CHARACTERISTICS, IMPACT, AND TOLERANCE OF PARTIAL DISK FAILURES

by

Lakshmi Narayanan Bairavasundaram

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2008

To my parents and my sister

ACKNOWLEDGMENTS

I would like to thank my advisors, Andrea and Remzi, for making my Ph.D. experience truly wonderful. I have learned a great deal from them about how to do research as well as how to present my work. Even more than that, I have learned the importance of commitment and zeal for research; I'll never forget that Remzi sent me an email with comments about a paper in the middle of the night – as he was waiting for their daughter Maddy to be born. Finally, I cannot thank them enough for the care and concern they showed for my well-being throughout my Ph.D.

I would like to thank my other thesis-committee members, Somesh Jha, Ben Liblit, Mike Swift, and Kjell Doksum, for their insights, questions, and advice for my research. I really enjoyed working with Mike Swift on a paper that is a part of this dissertation.

During my six years in Madison, I have thoroughly enjoyed working with my colleagues Nitin Agrawal, John Bent, Nate Burnett, Tim Denehy, Haryadi Gunawi, Todd Jones, Shweta Krishnan, Andrew Krioukov, Joe Meehan, Florentina Popovici, Vijayan Prabhakaran, Abhishek Rajimwale, Meenali Rungta, Muthian Sivathanu, and Swami Sundararaman. Collaborating with Muthian and Vijayan during my initial years here has influenced my research a great deal. Haryadi has been an awesome officemate all of these years (he's the kind that furnishes the office with a nice couch).

I have benefited greatly from internships at IBM T.J. Watson Research Center, Intel Corporation, and NetApp, Inc. I would like to thank the companies as well as my mentors and managers there: Bulent Abali and Mohammad Banikazemi at IBM Research, Steve Bennett, Alain Kagi, and Rich Uhlig at Intel, and David Ford, Garth Goodson, Stephen Harpster, and Shankar Pasupathy at NetApp. In addition, I had a great experience working with Jiri Schindler, Kiran Srinivasan, and Randy Thelen at NetApp. The data on disk failures that I analyzed at NetApp forms an important part of my dissertation. I would like to thank NetApp, Inc. for giving me the opportunity to do

the analysis. During my internship at NetApp in 2007, I also had the opportunity to collaborate with Bianca Schroeder. I would like to thank her for sharing her insights on disk failures, system failures and data analysis with me.

A lot of my motivation for pursuing a Ph.D. came from my experiences as an undergrad at Anna University in Chennai, India. I would like to especially thank my advisor there, Ranjani Parthasarathi, for instilling in me the love for research.

I would like to thank my roommates of five years, Gautham and Nitin, for making my stay in Madison simply amazing. They have appreciated the best of me and put up with the worst of me. They have indeed been my family here. I loved the passionate arguments with “Gau” about things completely unrelated to us. I’ve always been surprised at how Nitin can put up with me both at work and at home. I would also like to thank Anoop for being a part of the family for the two years he was in Madison.

I consider myself very lucky to have made some great friends during my years in Madison. The company of Igor, Jessie, Koushik, “Lacrosse”, Megan, Michelle, and Sharon made the winters seem much milder than they probably were. The hours and hours talking about everything and nothing with Igor were just wonderful. Jessie has an uncanny knack for knowing when I am feeling down and cheering me up. She also made sure I ate on days close to deadlines when food is typically low on the priority list.

I am very thankful for the splendid support that my friends from high school and college have given me. Naveen, Pothi, and Rajesh have always been there for me. The conference calls with Arumugam, Pradeep, Sai, Sibi, and Vijayaraghavan have been the times I have laughed the hardest (much needed!). Sam and Vijay have given me great support. My graduate school years have been punctuated with awesome trips to meet all of them.

Finally, I would like to thank my family, without whose love and support this Ph.D. would have been impossible. My sister, Archana, has cheered me on for as long as I can remember. It is unbelievable how much my success means to her; I even recall an instance in our childhood when she teared up because I didn’t score as high on an exam as she thought I deserved. Her husband, Bhupesh, has now joined her in cheering me on. My parents have sacrificed much in making great

education opportunities a possibility. They have always been there for me when I needed them. They were with me the first day of kindergarten, and they were with me the day of my Ph.D. defense (they even prepared and brought me lunch!). I dedicate this dissertation to my family.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

Page

Page

DISCARD THIS PAGE

LIST OF TABLES

Table Page

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

Figure Page

CHARACTERISTICS, IMPACT, AND TOLERANCE OF PARTIAL DISK FAILURES

Lakshmi Narayanan Bairavasundaram

Under the supervision of Professors Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau

At the University of Wisconsin-Madison

Hard-disk failures are one of the primary causes of data loss. Most disk failures are partial failures, where only some sectors are unavailable due to a latent sector error or some blocks are silently corrupted. This dissertation focuses on all aspects of such partial disk failures – their *characteristics*, their *impact* on different systems, and techniques that can be used *tolerate* them.

We perform the first large-scale study of partial disk failures, involving 1.53 million disks. We find that partial disk failures affect a large percentage of disks. We also find that (i) SATA drives have a higher probability of developing partial disk failures and (ii) failures are not independent; failures within the same disk have high spatial and temporal locality.

We examine the impact of partial disk failures on a variety of systems. We use model checking to examine data protection in RAID systems, and find that most schemes do not protect against one or more failures, leading to data loss. We apply *type-aware fault injection* to examine the impact of partial disk failures on the virtual-memory systems of Linux, FreeBSD, and Windows XP. We find that these systems use simplistic or inconsistent failure-handling policies. We analyze the impact of corrupt on-disk pointers on file systems NTFS and ext3. We find that these systems do not use fault-tolerance techniques effectively, resulting in data loss.

We have built an *N-version file system* to tolerate partial disk failures. This system stores data in N different file systems, thereby eliminating the reliance on a single complex file system. Our system uses existing file systems, such as ext3 and JFS, thus avoiding the development costs of building different versions. Our experiments show that an N-version file system significantly reduces the probability of data loss.

Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau

ABSTRACT

Hard-disk failures are one of the primary causes of data loss in both enterprise storage systems and personal computers. Most disk failures are partial failures, where only some sectors are unavailable due to a latent sector error or some blocks are silently corrupted. This dissertation focuses on all aspects of such partial disk failures – their *characteristics*, their *impact* on different systems, and techniques that can be used *tolerate* them.

We perform the first large-scale study of partial disk failures, involving 1.53 million disks in more than 50,000 storage systems. We find that partial disk failures affect a large percentage of disks (*e.g.*, in the worst case, latent sector errors affect up to 20% of the disks in 2 years). We also find that (i) inexpensive SATA drives have a higher probability of developing partial disk failures, (ii) failures are not independent; failures within the same disk have high spatial and temporal locality, and (iii) many failures are detected by background scans of disk blocks called "disk scrubbing."

