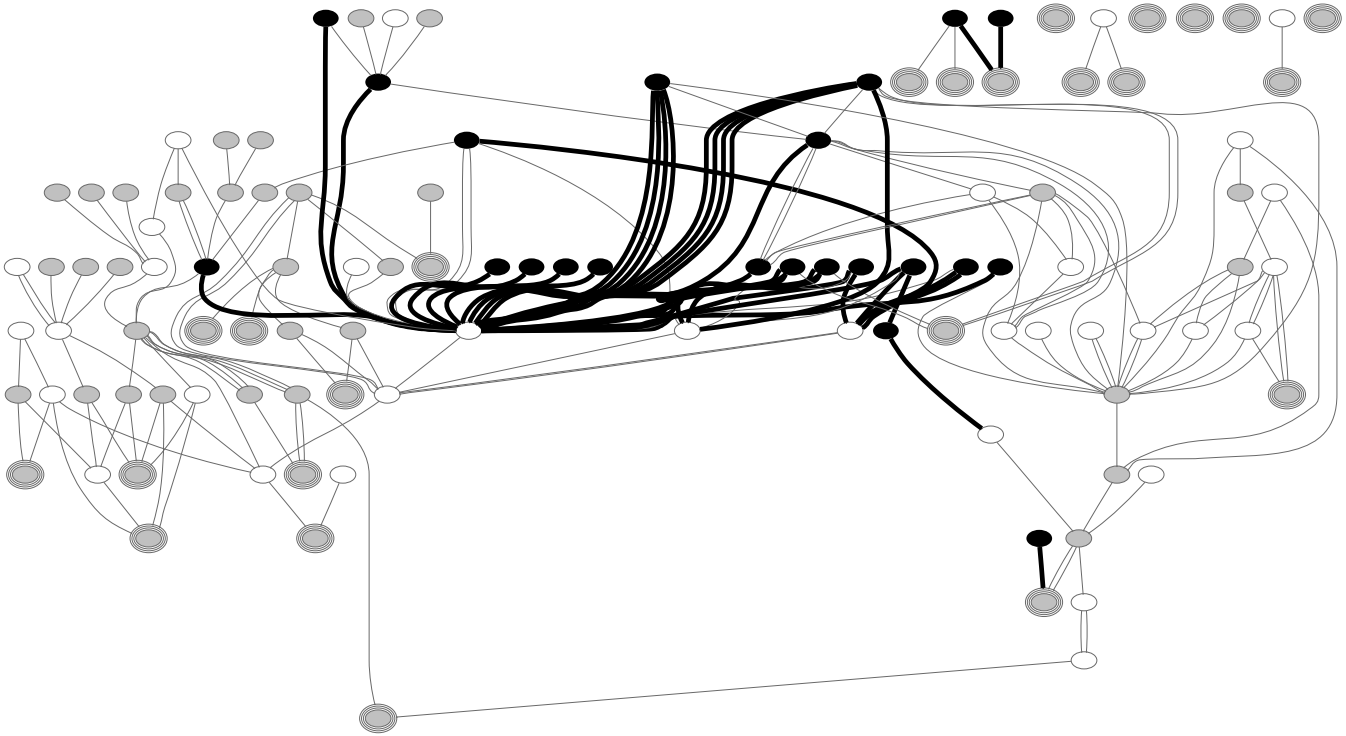


Figure 5.5: **EDP output for ext3.** *Please see caption in Figure 5.4.*

ReiserFS [35 bad / 218 calls, 16%]

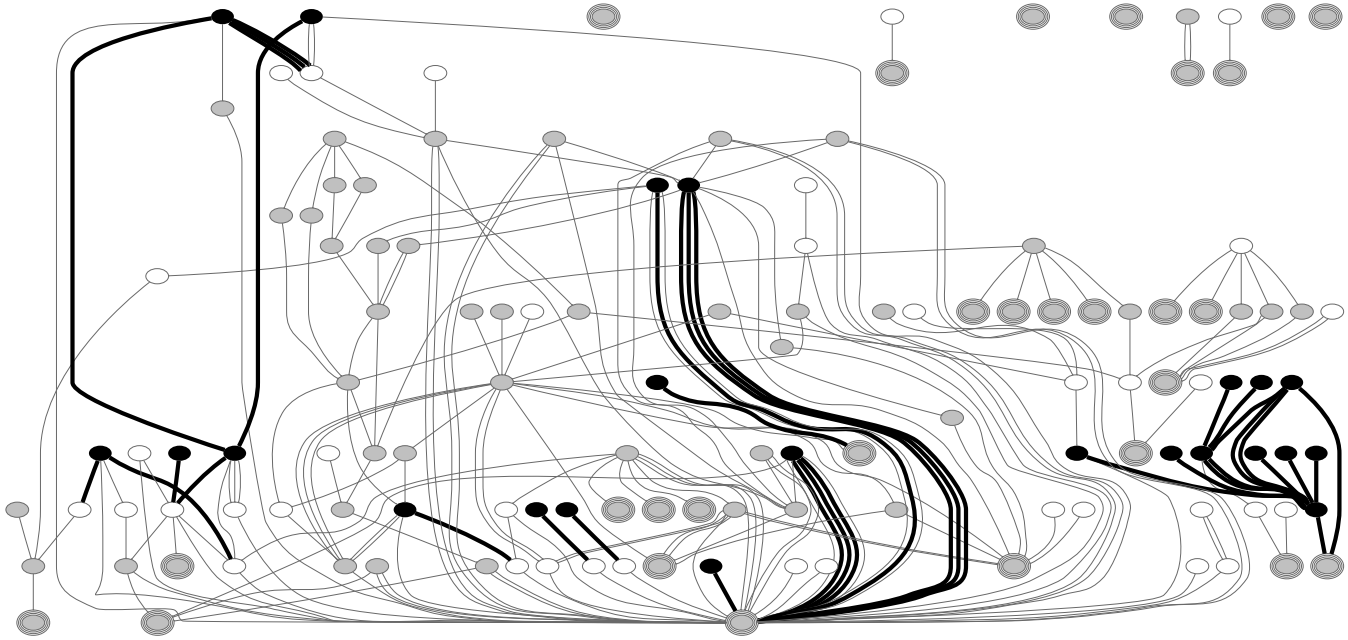


Figure 5.6: **EDP output for ReiserFS.** *Please see caption in Figure 5.4.*

IBM JFS [61 bad / 340 calls, 18%]

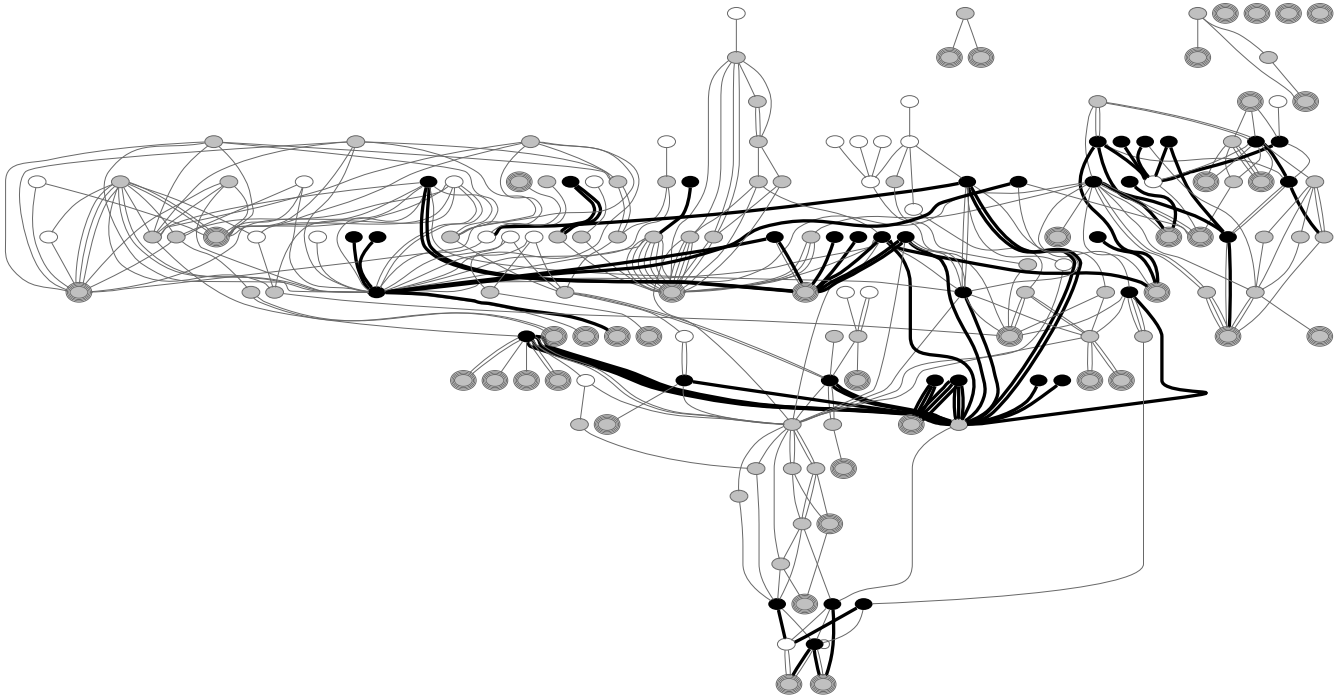


Figure 5.7: **EDP output for IBM JFS.** *Please see caption in Figure 5.4.*

NFS Client [54 bad / 446 calls, 12%]

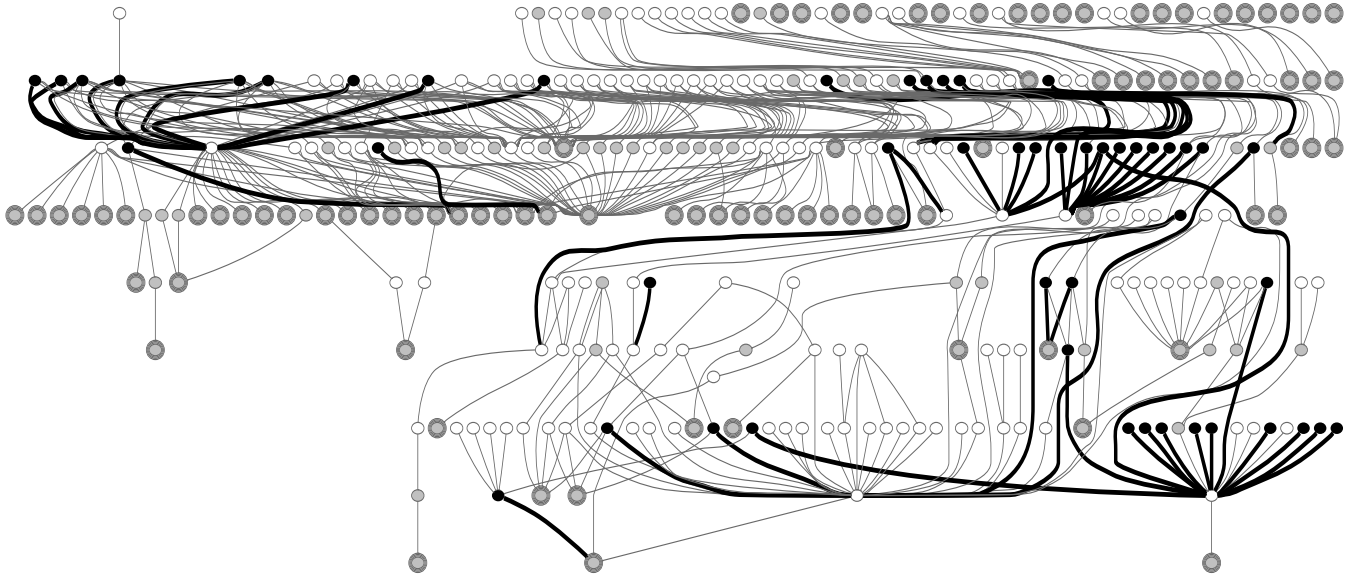


Figure 5.8: **EDP output for NFS Client.** *Please see caption in Figure 5.4.*

XFS [105 bad / 1453 calls, 7%]

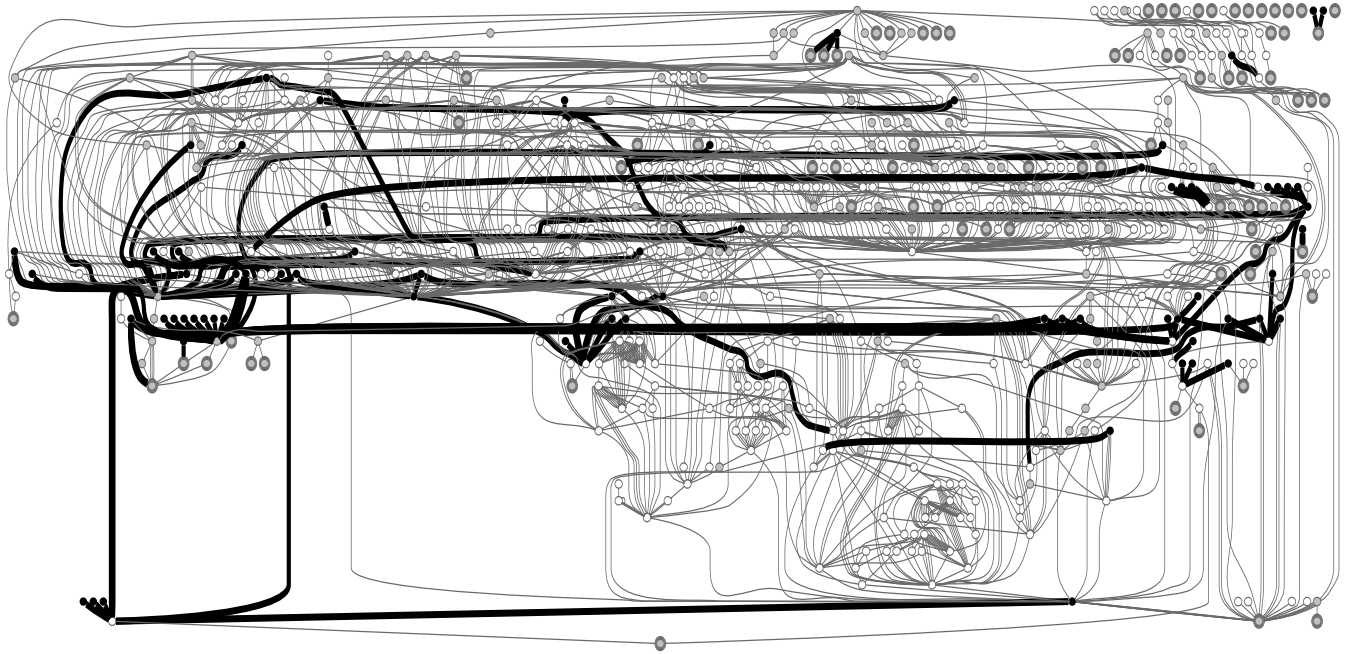


Figure 5.9: **EDP output for XFS.** *Please see caption in Figure 5.4.*

File Systems

	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
XFS	**101**	1457	71	6.9	1.4
Virtual FS	**96**	1149	34	8.4	2.9
IBM JFS	**95**	390	17	24.4	5.6
ext3	**80**	362	12	22.1	7.2
NFS Client	**62**	482	18	12.9	3.6
CIFS	**43**	339	21	12.7	2.1
ReiserFS	**42**	399	24	10.5	1.8
Mem. Mgmt.	**40**	351	20	11.4	2.0
Apple HFS+	**25**	98	7	25.5	3.7
JFFS v2	**24**	153	11	15.7	2.2
Apple HFS	**20**	76	5	26.3	4.8
SMB	**19**	196	6	9.7	3.5
ext2	**18**	103	6	17.5	3.3
AFS	**16**	62	7	25.8	2.6
NTFS	**15**	186	18	8.1	0.9
NFS Server	**15**	265	14	5.7	1.2
NCP	**13**	169	5	7.7	2.6
UFS	**12**	44	5	27.3	2.6
JBD	**10**	43	4	23.3	2.6
FAT	**9**	81	4	11.1	2.9
Plan 9	**9**	80	4	11.2	2.4
System V	**7**	30	3	23.3	3.2
JFFS	**7**	56	5	12.5	1.4
UDF	**6**	50	9	12.0	0.7
MSDOS	**5**	39	1	12.8	9.3
VFAT	**4**	39	1	10.3	5.0
Minix	**4**	31	4	12.9	1.2

Table 5.3: **Error-broken channels due to unsaved error codes.** *Tables 5.3, 5.4 and 5.5 report the number of bad calls found across all file systems and storage device drivers in Linux 2.6.15.4. In each table, from left to right column we report the name of the subsystem, the number of bad calls, the number of error channels (i.e., the number of calls to functions that propagate error codes), the size of the subsystem, the fraction of bad calls over all error-related calls (ratio of 2nd and 3rd column), and finally the number of violations per Kloc (ratio of 2nd and 4th column). We categorize a directory as a subsystem.*

File Systems (Cont'd)

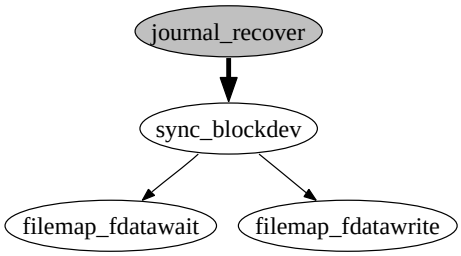
	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
FUSE	**4**	48	3	8.3	1.5
Automounter4	**4**	53	2	7.5	2.7
NFS Lockd	**3**	21	4	14.3	0.8
Relayfs	**2**	5	1	40.0	2.7
Partitions	**2**	3	4	66.7	0.6
ISO	**2**	19	3	10.5	0.7
HugeTLB Sup	**2**	10	1	20.0	3.0
Compr. ROM	**2**	3	1	66.7	4.5
ADFS	**2**	30	2	6.7	1.3
sysfs sup.	**1**	29	2	3.4	0.8
romfs sup.	**1**	3	1	33.3	2.4
ramfs sup.	**1**	6	1	16.7	6.0
QNX 4	**1**	8	2	12.5	0.9
proc fs sup.	**1**	44	6	2.3	0.2
OS/2 HPFS	**1**	18	6	5.6	0.2
FreeVxFS	**1**	4	2	25.0	0.7
EFS	**1**	3	1	33.3	1.4
devpts	**1**	2	1	50.0	6.2
Boot FS	**1**	9	1	11.1	1.2
BeOS	**1**	5	3	20.0	0.5
Automounter	**1**	41	2	2.4	1.0
Amiga FFS	**1**	34	3	2.9	0.3
exportfs sup.	**0**	1	1	0.0	0.0
Coda	**0**	149	3	0.0	0.0
Total	**833**	7278	366	–	–
Average	**16.3**	142.7	7.2	**17.0**	**2.4**