We examine the impact of partial disk failures on a variety of systems. We use model checking to examine data protection in RAID systems. We find 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. We apply *type-aware fault injection* to examine the impact of partial disk failures on the virtual-memory systems of Linux, FreeBSD, and Windows XP. We find that these systems use simplistic or inconsistent failure-handling policies, thus causing data corruption and system-security violations. We analyze the impact of corrupt on-disk pointers on two file systems, NTFS and ext3. We find that these systems do not use available fault-tolerance techniques effectively, resulting in data loss and non-mountable file systems. Overall, we find that a single system cannot be depended upon to reliably store data.

We have built an *N-version file system* to tolerate partial disk failures. This system stores and retrieves data from N different file systems, thereby eliminating the need to rely on a single complex file system. Our system uses existing file systems, such as ext3 and JFS, thus avoiding the development costs of building different versions. We show using fault-injection experiments that an N-version file system significantly reduces the probability of data loss.

Chapter 1

Introduction

Much of the value people place in computer systems stems from the value of the data stored therein. Today, this emphasis on data is true not only for enterprise systems in corporate or government settings, but also for personal computers, which store valuable photos, home videos, and important documents such as tax returns. By some estimates, about 5 exabytes of information was produced in the world in just a single year (2002), and ninety-two percent of this new information was stored on magnetic media, mostly on hard disk drives [87]. Thus, it is primarily the information stored in our computer systems that makes them so valuable to us.

Given the rising importance of information, it is not surprising that data reliability and integrity are considered vital to storage systems. Performance problems can be tuned, tools can be added to cope with management issues, but data loss is seen as catastrophic. As Keeton *et al.* state, data unavailability may cost a company "... more than $1 million/hour" and the price of data loss is "even higher" [78].

Unfortunately, disk drives fail, and they fail more often than manufacturers expect them to [118]. Even in well-designed, high-end systems, disk-related errors are still one of the main causes of potential trouble [84]. With the increasing amount of valuable information stored on hard disks, the onus is now on file systems and storage systems to handle disk failures, thereby preserving data over long periods of time.

For many years, file-system and storage-system designers have assumed that disks operate in a "fail-stop" manner [116]; within this classic model, the disks either are working perfectly, or fail absolutely and in an easily-detectable manner. The failure model presented by current disk drives, however, is much more complex. For example, drives can exhibit *latent sector errors* [35, 77, 119],

where a disk block or set of blocks are inaccessible. Worse, disk blocks sometimes become *silently corrupted* [18, 54, 130, 131]. We refer to this complex set of failures as *partial disk failures*.

Partial disk failures occur due to a variety of reasons; particles within disk drives could cause scratches thereby rendering sectors unreadable [5, 119]; unacceptably high gaps between the disk read/write head and the medium cause data to be written poorly [12]; firmware bugs could cause corrupted data to be returned or cause writes to complete without actually writing data to the magnetic medium [48, 54, 130, 131]; software bugs in file systems and device drivers could corrupt data [34, 45, 133, 148, 150].

The goals of this dissertation are three-fold: first, to examine the magnitude of the threat that partial disk failures pose and identify the characteristics of such failures; second, to evaluate the impact of partial disk failures on current systems; third, to develop techniques for tolerating these failures.

We address the goals of this dissertation as follows. First, we analyze the occurrence and characteristics of two important classes of partial disk failures – latent sector errors and silent data corruptions [12, 13]. Second, we examine the impact of these partial disk failures on various storage-stack components – enterprise RAID systems [83], virtual-memory systems [11], and file systems [14, 15]. Third, we develop an N-version file system, a solution for tolerating all partial disk failures that affect file systems, including data corruption due to file-system bugs. The following sections elaborate on each of these contributions of the dissertation.

1.1 Characteristics of Partial Disk Failures

Detailed knowledge of real-world failure characteristics is essential for building systems that can tolerate failures effectively and for examining whether existing systems meet fault-tolerance goals. For example, Gray [51] used information about system failures in the field to garner the insight that system administration was the primary cause of failures at that time. Such insight has proven extremely valuable: it has spurred research efforts to (i) verify whether the observation is true in other systems [91, 96], (ii) examine how resilient systems are to human errors [79,

143], and (iii) develop techniques to handle potential problems and failures introduced by operator mistakes [27, 126].

The need for real-world data applies to partial disk failures as well. Unfortunately, there is little field data available about partial disk failures. This situation persists despite the recent influx of data on absolute disk failures [41, 102, 118]. Therefore, current techniques to handle partial disk failures have to be built based on anecdotal information and back-of-the-envelope calculations [119].

In this dissertation, we present the first large-scale studies of partial disk failures. We focus on two important types of partial disk failures: latent sector errors and silent data corruptions. These two types of failures are important since both of them could lead to data loss.

To perform the study, we analyze data from more than 50,000 production and development storage systems developed by NetApp™ and installed at many of their customer sites. The data pertains to partial disk failures affecting the 1.53 million disks used by these storage systems.

Through our study, we answer several important questions about partial disk failures, including:

- What is the magnitude of this threat? What are the average and worst-case percentages of disks affected by partial disk failures in given period of time?
- What factors impact the development of partial disk failures? For instance, do older disks have a higher probability of being affected? The world is moving to larger capacity disk drives; does this exacerbate the problem of partial disk failures?
- Are partial disk failures independent events? Some file systems use intra-disk redundancy to overcome partial disk failures [20, 88]; should these systems be concerned about spatial locality of partial disk failures?
- Are the partial disk failures in our study and other disk errors like not-ready-condition errors independent occurrences?
- What techniques are useful for detecting partial disk failures? Is “disk scrubbing,” a periodic scan of all disk blocks, useful for detecting partial disk failures?

One important lesson we learn from the study is that partial disk failures affect a significant percentage of disk drives; latent sector errors affect up to 20% of the drives of one of the SATA disk models in just 2 years. Such a high percentage implies a need for maintaining redundant information to protect against data loss. Single-disk systems should strive for intra-disk redundancy, perhaps in the form of replicated file-system metadata [20, 88, 104], while RAID [101] systems should consider protecting against double disk failures [3, 22, 35, 57, 59, 98].

1.2 Impact of Partial Disk Failures

Given that partial disk failures affect a large number of disks, it is important to understand how the partial disk failures impact current systems. If existing systems handle partial disk failures in an efficient manner, then there is no need for new techniques; if they are somewhat inefficient, such a study would show us where the systems fall short and suggest improvements to the failure-handling techniques used by the systems.

Many different types of systems use disk drives directly, including RAID systems, file systems, virtual-memory systems, and database systems. In this dissertation, we examine the impact of partial disk failures on three types of systems: RAID systems, virtual-memory systems, and file systems.

We use model checking [73] to examine the design of data protection in RAID systems, and extend a fault-injection technique called type-aware fault injection [104] to examine virtual-memory systems and file systems. We first outline our analysis of RAID, then briefly describe type-aware fault injection, then discuss our analyses of virtual-memory systems and file systems.

1.2.1 RAID Systems

RAID (Redundant Array of Independent Disks) stores data on multiple disks in a redundant fashion in order to survive the failure of one or more of the disks [101]. Since it was originally proposed, it has been employed in nearly every enterprise storage system [43, 65, 68, 94].

RAID is specifically targeted towards handling disk failures; therefore, one would expect a thorough and verifiable failure-handling scheme. Although getting an implementation to work

correctly may be challenging (often involving hundreds of thousands of lines of code [146]), one could feel confident that the design properly handles the expected failures. Indeed, over the years, analytical modeling has been used to evaluate the fault-tolerance capability of RAID systems [17, 39, 49, 76, 119] under the assumption that disk failures, whether absolute or partial, will be detected by the RAID system. While such an assumption would hold true for absolute failure or any partial failure that is reported by the disk drive (*e.g.*, a latent sector error), it is not necessarily true for silent data corruptions. Failures that cause silent data corruption considerably complicate the construction of correctly-designed protection strategies.

A number of techniques have been developed and used in enterprise RAID systems to cope with silent data corruption. For example, various forms of checksumming can be used to detect corruption [18, 129]; combined with redundancy (*e.g.*, mirrors or parity), checksumming enables both the detection of and recovery from certain classes of corruptions. However, given the broad range of techniques used (including sector checksums [18, 37, 65], block checksums [131], parental checksums [130], write-verify operations [131], identity information [107, 131], and disk scrubbing [37, 119, 130, 131], to list a few), exactly which techniques protect against which failures is sometimes unclear; worse, combining different approaches in a single system may lead to unexpected gaps in data protection.

We propose an approach based on model checking [73] to analyze the design of protection schemes in RAID systems. We develop and apply a simple *model checker* to examine different data protection schemes. We first implement a simple logical version of the protection scheme under test; the model checker then applies different sequences of read, write, and partial-failure events to exhaustively explore the state space of the system, either producing a chain of events that lead to data loss or a "proof" that the scheme works as desired. We apply the model checker on various real single-parity schemes used in enterprise systems and show that all of the schemes could lead to data loss under a single silent data corruption; we find that many of these systems suffer from a general problem that we call *parity pollution*, wherein corruption to a disk block on a data disk can spread to the parity disk, thereby rendering the data unrecoverable. In addition to analyzing existing schemes, we identify a protection scheme that can handle our entire set of

partial disk failures; this scheme uses several techniques including block checksums, both logical and physical identity information, and version mirroring.

We also show how a system designer can combine real data of failure probability (from our study of the characteristics of partial disk failures) with our model checker's results for a given scheme to arrive upon a final estimation of data-loss probability for that scheme. Doing so enables one to compare different protection approaches and determine which is best given the current environment.

1.2.2 Type-Aware Fault Injection

We developed type-aware fault injection in earlier work [104] to study the impact of certain types of partial disk failures on commodity file systems. Our approach is to inject faults just beneath the system under test and observe how the system responds. Many standard fault injectors [26, 123] that take this approach 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 failure policy 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*, which builds on our previous work with "semantically-smart" disk systems [16, 125, 126, 127]. 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 failure policy, allowing us to extract the different strategies that a system applies for its different data structures. In addition, we believe that different code paths in the system may not respond in the same manner even when the same type of disk block is failed. Therefore, we also use a suite of fine-grained workloads to test failure behavior for each type of disk block.

Previously, we have used type-aware fault injection to study how commodity file systems (ext3, JFS, ReiserFS, and NTFS) respond to block read and write errors and completely-corrupt disk

blocks [104]. The study found that file systems use illogically-inconsistent failure policies, and do not detect partial disk failures in various scenarios.

In this dissertation, we extend type-aware fault injection to work with virtual-memory systems and study how they respond to block read and write errors and completely-corrupt disk blocks. We also extend the technique to study the impact of specific corruptions to on-disk pointers of file systems.

1.2.3 Virtual-Memory Systems

A virtual-memory system is an integral part of most operating systems, and like file systems, is a significant user of disk storage. The virtual-memory system uses disk space to store memory pages that are not expected to be of immediate use, thereby freeing-up physical memory for other memory pages. When a page stored on disk is accessed again, it is brought back into physical memory. Thus, the virtual-memory system is responsible for handling disk errors that occur to these memory pages.

Since the virtual-memory system is an integral element of the storage stack, it is important to understand how a virtual-memory system responds to partial disk failures. To do so, we apply type-aware fault injection to study the virtual-memory systems of two operating systems, Linux 2.6.13 and FreeBSD 6.0, in detail. We also perform a preliminary study of the Windows XP virtual-memory system.

From our experiments, we find that these virtual-memory systems are not well-equipped to deal with partial disk failures. Like the file systems studied in prior work [104], the virtual-memory systems use policies that are illogically-inconsistent (*e.g.*, in FreeBSD, a read error for a user data page may result in a error report in one case, while it results in kernel panic in another). In addition, we find that the failure-handling routines in virtual-memory systems have bugs. In most cases, the failure-handling policy is simplistic, and in some cases, even absent. This disregard for partial disk failures leads to many problems, ranging from loss of physical memory abstraction, to further data corruption, and even to system-security violations.

1.2.4 File Systems

A file system is a crucial component of the storage stack; most applications use file systems to store data. In commodity systems, such as desktops and laptops, file systems are also tasked with the responsibility of ensuring that data is stored reliably. While we have analyzed how file systems respond to block read and write errors and completely-corrupt disk blocks in previous work [104], here we develop a thorough understanding of how file systems respond to more nuanced forms data corruption. In particular, we corrupt the on-disk pointers of file systems.

Although any block on disk may become corrupt, some corruptions are more damaging than others. If a data block of a file is corrupt, then only the application that reads the file is impacted. However, if a disk block belonging to file-system metadata is corrupt, then the entire file system can be affected; for example, a corrupt on-disk pointer incorrectly referring to data belonging to a different data structure can cause that data to be overwritten and corrupted as well. Therefore, an integral part of ensuring the long-term availability of data is ensuring the reliability and availability of pointers, the *access paths* to data.

File systems today use a variety of techniques to protect against corruption. ReiserFS, JFS and Windows NTFS perform lightweight checks to detect corruption like type checking [104]; that is, ensuring that the disk block being read contains the expected data type. In order to recover from corruption, most systems rely on replicated data structures. For example, JFS and NTFS replicate key data structures, giving them the potential to recover from corruption of these structures [20, 128].

We seek to evaluate how a set of corruption-handling techniques work in reality. To analyze the file systems, we develop *type-aware pointer corruption*, an extension of type-aware fault injection. Type-aware pointer corruption explores failure behavior by systematically changing the values of only one disk pointer of each type in the file system and observing its behavior. Further, it corrupts the pointers to refer to each type of data structure, instead of to random disk blocks. The technique is successful because different block types are used differently by the file system, thus causing the blocks and the pointers that point to them to be protected differently.

We apply type-aware pointer corruption on two widely-used file systems, Windows NTFS and Linux ext3. We examine their use of type checking, sanity checking, and replication to deal with corrupt pointers, and verify whether these techniques work well in practice. The study of NTFS is particularly interesting since it is a closed-source system for which little information is available about exact failure policies.

We find that both file systems fail to recover from many pointer corruptions despite the availability of redundant information. This failure to recover is due to poor use of techniques like type checking and replication.