Table 5.4: **Error-broken channels due to unsaved error codes (Cont'd).** *Tables 5.3, 5.4 and 5.5 report the number of bad calls found across all file systems and storage device drivers in Linux 2.6.15.4. Please see the caption in Table 5.3.*

Storage Drivers

	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
SCSI (root)	**123**	628	198	19.6	0.6
IDE (root)	**53**	223	15	23.8	3.5
Block Dev (root)	**39**	195	36	20.0	1.1
Software RAID	**31**	290	32	10.7	1.0
SCSI (aacraid)	**30**	76	7	39.5	4.8
SCSI (lpfc)	**14**	30	16	46.7	0.9
Blk Dev (P-IDE)	**11**	17	8	64.7	1.5
SCSI aic7xxx	**8**	62	37	12.9	0.2
IDE (pci)	**5**	106	12	4.7	0.4
IDE legacy	**2**	3	3	66.7	0.8
Blk Layer Core	**2**	65	8	3.1	0.3
SCSI megaraid	**1**	30	6	3.3	0.2
Blk Dev (Eth)	**1**	5	2	20.0	0.7
SCSI (sym53c8)	**0**	6	10	0.0	0.0
SCSI (qla2xxx)	**0**	8	49	0.0	0.0
Total	**320**	1744	430	–	–
Average	**21.3**	116.3	28.6	**22.4**	**1.1**

Table 5.5: **Error-broken channels due to unsaved error codes (Cont'd).** *Tables 5.3, 5.4 and 5.5 report the number of bad calls found across all file systems and storage device drivers in Linux 2.6.15.4. We categorize a directory as a subsystem. Thus, for storage drivers, since different SCSI device drivers exist in the first-level of the* `scsi/` *directory, we put all of them as one subsystem. SCSI device drivers that are located in different directories (e.g.,* `scsi/lpfc/`, `scsi/aacraid/`*) are categorized as different subsystems. The same principle is applied to IDE. Please see the caption in Table 5.3.*




```
journal_recover()
  /* BROKEN CHANNEL */
  sync_blockdev();

sync_blockdev()
  ret = fm_fdatawrite();
  err = fm_fdatawait();
  if(!ret) ret = err;
  /* PROPAGATE EIO */
  return ret;
```

Figure 5.10: **Silent error in journal recovery.** *In the figure on the left, EDP marks `journal_recover` as a termination endpoint of a broken channel. The code snippet on the right shows that `journal_recover` ignores the `EIO` propagated by `sync_blockdev`.*

```
 1 // fs/buffer.c
 2 int sync_dirty_buffer (buffer_head* bh) {
 3     ...
 4     return ret; // RETURN ERROR CODE
 5 }
 6 // reiserfs/journal.c
 7 int flush_commit_list() {
 8     sync_dirty_buffer(bh); // UNSAVED EC
 9     if (!buffer_uptodate(bh)) {
10         return -EIO;
11     }
12 }
```

To ensure that the number of false positives we report is not overly large, we manually analyze the code snippets around the bad calls we found to check whether a second error code is being checked. Note that this manual process can be automated if we incorporate all types of error codes into EDP. We have found only a total of 39 false positives out of 1192 bad calls, which have been excluded from the numbers we report in this paper. Thus, the high numbers in Tables 5.3, 5.4, and 5.5 provide a hint to a real and critical problem.

Silent Failures: Manifestations of Unsaved Error Codes

To show that unsaved error codes represent a serious problem that can lead to silent failures, we injected disk block failures in two subsystems, JBD and NFS. For injecting the faults, we use our methodology described in Section 2.4.

First, as shown in Figure 5.10, one serious silent failure arises during file system recovery: the journaling block device layer (JBD) does not properly propagate any block write failures, including inode, directory, bitmap, superblock, and other block write failures. EDP unearths these silent failures by pinpointing the `journal_recover` function, which is responsible for file system recovery, as it calls `sync_blockdev` to flush the dirty buffer pages owned by the block device. Unfortunately, `journal_recover` does not save the error code propagated by `sync_blockdev` in the case of block write failures. This is an example where the error code is dropped in the middle of its propagation chain; `sync_blockdev` correctly propagates the `EIO` error codes received from the two function calls it makes.

Second, a similar problem occurs in the NFS server code. From a similar failure injection experiment, we found that the NFS client is not informed when a write failure occurs during a `sync` operation. In the experiment, the client updates old data and then sends a `sync` operation with the data to the NFS server. The NFS server then invokes the `nfsd_dosync` operation, which mainly performs three operations similar to the `sync_blockdev` call above. First, the NFS server writes dirty pages to the disk; second, it writes dirty inodes and the superblock to disk; third, it waits until the ongoing I/O data transfer terminates. All these three operations could return error codes, but the implementation of `nfsd_dosync` does not save any return values. As a result, the NFS client will never notice any disk write failures occurring in the server. Thus, even a careful, error-robust client cannot trust the server to inform it of errors that occur.

In the NFS server code, we might expect that at least one return value would be saved and checked properly. However, no return values are saved, leading one to question whether the returned error codes from the `write` or `sync` operations are correctly handled in general. It could be the case that the developers are not concerned about write failures. We investigate this hypothesis in Section 5.3.2.

5.2.2 Unchecked Error Codes

Lastly, we report the number of error-broken channels due to a variable that contains an error code not being checked or used in the future. For example, in the IBM JFS code below, `rc` carries an error code propagated from `txCommit` (line 4), but `rc` is never checked.

```
1 // jfs/jfs_txnmgr.c
2 int jfs_sync () {
3     int rc;
4     rc = txCommit(); // UNCHECKED 'rc'
```

```
5     // No usage or check of 'rc'
6     // after this line
7 }
```

This analysis can also report false positives due to the double error code problem described previously. In addition, we also find the problem of *overloaded variables* that contribute as false positives. We define a variable to be overloaded if the variable could contain an error code or a data value. For instance, `blknum` in the QNX4 code below is an example of an overloaded variable:

```
1  // qnx4/dir.c
2 int qnx4_readdir () {
3     int blknum;
4     struct buffer_head *bh;
5     blknum = qnx4_block_map();
6     bh = sb_bread (blknum);
7     if (bh == NULL)
8         // error
9 }
```

In this code, `qnx4_block_map` could return an error code (line 5), which is usually a negative value. `sb_bread` takes a block number and returns a buffer head that contains the data for that particular block (line 6). Since a negative block number will lead to a `NULL` buffer head (line 7), the error code stored in `blknum` does not have to be explicitly checked. The developer believes that the other part of the code will catch this error or eventually raise related errors. This practice reduces the accuracy of our static analysis.

Since the number of unchecked error code reports is small (only 21 reported), we were able to remove the false positives and find a total of 3 unchecked error codes in file systems (CIFS, NFS Server, and JFS) and 2 in storage drivers (software RAID and loopback driver).

5.2.3 Overwritten Error Codes

Broken channels can also be caused by *overwritten error codes*, in which the container that holds the error code is overwritten with another value before the previous error is checked. For example, the CIFS code below overwrites (line 6) the previous error code received from another call (line 4).

```
1 // cifs/transport.c
2 int SendReceive () {
```

```
 3     int rc;
 4     rc = cifs_sign_smb(); // PROPAGATE E.C.
 5     ... // No use of 'rc' here
 6     rc = smb_send(); // OVERWRITTEN
 7 }
```

Currently, EDP detects overwritten error codes, but reports too many false positives to be useful. The biggest problem we have encountered is due to the nature of the error hierarchy: in many cases, a less critical error code is overwritten with a more critical one. For example, in the memory management code below, when first encountering a page error, the error code is set to `EIO` (line 6). Later, the function checks whether the flags of a `map` structure carry a no-space error code (line 8). If so, the `EIO` error code is overwritten (line 9) with a new error code `ENOSPC`.

```
 1 // mm/filemap.c
 2 int wait_on_page_writeback_range (pg, map) {
 3     int ret = 0;
 4     ...
 5     if (PageError(pg))
 6         ret = -EIO;
 7     ...
 8     if (test_bit(AS_ENOSPC, &map->flags))
 9         ret = -ENOSPC;
10     if (test_bit (AS_EIO, &map->flags))
11         ret = -EIO;
12     return ret;
13 }
```

Manually inspecting the results obtained from EDP (only 12 reported), we have identified five real cases of overwritten error codes: one each in AFS and FAT, and three in CIFS.

5.3 Analysis of Results

In the following sections, we present four analyses whereby we try to uncover the root causes and impact of incomplete error propagation. Since the number of unchecked and overwritten error codes is small, we only consider unsaved error codes (bad calls) in our analyses; thus we use "bad calls" and "broken channels" interchangeably from now on. First, we made a correlation between robustness and complexity (Section 5.3.1). Second, we analyzed whether file systems and storage device drivers give different treatment to errors occurring in I/O read vs. I/O write

	By % Broken		By Viol/Kloc	
Rank	FS	Frac.	FS	Viol/Kloc
1	IBM JFS	24.4	ext3	7.2
2	ext3	22.1	IBM JFS	5.6
3	JFFS v2	15.7	NFS Client	3.6
4	NFS Client	12.9	VFS	2.9
5	CIFS	12.7	JFFS v2	2.2
6	MemMgmt	11.4	CIFS	2.1
7	ReiserFS	10.5	MemMgmt	2.0
8	VFS	8.4	ReiserFS	1.8
9	NTFS	8.1	XFS	1.4
10	XFS	6.9	NFS Server	1.2

Table 5.6: **Least Robust File Systems.** *The table shows the ten least robust file systems using two ranking systems. In the first ranking system, file system robustness is ranked based on the fraction of broken channels over all error channels (the 5th column of Table 5.3). The second ranking system sorts file systems based on the number of broken channels found in every Kloc (the 6th column of Table 5.3).*

operations (Section 5.3.2). From that analysis we find that many write errors are neglected; hence we perform the next study in which we try to answer whether ignored errors are corner-case mistakes or intentional choices (Section 5.3.3). In the final analysis, we analyze whether chained error propagation and inter-module calls play major parts in causing incorrect error propagation (Section 5.3.4).

5.3.1 Complexity and Robustness

In our first analysis, we would like to correlate the number of mistakes in a subsystem with the complexity of that subsystem. For file systems, XFS with 71 Kloc has more mistakes than other, smaller file systems. However, this does not necessarily imply that XFS is the least robust file system. Table 5.6 sorts the robustness of each file system based on two rankings: *percentage-broken* and *viol/kloc* rankings. In the first ranking system, file system robustness is ranked based on the fraction of broken channels over all error channels (the 5th column of Table 5.3). The second ranking system sorts file systems based on the number of broken channels found in every Kloc (the 6th column of Table 5.3). In both rankings, we only include file systems that are at least 10 Kloc in size with at least 50 error-related calls (*i.e.* we only consider "complex" file systems).

A noteworthy observation is that ext3 and IBM JFS are ranked as the two least

robust file systems. This fact affirms our earlier findings on the robustness of ext3 and IBM JFS [106]. In this prior work, we found that ext3 and IBM JFS are inconsistent in dealing with different kinds of disk failures. Thus, it might be the case that these inconsistent policies correlate with inconsistent error propagation.

Among storage device drivers, it is interesting to compare the robustness of the SCSI and IDE subsystems. If we compare SCSI and IDE subsystems using the percentage-broken ranking system, SCSI and IDE are almost comparable (21% vs. 18%). However, if we compare them based on the viol/kloc ranking system, then the SCSI subsystem is almost four times more robust than IDE (0.6 vs. 2.1 errors/Kloc). Nevertheless it seems the case that SCSI utilizes basic error codes much more than IDE does.

When the robustness of storage drivers and file systems is compared using the percentage-broken ranking, on average storage drivers are less robust compared to file systems (22% vs. 17%, as reported in the last rows of Table 5.3). On the other hand, in the viol/kloc ranking system, storage drivers are more robust compared to file systems (1.1 vs. 2.4 mistakes/Kloc). From our point of view, the percentage-broken ranking system is more valid because a subsystem could be comprised of submodules that do not necessarily use error codes; what is more important is the number of bad calls in the population of all error-related calls.

5.3.2 Neglected Write Errors

As mentioned in Section 5.2.1, we have observed that error codes propagated in `write` or `sync` operations are often ignored. Thus, we investigate how many write errors are neglected compared to read errors. This study is motivated by our findings in that section as well as by our earlier findings that at least for ext3, read failures are detected, but write errors are often ignored [106].

To perform this study, we filter out calls that do not relate to read and write operations. Since it is impractical to do that manually, we use a simple string comparison to mark calls that are relevant to our analysis. That is we only take a caller→callee pair where the callee contains the string `read`, `write`, `sync`, or `wait`. We include `wait`-type calls because in many cases `wait`-type callees (*e.g.*, `filemap_datawait`) represent waiting for one or more I/O operations and could return error information on the operation. Thus, in our study, `write`-, `sync`-, and `wait`-type calls are categorized as write operations.

The upper half of Table 5.7 reports our findings. The last column shows how often errors are ignored in the file system code. Interestingly, file systems have a tendency to correctly handle error codes propagated from `read`-type calls, but not those from `write`-type calls (4.3% vs. 19.6%). The 29 (4.3%) unsaved read error

Callee Type	Bad Calls	EC Calls	**Frac. (%)**
Read*	26	603	**4.3**
Sync	70	236	**29.7**
Wait	27	70	**38.6**
Write	80	598	**13.4**
Sync+Wait+Write	177	904	**19.6**
Specific Callee			
`filemap_fdatawait`	22	29	**75.9**
`filemap_fdatawrite`	30	47	**63.8**
`sync_blockdev`	15	21	**71.4**

Table 5.7: **Neglected write errors in file system code.** *The table shows that read errors are handled more correctly than write errors. The upper table shows the fraction of bad calls over four category of calls: read, sync, wait, and write. The later three can be categorized as a write operation. The lower table shows neglected write errors for three specific functions. The 29 (*) violated read calls are all related to readahead and asynchronous read; in other words, all error codes returned in synchronous reads are being saved and checked.*

codes are all found in readahead operations in the memory management subsystem; it might be acceptable to ignore prefetch read errors because such reads can be reissued in the future whenever the page is actually read.

As discussed in Section 5.2.1, a function could return more than one error code at the same time, and checking only one of them suffices. However, if we know that a certain function only returns a single error code and yet the caller does not save the return value properly, then we know that such a call is really a flaw. To find real flaws in the file system code, we examined three important functions that we know only return single error codes: `sync_blockdev`, `filemap_fdatawrite`, and `filemap_fdatawait`. A file system that does not check the returned error codes from these functions would obviously let failures go unnoticed in the upper layers.

The lower half of Table 5.7 reports our findings. Many error codes returned from the three methods are simply not saved ($> 63\%$ in all cases). Two conclusions might be drawn from this observation. First, this could suggest that higher-level recovery code does not exist (since if it exists, it will not be invoked due to the broken error channel), or it could be the case that errors are intentionally neglected. We consider this second possibility in greater detail in the next section.

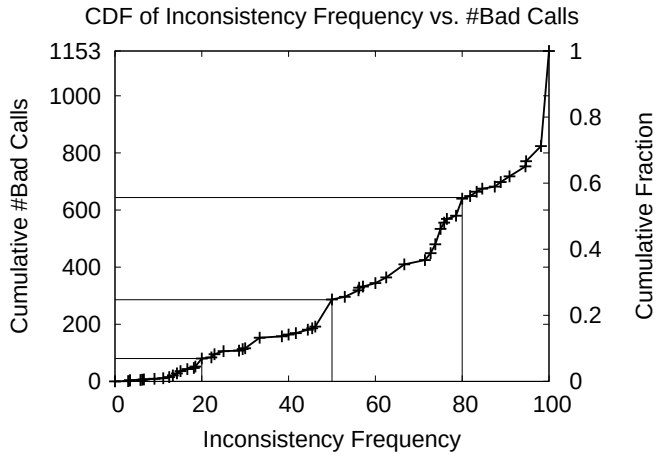


Figure 5.11: **Inconsistent calls frequency.** *The figure shows that inconsistent calls are not corner-case bugs. The x-axis represents the inconsistent-call frequency of a function. x=20% means that there is one bad call out of five total calls; x=80% means that there are four bad calls out of five total calls. The left y-axis counts the cumulative number of bad calls. For example, below the 20% mark, there are 80 bad calls that have an inconsistent-call frequency of less than 20%. As reported in Tables 5.4 and 5.5, there exist a total of 1153 bad calls. The right y-axis shows the cumulative fraction of bad calls over the 1153 bad calls.*

5.3.3 Inconsistent Calls: Corner Case or Majority?

In this section, we consider the nature of *inconsistent* calls. For example, we found that 1 out of 33 calls to `ide_setup_pci_device` does not save the return value. One would probably consider this single call as an inconsistent implementation because the majority of the calls to that function save the return value. On the other hand, we also found that 53 out of 54 calls to `unregister_filesystem` do not save the return error codes. Assuming that most kernel developers are essentially competent, this suggests that it may actually be safe to not check the error code returned from this particular function.

To quantify inconsistent calls, we define the *inconsistent call frequency* of a function as the ratio of bad calls over all error-related calls to the function, and correlate this frequency with the number of bad calls to the function. For example, the inconsistent call frequencies for `ide_setup_pci_ blockdev` and `unregister_filesystem` are 3% (1/33) and 98% (53/54) respectively and the numbers of bad calls are 1 and 53 respectively.

Figure 5.11 plots the cumulative distribution function of this behavior. The graph could be seen as a means to prioritize which bad calls to fix first. Bad calls that fall below the 20% mark could be treated as *corner cases* (*i.e.*, we should be suspicious on one bad call in the midst of four good calls to the same function). On the other hand, bad calls that fall above the 80% mark could hint that either different developers make the same mistake and ignore it, or it is probably safe to make such a "mistake".

One perplexing phenomenon visible in the graph is that around 871 bad calls fall above the 50% mark. In other words, they cannot be considered as corner-case bugs; the developers might be aware of these bad calls, but probably just ignore them. One thing we have learned from our recent work on file system code is that if a file system does not know how to recover from a failure, it has the tendency to just ignore the error code. For example, ext3 ignores write failures during checkpointing simply because it has no recovery mechanism (*e.g.*, chained transactions [60]) to deal with such failures. Thus, we suspect that there are deeper design shortcomings behind poor error code handling; error code mismanagement may be as much symptom as disease.

Our analysis is similar to the work of Engler *et al.* on findings bugs automatically [40]. In their work, they use existing implementation to imply beliefs and facts. Applying their analysis to our case, the bad calls that fall above the 80% mark might be considered as good calls. However, since we are analyzing the specific problem of error propagation, we use that semantic knowledge and demand a discipline that promotes checking an error code in all circumstances, rather than one that follows majority rules.

5.3.4 Characteristics of Error Channels

Finally, we study whether the characteristic of an error channel has an impact on the robustness of error code propagation in that channel. In particular, we explore two characteristics of error channels: one based on the error propagation distance and one based on the location distance (inter- vs. intra-file calls).

With the first characteristic, we would like to find out whether error codes are lost near the generation endpoint or somewhere in the middle of the propagation chain. We distinguish two calls: direct-error and propagate-error calls. In a *direct-error call*, the callee is an error-generation endpoint. In a *propagate-error call*, the callee is not a generation endpoint; rather it is a function that propagates an error code from one of the functions that it calls (*i.e.*, it is a function in the middle of the propagation chain). Next, we define a *bad* direct-error (or propagate-error) call as a direct-error (or propagate-error) call that does not save the returned error code.

	Bad Calls	EC Calls	**Frac. (%)**
File Systems			
Inter-module	307	1944	**15.8**
Inter-file	367	2786	**13.2**
Intra-file	159	2548	**6.2**
Storage Drivers			
Inter-module	48	199	**24.1**
Inter-file	92	495	**18.6**
Intra-file	180	1050	**17.1**

Table 5.8: **Calls based on location distance.** *The table shows that the fraction of bad calls in inter-module calls is higher than the one in inter-file calls. Similarly, inter-file calls are less robust than intra-file calls. Note that "inter-file" refers to cross-file calls within the same module. Inter-file calls across different modules are categorized as inter-module.*

Initially, we assumed that the frequency of bad propagate-error calls would be higher than that of bad direct-error calls; we assumed error codes tend to be dropped in the middle of the chain rather than near the generation endpoint. It turns out that the number of bad direct-error and propagate-error calls are similar for file system code but the other way around for storage driver code. In particular, for file systems, the ratio of bad over all direct-error calls is 10%, and the ratio of bad over all propagate-error calls is 14%. For storage drivers, they are 20% and 15% respectively.

For the second characteristic, we categorized calls based on the location distance between a caller and a callee. In particular, we distinguish three calls: inter-module, inter-file (but within the same module), and intra-file calls. Table 5.8 reports that intra-file calls are more robust than inter-file calls, and inter-file calls are more robust than intra-file calls. For example, out of 1944 inter-module calls in which error codes propagate in file system, 307 (16%) of them are bad calls. However, out of 2786 inter-file calls within the same module, there are only 367 (13%) bad calls. Intra-file calls only exhibit 6% bad calls. The same pattern occurs in storage device drivers. Thus, we conclude that the location distance between the caller and the callee plays a role in the robustness of the call.

5.4 Conclusion

In this chapter, we have analyzed the file and storage systems in Linux 2.6 and found that error codes are not consistently propagated. In the beginning of each chapter of this dissertation, we have reprinted some developer comments we found near some problematic cases (filenames and line numbers are shown inside the parentheses). Unfortunately, there are more:

> "*Retval ignored?*" – in SCSI (sg.c, 2612)
>
> "*Todo: handle failure.*" – in SCSI (mac53c94.c, 504)
>
> "*Can this catch a write error?*" – in SCSI (osst.c, 737)
>
> "*FIXME: Handle lost commands*" – in SCSI (scsi_error.c, 1139)
>
> "*Not much we can do if it fails anyway, ignore rc.*" – in CIFS (file.c, 553)
>
> "*Ignore errors.*" – in NCPFS (dir.c, 259)
>
> "*Never mind errors we might get here.*" – in XFS (xfs_mount.c, 1177)

These comments from developers indicate part of the problem: even when the developers are aware they are not properly propagating an error, they do not know how to implement the correct response. Given static analysis tools to identify the source of bugs (such as EDP), developers may still not be able to fix all bugs in a straightforward manner.

Due to these observations, we believe it is thus time to rethink how failures are managed in large systems. Preaching that developers follow error handling conventions and hoping the resulting systems work as desired seems naive at best. New approaches to error detection, propagation, and recovery are needed.

For future work, we advocate two approaches to help file and storage system programmers avoid these types of mistakes. First, we propose building systems with *semantic error codes*; with this approach, the system does not blindly believe in the success or failure signal reported by an error code but instead performs extra checks to confirm whether the corresponding operation is successful or not. This technique is similar to dynamic verification techniques [43]. Second, we propose adopting the *malloc-free* paradigm for error codes [82]. Specifically, once an error code is generated, it is treated as immutable and can only be destroyed if it

transforms to another error code or the corresponding failure is handled. If there is a “dangling” error code, then the system has forgotten to check or handle certain faults. This new architecture ensures that errors do not disappear easily, hence reducing the instances of silent failure.

Chapter 6

I/O Shepherding: A New Reliability Infrastructure

"Note: todo: log error handler."
– A comment in IBM JFS (jfs_logmgr.c, line 222)

Modern disks, due to their complex and intricate nature [11], have a wide range of "interesting" failure modes, including latent sector faults [79], block corruption [46, 53], transient faults [135], and whole-disk failure [115]. To store data reliably, file systems need to handle all these failures properly. Our analysis in Section 3.2 reveals that unfortunately file system failure handling is broken, primarily due to the diffusion of I/O failure handling; the code that detects I/O failures and performs recovery (such as retry or stopping the file system) is spread over different places. This eventually leads to several problems. First, failure policies are *illogically inconsistent*; different failure handling techniques are used even under similar failure scenarios unintentionally. Second, failure policies and mechanisms are tangled; it is hard to separate failure policies (*e.g.*, "detect block corruption") from their implementation (*e.g.*, "read from a replica"). As a result of this tangled policy and mechanism, neither can be modified without affecting the other, resulting in an inflexible failure handling system.

As a way to mitigate the aforementioned problems, this chapter presents the design, implementation, and evaluation of a new reliability infrastructure for file systems called *I/O shepherding*. With I/O shepherding, the reliability policies of a file system are well-defined, easy to understand, powerful, and simple to tailor to environment and workload. The I/O shepherd achieves these ends by interposing

on each I/O that the file system issues. The shepherd then takes responsibility for the "care and feeding" of the request, specifically by executing a *reliability policy* for the given block. Simple policies will do simple things, such as issue the request to the storage system and return the resulting data and error code (success or failure) to the file system above. However, the true power of shepherding lies in the rich set of policies that one can construct, including sophisticated retry mechanisms, strong sanity checking, the addition of checksums to detect data corruption, and mirrors or parity protection to recover from lost blocks or disks.

The rest of this chapter is organized as follows. We first present the goals of file system reliability (Section 6.1) and then the design of I/O shepherding (Section 6.2). To show how we can easily specify reliability policies in this framework, Section 6.3 presents some examples of policies that file system administrators can specify. Section 6.4 then shows how we take an existing journaling file system, Linux ext3, and transform it into a shepherding-aware file system, which we call CrookFS. As part of this implementation, we also introduce a novel concept of *chained transactions* (Section 6.4.1), which is a solution to the major problem of failed intentions we found in journaling file systems (as described in Section 3.3). Finally, in Section 6.5, we explore how to craft reliability policies and evaluate their overheads.

6.1 Goals

The single underlying design principle of this work is that *reliability should be a first-class file system concern.* We believe a reliability framework should adhere to the following three goals: simple specification, powerful policies, and low overhead.

Simple specification: We believe that system developers should be able to specify reliability policies simply and succinctly. Writing code for reliability is usually complex, given that one must explicitly deal with both misbehaving hardware and rare events; it is especially difficult to ensure that recovery actions remain consistent in the presence of system crashes. We envision that file system administrators will take on the role of fault *policy writers*; the I/O shepherd should ease their task.

The I/O shepherd simplifies the job of a policy writer in two ways. First, all reliability policies are written in a single locale (*i.e.*, the shepherd layer). This is achieved by routing all I/O requests to the shepherd layer first. As the shepherd interposes on each request, it can apply the desired reliability

policies to the request. By writing policies in a centralized fashion, policies are easier to maintain and debug.

Second, the I/O shepherd provides a diverse set of detection and recovery primitives that hide much of the complexity. For example, the I/O shepherd takes care of both the asynchrony of initiating and waiting for I/O and keeps multiple updates and new metadata consistent in the presence of crashes. Policy writers are thus able to stitch together the desired reliability policy with relatively few lines of code; each of the complex policies we craft (Section 6.5) requires fewer than 80 lines of code to implement. The end result: less code and (presumably) fewer bugs.

Powerful policies: We believe the reliability framework should enable not only correct policies, but more powerful policies than currently exist in commodity file systems today. Specifically, the framework should enable composable, flexible, and fine-grained policies.

A *composable* policy allows the file system to use different sequences of recovery mechanisms. For example, if a disk read fails, the file system can first retry the read; if the retries continue to fail, the file system can try to read from a replica. With shepherding, policy writers can compose basic detection and recovery primitives in the manner they see fit.

A *flexible* policy allows the file system to perform the detection and recovery mechanisms that are most appropriate for the expected workload and underlying storage system. For example, one may want different levels of redundancy for temporary files in one volume and home directories in another. Further, if the underlying disk is known to suffer from transient faults, one may want extra retries in response. With I/O shepherding, administrators can configure these policy variations for each mounted volume.

A *fine-grained* policy is one that takes different recovery actions depending on the block that has failed. Different disk blocks have different levels of importance to the file system; thus, some disk faults are more costly than others and more care should be taken to prevent their loss. For example, the loss of disk blocks containing directory contents is catastrophic [123]; therefore, a policy writer can specify that all directory blocks be replicated. With I/O shepherding, policies are specified as a function of block type.

Low overhead: Users are unlikely to be willing to pay a large performance cost for improved reliability. For reasonable performance, we have found that it is critical to properly integrate reliability mechanisms with the consistency

management, layout, caching, and disk scheduling subsystems. Of course, reliability mechanisms do not always add overhead; for example, a smart scheduler can utilize replicas to improve read performance [71, 148].

6.2 Architecture

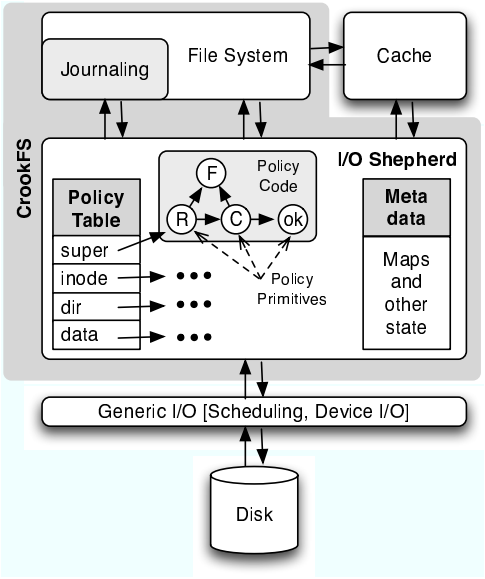


Figure 6.1: **System Architecture.** *The architecture of a file system containing an I/O shepherd is shown. The file system proper (including journaling) and caching subsystems sit above the I/O shepherd, but have been modified in key locations to interact with the shepherd as necessary. The shepherd itself consists of a policy table, which points to policy code that dictates the detection and recovery strategy for that particular block type. Beneath the shepherd is the generic I/O layer (including disk scheduling, which is slightly modified as well) and one (or more) disks.*

To manage the storage system in a reliable way, the I/O shepherd must be able to interpose on every I/O request and response, naturally leading to an architecture in which the shepherd is positioned between the file system and disks (Figure 6.1). As shown therein, I/O requests issued from different components of a file system (*e.g.*, the file system, journaling layer, and cache) are all passed to the I/O shep-

herd. The shepherd unifies all reliability code in a single location, making it easier to manage faults in a correct and consistent manner. The shepherd may modify the I/O request (*e.g.*, by remapping it to a different disk block) or perform additional requests (*e.g.*, by reading a checksum block) before sending the request to disk. When a request completes, the response is again routed through the shepherd, which performs the desired fault detection and recovery actions, again potentially performing more disk operations. After the shepherd has executed the reliability policy, it returns the response (success or failure) to the file system.

6.2.1 Policy Table and Code

With I/O shepherding, the reliability policy of the file system is specified by a *policy table*; this structure specifies which code to execute when the file system reads or writes each type of on-disk data structure (*e.g.*, superblock, inode, or directory). Each entry in the table points to *policy code*, which defines the sequence of actions taken for a block of a particular type. For example, given an ext3-based file system, a policy writer can specify a different policy for each of its 13 block types (as shown in Table 6.1). The table could thus mandate replication of the superblock, checksum protection for other metadata, and an aggressive retry scheme for user data.

Although this design does not directly support different policies for individual files, the I/O shepherd allows a different policy table per mounted file system. Thus, administrators can tailor the policy of each volume, a basic entity they are accustomed to managing. For example, a `/tmp` volume could employ little protection to obtain high performance while an archive could add checksums and parity to improve reliability at some performance cost.

6.2.2 Policy Metadata

To implement useful policies, an I/O shepherd often requires additional on-disk state to track the location of various blocks it is using (*e.g.*, the location of checksums, replicas, or parity blocks). Thus, to aid in the management of persistent metadata, the I/O shepherd framework provides *maps*. Some commonly used maps are a `CMap` to track checksum blocks, an `RMap` to record bad block remappings, and an `MMap` to track multiple replicas.

A policy can choose to use either a *static* or *dynamic* map for a particular type of metadata. With static mapping, the association between a given on-disk block and its checksum or replica location is fixed when the file system is created. With a dynamic map, new associations between blocks can be created over time.

Ext3 Structures	Read Policy	Write Policy
inode	`ChecksumWrite()`	`ChecksumRead()`
directory	`ChecksumWrite()`	`ChecksumRead()`
data bitmap	`ChecksumWrite()`	`ChecksumRead()`
inode bitmap	`ChecksumWrite()`	`ChecksumRead()`
indirect	`ChecksumWrite()`	`ChecksumRead()`
data	`RetryWrite()`	`RetryRead()`
super	`MirrorWrite()`	`MirrorRead()`
group descriptor	`ChecksumWrite()`	`ChecksumRead()`
journal super	`ChecksumWrite()`	`ChecksumRead()`
journal revoke	`ChecksumWrite()`	`ChecksumRead()`
journal descriptor	`ChecksumWrite()`	`ChecksumRead()`
journal commit	`ChecksumWrite()`	`ChecksumRead()`
journal data	`ChecksumWrite()`	`ChecksumRead()`

Table 6.1: **Policy Table.** *The table presents an example of a policy table that a policy writer can specify. The policy table specifies different policies for each of the 13 block types in ext3. For example, the table mandates replication of the superblock, checksum protection for other metadata, and an aggressive retry scheme for user data. Section 6.3 will show the implementation of some of policy code.*

There are obvious trade-offs to consider when deciding between static and dynamic maps. Static maps are simple to maintain but inflexible; for example, if a static map is used to track a block and its copy, and one copy becomes faulty due to a latent sector error, the map cannot be updated with a new location of the copy.

Dynamic maps are more flexible, as they can be updated as the file system is running and thus can react to faults as they occur. However, dynamic maps must be reflected to disk for reliability. Thus, updating dynamic maps consistently and efficiently is a major challenge; we describe the problem and our approach to solving it in more detail in Section 6.4.1.

6.2.3 Policy Primitives

To ease the construction of policy code, the shepherd provides a set of *policy primitives*. The primitives hide the complexity inherent to reliable I/O code; specifically, the primitives ensure that policy code updates on-disk structures in a single transaction. Clearly, a fundamental tension exists here: as more functionality is encapsulated in each primitive, the simpler the policy code becomes, but the less control one has over the reliability policy. Our choice has generally been to expose more control to the policy writers.

The I/O shepherd provides five classes of reliability primitives. All primitives return failure when the storage system itself returns an error code or when blocks do not have the expected contents.

Read and Write: The I/O shepherd contains basic primitives for reading and writing either a single block or a group of blocks concurrently from disk. A specialized primitive reads from mirrored copies on disk: given a list of blocks, it reads only the block that the disk scheduler predicts has the shortest access time.

Integrity: On blocks that reside in memory, primitives are provided to compute and compare checksums, compare multiple blocks, and perform strong sanity checks (*e.g.*, checking the validity of directory blocks or inodes).

Higher-level Recovery: The I/O shepherd contains primitives to stop the file system with a panic, remount the file system read-only, or even reboot the system. Primitives are also provided that perform semantic repair depending upon the type of the block (*e.g.*, an inode or a directory block) or that run a full `fsck` across the disk.

Persistent Maps: The I/O shepherd provides primitives for looking up blocks in an indirection map and for allocating (or reallocating and freeing) new entries in such a map (if it is dynamic).

Layout: To allow policy code to manage blocks for its own use (*e.g.*, for checksums, remapped blocks, and replicas), the I/O shepherd can allocate blocks from the file system. One primitive exposes information about the current layout in the file system while a second primitive allocates new blocks, with hooks to specify preferences for block placement. With control over block placement, policy code can provide trade-offs between performance and reliability (*e.g.*, by placing a replica near or far from its copy).

6.3 Example Policy Code

With all the shepherd's features mentioned in the previous section, writing reliability policies within the shepherd is more straightforward than in current approaches. In this section, we show how the I/O shepherd enables one to specify reliability policies that are traditionally implemented across different levels of the storage stack. For example, one can specify policies that operate on a single block and are often performed within disks (*e.g.*, retrying, remapping, and checksums), policies

that operate across multiple blocks or multiple disks (*e.g.*, mirrors and parity), and finally, one can specify policies requiring semantic information about the failed block and are usually performed by the file system (*e.g.*, stopping the file system, data structure repair, and fsck). A shepherd enables policies that compose all of these strategies.

We now show the simplicity and power of the shepherd through a number of examples. The names of all policy primitives begin with `IOS` for clarity. We simplify the pseudo-code by ignoring some of the error codes that are returned by the policy primitives, such as `IOS_MapLookup` and `IOS_MapAllocate`.

The first example policy is based loosely on NTFS [106]. The NTFS policy tries to keep the system running when a fault arises by first retrying the failed read or write operation a fixed number of times; if it is unable to complete the operation, the fault is simply propagated to the application. We show the read version of the code here (the write is similar).

```
NTFSRead(DiskAddr D, MemAddr A)
  for (int i = 0; i < RETRY_MAX; i++)
      if (IOS_Read(D, A) == OK)
          return OK;
  return FAIL;
```

The code above takes the disk address D on which the file system issues a read. The `IOS_Read` primitive sends the read request to the disk and put the content from the disk address to the memory address A. If the read operation is successful, the code returns success to the file system. Otherwise, the policy will repeat the read for `RETRY_MAX` times.