1.3 Tolerating Partial Disk Failures with N-Version File Systems

This section describes our solution for tolerating partial disk failures in personal computer systems. This solution is influenced by the lessons learned from our study of partial disk failures and our analysis of file systems:

- Partial disk failures do occur; they affect a significant percentage of disk drives. These failures affect a higher percentage of inexpensive SATA disk drives that are used in our desktops and laptops. In fact, latent sector errors affect up to 20% of the drives of one of the SATA disk models in just 2 years.

- Commodity file systems (that use the SATA disk drives) are extremely poor at handling disk failures; they use inconsistent policies and contain bugs in failure-handling code. As a result, they fail to detect many instances of data corruption and do not leverage available replication to recover from corruption.

In addition to these lessons, recent research has shown that file systems themselves contain many bugs [148, 149, 150]. These bugs could potentially cause data loss or corruption. The bugs and poor use of failure-handling techniques exist despite the file systems being widely-used and potentially well-tested. Therefore, we believe that one cannot rely on a single file system to handle all partial disk failures, including file-system bugs.

Our solution to the problem of partial disk failures is an *N-version file system*. An N-version file system is an instance of N-version software [6]. In an N-version file system, data is stored in N *different* file systems. All file operations performed by the user are received by a simple software layer that then performs the operation on all the file systems and delivers the majority result to the user. Thus, we eliminate the reliance on a single complex file system, and place it on a simpler software layer. We design the N-version file system with simplicity as one of its important goals, and from our experience in building it, we find that it can be kept simple.

One major issue in building an N-version-software system is the high development costs associated with formulating a common specification for the system, and creating N different versions of the system. In order to reduce these costs, we hypothesize that for an N-version file system, (i) we can use an existing specification, such as POSIX, as the common specification, and (ii) we can use existing file systems, such as ext3, JFS, etc., as the N different file-system versions. In building an N-version file system using an unmodified specification and existing file systems, we verify these hypotheses.

A second issue in using an N-version file system is the high performance and disk-space overheads introduced by storing and retrieving data from N file systems instead of one. Our solution to this issue is to use a block-level single-instance store underneath the file systems. A block-level single-instance store uses content hashing to identify disk blocks with the same content; it then stores a single copy of these blocks on disk. In an N-version file system, user data stored in the different file systems will have the same content and will therefore be coalesced into a single block, while file-system metadata of different file systems will have different contents and will not be coalesced. Therefore, a single-instance store protects against partial disk failures that affect metadata (thereby protecting the important access paths to data), but not against failures that affect data blocks. A single-instance store is especially useful in cases where file-system bugs are the main contributors to partial disk failures.

Our 3-version file system uses ext3, JFS, and ReiserFS as *child file systems* to store data. We evaluate its reliability against that of the individual child file systems using fault injection, and find

that the 3-version file system successfully recovers from almost all scenarios where a child file system has incorrect contents or is affected by a partial disk failure.

1.4 Overview

The rest of this dissertation is organized as follows.

Background: Chapter 2 provides a background on the storage stack and disk drives, disk and storage-stack failures, a taxonomy of failure-handling techniques used within a single disk, and the type-aware fault-injection technique we have developed to analyze systems.

Characteristics: Chapter 3 presents our study of the characteristics of two important types of partial disk failures, latent sector errors and silent data corruptions; we analyze the impact of factors such as disk age, properties of errors such as spatial locality, and the efficacy of different methods used to detect partial disk failures.

Impact: Chapters 4, 5, and 6 discuss our analyses of the impact of partial disk failures on RAID systems, virtual-memory systems, and file systems; Chapter 4 presents our model-checking-based analysis of the effectiveness of schemes used in enterprise RAID systems to detect and recover from partial disk failures; Chapter 5 details our analysis of the failure policies of virtual-memory systems using type-aware fault injection; Chapter 6 discusses the impact of corrupt on-disk pointers on file systems.

Tolerance: Chapter 7 presents our design and evaluation of N-version file systems, our solution for tolerating all partial disk failures, including file-system bugs.

Related Work: Chapter 8 summarizes research efforts focusing on characterization of system and storage failures, techniques used to analyze the failure behavior of systems, techniques used to handle disk failures, and N-version programming.

Conclusions and Future Work: Chapter 9 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.

Chapter 2

Background

This chapter provides a background on various aspects integral to this dissertation. First, we provide a brief overview of the storage stack in a computer system, focusing on the lowest component of the stack, disk drives (Section 2.1). Second, we discuss failures that occur in the storage stack and describe specific partial disk failures that are addressed in this dissertation (Section 2.2). Third, we discuss RAID [101], a technique used in nearly every enterprise storage system to handle disk failures (Section 2.3). This discussion serves as an overview; Chapter 4 is a more detailed examination of enterprise RAID data protection. Fourth, we present the IRON taxonomy [104] of failure-handling policies (Section 2.4). This discussion also serves as an overview of the different techniques that may be used within a single disk to handle partial disk failures. Last, we present a fault-injection technique called *type-aware fault injection* [104]. We use the IRON taxonomy and type-aware fault injection to analyze the failure behavior of both file systems and virtual-memory systems in later chapters.

2.1 Storage Stack

A storage stack is an integral part of most computer systems. The role of the storage stack is to provide a means to store data. As in a communication protocol stack, the different layers of the stack use the abstraction provided by the layer below to build the abstraction and services that it provides to layers above. Figure 2.1 shows the storage stack in a typical computer system. It consists of hardware, software, as well as firmware components.

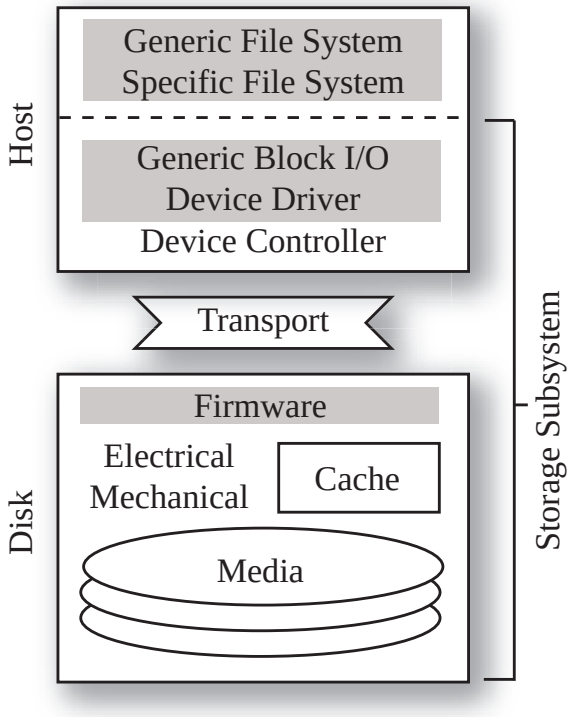

Figure 2.1 **The storage stack.** *We present a schematic of the entire storage stack. At the top is the file system; beneath are the many layers of the storage subsystem. Gray shading implies software or firmware, whereas white (unshaded) is hardware.*

At the bottom of the storage stack is the disk drive. 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.*, Fibre-Channel). At the top of the stack is the host, in which 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). On top of the generic block-I/O layer 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 [82]) is positioned between the two.

2.1.1 Disk Drives

Hard disk drives serve as the primary storage medium both in enterprise environments and in personal computers. As discussed above, they are at the lowest level of the storage stack.

Disk drives are complex entities; as shown in Figure 2.1, they contain media, mechanical, electrical, memory, and firmware components. In a hard disk, data is recorded on *platters* coated with a ferromagnetic material by magnetizing the material in a specific direction, and data is read by detecting the direction in which the material is magnetized. Disks typically have multiple platters, where each platter usually has two surfaces, each accessed by a dedicated read/write head. A single surface is divided into tens of thousands of concentric circular tracks and each track is subdivided into *sectors*, the smallest addressable unit of data access, usually 512 bytes in size. Each sector is protected by error correcting codes (ECC). The number of sectors on each track varies depending on whether the track is close to the center of the disk or farther away, with tracks farther away containing more sectors.

Beyond the magnetic medium, there are mechanical (*e.g.*, the motor and arm assembly) and electrical components (*e.g.*, buses) that read and write the data. 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 Seagate drive circa 2005 contains roughly 400,000 lines of code [38]).

Disks typically use one of two different interfaces to communicate with the host: SCSI [135] or ATA [136]. SCSI and ATA disks are different in nearly every component of the disk – mechanics, materials, electronics, and firmware [5]; SCSI disks are built towards achieving better performance and reliability characteristics, and are therefore more expensive. While there is no inherent interface-related requirement that SCSI disks and ATA disks be built differently, they are built differently since they address different market segments. Over the years, SCSI disks have typically been used in enterprise systems involving mission-critical or business-critical applications, while ATA disks have typically been used in personal computers. More recently, ATA disks are being increasingly used in various enterprise systems as well (*e.g.*, archival, or backup storage systems) [12, 48]. In this dissertation, we also refer to ATA disks as *nearline* disks and SCSI disks as *enterprise* disks. Note that while the logical interface (command set) is SCSI in an enterprise disk, the physical interface (connector/transport) may be different (*e.g.*, Fibre-Channel).

The disk interface abstracts the disk as a linear array of equal sized blocks each identified by a logical block number (LBN). Internally, the disk reserves a small portion of sectors called *spares*, which are not initially mapped to a particular LBN. The disk firmware can map a spare sector to the LBNs of failed sectors. Today's disk drives allocate a few thousand spare sectors for such *re-mapping*.

2.2 Disk Failures

This section provides a background on disk failures, with a focus on partial disk failures. We first discuss the different sources of failures in the storage stack and then describe specific types of partial disk failures. 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.

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 [119]. 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 [5]. 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.

Electrical: A power spike or surge can damage in-drive circuits and hence lead to drive failure [138]. Thus, electrical problems can lead to entire disk failure.

Drive firmware: Interesting errors arise in the drive controller, which consists of many thousands of lines of real-time, concurrent firmware. For example, disks have been known to return correct data but circularly shifted by a byte [85] or have memory leaks that lead to intermittent failures [138]. One of the disk drive models in our study of partial disk failures [13] had a bug specific to flushing the disk's write cache. Upon reception of a cache flush command, the disk drive sometimes returned success without committing the data to

the disk medium. If, for any reason, the disk was then power-cycled the data just written was lost. This type of silent data corruption is called a "lost write" [131]. In summary, drive firmware bugs often lead to sticky or transient block corruption.

Transport: The transport connecting the drive and host can also be problematic. For example, a study of a large disk farm [137] 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 buggy. 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 silent data corruption [145]. 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 [54]. Others have also observed IDE protocol version problems that yield corrupt data [48]. In summary, controller problems can lead to transient block failure and silent 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 [34, 45, 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 silent 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 [104, 148, 149, 150].

2.2.2 Types of Partial Disk Failures

We now describe specific types of partial disk failures related to this dissertation. Many of the problems described in the previous subsection result in one or more of the following partial disk failures.

Latent sector error: This error occurs when the disk drive cannot read or write a particular disk sector or when the disk encounters an uncorrectable ECC error. Any data previously stored in the sector is usually lost. Causes of latent sector errors include: (a) medium imperfections, (b) loose particles causing medium scratches, (c) "high-fly" writes leading to incorrect bit patterns on the medium, (d) rotational vibration, (e) read/write head hitting a bump or medium, and (f) off-track reads or writes. Latent sector errors are detected and reported by the disk drive. They are called latent because this detection and report occurs only when the sector is accessed by the system and the error is hidden until such time.

Before reporting latent sector errors, disks typically perform error correction with multiple retries of a given operation. Additionally, after a (configurable) number of unsuccessful retries, disk drives can automatically *re-map* failed writes to spare sectors. Sparing and re-mapping can only occur on detected write errors; read errors require higher-level mechanisms such as RAID reconstruction to obtain the lost data.

Not-ready-condition error: These errors are reported by the disk drive when the drive is not ready to handle a command from the host. This error could also indicate that the disk itself is not accessible. These errors are often resolved by systems by waiting and retrying the disk operation. These errors do not lead to permanent data loss unless the disk has experienced a complete failure.

Recovered error: These "errors" are warnings issued by the disk drive. They occur in scenarios very similar to that of latent sector errors. The only (but crucial) difference is that in this case, disk-level retries and error correction successfully recover data from a sector (although the operation failed the first time it was tried).

Silent data corruption: This partial disk failure is a situation where the data in a disk block is incorrect. The corruption could be caused by any element of the storage stack. The causes are typically bugs in either software or firmware components. The main issue with silent data corruption is that it is not reported by the disk drive or any other hardware component. Various forms of silent data corruption could occur. The remaining partial disk failures below are different forms of silent data corruption.

Bit corruption: These corruptions are scenarios in which the bits stored in a disk block get corrupted, or the bits are modified by some element of the storage stack during a write operation.

Torn write: These corruptions are scenarios in which the disk drive ends up writing only a portion of the original disk request. Often, this occurs when the drive is power-cycled in the middle of processing the write request. Therefore, future reads return some sectors with the new data and some with the old data.

Lost write: These corruptions occur when buggy firmware components return a success code to indicate completion of a write, but do not perform the write to the disk medium in reality.

Misdirected write: These corruptions occur when buggy firmware writes the data in a write request to the wrong disk or the wrong location within a disk. The effect of this error is two-fold: the original disk location does not receive the write it is supposed to receive (lost write), while the data in a different location is overwritten (with effects similar to bit corruption, torn write, or lost write depending on how the disk block is used).

2.3 RAID

RAID stands for “Redundant Array of Independent Disks”. It is the general name for techniques that store data on multiple disks in order to survive the failure of one or more of the disks [101]. Since it was originally proposed, it has been employed in nearly every enterprise storage system. RAID has also been implemented using both hardware and software; various off-the-shelf hardware RAID cards [1] and software implementations [47] are available; even in the

case of enterprise systems, some have adopted a hardware-based approach [37], while some use a software-based approach [94]. In the storage stack (Figure 2.1), the RAID layer can be inserted just below the generic block I/O layer if it is implemented in software or as the device controller itself if it is implemented in hardware.