The second example policy loosely emulates the behavior of ReiserFS [106]. This policy chooses reliability over availability; whenever a write fault occurs, the policy simply halts the file system by calling the `IOS_Stop` primitive. By avoiding updates after a fault, this conservative approach minimizes the chance of further damage.

```
ReiserFSWrite(DiskAddr D, MemAddr A)
  if (IOS_Write(D, A) == OK)
      return OK;
  else
      IOS_Stop(IOS_HALT);
```

The next two examples show the ease with which one can specify policies that detect block corruption. First, the `SanityRead` policy performs type-specific sanity checking on the read block using a shepherd primitive (`IOS_SanityCheck`). Note in this example how the block type can be passed to and used by policy code.

For example, if the block is an inode block, then the `IOS_SanityCheck` primitive will perform specific inode checks (*e.g.*, an inode being used should not have a zero modification time).

```
SanityRead(DiskAddr D, MemAddr A, BlockT T)
   if (IOS_Read(D, A) == FAIL)
      return FAIL;
   return IOS_SanityCheck(A, T);
```

Second, the `ChecksumRead` policy below uses checksums to detect block corruption; the policy code first finds the location of the checksum block by looking up the checksum map (`CMap`), then concurrently reads both the stored checksum and the data block (the checksum may be cached), and then compares the stored and newly computed checksums.

```
ChecksumRead(DiskAddr D, MemAddr A)
   DiskAddr cAddr;
   ByteOffset off;
   CheckSum onDisk;
   // find the checksum block
   IOS_MapLookupOffset(CMap, D, &cAddr, &off);
   // read from checksum and D concurrently
   if (IOS_Read(cAddr, &onDisk, D, A)==FAIL)
      return FAIL;
   // compare the stored and computed checksums
   CheckSum calc = IOS_Checksum(A);
   return IOS_Compare(onDisk, off, calc);
```

The next two examples compare how static and dynamic maps can be used for tracking replicas. First, the `StaticMirrorWrite` policy code assumes that the mirror map, `MMap`, was configured for each block when the file system was created. Thus, the code looks simple. This kind of policy code is useful for on-disk structures that are stored in static locations (*e.g.*, inodes in ext3).

```
StaticMirrorWrite(DiskAddr D, MemAddr A)
   DiskAddr copyAddr;
   IOS_MapLookup(MMap, D, &copyAddr);
   // write to both copies concurrently
   return (IOS_Write(D, A, copyAddr, A));
```

Second, `DynMirrorWrite` checks to see if a copy already exists for the block being written to; if the copy does not exist, the code picks a location for the mirror and allocates (and persistently stores) an entry in `MMap` for this mapping. Note that this policy needs to do more work than the static one; when a replica does

not exist, this policy needs to ask the file system to allocate a new block (via the `PickMirrorLoc` primitive) and stores this new location in the mirror map (via the `IOS_MapAllocate` primitive). This kind of policy code is useful for on-disk structures that are allocated dynamically on the fly (*e.g.*, data blocks in ext3).

```
DynMirrorWrite(DiskAddr D, MemAddr A)
   DiskAddr copyAddr;
   // copyAddr is set to mirrored block
   //   or NULL if no copy of D exists
   IOS_MapLookup(MMap, D, &copyAddr);
   if (copyAddr == NULL)
      PickMirrorLoc(MMap, D, &copyAddr);
      IOS_MapAllocate(MMap, D, copyAddr);
   return (IOS_Write(D, A, copyAddr, A));
```

The final two policy examples show how blocks can be remapped; the map of remapped blocks is most naturally a dynamic map, since the shepherd does not know *a priori* which writes will fail. The first remap code, `RemapWrite`, is responsible for the remapping; if a write operation fails, the policy code picks a new location for that block, allocates a new mapping for that block in `RMap`, and tries the write again.

```
RemapWrite(DiskAddr D, MemAddr A)
   DiskAddr remap;
   // remap is set to remapped block
   //   or to D if not remapped
   IOS_MapLookup(RMap, D, &remap);
   if (IOS_Write(remap, A) == FAIL)
      PickRemapLoc(RMap, D, &remap);
      IOS_MapAllocate(RMap, D, remap);
   return IOS_Write(remap, A);
```

The second remap code, `RemapRead`, checks `RMap` to see if this block has been previously remapped; the read to the disk is then redirected to the possibly new location. Of course, all of these policies can be extended, for example, by retrying if the disk accesses fail or stopping the file system on failure.

```
RemapRead(DiskAddr D, MemAddr A)
   DiskAddr remap;
   IOS_MapLookup(RMap, D, &remap);
   return IOS_Read(remap, A);
```

In summary, this section has shown how different policies can be specified easily within the shepherd. With the ease of writing policies, we enable not only

file system developers, but also file system administrators to write the policies that fit for their specific goals.

6.4 Implementation

A major challenge in implementing I/O shepherding is proper systems integration. We now describe how to integrate I/O shepherding into an existing file system, Linux ext3. For our prototype system, we believe that it is important to work with an existing file system in order to leverage the optimizations of modern systems and to increase the likelihood of deployment. We refer to the ext3 variant with I/O shepherding as CrookFS, named for the hooked staff of a shepherd.

Integrating shepherding with ext3 instead of designing a system from scratch does introduce challenges in that it requires changes to the file system consistency management routines, layout engine, disk scheduler, and buffer cache, as well as the addition of thread support. Many of these alterations are necessary to pass information throughout the system (*e.g.*, informing the disk scheduler where replicas are located so it can read the closer copy); some are required to provide improved control to reliability policies (*e.g.*, enabling a policy to control placement of on-disk replicas).

Of those changes, the most important interaction between the shepherd and the rest of the file system is in the consistency management subsystem. Most modern file systems use *write-ahead logging* to a journal to update on-disk structures in a consistent manner [64]. Policies developed in the shepherd often add new on-disk state (*e.g.*, checksums, or replicas) and thus must also update these structures atomically. In most cases, doing so is straightforward. However, as described in Section 3.3, we have found that journaling file systems suffer from a general *problem of failed intentions*, which arises when the intent as written to the journal cannot be realized due to disk failure during checkpointing. Thus, the shepherd incorporates *chained transactions*, a novel and more powerful transactional model that allows policies to handle unexpected faults during checkpointing and still consistently update on-disk structures. The shepherd provides this support *transparently* to all reliability policies, as the required actions are encapsulated in various systems primitives.

In this section, we devote most of our discussion to how we integrate CrookFS with ext3 journaling to ensure consistency (Section 6.4.1), and then describe integration with other key subsystems (Section 6.4.2). Finally, we present the complexity of CrookFS in Section 6.4.3.

6.4.1 Consistency Management

In order to implement some reliability policies, an I/O shepherd requires additional data (*e.g.*, checksum and replica blocks) and metadata (*e.g.*, persistent maps). Keeping this additional information consistent can be challenging. As an example, consider policy code that dynamically creates a replica of a block; doing so requires picking available space on disk for the replica, updating the mirror map to record the location of the replica, and writing the original and replica blocks to disk. One would like these actions to be performed atomically despite the presence of crashes.

Given that the goal of I/O shepherding is to enable highly robust file systems, we build upon the most robust form of journaling, data journaling. (Journaling basics have been described in more detail in Section 3.3). To understand how CrookFS uses the ext3 data journaling to maintain consistency, we begin with two strawman approaches (Section 6.4.1). Neither work; rather, we use them to illustrate some of the subtleties of the problem. We then present our working solution with chained transactions (Section 6.4.1).

Strawman Shepherds

In the *early strawman* approach, the shepherd interposes on the preceding journal writes to insert its own metadata for this transaction. This requires splitting policy code for a given block type into two portions: one for the operations to be performed on the journal write for that block and one for operations on a checkpoint. In the *late strawman*, the shepherd appends a later transaction to the journal containing the needed information. This approach assumes that the policy code for a given block is invoked only at checkpoint time. We now describe how both strawmen fail.

First, consider the `DynMirrorWrite` policy (presented in Section 6.3). On the first write to a block **D**, the policy code picks, allocates, and writes to a mirror block **C** (denoted `copyAddr` in the policy code); at this time, the data bitmap **B'** and the mirror map **M** are also updated to account for **C**. All of these actions must be performed atomically relative to the writing of **D** on disk.

The early strawman can handle the `DynMirrorWrite` policy, as shown in Figure 6.2. When the early strawman sees the entry for **D** written to the journal (T_1), it invokes policy code to allocate an entry for **C** in **M** and **B'** and to insert **M** and **B'** in the current transaction. When **D** is later checkpointed (T_2), similar policy code is again invoked so that the copy **C** is updated according to the mirror map **M**. With the early strawman, untimely crashes do not cause problems because all metadata is in the same transaction.

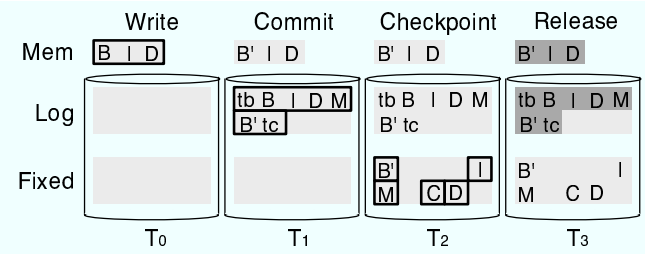


Figure 6.2: **Early Strawman for** `DynMirrorWrite`**.** *The figure shows how the early strawman writes to a replica of a data block* **D***. Both in-memory (top) and on-disk (bottom) states are shown.* **D** *is a data block,* **I** *an inode,* **B** *a bitmap,* **tb** *the beginning of a transaction, and* **tc** *the commit block. Darker gray shading indicates that blocks have been released after checkpointing.*

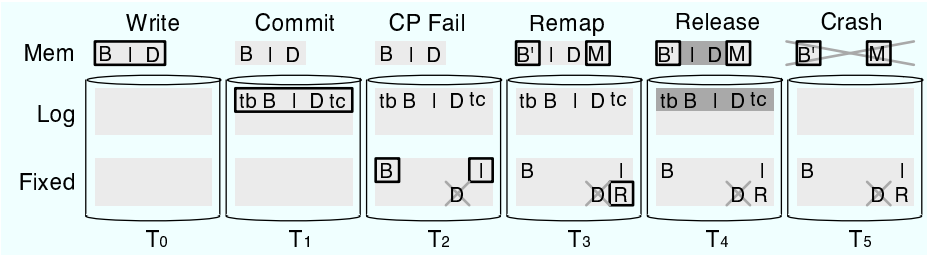


Figure 6.3: **Early Strawman for** `RemapWrite`**.** *The figure illustrates how the early strawman cannot deal with the problem of failed intentions.*

Now, consider the `RemapWrite` policy (presented in Section 6.3). This policy responds to the checkpoint failure of a block **D** by remapping **D** to a new location, **R** (denoted `remap` in the policy code). However, the early strawman cannot implement this policy. As shown in Figure 6.3, after the write to a data block **D** fails (T_2) the policy wants to remap block **D** to **R** (T_3), which implies that the bitmap and `RMap` are modified (**B'** and **M**). However, it is too late to modify the transaction that has been committed. Thus, if a crash occurs (T_5) after the transaction is released (T_4), all metadata changes will be discarded and the disk will be in an inconsistent state. Specifically, the data block **D** is lost since the modified `RMap` that has the reference to **R** has been discarded.

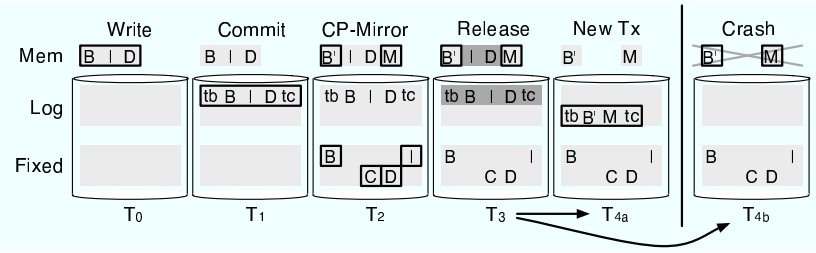


Figure 6.4: **Late Strawman for** `DynMirrorWrite`**.** *The figure shows the incorrect timing of the new transaction commit.*

More generally, the early strawman cannot handle any checkpoint failures that result in metadata changes, because it must calculate *a priori* to the actual checkpoint what will happen at checkpoint time. In Section 3.3, we have referred to this as the problem of *failed intentions*. Failed intentions occur when a commit to the journal succeeds but the corresponding checkpoint does not; the intent of the update (as logged in the journal) cannot be realized due to checkpoint failure.

We now examine the late strawman which only invokes policy code at checkpoint time. Given that it is "too late" to modify a transaction at checkpoint time, the late strawman adds another transaction with the new metadata. Unfortunately, the late strawman cannot correctly handle the `DynMirrorWrite` policy, as shown in Figure 6.4. During the checkpoint of block **D** (T_2), the late strawman invokes the policy code, creates and updates the copy **C** as desired. After this transaction has been released (T_3), a new transaction containing **B'** and **M** is added to the journal (T_{4a}). The problem with the late strawman is that it cannot handle a system crash that occurs between the two transactions (*i.e.*, T_{4b}, which can occur between T_3 and T_{4a}): **D** will not be properly mirrored with a reachable copy. When the file system recovers from this crash, it will not replay the transaction writing **D** (and **C**) because it has already been released and it will not replay the transaction containing **B'** and **M** because it has not been committed; as a result, copy **C** will be unreachable. Thus, the timing of the new transaction is critical and must be carefully managed, as we will see below.

Chained Transactions

We solve the problem of failed intentions with the development of *chained transactions*. With this approach, like the late strawman, all metadata changes initiated

by policy code are made at checkpoint time and are placed in a new transaction; however, unlike the late strawman, this new chained transaction is committed to the journal *before* the old transaction is released. As a result, a chained transaction makes all metadata changes associated with the checkpoint appear to have occurred at the same time.

To illustrate chained transactions we consider a reliability policy that combines mirroring and remapping. We consider the case where this is not the first write to the block **D** (*i.e.*, an entry in the mirror map should already exist) and it is the write to the copy **C** that fails. Code paths *not* taken are *slanted.*

```
RemapMirrorWrite(DiskAddr D, MemAddr A)
   DiskAddr copy, remap;
   Status status1 = OK, status2 = OK;
   IOS_MapLookup(MMap, D, &copy);
    // remap is set to D if not remapped
   IOS_MapLookup(RMap, D, &remap);
   if (copy == NULL)
      PickMirrorLoc(MMap, D, &copy);
      IOS_MapAllocate(MMap, D, copy);
   if (IOS_Write(remap, A, copy, A) == FAIL)
      if (IOS_Failed(remap))
         PickRemapLoc(RMap, D, &remap);
         IOS_MapAllocate(RMap, D, remap);
         status1 = IOS_Write(remap, A);
      if (IOS_Failed(copy))
         PickMirrorLoc(MMap, D, &copy);
         IOS_MapAllocate(MMap, D, copy);
         status2 = IOS_Write(copy, A);
   return ((status1==FAIL)||(status2==FAIL));
```

Figure 6.5 presents a timeline of the activity in the system with chained transactions. With chained transactions, committing the original transaction is unchanged as seen at times T_0 and T_1 (policy code will be invoked when each of the blocks in the journal is written, but its purpose is to implement the reliability policy of the journal itself). When the data block **D** is checkpointed, the `RemapMirrorWrite` policy code is invoked for **D**. The policy code finds the copy location of **C** (denoted `copy`) and the potentially remapped location of **R** (denoted `remap`). In our example, we assume that writing to the copy **C** fails (T_2); in this case, the policy code allocates a new location for **C** (hence dirtying the bitmap, **B'**), writes the copy to a new location, and updates the mirror map **M'** (T_3). Our integration of the shepherd primitive, `IOS_MapAllocate`, with the ext3 journaling layer ensures that the chained transaction containing **B'** and **M'** is committed to the journal (T_4) before releasing the original transaction (T_5). At time T_6 (not shown), when the chained

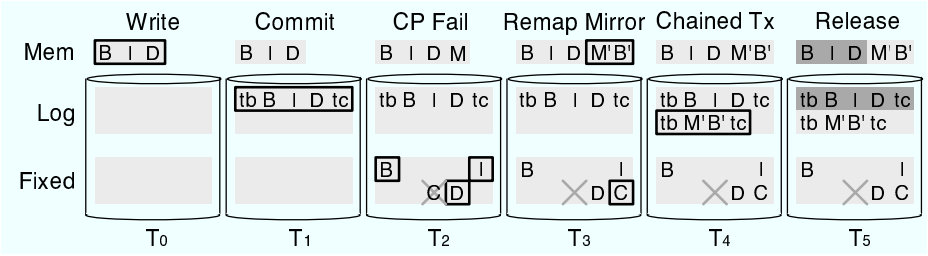


Figure 6.5: **Chained Transactions for `RemapMirrorWrite`.** *The figure shows how chained transactions handle failed intentions.*

transaction is checkpointed, the blocks **B'** and **M'** are finally written to their fixed locations on disk; given that these are normal checkpoint writes, the relevant policy code will be applied to these updates.

With a chained transaction, a crash cannot occur "between" the two related transactions, because the second transaction is always committed before the first is released. If the system crashes before the first transaction is released, all operations will be replayed.

Chained transactions ensure that shepherd data and metadata are kept consistent in the presence of crashes. However, if one is not careful, chained transactions introduce the possibility of deadlock. Specifically, because CrookFS now holds the previous checkpoint while waiting to commit the chained transaction, we must avoid the two cases that can lead to circular dependencies. First, CrookFS must ensure that sufficient space exists in the journal for all chained transactions; this constrains the number of remappings (and subsequent chained transactions) that can occur in any policy code. Second, CrookFS must use shadow copies when updating a shepherd metadata block that exists in a previous transaction (just as it does for its own metadata), instead of acquiring locks.

Non-Idempotent Policies

To maintain consistency, all failure scenarios must be considered, including repeated crashes during recovery. Repeated crashes will cause CrookFS to replay the same transactions and their corresponding checkpoints. In such scenario, only *idempotent* policy code will work correctly.

For example, consider a policy that protects data blocks with parity. Although parity can be computed using an idempotent equation ($P = D1 \oplus D2 \oplus ... \oplus Dn$),

this approach performs poorly because $N - 1$ blocks must be read every time a block is modified. However, the more efficient way of computing parity ($P_{new} = P_{old} \oplus D_{old} \oplus D_{new}$) is non-idempotent since P_{old} and D_{old} will be overwritten with P_{new} and D_{new}, respectively. Thus, repeated journal replays will incorrectly calculate the parity block.

To handle non-idempotent policies such as parity, CrookFS provides an old-value logging mechanism [52]. The old-value log annotates versions to distinguish old and new values, and writes the old data and its corresponding version into the log atomically. Thus, non-idempotent policy code must take care to read the old values and log them into the old-value log, using support within CrookFS. Simplified policy code for `ParityWrite` is as follows.

```
ParityWrite(DiskAddr D, MemAddr aNew)
  DiskAddr P;
  MemAddr  aOld, apOld, apNew;
  IOS_MapLookup(PMap, D, &P);
  if (IOS_ReadStable(D,aOld,P,apOld) == FAIL)
      return FAIL;
  if (IOS_WriteAndLog(D,aOld,P,apOld) == FAIL)
      return FAIL;
  apNew = ComputeParity(apOld, aOld, aNew);
  return (IOS_Write(D, aNew, P, apNew));
```

Reliability of the Journal and Shepherd Maps

A final complication arises when the reliability policy wishes to increase the protection of the journal or of shepherd metadata itself. Although there are a large number of reasonable policies that do not add protection features to the journal (since it is only read in the event of an untimely crash), some policies might wish to add features (*e.g.*, replication or checksumming). The approaches we describe above do not work for the journal, since they use the journal itself to update other structures properly. Thus, we treat journal replication, checksumming, and other similar actions as a special case, mandating a restricted set of possible policies.

6.4.2 System Integration

To build effective reliability policies, the shepherd must interact with other components of the existing file system. Below, we discuss these remaining technical hurdles.

Semantic Information: To implement fine-grained policies, the shepherding layer must have information about the current disk request; in our implementation, shepherding must know the type of each block (*e.g.*, whether it is an inode or a directory) and whether the request is a read or a write in order to call the correct policy code as specified in the policy table.

In the case where requests are issued directly from the file system, acquiring this information is straightforward: the file system is modified to pass the relevant information with each I/O call. When I/O calls are issued from common file system layers (*e.g.*, the generic buffer cache manager), extra care must be taken. First, the buffer cache must track block type for its file blocks and pass this information to the shepherd when calling into it. Second, the buffer cache must only pass this information to shepherd-aware file systems. A similar extension was made to the generic journaling and recovery layers to track the type of each journaled block.

Threads: I/O shepherding utilizes threads to handle each I/O request and any related fault management activity. A thread pool is created at mount time, and each thread serves as an execution context for policy code. Thus, instead of the calling context issuing a request directly and waiting for it to complete, it enqueues the request and lets a thread from the pool handle the request. This thread then executes the corresponding policy code, returning success or failure as dictated by the policy. When the policy code is complete, the caller is woken and passed this return code.

We have found that a threaded approach greatly simplifies the task of writing policy code, where correctness is of paramount importance; without threads, policy code was split into a series of event-based handlers that executed before and after each I/O, often executing in interrupt context and thus quite difficult to program. A primary concern of our threaded approach is overhead, which we explore in Section 6.5.2.

Legacy Fault Management: Because the shepherd now manages file system reliability, we removed the existing reliability code from ext3. Thus, the upper layers of CrookFS simply propagate faults to the application. Note that some sanity checks from ext3 are kept in CrookFS, since they are still useful in detecting memory corruption.

One issue we found particularly vexing was correct error propagation; a closer look revealed that ext3 often accidentally changed error codes or ignored them altogether. In the previous chapter we have presented our static analysis tool to find these bugs so we could fix them.

Layout Policy: Fault management policies that dynamically require disk space (*e.g.*, for checksum or replica blocks) must interact with the file system layout and allocation policies. Since reliability policies are likely to care about the location of the allocated blocks (*e.g.*, to place blocks away from each other for improved reliability), we have added two interfaces to CrookFS. The first exposes information about the current layout in the file system. The second allows a reliability policy to allocate blocks with options to steer block placement. Policy code can use these two interfaces to query the file system and request appropriate blocks.

Disk Scheduling: For improved performance, the disk scheduler should be integrated properly with reliability policies. For example, the scheduler should know when a block is replicated, and access the nearer block for better performance [71, 148].

We have modified the disk scheduler to utilize replicas as follows. Our implementation inserts a request for each copy of a block into the Linux disk scheduling queue; once the existing scheduling algorithm selects one of these requests to be serviced by disk, we remove the other requests. When the request completes, the scheduler informs the calling policy which replica was serviced, so that faults can be handled appropriately (*e.g.*, by trying to read the other replica). Care must be taken to ensure that replicated requests are not grouped and sent together to the disk.

Caching: The major issue in properly integrating the shepherd with the existing buffer cache is ensuring that replicas of the same data do not simultaneously reside in the cache, wasting memory resources. By placing the shepherd beneath the file system, we circumvent this issue entirely by design. When a read is issued to a block that is replicated, the scheduler decides to read one copy or the other; while this block is cached, the other copy will never be read, and thus only a single copy can reside in cache.

Multiple Disks: One final issue arose from our desire to run CrookFS on multiple disks to implement more interesting reliability policies. To achieve this, we mount multiple disks using a concatenating RAID driver [33]. The set of disks appears to the file system as one large disk, with the first portion of the address space representing the first disk, the second portion of the address space representing the second disk, and so forth. By informing CrookFS of the boundary addresses between disks, CrookFS allocation policies can place data as desired across disks (*e.g.*, data on one disk, a replica on another).

Changes in Core OS	
Chained transactions	26
Semantic information	600
Layout and allocation	176
Recovery	108
Total	*910*
Shepherd infrastructure	
Thread model	900
Data structures	743
Read/Write + Chained Transactions	460
Layout and allocation	66
Scheduling	220
Sanity + Checksums + fsck + Mirrors	429
Support for multiple disks	645
Total	*3463*

Table 6.2: **CrookFS Code Complexity.** *The table presents the amount of code added to implement I/O shepherding as well as a breakdown of where that code lives. The number of lines of code is counted by tallying the number of semi-colons in code that we have added or changed.*

6.4.3 Code Complexity

Table 6.2 summarizes the code complexity of CrookFS. The table shows that the changes to the core OS were not overly intrusive (*i.e.*, 910 C statements were added or changed); the majority of the changes were required to propagate the semantic information about the type of each block through the file system. Many more lines of code (*i.e.*, 3463) were needed to implement the shepherd infrastructure itself. We are hopeful that incorporating I/O shepherding into file systems other than ext3 will require even smaller amounts of new code, given that much of the infrastructure can be reused.

6.5 Crafting a Policy

We now explore how I/O shepherding simplifies the construction of reliability policies. We make two major points in this section. First, the I/O shepherding framework does not add a significant performance penalty. Second, a wide range of useful policies can be easily built in CrookFS, such as policies that propagate errors, perform retries and reboots, policies that utilize parity, mirroring, sanity checks,

Propagate	8	Mirror	18
Reboot	15	Sanity Check	10
Retry	15	Multiple Lines of Defense	39
Parity	28	D-GRAID	79

Table 6.3: **Complexity of Policy Code.** *The table presents the number of semicolons in the policy code evaluated in Section 6.5.*

and checksums, and policies that operate over multiple disks. Overall, we find that our framework adds less than 5% performance overhead on even I/O-intensive workloads and that no policy requires more than 80 lines of policy code to implement. Table 6.3 reports the number of lines of code to implement each reliability policy.

We also make two relatively minor points. First, effective yet simple reliability policies (*e.g.*, retrying requests and performing sanity checks) are not consistently deployed in commodity file systems, but they should be to improve availability and reliability. Second, CrookFS is integrated well with the other components of the file system, such as layout and scheduling.

6.5.1 Experimental Setup

The experiments in this section were performed on an Intel Pentium 4 machine with 1 GB of memory and up to four 120 GB 7200 RPM Western Digital EIDE disks (WD1200BB). We used the Linux 2.6.12 operating system and built CrookFS from ext3 therein.

To evaluate the performance of different reliability policies under fault-free conditions, we use a set of well-known workloads: PostMark [80], which emulates an email server, a TPC-B variant [138] to stress synchronous updates, and SSH-Build, which unpacks and builds the ssh source tree. Table 6.4 shows the performance on PostMark, TPC-B, and SSH-Build of eight reliability policies explored in more detail in this section, relative to unmodified ext3.

To evaluate the reliability policies when faults occur, we stress the file system using type-aware fault injection with a pseudo-device driver (Section 3.2.2). To emulate a block failure, the pseudo-device simply returns the appropriate error code and does not issue the operation to the underlying disk. To emulate corruption, the pseudo-device changes bits within the block before returning the data. The fault injection is type aware in that it can be selectively applied to each of the 13 different block types in ext3 (as shown in Table 6.1).

	PostMark	TPC-B	SSH-Build
Linux ext3	1.00	1.00	1.00
Propagate	1.00	1.05	1.01
Retry and Reboot	1.00	1.05	1.01
Parity	1.14	1.27	1.02
Mirror$_{Near}$	1.59	1.41	1.04
Mirror$_{Far}$	1.65	1.87	1.06
Sanity Check	1.01	1.05	1.01
Multiple Lines of Defense	1.10	1.28	1.01

Table 6.4: **Performance Overheads.** *The table shows the performance overhead of different policies in CrookFS relative to unmodified ext3. Three workloads are run: PostMark, TPC-B, and ssh. Each workload is run five times; averages are reported (there was little deviation). Running times for standard ext3 on PostMark, TPC-B, and SSH-Build are 51, 29.33, 68.19 seconds respectively. The Multiple Lines of Defense policy incorporates checksums, sanity checks, and mirrors.*

6.5.2 Propagate

The first and most basic question we answer is: how costly is it to utilize the shepherding infrastructure within CrookFS? To measure the basic overhead of I/O shepherding, we consider the simplest reliability policy: a null policy that simply propagates errors through the file system. This basic propagate policy is extremely simple, requiring only 8 statements.

The second line of Table 6.4 reports the performance of the propagate policy, normalized with respect to unmodified Linux ext3. For the propagate policy, the measured slowdowns are 5% or less for all three workloads. Thus, we conclude that the basic infrastructure and its threaded programming model do not add noticeable overhead to the system.

6.5.3 Reboot vs. Retry

We next show the simplicity of building robust policies given our I/O shepherding framework. We use CrookFS to implement two straightforward policies: the first halts the file system upon a fault (with optional reboot); the second retries the failed operation a fixed number of times and then propagates the error code to the application. The pseudo code for these two policies was presented earlier (Section 6.3). As shown in Table 6.3 the actual number of lines of code needed to implement these policies is very small: 15 for each.

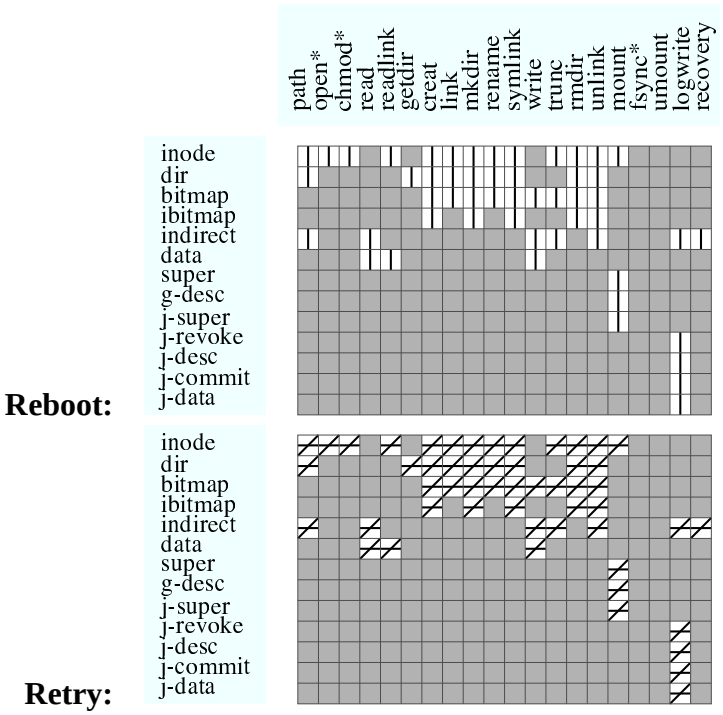


Figure 6.6: **Comparison of Ext3, Reboot, and Retry Policies.** *The table shows how CrookFS with a reboot (top) and a retry (bottom) policy react to read faults (compared to the original ext3 recovery behavior shown in Figure 3.3 in Section 3.2). Along the x-axis, different workloads are shown; each workload stresses either a posix API call or common file system functionality (e.g., path lookup). Along the y-axis, the different data structures of the file system are presented. Each (x,y) location presents the results of a read fault injection of a particular data structure (y) under the given workload (x). The four symbols (/, −, |, and O) represent the detection and recovery techniques used by the file systems.*

To demonstrate that CrookFS implements the desired reliability policy, we inject type-aware faults on read operations. To stress many paths within the file system, we utilize a synthetic workload that exercises the POSIX file system API. The three graphs in Figure 6.6 show how the default ext3 file system, the Reboot, and Retry policies respond to read faults for each workload and for each block type. The top graph (taken from Figure 3.3 in Section 3.2.3) shows that the default ext3 file system does not have a consistent policy for dealing with read faults; for example, when reading an indirect block fails as part of the file writing workload, the

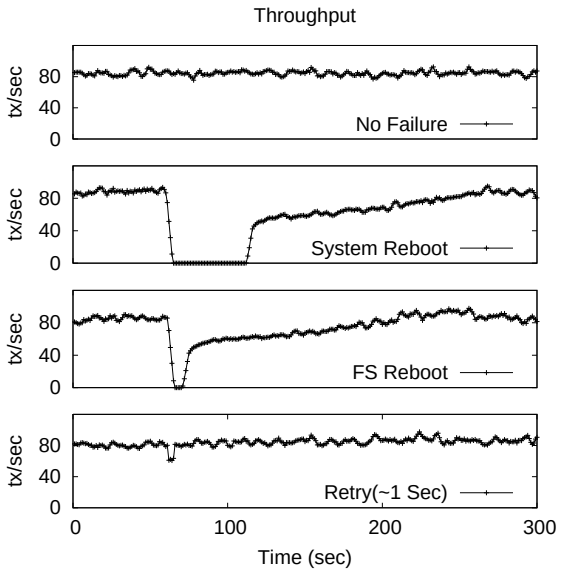


Figure 6.7: **Reboot vs. Retry (Throughput).** *The throughput of PostgreSQL 8.2.4 running pgbench is depicted. The database is initialized with 1.5 GB of data, and the workload performs a series of simple SELECTs. Four graphs are presented: the first with no fault injected (top), and the next three with a transient fault. The bottom three graphs show the different responses from three different policies: full system reboot, file system reboot, and retry.*

error is not even propagated to the application.

The middle and bottom graphs of Figure 6.6 show CrookFS is able to correctly implement the Reboot and Retry policies; for every workload and for every type of block, CrookFS either stops the file system or retries the request and propagates the error, as desired. Further, during fault-free operation, the CrookFS implementation of these two policies has negligible overhead; Table 6.4 shows that the performance of these two policies is equivalent to the simple Propagate policy on the three standard workloads.

Figure 6.7 compares the availability implications of system reboot, a file system microreboot (in which the file system is unmounted and remounted), and retrying in the presence of a transient fault. For these experiments, we measure the throughput of PostgreSQL 8.2.4 running a simple database benchmark (pgbench) over time; we inject a single transient fault in which the storage system is unavailable for one second. Not surprisingly, the full reboot can be quite costly; the system takes nearly

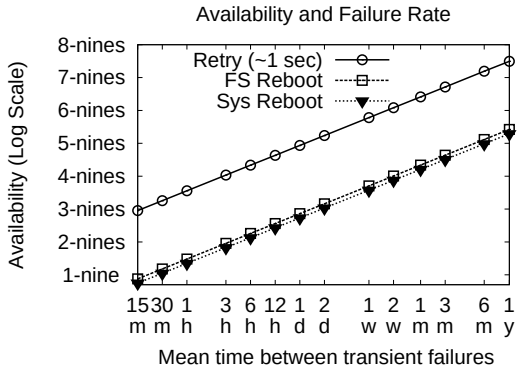


Figure 6.8: **Reboot vs. Retry (Availability).** *The graph shows the computed availability (in terms of "nines") plotted versus the mean time between transient failures for the three policies: full system reboot, file system reboot, and retry. The system is considered "available" when its delivered performance is within 10% of average steady-state performance.*

a minute to reboot, and then delivers lower throughput for roughly another minute as the cache warms. The microreboot fairs better, but still suffers from the same cold-cache effects. Finally, the simple retry is quite effective in the face of transient faults.

Given these measurements, one can calculate the impact of these three reliability policies on system availability. Figure 6.8 plots system availability as a function of the frequency of transient faults, assuming that unavailability is due only to transient faults and that the system is available when its delivered throughput is within 10% of its average steady-state throughput. To calibrate the expected frequency of transient faults, we note that although most disks encounter transient faults only a few times a year, a poorly-behaving disk may exhibit a transient fault once per week [13]. Given a weekly transient fault, the reboot strategy has availability of only "three 9s", while the retry strategy has "six 9s".

In summary, it is well known that rebooting a system when a fault occurs has a large negative impact on availability; however, many commodity file systems deployed today still stop the system instead of retrying an operation when they encounter a transient error (*e.g.*, ext3 and ReiserFS [106]). With CrookFS, one can easily specify a consistent retry policy that adds negligible slowdown and can improve availability markedly in certain environments.

6.5.4 Parity Protection

With the increasing prevalence of latent sector errors [13], file systems should contain reliability policies that protect against data loss. Such protection is usually available in high-end RAIDs [30], but not in desktop PCs [106]. For our next reliability policy, we demonstrate the ease with which one can add parity protection to a single drive so that user data can survive latent sector errors.

The parity policy is slightly more complex than the retry and reboot policies, but is still quite reasonable to implement in CrookFS; as shown in Table 6.3, our simple parity policy requires 28 lines of code. As described in Section 6.4.1, calculating parity efficiently is a non-idempotent operation, and thus the policy code must perform old-value logging. We employ a static parity scheme, which adds one parity block for k file system blocks (k is configured at boot time). A static map is used to locate parity blocks.