Many different *levels* of RAID have been proposed [101]; the levels differ in the kind of redundant data that is stored, and hence in the number of disk failures that can be tolerated, the extra space used, and the performance overheads. One popular RAID level is RAID-1 or "mirroring," wherein each data block is stored on two different disk drives, so that the failure of one of the disks does not lead to any data loss. Other popular RAID levels include RAID-4 and RAID-5, wherein parity information is calculated for corresponding blocks on a set of disks and is stored on another disk; in the case of RAID-4, the parity block is always stored on a separate parity disk, while in RAID-5, each disk performs the role of the parity disk for a portion of the disk blocks. The set of corresponding blocks for which parity is calculated is referred to as a *stripe*.

Over the years, various kinds of enhancements have been developed for RAID. One example is the AutoRAID system [146], where different data blocks in the storage system are automatically stored at different RAID levels, depending on their usage. Also, while the original RAID levels (up to 5) guaranteed protection against a single disk failure, numerous schemes have been developed to tolerate double disk failures [3, 22, 35, 57, 59, 98]

The primary role of RAID is to tolerate complete disk failures, while also offering protection against errors such as latent sector errors that are reported by the disk drive. More recent RAID systems also offer varying degrees of protection against data corruption [24, 37, 65, 131]. In some of these cases, extra protection is possible due to the close interaction of the file system and RAID layers [24, 130, 131]. We identify some implications of partial disk failures for RAID-system design in discussing the characteristics of failures in Chapter 3 and then perform a detailed examination of the exact data protection offered by various RAID systems in Chapter 4.

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	Detects corruption in end-to-end way

Table 2.1 **IRON detection taxonomy.** *The table describes the different levels of detection in the IRON taxonomy.*

2.4 IRON Taxonomy

We now describe an extended version of the IRON taxonomy of failure-handling strategies that we developed in previous work [104]. 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.

To cope with partial disk failures, storage-stack components may include machinery to *detect* (Level D) these failures, *react* (Level R) to them, and also *prevent* them (Level P). Tables 2.1, 2.2, and 2.3 present our IRON detection, reaction, and prevention taxonomies, respectively. We have found from experience that this taxonomy can be used to sufficiently describe the failure-handling strategies of various file systems and virtual-memory systems. However, 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 [4, 146]; indeed, we have extended the taxonomy to include a prevention axis since it was originally proposed.

2.4.1 Detection Levels

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

Level	Technique	Comment
R_{Zero}	No recovery	Assumes disk works
R_{Report}	Report error	Informs user
R_{Record}	Record that operation did not succeed	Stops/pauses dependent actions
R_{Stop}	Stop activity (crash, prevent writes)	Limit amount of damage
R_{Guess}	Return "guess" at block contents	Could be wrong; failure hidden
R_{Retry}	Retry read or write	Handles failures that are transient
R_{Repair}	Repair data structs	Could lose data
R_{Remap}	Remaps block or file to different locale	Assumes disk informs system of failure
$R_{Redundancy}$	Block replication or other forms	Enables recovery from loss/corruption

Table 2.2 **IRON reaction taxonomy.** *The table describes the different levels of reaction in the IRON taxonomy.*

Level	Technique	Comment
P_{Zero}	No prevention	Assumes disk works
$P_{Remember}$	Remembers disk errors	Prevents usage of blocks with errors
P_{Reboot}	Periodically re-initializes the system	Tries to avoid bugs due to excess state
$P_{LoadBalance}$	Balances the read/write load on blocks	Attempts to reduce "wear" on blocks
P_{Scan}	Performs read/write checks	Detects possibly "sticky" block errors

Table 2.3 **IRON prevention taxonomy.** *The table describes the different levels of prevention in the IRON taxonomy.*

$Zero$**:** 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 the analysis of various systems, this approach is surprisingly common (although often it is applied unintentionally).

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

$Sanity$**:** The system can verify that the data structures stored on disk are consistent. This check can be performed either within a single block or across blocks. Two kinds of checks that can be performed are type checks and sanity checks. Type checking verifies that a disk block contains a specific type of data structure (such as an inode). Typically, type information for a disk block is encoded in the form of a "magic" number and stored in the disk block. Sanity checking verifies that certain values in data structures follow constraints. For example, a pointer value in the data structure can be compared with well-known values, such as locations of metadata structures like the boot sector or the size of the disk partition, to ensure that the pointer is not corrupt.

$Redundancy$**:** The final level of the detection taxonomy is redundancy. Many forms of redundancy can be used to detect block corruption. For example, as discussed in the previous subsection, *checksumming* has been used in reliable systems for years to detect corruption [18] and has recently been applied to improve security as well [99, 129]. Checksums are particularly well-suited for detecting corruption due to firmware components (*e.g.*, a buggy controller that misdirects a disk write 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, 130]; specifically, a checksum that is stored along with the data it checksums will not detect such misdirected or lost writes. We discuss such issues in greater detail in analyzing the protection offered by enterprise RAID systems (Chapter 4).

2.4.2 Reaction Levels

Level R of the IRON taxonomy describes techniques used in reacting to block failure within a single disk drive. These techniques handle both latent sector errors and block corruptions.

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

$Report$**:** A straightforward reaction strategy is to report errors up through the system; for example, the file system informs the application that an error occurred and assumes the client program or user will respond appropriately to the problem.

$Record$**:** At this level, the system records that the I/O operation did not succeed. This level prevents the system from performing any action that assumes successful completion of the I/O operation. For example, when a write error is detected and the system records the error, it does not free the "dirty" memory page assuming that it has been successfully written out to disk, thus avoiding data loss.

$Stop$**:** One way to react to a disk failure is to stop current 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 mechanism turns all *detected* disk failures into fail-stop failures and likely preserves the integrity of on-disk data structures. 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. In the case of a file system, one can choose a less drastic approach and 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 unrelated processes to continue.

$Guess$**:** As recently suggested by Rinard *et al.* [110], another possible reaction to a failed block read would be to manufacture a response, perhaps allowing the system to keep running in spite of a failure. The negative is that an artificial response may be less desirable than failing.

Retry: A simple response to failure is to retry the failed operation. Retry can appropriately handle transient errors, but wastes time retrying if the failure is indeed permanent.

Repair: If a system can detect an inconsistency in its internal data structures, it can likely repair them. For example, in a file system, a block that is not pointed to, but is marked as allocated in a bitmap, could be freed.

Remap: Similar to sector remapping performed by disk drives, systems can perform block remapping. This technique can be used to fix errors that occur when writing a block, but cannot recover from failed reads. Specifically, when a write to a given block fails, the 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 [101], by adding a parity block per block group, a system can tolerate the unavailability or corruption of one block in each such group.

2.4.3 Prevention Levels

Level P techniques can be used to reduce the probability of occurrence of partial disk failures or of encountering them. This prevention axis is a new addition to the IRON taxonomy; it was not a part of the taxonomy when it was originally proposed [104].

Zero: In the simplest case, the system does not use any special prevention techniques; the system assumes either that the disk works or that errors can be dealt with when they occur.