To help configure the value of k, we examine the trade-off between the probability of data loss and space overhead. Figure 6.9 shows both the probability of data loss (bottom) and the space overhead (top) as a function of the size of the parity set. To calculate the probability of data loss, we utilize recent work reporting the frequency of latent sector errors [13], as described in the figure caption. The bottom graph shows that using too large of a parity set leads to a high probability of data loss; for example, one parity block for the entire disk (the rightmost point) has over a 20% chance of data loss. However, the top graph shows that using too small of a parity set leads to high space overheads; for example, one parity block per file system block (the leftmost point) is equivalent to mirroring and wastes half the disk. A reasonable trade-off is found with parity sets between about 44 KB and 1 MB ($k = 10$ and $k = 255$); in this range, the space overhead is reasonable (*i.e.*, less than 10%) while the probability of loss is small (*i.e.*, less than 0.001%). In the rest of our parity policies, we use parity sets of $k = 10$ blocks.

Adding parity protection to a file system can have a large impact on performance. Figure 6.10 shows the performance of the parity policy for sequential and random access workloads that are either read or write intensive. The first graph shows, given no faults, that random reads perform well; however, random updates are quite slow. This result is not surprising, since each random update requires reading the old data and parity and writing the new data and parity; on a single disk, there is no overlap of these I/Os and hence the poor performance. The second graph shows that when there are no faults, the performance impact of parity blocks on sequential I/O is minimal, whether performing reads or writes. The parity policy code optimizes sequential write performance by buffering multiple updates to a parity block and then flushing the parity block in a chained transaction. Finally,

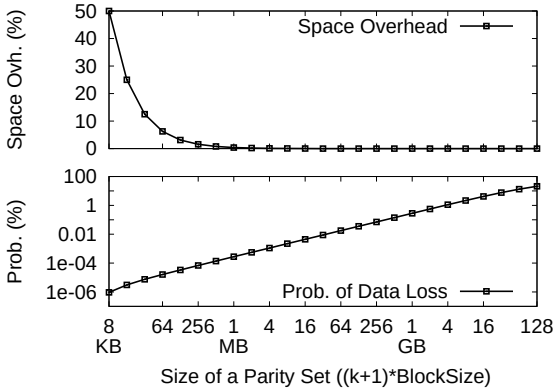


Figure 6.9: **Overhead and Reliability with Parity.** *The bottom graph shows the probabilty of data loss and the top graph the space overhead, as the size of the parity set is increased from 2 4-KB blocks (equivalent to mirroring) to one parity block for the entire disk. To compute probability of data loss, we focused on the roughly 1 in 20 ATA disks that exhibited latent sector errors; for those disks, the data in [13] reports that they exhibit roughly 0.01 errors per GB per 18 months, or a block failure rate F_B of 2.54×10^{-8} errors per block per year. Thus, if one has such a disk, the odds of at least one failure occurring is $1 - P(NoFailure)$ where $P(NoFailure) = (1 - F_B)^N$ on a disk of size N. For a 100 GB disk, this implies a 63% chance of data loss. A similar analysis is applied to arrive at the bottom graph above, but assuming one must have 2 (or more) failures within a parity set to achieve data loss. Note that our analysis assumes that latent sector errors are independent.*

given a latent sector error on each initial read, read performance is significantly slower because the data must be reconstructed; however, this is (hopefully) a rare case.

In summary, CrookFS can be used to add parity protection to file systems. Although parity protection can incur a high performance cost for random update-intensive workloads (*e.g.*, TPC-B in Table 6.4), it still adds little overhead in many cases. We believe that parity protection should be considered for desktop file systems, since it enables important data to be stored reliably even in the presence of problematic disks.

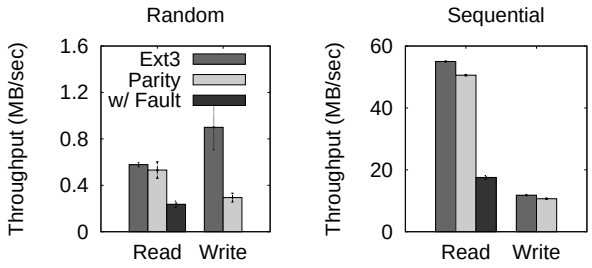


Figure 6.10: **Parity Throughput.** *The figure plots the throughput of the parity policy under some simple microbenchmarks. For sequential writes, we simply write 24 MB to disk. For random reads and writes, we either read or update random 4-KB blocks in a large (2 GB) file. For reads, both the normal and failure cases are reported; failures are injected by causing each initial read to fail which triggers reconstruction. Each experiment is repeated 60 times; averages and standard deviations are reported.*

6.5.5 Mirroring

For parity protection, we assumed that the parity location was determined when the file system was created. However, to improve performance or reliability, more sophisticated policies may wish to control the location of redundant information on disk. We explore this issue in the context of a policy that mirrors user data blocks. The code for this policy has been presented (Section 6.3); implementing it requires 18 statements, as shown in Table 6.3.

We first examine the cost of mirroring during writes. The leftmost graph of Figure 6.11 presents the results of a simple experiment that repeatedly writes a small 4 KB block to disk synchronously. Three approaches are compared. The first approach does not mirror the block (None); the second does so but places the copy as near to the original as possible (Near); the third places the copy as far away as possible (Far). As one can see, placing the copy nearby is nearly free, whereas placing the blocks far away exacts a high performance cost (a seek and a rotation). However, in terms of reliability, the far strategy is better as spatial localized faults could occur.

However, when reading back data from disk, spreading mirrors across the disk surface can improve performance [71, 148]. The rightmost graph of the figure shows an experiment in which a process reads a random block alternately from

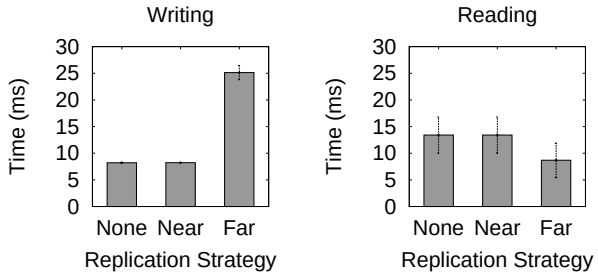


Figure 6.11: **Mirroring: Layout and Scheduling.** *The leftmost graph shows the average cost of writing a small file (4 KB) synchronously to disk, under three different replication strategies. The rightmost graph shows the average cost of reading a random 4 KB block alternately from two files. Different replication strategies are used; "None" indicates no replication, "Near" that replicas are placed as close to the original as possible, and "Far" that replicas are placed approximately 20 GB away).*

each of two files placed on opposite ends of the disk. Without replication (None), performance is poor, incurring high seek costs. With the file replica near its original (Near), there is also no benefit, as expected. Finally, with replicas far away, read performance improves dramatically: the scheduler is free to pick the copy to read from, reducing access time by nearly a factor of two.

In summary, the best choice for mirror locations is highly nuanced and depends on the workload. As expected, when the workload contains a significant percentage of metadata operations, performance suffers with mirroring, regardless of the mirror location (*e.g.*, the PostMark and TPC-B workloads shown in Table 6.4). However, in other cases, the location does matter. If spatially localized faults are likely, or read operations dominate (*e.g.*, in a transactional workload), the Far replication strategy is most appropriate; however, if data write performance is more critical (*e.g.*, in an archival scenario), the Near strategy may be the best choice. In any case, CrookFS can be used to dynamically choose different block layouts within a reliability policy.

6.5.6 Sanity Checks

Our next policy demonstrates that CrookFS allows different reliability mechanisms to be applied to different block types. For example, different sanity checks can be

applied to different block types; we have currently implemented sanity checking of inodes.

Sanity checking detects whether a data structure has been corrupted by comparing each field of the data structure to its possible values. For example, to sanity check an inode, the mode of an inode is compared to all possible modes and pointers to data blocks (*i.e.*, block numbers) are forced to point within the valid range. The drawback of sanity checks are that they cannot detect bit corruption that does not lead to invalid values (*e.g.*, a data block pointer that is shifted by one is considered valid as long as it points within the valid range).

Table 6.3 shows that sanity checks require only 10 statements of policy code, since the I/O shepherd contains the corresponding primitive. To evaluate the performance of inode sanity checking, we constructed two inode-intensive workloads: the first reads one million inodes sequentially while the seconds reads 5000 inodes in a random order. Our measurements reveal that sanity checking incurs no measurable overhead relative to the baseline Propagate policy, since the sanity checks are performed at the speed of the CPU and require no additional disk accesses. As expected, sanity checks also add no overhead to the three workloads presented in Table 6.4.

In conclusion, given that sanity checking has no performance penalty, we believe all file systems should sanity check data structures; we note that sanity checking can be performed in addition to other mechanisms for detecting corruption, such as checksumming. Although file systems such as ext3 do contain some sanity checks, it is currently done in an *ad hoc* manner and is diffused throughout the code base. Due to the centralized architecture of I/O shepherding, CrookFS can guarantee that each block is properly sanity checked before being accessed.

6.5.7 Multiple Levels of Defense

We next demonstrate the use of multiple data protection mechanisms within a single policy. Specifically, the multiple levels of defense policy uses checksums and replication to protect against data corruption. Further, for certain block types, the policy employs repair routines when a structure does not pass a checksum match but looks mostly "OK" (*e.g.*, all fields in an inode are valid except time fields). Finally, if all of these attempts fail to repair metadata inconsistencies, the system unlocks the block, queues any pending requests, runs `fsck`, and then remounts and begins running again. As indicated in Table 6.3, the multiple levels of defense policy is one of the more complex policies, requiring 39 lines of code.

Figure 6.12 shows the activity over time in a system employing this policy for four different fault injection scenarios; in each case, the workload consists of

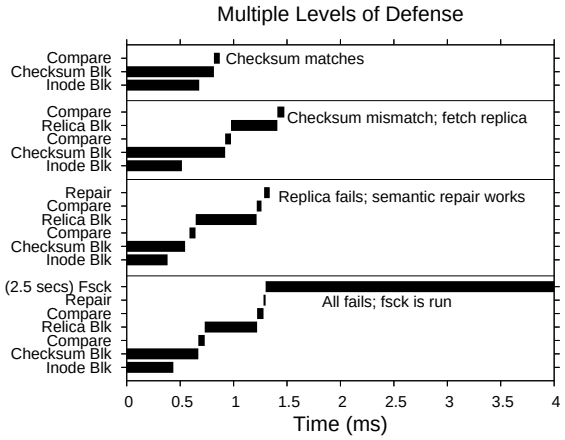


Figure 6.12: **A Multi-Level Policy.** *The figure shows four different runs of the multiple lines of defense policy. From top to bottom, each experiment induces a new fault and the y-axis highlights which action the system takes. These experiments use UML, which impacts absolute timing.*

reading a single inode. The topmost part of the timeline shows what happens when there are no disk faults: the inode and its checksum are read from disk and the checksums match, as desired. In the next experiment, we inject a single disk fault, corrupting one inode; in this case, when the policy sees that the checksums do not match, it reads the alternate inode which matches, as desired. In the third, we perform a small corruption of both copies of the inode; here, the policy finds that neither inode's calculated checksum matches the stored checksum, but finds that the inode looks mostly intact and can be repaired simply (*e.g.*, clears a non-zero dtime because the inode is in use). In our final experiment, we corrupt both copies of the inode more drastically. In this case, all of these steps fail to fix the problem, and the policy runs the full `fsck`; when this action completes, the file system remounts and continues serving requests (not shown).

The performance overhead of adding multiple levels of defense for inode blocks is summarized in Table 6.4. Given no faults, the basic overheads of this policy are to verify and update the inode checksums and to update the inode replicas. Although updating on-disk checksums and replicas is costly, performing multiple levels of defense has a smaller performance penalty than some other policies since the checks are applied only to inode blocks.

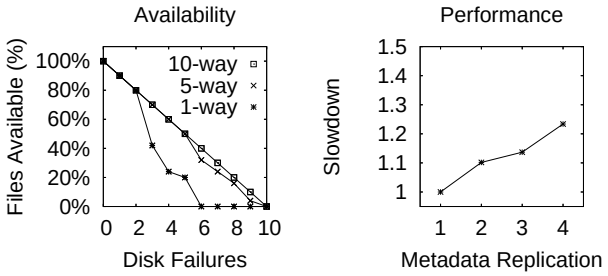


Figure 6.13: **D-GRAID Availability and Performance.** *The graphs show the availability and performance of D-GRAID on a workload creating 1000 4-KB files. On the left, each line varies the number of metadata replicas, while increasing the number of injected disk failures along the x-axis up to the full size of an emulated 10-disk array. The y-axis plots the percentage of files available. On the right, performance on four disks is shown as the number of metadata replicas increases; the y-axis shows slowdown compared to a single copy.*

6.5.8 D-GRAID

To demonstrate the full flexibility of CrookFS, we consider a fine-grained reliability policy that enacts different policies for different on-disk data types. We explore this policy given multiple disks. In this final policy, we implement D-GRAID style replication within the file system [123]. In D-GRAID, directories are widely replicated across many disks and a copy of each file (*i.e.*, its inode, all data blocks, and any indirect blocks) is mirrored and isolated within a disk boundary. This strategy ensures graceful degradation in that the failure of a disk does not render all of the data on the array unavailable.

With shepherding, the D-GRAID policy is straightforward to implement. First, the policy code for important metadata (such as directories and the superblock) specifies a high degree of replication. Second, the policy for inode, data, and indirect blocks specifies that a mirror copy of each block should be dynamically allocated to a particular disk. As indicated in Table 6.3, the D-GRAID policy requires 79 statements; although this is more than any of the other policies, it is significantly less than was required for the original D-GRAID implementation [123].

Figure 6.13 presents the availability and performance of CrookFS D-GRAID. The leftmost graph shows the availability benefit: with a high degree of metadata replication, most files remain available even when multiple disks fail. The right-

most graph shows performance overhead; as the amount of metadata replication is increased from one to four, the time for a synchronous write (and thus metadata-intensive) benchmark increases by 25%.

In conclusion, CrookFS is particularly interesting given multiple disks since it enables the file system to add reliability features across the disks without a separate volume manager (much like ZFS [130]). Due to the ability of CrookFS to enact different policies for different block types, we are able to implement even relatively complex reliability policies, such as D-GRAID.

6.6 Conclusion

In this paper, we have described a flexible approach to reliability in file systems. I/O Shepherding provides a way to tailor reliability features to fit the needs of applications and the demands of the environment. Through its basic design, shepherding makes sophisticated policies simple to describe; through careful integration with the rest of the system, shepherding implements policies efficiently and correctly.

6.6.1 Porting the Shepherd

Similar to other interfaces internal to the OS [84], we believe that multiple file systems can leverage the same functionalities we have provided in our current shepherding framework. Hence, I/O shepherding can be seen as a general layer of which all file systems can take advantage.

At this point, we have ported shepherding to Linux ext2 and partially to ReiserFS. The ext2 port has been straightforward, as it is simply a non-journaling version of ext3. Thus, we removed all consistency management code and embrace the ext2 philosophy of writing blocks to disk in any order. ReiserFS has been more challenging as it utilizes entirely different on-disk structures than the ext family. Thus far, we have successfully built simple policies. Through this work, we are slowly gaining confidence about the general applicability of the shepherding approach.

6.6.2 Lessons

Adding reliability through the I/O shepherd was simple in some ways and challenging in others. In the process of building the environment, we have learned a number of lessons.

Interposition simplifies fault management. One of the most powerful aspects of I/O Shepherding is its basic design: the shepherd interposes on all I/O and thus can implement a reliability policy consistently and correctly. Expecting reliability from code that is scattered throughout is unrealistic.

Block-level interposition can make things difficult. The I/O shepherd interposes on block reads and writes that the file system issues. While natural for many policies (*e.g.*, replicating blocks of a particular type), block-level interposition makes some kinds of policies more difficult to implement. For example, implementing stronger sanity checks on directory contents (which span many blocks) is awkward at best. Perhaps a higher-level storage system interface would provide a better interposition target.

Shepherding need not be costly. The shepherd is responsible for the execution of all I/O requests in the system. Careful integration with other subsystems is essential in achieving low overheads, with particular attention paid to the concurrency management infrastructure.

Good policies stem from good information. Although not the main focus of this paper, shaping an appropriate reliability policy clearly requires accurate data on how the disks the system is using actually fail as well as the nature of the workloads that run on the system. Fortunately, more data is becoming available on the true nature of disk faults [13, 115]; systems that deploy I/O shepherding may also need to incorporate a fault and workload monitoring infrastructure to gather the requisite information.

Fault handling in journaling file systems is challenging. By its nature, write-ahead logging places intentions on disks; reactive fault handling by its nature must behave reasonably when these intentions cannot be met. Chained transactions help to overcome this inherent difficulty, but at the cost of complexity (certainly it is the most complex part of our code). Alternate simpler approaches would be welcome.

Fault propagation is important (and yet often buggy). An I/O shepherd can mask a large number of faults, depending on the exact policy specified; however, if a fault is returned, the file system above the shepherd is responsible for passing the error to the calling application. Unfortunately, we have found many bugs in error propagation. In the previous chapter, we have presented a static analysis tool that shows where error-codes are dropped in file systems and storage drivers.

Chapter 7

Related Work

This chapter discusses various research efforts and real systems that are related to this dissertation. We first discuss literature on building more robust file systems and then close this chapter with with other efforts in analyzing system robustness.

7.1 Building Robust File and Storage Systems

This section discusses three classes of approach in building more reliable file and storage systems: adding some forms of redundancy (*e.g.*, replication, checksumming), using specification, and redesigning how systems are built.

7.1.1 Adding Redundancy

Our work 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 [6]. 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 [46]. With I/O shepherding, these techniques can be incorporated into the file system, where all applications can benefit from them. Note that I/O shepherding is 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.

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 [30]. Further, virtually all Network Appliance products use checksumming to detect block corruption [69]. Similarly, systems such as the Tandem NonStop kernel [18] include end-to-end checksums, to handle problems such as misdirected writes [18].

Sivathanu *et al.* also embraces more replication within a RAID-5 storage array [123]. They find that, in a RAID-5 storage array, if one disk fails before another is repaired, the entire array is corrupted. Until a time-consuming restore from backup, the entire array remains unavailable, although most disks are still operational. Thus, they propose D-GRAID to ensure that most files within the file system remain available even when an unexpectedly high number of faults occur. This done by replicates naming and system meta-data structures of the file system to a high degree while using standard redundancy techniques for data. Thus, with a small amount of overhead, excess failures do not render the entire array unavailable. Instead, the entire directory hierarchy can still be traversed, and only some fraction of files will be missing, proportional to the number of missing disks.

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 [92]. 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 [148]. 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.

Checksumming is also becoming more commonplace to improve system security. For example, both Patil *et al.* [100] and Stein *et al.* [126] 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 Sun ZFS is a good example of a file system that uses IRON techniques [23]. ZFS uses checksums to detect block corruption and employs redundancy across multiple drives to ensure recoverability.