Remember: A basic prevention strategy that can be used is to remember that a specific block is "bad" once the system has had at least one bad experience in using the block. This strategy could prevent future data loss.

Reboot: A phenomenon that has been observed for a long time is that systems are either less likely to fail or faults are cured if the systems are rebooted or reinitialized [28] (since the systems can be rid of effects of transient bugs accumulated over time). This fact can be used as a failure prevention strategy by periodically rebooting subsystems [69]. For example, the rebooting strategy for a file system could range from unmounting and re-mounting the file system periodically to even re-initializing the drivers and disk controllers.

LoadBalance: This prevention technique attempts to reduce the wear on disk blocks by balancing the load on them. An example of this technique is the use of wear-leveling in file systems for flash drives (like JFFS2 [106, 147]).

Scan: The final prevention technique is scanning the disk for bad blocks by performing accesses, perhaps with bogus data. This technique is used in RAID systems to weed out potential bad blocks – the process is called "disk scrubbing" [77, 119]. Systems can scan the disk periodically during disk idle time or by using freeblock scheduling [86] and avoid using disk blocks found to be "bad" in the scan.

2.5 Analysis of Failure Behavior

In this section, we first describe a fault-injection technique that we developed in previous work [104] to uncover the disk-failure-handling policy of systems, and then present an overview of the results we obtained when we applied our technique on commodity file systems.

2.5.1 Type-Aware Fault Injection

The primary objective of our fault-injection technique is to determine which IRON detection and reaction techniques each system uses and the assumptions each makes about how the underlying storage system can fail. By comparing the failure policies across systems, we can learn not only which systems are the most robust to partial disk failures, but also suggest improvements for each. We have used this methodology in studying the behavior of both virtual-memory systems

(Chapter 5) and file systems ([104] and Chapter 6). This section provides a brief outline of the methodology; details specific to each study are described in the corresponding chapters.

Our approach is to inject faults just beneath the system under test and observe how the system responds. If the fault policy is 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 failure policy 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.

Therefore, to extract the failure policy of a system, we must trigger all 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 all relevant code paths in combination with induced faults on all data structures.

Type Awareness: Many standard fault injectors [26, 123] 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 failure policy 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*, which builds on our previous work with "semantically-smart" disk systems [16, 125, 126, 127]. 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 in reverse-engineering failure policy, allowing us to discern the different strategies that a 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 believe 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.

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 [26], 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 [109]. However, given that we are characterizing how a specific 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.

2.5.2 Analysis of File Systems

We have performed a failure-policy analysis for four commodity file systems: ext3 [141], ReiserFS (version 3) [108], and IBM's JFS [20] on Linux and NTFS [128] on Windows XP; we have analyzed the impact of read errors, write errors, and corruption of entire disk blocks in these file systems. In this subsection, we first present the results we found for JFS, then summarize the findings of the entire study [104].

Figures 2.2 and 2.3 present the detection and reaction techniques used by JFS to handle read, write, and corruption 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 corresponds to how JFS 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. Table 2.4 summarizes our observations from the experiments.

We now summarize the observed response to partial disk failures for all of the file systems in the study.

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

JFS: The kitchen sink. JFS is the least consistent and most diverse in its failure detection and 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. We draw more detailed conclusions about NTFS behavior in analyzing its response to corrupt pointers in Chapter 6.

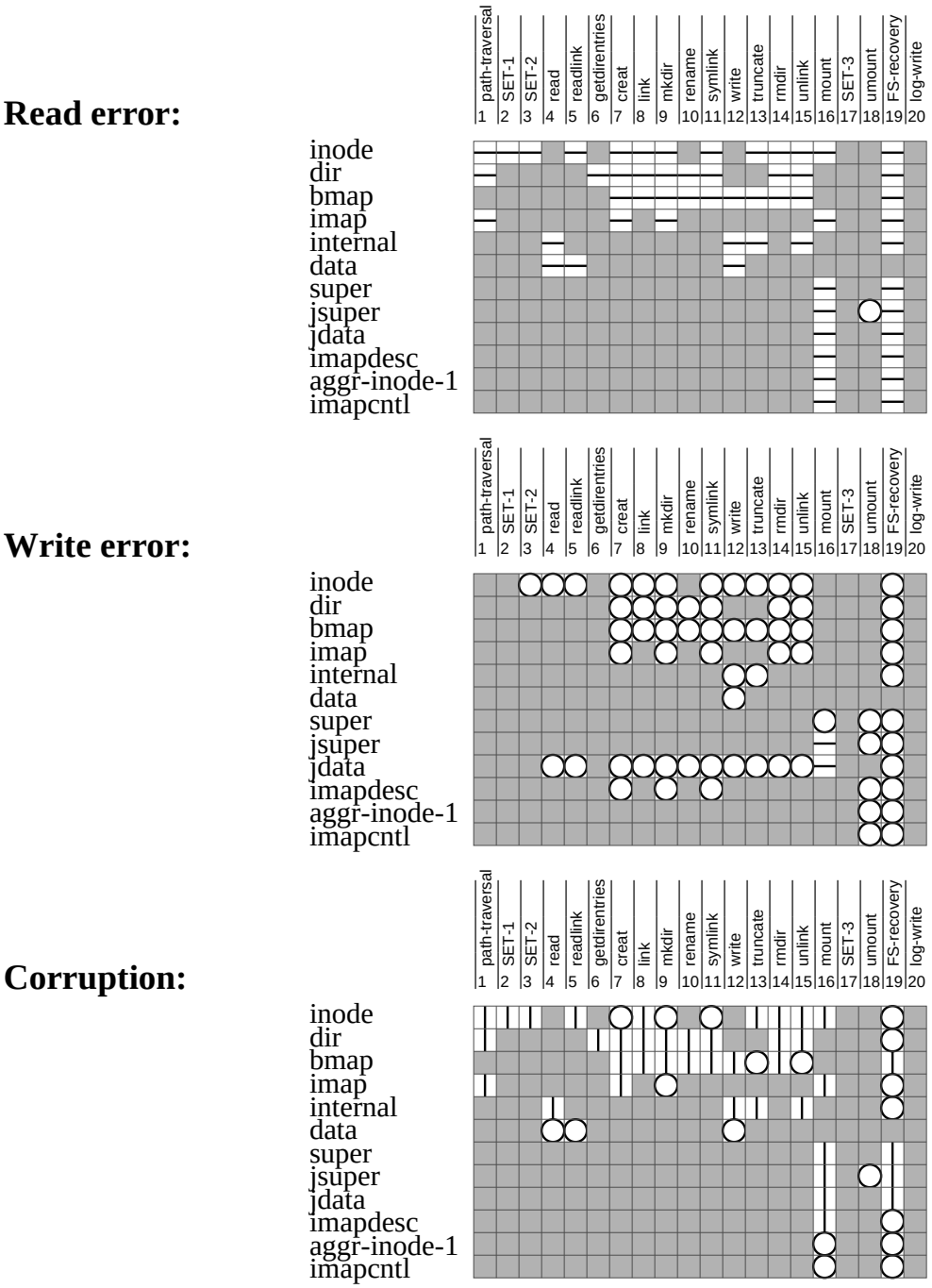

Figure 2.2 **JFS detection policies.** *The tables indicate the detection policies of JFS for read, write, and corruption faults injected for each block type across a range of workloads. Each row corresponds to a block type and each column corresponds to a file operation. The symbols are* [◯] *for* D_{Zero}, [−] *for* $D_{ErrorCode}$, *and* [|] *for* D_{Sanity}. *A gray box indicates that the workload is not applicable for the block type. If multiple policies are observed, the symbols are superimposed.*