7.1.2 Using Specification

Specification languages like Alloy [74] and Z [31] are useful for describing constraints of a system and then finding violations of that model. Demsky and Rinard took this approach for writing a specification for a simplified Linux file system [32].

For example, they can express consistency constraints such as “the inode bitmap is consistent with the use of inodes”, “blocks are not shared between files or other disk structures”, and “inode reference counts are correct.” Furthermore, they took the specification for automatically repairing file systems [32]; their automated repair finds the cheapest way to repair the system such that it satisfies the constraints again. For example, if two inodes share the same data block, the cheapest repair could simply remove one of the pointers; however, this may not be the desired result. In fact, there are many ways to solve the problem: the inode with the earliest modification time could release the block [93], the block could be cloned (e2fsck’s way), or the operator could decide. Thus, we believe the drawback of their work is that it does not allow one to naturally express the repairs that should be performed when violations are discovered.

Specifications are also useful to ensure correct data accesses. For example, Sivathanu *et al.* propose the notion of a type-safe disk (TSD) [122], a disk system that has knowledge of the pointer relationships between blocks. With this knowledge, a TSD can enforce invariants on data access, providing better data integrity and security. For example, it can enforce the invariant that for a block to be accessed, a parent block pointing to this block should have been accessed in the recent past. With this invariant, it is impossible for a buggy file system to access an unallocated block.

Developers could also use simple specifications to ensure correct error-code propagation. For example, developers could adopt a simple set-check-use methodology [21], *i.e.*, before using a value that was set before, one should check the corresponding error code. However, it is interesting to see that this simple practice has not been applied thoroughly in file systems and storage device drivers. As mentioned in Section 5.4, we suspect that there are deeper design shortcomings behind poor error-code handling.

7.1.3 Redesigning Systems

Numerous researchers have recently explored the advantages of using declarative languages in other domains. DeTreville introduced a checkable declarative approach to system configuration that improves system integrity and makes systems more dependable [34]. With this declarative framework, one can apply a system model to a set of system parameters to produce a statically typed, fully configured system instance. Loo *et al.* implemented P2, a system that uses a declarative logic language to express overlay networks in a highly compact and reusable form [89]. With P2, they can specify Chord [127], a peer-to-peer lookup protocol, in 47 simple logic rules, versus thousands of lines of code for the MIT Chord reference imple-

mentation.

Aspect-oriented programming [29, 83] addresses the general issue that code to implement certain high-level properties (*e.g.*, "performance") is *scattered* throughout systems, much as we observed that fault-handling is often diffused through a file system. Aspect-oriented programming provides language-level assistance in implementing these "crosscutting concerns," by allowing the relevant code to be described in a single place and then "weaved" into the code with an aspect compiler. Thus, one could consider building I/O Shepherding with aspects; however, the degree of integration required with OS subsystems could make this quite challenging.

Our work also drew inspiration from both the Congestion Manager (CM) [9, 17] and Click [95]. CM centralizes information about network congestion within the OS, thus enabling multiple network flows to utilize this knowledge and improve their behavior; in a similar manner, the I/O Shepherd centralizes both information and control and thus improves file system reliability. Click is a modular system for assembling customized routers [95]. We liked the clarity of Click router configurations from basic elements, and as a result there are parallels in our policies and primitives.

We also note that our chained transactions (Section 6.4.1) are similar to *compensating transactions* in the database literature [85]; both deal with the case where committed transactions are treated as irreversible, and yet there is a need to change them. In databases, this situation commonly arises when an event external to the transactional setting occurs (*e.g.*, a customer returns a purchase); in our case, we use chained transactions in a much more limited manner, specifically to handle unexpected disk failures during checkpointing.

7.2 Robustness Analysis

In this section, we present related work that analyzes system robustness, with a focus on file systems and storage systems. We first discuss techniques that use fault injection, and then formal techniques such as model checking and static analysis, and finally monitoring and modeling techniques.

7.2.1 Fault Injection

The fault-tolerance community has worked for many years on techniques for injecting faults into a system to determine its robustness [19, 24, 77, 120, 139]. These techniques differ in various ways (*e.g.*, the types of faults they can inject, the ease of

use of the framework, the monitoring capability). Some simulate hardware faults such as processor, memory, and bus faults. For example, FIAT (Fault Injection-based Automated Testing) simulates the occurrence of hardware errors by altering the contents of memory or registers [19]. Others simulate software faults. For example, FINE (Fault Injection and moNitoring Environment) injects software faults (*e.g.*, pointer errors) into an operating system and traces the execution flow of the kernel [77].

The FTAPE (Fault Tolerance And Performance Evaluator) framework [139] is closely related to our work. It consists of a workload generator and a device-driver-level disk-fault injector (which injects disk errors, but not corruption). FTAPE injects faults by automatically determining the time and location that will maximize fault propagation. The authors show that this approach leads to higher errors to faults ratio, an indication that fault-tolerant mechanisms are being well-exercised. Unlike our approach, the FTAPE fault injector does not inject type-aware faults. Also, while in our framework a fault is initiated upon an I/O read/write, they use stress-based injection techniques to inject faults during high workload activities.

Another related fault-injection study is an analysis by Siewiorek *et al.* [120]. In their analysis, they test the dependability of a file system's libraries by corrupting file pointers. Unlike our approach to pointer corruption, they do not corrupt pointers in other metadata structures and do not use type-aware corruption values.

Also closely related to our work is Brown and Patterson's work on RAID failure analysis [24]. Therein the authors suggest that the 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. Specifically, they find that while the Linux version is paranoid about transient errors and values application performance over reconstruction upon failure, the Windows and Solaris versions tolerate transient errors better and perform reconstruction more aggressively. Similar to their work, our goal is also to discover "failure policy", but target the file system (not RAID), hence requiring a more complex type-aware approach.

In recent work, Johansson analyzes run-time error propagation based on interface observations [75]. Specifically, an error is injected at the OS-driver interface by changing the value of a data parameter. By observing the application-OS interface after the error injection, they reveal whether errors occurring in the OS environment (device drivers) will propagate through the OS and affect applications. This run-time technique is complementary to our work, especially to uncover the eventual bad effects of error-broken channels.

7.2.2 Formal Techniques

Model checking is a formal technique that has been used over the years to analyze a variety of systems [76]. Recently, Bairavasundaram *et al.* use model checking to examine data protection in RAID systems [86]. They found that schemes in many RAID systems are broken; they do not protect against one or more failures, leading to unrecoverable data loss or corrupt data being returned to applications. Yang *et al.* also have adapted model checking to analyze real operating system code [96], and subsequently find bugs in file systems [147]. 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.

Static analysis is another formal technique that has been used to study file systems. For example, Yang *et al.* uses symbolic execution to automatically find bugs in file system code that sanity checks on-disk data structure values [146]. In this work, they found bugs in ext2, ext3, and JFS. These bugs could potentially cause a kernel panic or allow buffer overflow attacks when a malicious disk image is mounted. Meta-level compilation (MC) [39, 40] enables a programmer to write simple, system-specific compiler extensions to automatically check software for rule violations. With their work, one can find broken channels by specifying a rule such as "a returned variable must be checked." Compared to their work, our EDP presents more information on how error propagates and convert it into graphical output for ease of analysis and debugging.

Our EDP tool is also similar to Jex [111]. While Jex is a static analysis tool that determines exception flow information in Java programs, our tool determines the error code flow information within the Linux kernel.

7.2.3 Monitoring and Modeling

Systems can also be stress tested and monitored to understand their failure characteristics. Gray *et al.* measure the disk error rates in SATA drives by moving several petabytes of data [51]. They run programs in office-like and data-center-like setups that write and read data from large files and compare the checksum of the data looking for uncorrectable read errors. They measure about 30 uncorrectable bit errors as seen by the file system and 4 errors at the application level.

In a larger-scale study, Bairavasundaram *et al.* analyze data collected from production storage systems over 32 months across 1.53 million disks [13]. They ana-

lyze factors that impact latent sector errors, observe trends, and explore their implications on the design of reliability mechanisms in storage systems. They find that almost 20% of nearline (SATA) disk drives are afflicted by latent sector errors in 2 years of use, and that latent sector errors show high spatial and temporal locality.

In a subsequent study, Bairavasundaram *et al.* analyze corruption instances recorded in production storage systems [14]. They find that more than 400,000 instances of checksum mismatches over the 41-month period. They also find many interesting trends among these instances including: (i) nearline disks (and their adapters) develop checksum mismatches an order of magnitude more often than enterprise class disk drives, (ii) checksum mismatches within the same disk are not independent events and they show high spatial and temporal locality, and (iii) checksum mismatches across different disks in the same storage system are not independent.

These statistics and characteristics of failures can be further used for analytically modeling systems reliability. For example, Gibson develops an analytical model of the reliability of redundant disk arrays [48]. He discusses four different models ranging from a simple one that considers independent disk failures, to more complex models that include spare disks and dependency among disk unit failures. The models are validated using software simulation. In similar work, Kari develops reliability models which includes both sector faults and disk unit faults. However, he treats sector failures as independent events and does not account for the spatial locality in sector errors. [78]. In the world of archiving, Baker *et al.* model the reliability of long-term replicated storage systems [16]. They consider correlated failures that might occur due to spatial locality, assuming that the correlated failures are exponentially distributed.

Chapter 8

Future Work and Conclusions

> "*We can't do anything about an error here.*"
> – A comment in ReiserFS (inode.c, line 75)

Years of research on the design and implementation of local file systems has led to an abundance of innovations, many of which are realized in modern systems. For example, many performance enhancements have been suggested and evaluated in order to improve read and write performance [92, 94, 112]. Scalability has also been a focus, with the development of more advanced structures [132]. Consistency management has also received a great deal of attention, with work on journaling and soft updates demonstrating how to safely update on-disk structures [119]. Finally, search functionality has also been incorporated within the file system [47]. However, not all salient aspects of file system design and implementation have been as carefully studied. In particular, the *failure-handling subsystems* of modern file systems have largely been ignored.

Disks are one of the primary causes of failure in modern storage systems [87], and the manner in which their failures arise is becoming more complex. The simple view that disks either work or fail completely no longer holds. The reality today, disks not only exhibit whole-disk failure [115] but also partial and temporal failures, including latent sector faults [13, 79], block corruption [14, 46, 53], and transient faults [135]. As disk failure modes increase in their richness, file system failure handling comes into sharp focus. Thus, this becomes the focus of our work.

In this dissertation, we started with our analysis of three reliability components present in many modern file systems: the file system checker, failure policy, and journaling (Chapter 3). Ironically, we found that these subsystems are deficient in

handling partial disk failures, leading to many serious problems such as unmountable file systems and silent data loss. We note that these subsystems have been in active development for more than one decade. The fact that they are still problematic hints that dealing with disk failures is not easy. In fact, the developer comment quoted in the beginning of this chapter indicates that even when the developers are aware of the problems, they do not know how to respond.

The results of our analysis call for novel solutions towards building more reliable storage systems. Therefore, we have presented our approaches to solving the problems we have found. First, we introduced SQCK, a robust file system checker that employs declarative query language (Chapter 4). Next, we built EDP, a static analysis tool that shows error-codes propagation in file systems and storage drivers (in Chapter 5). Finally, we presented I/O shepherding, a simple yet powerful way to build robust and centralized failure policies within a file system (Chapter 6).

In this chapter, we first summarize our analysis and solutions (Section 8.1). We then list a set of lessons we learned from years of researching file system reliability (Section 8.2). Finally, we outline future directions where our work can possibly be extended (Section 8.3).

8.1 Summary

This dissertation is mainly divided in two parts: analysis of file system reliability components and our solutions to the problems we have found. We choose to focus on local file systems due to their ubiquitous presence and new challenges they present. We summarize each part in turn.

8.1.1 Analysis of File System Reliability Components

The first part of this dissertation is about analyzing how three reliability components present in many modern file systems react to partial disk failures. First, we evaluated a popular file system checker, e2fsck, the Linux ext2 checker. We injected corruptions to ext2 on-disk pointers and found that some repairs are buggy (making the repaired file system more corrupted) and some repairs are missing (leaving some corruptions unattended). We believe these problems exist because e2fsck is a complex piece of code; it performs more than 120 data structure repairs in in more than ten thousand lines of low-level C code, which is hard to reason about. As a result, it is difficult for e2fsck to combine the many pieces of information available and to ensure that all checks and repairs are done in the correct order. Other than e2fsck, other checkers (*e.g.*, ReiserFS and XFS checkers) are unfortunately writ-

ten in the same way. Thus, we believe these other checkers might have the same weaknesses as in e2fsck.

Second, we looked into failure policy, the file system component responsible for dealing with disk failures. We injected block-level read and write errors and corruptions to four commodity file systems (Linux ext3, ReiserFS, JFS, and Windows NTFS). Our findings point us to a major problem of diffused handling; policies that deal with disk failures are scattered in more than one hundred places across different sections of the file system code. This diffused handling causes policies to be inconsistent, buggy, and inflexible to change: different recovery actions are employed under similar failure scenarios, error-codes are dropped incorrectly leading to serious silent failures, and changing one simple policy requires modifications in many places.

Finally, we analyzed how journaling reacts to write failures. We uncovered that the current journaling framework cannot perform any checkpoint failure recoveries that result in metadata changes. We call this a problem of *failed intentions*. With this flaw, even a simple block remapping during a checkpoint failure cannot be done at the file system level. As a result, many modern file systems that employ journaling (such as ext3, IBM JFS, and ReiserFS) ignore checkpoint failure. Thus, the fact that we cannot recover from a checkpoint failure properly with the current journaling scheme is disastrous.

8.1.2 Towards Building Reliable Storage Systems

In the second part of this dissertation, we presented our approaches in building a new generation of robust file systems. First, we re-architected the file system checker by introducing SQCK, a robust file system checker that employs declarative query language. By writing hundreds of checks and repairs in a query language (*e.g.*, SQL), the high-level intent of the checker can be specified in a clear and compact manner. We showed that SQCK is able to perform the same functionality as e2fsck with surprisingly elegant and compact queries; we wrote e2fsck in 150 queries in about 1100 lines of SQL statement. We also showed that SQCK can improve upon the traditional checks and repairs; SQCK ensures correct ordering of repairs and enables new repairs to be plugged-in easily. SQCK achieves this simplicity and completeness with little cost to performance. Overall, we believe that the SQCK-style declarative approach will lead to a new generation of simpler, more robust, and more complete file system checking and repair.

Second, we presented Error Detection and Propagation (EDP), a static analysis tool that shows how error codes flow through the file system and storage drivers. EDP performs a dataflow analysis by constructing a function-call graph showing

how error codes propagate through return values and function parameters. We have applied EDP analysis to all file systems and 3 major storage device drivers (SCSI, IDE, and Software RAID) implemented in Linux 2.6. We found that *error handling is occasionally correct*. Specifically, we observed that low-level errors are sometimes lost as they travel through the many layers of the storage subsystem: out of the 9022 function calls through which the analyzed error codes propagate, we found that 1153 calls (13%) do not correctly save the propagated error codes. Our detailed analysis shows that many violations are not corner-case mistakes; the return codes of some functions are consistently ignored. For example, I/O write operations are more likely to neglect error codes than I/O read operations. This makes us suspect that the omissions are intentional, which again hints that dealing with disk failures is not easy.

Finally, we designed, implemented, and evaluated a new reliability infrastructure for file systems called *I/O shepherding* [60]. With I/O shepherding, the reliability policies of a file system are well-defined, easy to understand, powerful, and simple to tailor to environment and workload. The I/O shepherd achieves these ends by interposing on each I/O that the file system issues and executing a reliability policy for the given I/O. Thus, all disk fault-management policies are localized within the shepherd. Also, as part of this framework, we introduce *chained transactions*, a novel and more powerful transactional model that allows policies to handle unexpected faults during checkpointing and still consistently update on-disk structures. We showed that I/O shepherding enables simple, powerful, and correctly-implemented reliability policies by implementing an increasingly complex set of policies, incorporating data protection techniques such as retry, parity, mirrors, checksums, sanity checks, and data structure repairs; even complex policies can be implemented in less than 100 lines of code.

8.2 Lessons Learned

In this section, we present a list of general lessons we learned while working on this dissertation.

Reliability as a first class citizen: Traditionally, systems have been built with high performance as the primary goal. As a result, reliability features are not designed carefully. For example, we have shown that, in the e2fsck code, its subcomponents such as loader, scanner, checker, and repair are intermixed. We have also observed that, in the file system code, reliability features are buried deep within the code. These unelegant designs make both the intent

and the realization of the approach to reliability difficult to understand or evolve.

Furthermore, when reliability is not a first class citizen, building new reliability features on top of traditional systems is hard to achieve. For example, regardless of two decades of file system development, modern file systems still do not have online recovery; in the current state, a damaged file system needs to be taken down to be repaired, which greatly reduces availability. As Henson states, efficient online recovery is hard to build because file system data structures were not designed with "repair-driven" in mind [66]. We believe in the future, when systems are built from scratch, all aspects of reliability should be considered in the first place [26, 66, 73].

Complexity is the enemy of reliability: Recovery code is complex and hard to get right. Current approaches describe recovery in thousands of lines of low-level C code and it is scattered throughout. Thus, we advocate a higher-level strategy where the logic of reliability policies can be described clearly and concisely. This way, the completeness and correctness of the policies can be reasoned about in a straightforward manner.

We believe this lesson can be taken more broadly. The next generation software will contain many more features than today's software. If we still write these systems in low-level system languages such as C code, we believe failures will not be manageable. Unfortunately, that is the state-of-the-art of how we build large systems such as file system and operating systems. Thus, it is a truly great challenge to come up with higher-level approaches that can describe how big and complex systems should operate.

Interfaces to support reliability testing: Related to the first lesson above, we believe that if a system is built with reliability in mind, it should provide suitable interfaces that enable a variety of reliability testing. For our case, interfaces that provide type information would have helped greatly. In our experience, to perform our fault-injection experiments, we must change a considerable amount of code. More specifically, we must modify ext3, the generic buffer cache layer, and the journaling layer to pass semantics of the file system into our block-level fault injection. This is because, in the current framework, file system semantics are lost for all I/Os issued via the generic buffer cache manager and journaling layer.

We also experience a similar ordeal when analyzing e2fsck; when e2fsck reads a block of a particular type from the file system image, it simply uses

the `read()` POSIX interface. Hence, the intent of the read is not explicitly stated. Ideally, for the purpose of testing, we would like to have another wrapper that specifies more explicitly the read intent (*e.g.*, `read(blockType,..)`, which would eventually call the POSIX interface. This way, any testing that leverages type information can perform fault-injections cleanly within this new layer.

The need for formal specification: With the I/O shepherding framework, we have the machinery to implement good and complex policies, however we have not provided a method to reason about the correctness of the policies. For example, one might want to define a property such as "there is no single-point of failure"; a policy writer that forgets to mirror the mirror map will break the rule because the mirror map is a single-point of failure.

Another example is the severe implication of failed intentions during checkpointing (as described in Section 6.4.1): On a successful `fsync`, a user is likely to think that the data is consistent as "guaranteed" by journaling file systems; however, if checkpoint failure is ignored, the consistency assurance no longer holds. Ironically, the journaling framework has been widely deployed for a decade [5, 20, 64, 132, 140], but only recently have we found this major flaw. Furthermore, we unearthed this flaw via our rigorous fault-injection experiment. If only there existed a formal specification of journaling that incorporates all possible failure models (*e.g.*, including partial disk failures), we believe that this flaw could be caught easily. Besides journaling, some systems use other forms of consistency management (*e.g.*, shadow paging [52]), and there are many reliability trade-offs in these approaches that are not yet well understood. We believe using formal specification will be highly useful in these other cases.

The need for longitudinal failure simulation: Measuring data reliability is hard. Our current method in measuring robustness is to inject one failure at a time and observe how the evaluated system behaves. In reality, failures do not happen at one time and the life-span of data could be of several years. Ideally, what we need is a longitudinal failure simulator that reflects how systems fail over a long period of time. This kind of simulator needs to incorporate the failure models of all parts that can fail (*e.g.*, partial disk failures, spatial locality, *etc.*), the corresponding real-world statistics, and also real workloads that run over many years. Given such as simulator, we will be able to answer high-level questions such as: "after two years, is all my data still available?", "how often are file systems taken down due to disk failures in one year?",

"after multiple corruptions occur over several months, can fsck repair the corruptions?", and many others. By having the answers to these high-level questions, we believe that the many problems we uncovered in Chapter 3 can be more correlated to real-world scenarios. Fortunately, the storage community recently has gathered large-scale statistics of low-level disk and memory failures [13, 14, 104, 115, 116]. These statistics will be an important foundation in building a longitudinal failure simulator.

8.3 Future Work

In terms of future work, our vision is to build highly-reliable and -available systems. This section outlines various directions for this vision.

8.3.1 Continuous Checker and Repair Utility

Traditionally, file systems rely on an offline checker utility, fsck [93], to repair all inconsistencies caused by corruptions. Unfortunately, as the name suggests, offline fsck can only work when the file system is not running. Since file system downtime is usually avoided in reality, offline fsck is run very rarely (*e.g.*, every 30 mounts). As a result, the occasional runs of offline fsck is risky for reliability; corruptions are not detected early in time, and hence, corrupt data may potentially be used by the file system.

Therefore, to improve file system reliability and availability, the file system should be armed with a continuous checker and repair utility. The checker guarantees that the file system does not use corrupt data structures, while the repair restores the file system consistency without the need to shut down the file system. As mentioned in the previous section, unfortunately, most of today's file systems lack such a utility [60, 66, 106]. To build one, several challenges must be addressed. Below we present the challenges and sketch our proposal.

First, to detect a corruption, the consistency of each data structure and all of its fields needs to be verified. This is an expensive process since the whole file system must be scanned in order to run the cross-checks. Checksumming can alleviate this cost, however it is often done at a coarse-grained level (*e.g.*, sector- or block-level); it does not pinpoint which data structures are corrupt within the block. To detect a corruption in a fine-grained manner, we recommend the use of *data-structure checksumming*, with which the file system can easily retain non-corrupt data structures and repair only the corrupt ones.

After corrupt data structures have been detected, one can use existing redundancy to repair them on-the-fly. However, the corruption detection and the redundancy could appear at different levels in the storage stack (*e.g.*, checksumming at the file system level, and parity at the RAID-level). Unfortunately, the current storage interface hides low-level information from the upper levels. Thus, we propose that the storage interface has to support *cooperative repair*. Specifically, a small interface is added such that the file system can delegate the repair to the underlying storage subsystem. Such an interface that exposes more information and control has been shown to be powerful in operating and networking systems [12, 59].

Third, since redundancy is not always available, not all important metadata can be repaired. For example, commonly directory names are not replicated. A corrupt directory name could cause its subdirectories to be untraversable. We suggest the use of a *summary database* which stores partial redundancy of important file system metadata. Metadata copies can be added or removed flexibly depending on the level of availability needed.

Lastly, when all forms of fast repair cannot fix the corruption, a full online fsck is needed. Designing a full online fsck is tricky because it could unsafely modify data structures that are being used. If not designed carefully, a complex management of in-kernel data-structures is required [66]. Thus, we propose a design of online fsck that follows one important rule: it should not perform removal of data structures that are in-use. This rule can be implemented by adding a *repair bit* in each of the file system data structures. The bit is set when the corresponding data structure is found corrupt (*i.e.*, the checksum is wrong). This marker guarantees that the file system can only use non-corrupt data structures. The online fsck then performs all types of repair (update, addition, and removal) on data structures that have been marked, but could only perform update and addition (but not removal) on those that can be in-use. Without the repair bit, an online fsck cannot distinguish which data structures are safely repairable on-the-fly.

8.3.2 Solving the Problem of Incorrect Error Propagation

With EDP, we are able to catch incorrect propagation of error codes that are stored and propagated mainly in integer containers. However, file and storage systems also use other specific error codes stored in complex structures. Moreover, we have not yet provided an elegant solution that prevents developers from making the same mistakes. Before laying out our future plan on these matters, we make two important observations that shed light on the complexity of building a complete and accurate static error propagation analysis.

The first one is about *error transformation*. Each layer uses different error

codes, thus an error code transforms and its error container also changes (*e.g.*, the block layer clears the uptodate bit stored in a buffer structure to signal I/O failure, the VFS layer simply uses generic error code such as `EIO` and `EROFS`). We have observed a path where an error container changes five times involving four different type of containers. A complete analysis must recognize all transformations along with the variables or containers that hold the errors.

The second observation is about *error channels*; error codes do not propagate through function call paths only, but also *asynchronous paths*. We briefly describe two examples of asynchronous paths and their complexities. First, when a lower layer interrupts an upper one to notify it of the completion of an I/O, the low-level I/O error code is usually stored in a structure located in the heap; the receiver of the interrupt should grab the structure and check the error it carries, but tracking this propagation through the heap is not straightforward. Another example occurs during journaling: a journal daemon is woken up somewhere in the `fsync()` path and propagates a journal error code via a global journal state.

By taking into account the observations above, there are two approaches we can take. The first one is to enhance our static analysis into a more complete and sound analysis by considering error transformation and asynchronous path. However, without complete error code specification, EDP will obviously miss violations that forget to check the unspecified error codes. Thus, each layer in the system must properly declare a set of error codes that it exposes to another layer [136].

The second approach is to build a new error propagation architecture that prohibits file system programmers to make the same mistakes. Here, we advocate two approaches. First, we propose building systems with *semantic error codes*; with this approach, the system does not blindly believe in the success or failure signal reported by an error code but instead performs extra checks to confirm whether the corresponding operation is successful or not. This technique is similar to dynamic verification techniques [43]. Second, we propose adopting the *malloc-free* paradigm for error codes [82]. Specifically, once an error code is generated, it is treated as immutable and can only be destroyed if it transforms to another error code or the corresponding failure is handled. If there is a "dangling" error code, then the system has forgotten to check or handle certain faults. This new architecture ensures that errors do not disappear easily, hence reducing the instances of silent failure.

8.3.3 Other Data Management Systems

In this dissertation, we have focused on the problems of and solutions for local file systems. However, there are other systems that are also responsible for managing

data, such as distributed file systems and database management systems. These systems are more complex than local file systems with many more components in their storage stack. As an example, the MySQL database management system consists of 425,000 lines of code [4]. We believe our analyses and solutions can directly contribute to these other data management systems.

As a first step, in fact, we have examined the effects of corruption on database management systems [125]. Through fault injection of the MySQL DBMS, we find that in certain cases, corruption can greatly harm the system, leading to untimely crashes, data loss, or even incorrect results. Overall, of 145 injected faults, 110 lead to serious problems. More detailed observations point us to three deficiencies: MySQL does not have the capability to detect some corruptions due to lack of redundant information, does not isolate corruptions from valid data, and does not have a proper framework for corruption handling.

Furthermore, we also find that MySQL offline checker is not comprehensive in the checks it performs, misdiagnosing many corruption scenarios and missing others. Sometimes the checker itself crashes; more ominously, its incorrect checking can lead to incorrect repairs. Overall, we find that the checker does not behave correctly in 18 of 145 injected corruptions, and thus can leave the DBMS vulnerable to the problems described above.

We note that these findings are similar to the ones we found in file systems. We suspect there are two reasons behind this. First, the impact of partial disk failures to data management systems has not been well examined in literature and in practice. Second, many data management systems also describe recovery at a very low-level: thousands of lines of C code. Therefore, beyond local file systems, our analyses and solutions can be of significant contributions for other data management systems.

8.3.4 Revisiting Failure Management

Finally, we believe that failure management in current systems should be revisited. In this dissertation, we have found a major flaw in current journaling frameworks, bugs in error-code propagation, and design problems in managing storage failures. In short, failure management is hard. This is more true based on our interactions with some file system developers (ext4, JFS, and CIFS). These developers are aware that failures are not always handled properly, however, they may still not be able to fix all the bugs in a straightforward manner; there are larger design issues. Guo and Engler also point out a similar observation; in their study of developer responses on bug reports, they report that developers tend to address easy-to-fix bugs and defer difficult (but possibly critical) bugs [63].

Fortunately, we have published a full report of our error propagation analy-

sis [57]. This report pinpoints all places where failures are ignored in Linux file systems, and thus, can be seen as a "database" of problems. The next step is to perform an in-depth study of this database in order to unearth as many design problems as possible.

8.4 Closing Words

"The price of reliability is the pursuit of the utmost simplicity. "
– C.A.R. Hoare, "The Emperor's Old Clothes", Turing Award Lecture (1980)

Data reliability is of utmost importance. As the future work section suggests, the journey does not end here; there are still many challenges to face. In this dissertation, we have adhered to two important principles that help us face the challenges of building more reliable storage systems.

First, *reliability should be a first-class storage system concern*. The reliability principle demands that storage systems anticipate and properly handle all types of failures. Hence, it is important to critically analyze how modern data management systems react to the different types of faults that can occur, how such faults propagate through the systems, and the broader-scale failure architecture.

Second, *complexity is the enemy of reliability*. Recovery code is complex and hard to get right. Current approaches describe recovery in thousands of lines of low-level C code and it is scattered throughout different sections of the code. Thus, we have advocated a higher-level strategy where the logic of reliability policies can be described clearly and concisely. This strategy aligns with what Tony Hoare said (quoted above), but this does not mean that we are simplifying the features of today's systems. In fact, we accept the fact that tomorrow's systems will be much larger and complex than current ones. Therefore, the new challenge that this dissertation has addressed is how we can design large, reliable systems with simplicity while still keeping them powerful.

Bibliography

[1] e2fsprogs.sourceforge.net.

[2] en.wikipedia.org/wiki/SQL.

[3] en.wikipedia.org/wiki/Ext4.

[4] www.coverity.com/html/press_story05_02_15_05.html.

[5] ReiserFS. en.wikipedia.org/wiki/ReiserFS.

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

[7] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

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

[9] David G. Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan, and Hari Balakrishnan. System Support for Bandwidth Management and Content Adaptation in Internet Applications. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 213–226, San Diego, California, October 2000.

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

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

[12] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 90–105, Bolton Landing (Lake George), New York, October 2003.

[13] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, California, June 2007.

[14] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 223–238, San Jose, California, February 2008.

[15] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Systematically Benchmarking the Effects of Disk Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

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

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

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

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

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

[21] Michael W. Bigrigg and Jacob J. Vos. The Set-Check-Use Methodology for Detecting Error Propagation Failures in I/O Routines. In *Workshop on Dependability Benchmarking (WDB '02)*, Washington, DC, June 2002.

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

[23] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

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

[25] Aaron B. Brown and David A. Patterson. Towards Availability 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.

[26] George Candea and Armando Fox. Crash-Only Software. In *The Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, May 2003.

[27] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

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

[29] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *ESEC/FSE-9*, September 2001.

[30] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 1–14, San Francisco, California, April 2004.

[31] Jim Davies and Jim Woodcock. *Using Z: Specification, Refinement, and Proof* . Prentice Hall, 1996.

[32] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *The 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, Anaheim, California, October 2003.

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

[34] John DeTreville. Making system configuration more declarative. In *The Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, Sante Fe, New Mexico, June 2005.

[35] DTI/PriceWaterHouse. Information Security Breaches Survey, 2008.

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

[37] EMC. EMC Centera: Content Addressed Storage System. www.emc.com, 2004.

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

[39] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions . In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.

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

[41] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *5th International Conference Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, Venice, Italy, January 2004.

[42] Dawson R. Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*, London, United Kingdom, July 2007.

[43] R.S. Fabry. Dynamic Verification of Operating System Decisions. *Communications of the ACM*, 16(11):659–668, November 1973.

[44] Rob Funk. fsck / xfs. lwn.net/Articles/226851.

[45] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

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

[47] Dominic Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann, 1999.

[48] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, University of California at Berkeley, 1991.

[49] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, Chicago, Illinois, June 2005.

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

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

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

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

[54] Albert Greenberg, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow (PRESTO '09)*, Seattle, Washington, August 2002.

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

[56] Weining Gu, Z. Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux Kernel Behavior Under Errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, pages 459–468, San Francisco, California, June 2003.

[57] Haryadi S. Gunawi. EDP Output for All File Systems. www.cs.wisc.edu/adsl/Publications/eio-fast08/readme.html.

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

[59] Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Deploying Safe User-Level Network Services with icTCP. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 317–332, San Francisco, California, December 2004.

[60] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 283–296, Stevenson, Washington, October 2007.

[61] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[62] Haryadi S. Gunawi, Cindy Rubio-Gonzalez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 207–222, San Jose, California, February 2008.

[63] Philip J. Guo and Dawson Engler. Linux kernel developer responses to static analysis bug reports. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, San Diego, California, June 2009.

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

[65] Val Henson. The Many Faces of fsck. lwn.net/Articles/248180/, September 2007.

[66] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *IEEE 2nd Workshop on Hot Topics in System Dependability (HotDep '06)*, Seattle, Washington, November 2006.

[67] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*, Snowbird, Utah, June 2001.

[68] Hitachi Data Systems. Hitachi Universal Storage Platform V. www.hds.com.

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

[70] HP. HP XC Clusters. www.hp.com.

[71] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 263–276, Brighton, United Kingdom, October 2005.

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

[73] Galen C. Hunt, James R. Larus, Martin Abadi, Paul Barham, Manuel Fahndrich, Chris Hawblitzel Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report 2005-135, Microsoft Research, 2005.

[74] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[75] Andreas Johansson and Neeraj Suri. Error Propagation Profiling of Operating Systems . In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, Yokohama, Japan, June 2005.

[76] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

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

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

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

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

[81] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.

[82] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., 1988.

[83] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

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

[85] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB 16)*, pages 95–106, Brisbane, Australia, August 1990.

[86] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 127–141, San Jose, California, February 2008.

[87] Larry Lancaster and Alan Rowe. Measuring Real World Data Availability. In *Proceedings of the LISA 2001 15th Systems Administration Conference*, pages 93–100, San Diego, California, December 2001.

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

[89] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

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

[91] Peter Lyman and Hal R. Varian. How Much Information? 2003. www2.sims.berkeley.edu/research/projects/how-much-info-2003/.

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

[93] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[94] Jeffrey C. Mogul. A Better Update Policy. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '94)*, Boston, Massachusetts, June 1994.

[95] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 217–231, Kiawah Island Resort, South Carolina, December 1999.

[96] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[97] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: An infrastructure for c program analysis and transformation. In *International Conference on Compiler Construction (CC '02)*, pages 213–228, April 2002.

[98] John Oates. Bank fined 3 millions pound sterling for data loss, still not taking it seriously. www.theregister.co.uk/2009/07/22/fsa_hsbc_data_loss.

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

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

[101] Dave Patterson. A new focus for a new century: Availability and maintainability ¿¿ performance. Keynote at FAST 2002, February 2002.

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

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

[104] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, pages 17–28, San Jose, California, February 2007.

[105] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, pages 802–811, Yokohama, Japan, June 2005.

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

[107] American Data Recovery. Data loss statistics. www.californiadatarecovery.com/content/adr_loss_stat.html.

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

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

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

[111] Martin P. Robillard and Gail C. Murphy. Designing Robust Java Programs with Exceptions. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering (FSE '00)*, San Diego, CA, November 2000.

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

[113] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, Dublin, Ireland, June 2009.

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

[115] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, pages 1–16, San Jose, California, February 2007.

[116] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, Seattle, Washington, June 2007.

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

[118] Simon CW See. Data Intensive Computing. In *Sun Preservation and Archiving Special Interest Group (PASIG '09)*, San Fancisco, California, October 2009.

[119] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 71–84, San Diego, California, June 2000.

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

[121] Aameek Singh, Madhukar Korupolu, and Kaladhar Voruganti. Zodiac: Efficient impact analysis for storage area networks. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, California, December 2005.

[122] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

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

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

[125] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Jeffrey F. Naughton. Impact of Disk Corruption on Open-Source DBMS. In *Proceedings of the 26th International Conference on Data Engineering (ICDE '10)*, Long Beach, California, March 2010.

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

[127] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *Proceedings of SIGCOMM '01*, San Diego, California, August 2001.

[128] Scott Studham. Commoditization of high performance storage: breaking into the next frontier. In *Scientific Computing and Instrumentation*, April 2004.

[129] Sun. Sun StorageTek 5800 System. www.sun.com.

[130] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[131] Rajesh Sundaram. The Private Lives of Disk Drives. www.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html, February 2006.

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

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

[134] Alexander Szalay and Jim Gray. 2020 Computing: Science in an exponential world. *Nature*, (440):413–414, March 2006.

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

[136] Douglas Thain and Miron Livny. Error Scope on a Computational Grid: Theory and Practice. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC 11)*, Edinburgh, Scotland, July 2002.

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

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

[139] 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.

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

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

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

[143] Glenn Weinberg. The Solaris Dynamic File System. members.visi.net/∼thedave/sun/DynFS.pdf, 2004.

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

[145] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[146] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *IEEE Security and Privacy (SP '06)*, Berkeley, California, May 2006.

[147] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implement ation (OSDI '04)*, San Francisco, California, December 2004.

[148] 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.