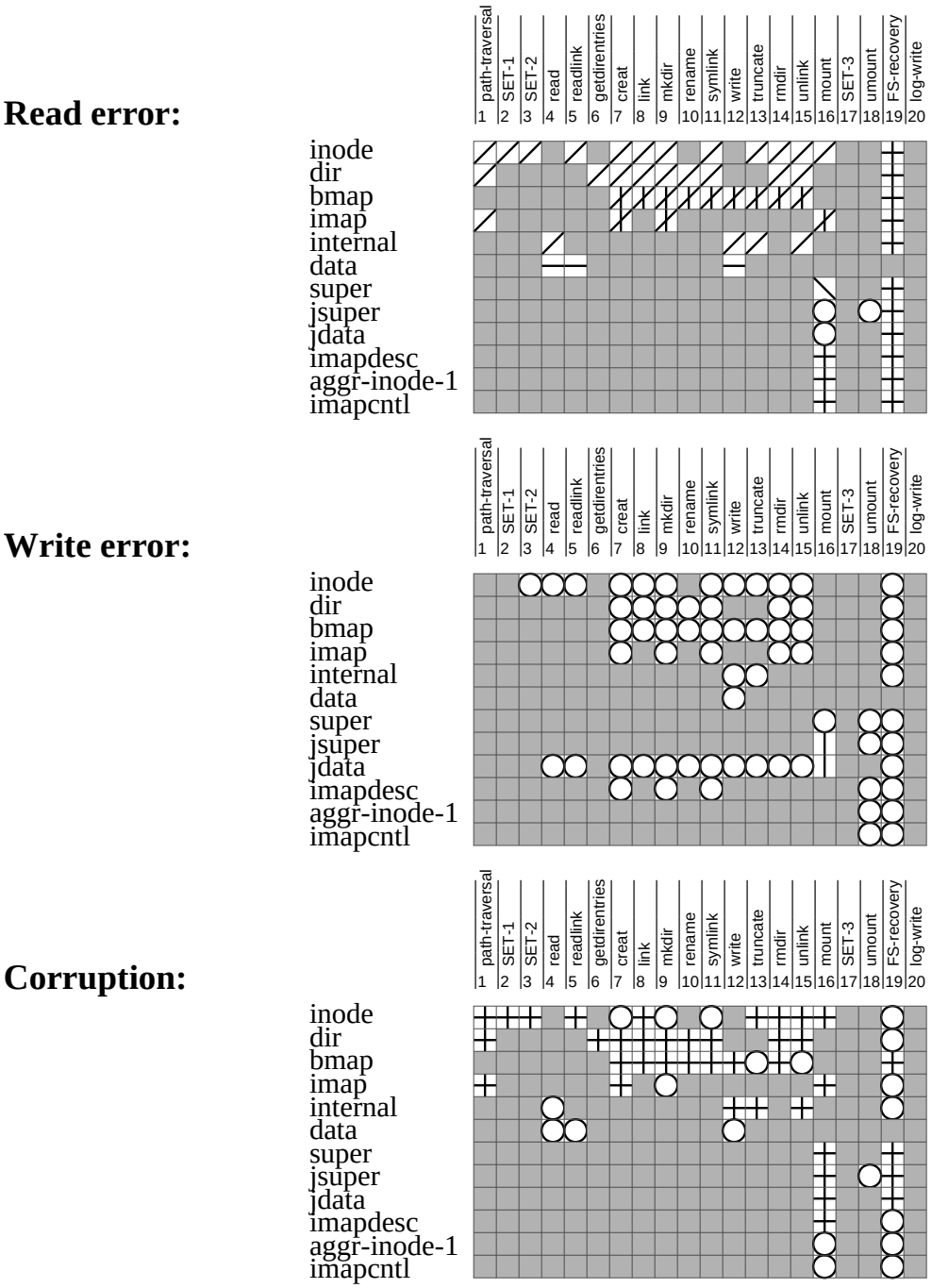

Figure 2.3 **JFS reaction policies.** *The tables indicate the reaction policies of JFS for read, write, and corruption faults injected for each block type across a range of workloads. Each row corresponds to a block type and each column corresponds to a file operation. The symbols are* [◯] *for* R_{Zero}, [/] *for* R_{Retry}, [−] *for* R_{Report}, [\] *for* $R_{Redundancy}$, *and* [|] *for* R_{Stop}. *A gray box indicates that the workload is not applicable for the block type. If multiple policies are observed, the symbols are superimposed.*

Detection

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

Reaction

The reaction strategies of JFS vary dramatically depending on the block type. For example, when an error occurs during a journal superblock write, JFS crashes the system (R_{Stop}); however, other write errors are ignored entirely (R_{Zero}). On a block read failure to the primary superblock, JFS accesses the alternate copy ($R_{Redundancy}$) to complete the mount operation; however, a corrupt primary results in a mount failure (R_{Stop}). Explicit crashes (R_{Stop}) are used when a `BMAP` or `IMAP` read fails. Error codes for all metadata reads are handled by generic file-system code called by JFS; this generic code attempts to recover from read errors by retrying the read a single time (R_{Retry}). Finally, the reaction for a failed sanity check is to report the error (R_{Report}) and remount the file system as read-only (R_{Stop}); during journal replay, a sanity-check failure causes the replay to abort (R_{Stop}).

Bugs and Inconsistencies

We found various problems with the JFS failure policy. First, while JFS has some built-in redundancy, it does not always use it as one would expect; for example, JFS does not use its secondary copies of aggregate inode tables (special inodes used to describe the file system) when an error code is returned for an aggregate inode read. Second, a blank page is sometimes returned to the user (R_{Guess}), although we believe this is not by design (*i.e.*, it is a bug); for example, this occurs when a read to an internal tree block does not pass its sanity check. Third, some bugs limit the utility of JFS reaction mechanisms. For example, although generic file-system code detects read errors and retries, a bug in the JFS implementation leads to ignoring the error; in most of these cases JFS subsequently detects a problem through sanity checks and reports an error (the figure shows only the $D_{ErrorCode}$ and R_{Retry} since the rest of the response occurred due to a bug); in some cases, the bug leads to JFS corrupting the file system.

Table 2.4 **JFS behavior details.** *This table describes the various detection and reaction policies used by JFS and also points out inconsistencies and bugs in the JFS failure handling.*

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.

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

Detection and Reaction: 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 file-system 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 [44]; 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 [126]. What we are criticizing are inconsistencies that are undesirable (and likely unintentional); for example, JFS will attempt to read the alternate superblock if a read failure occurs when reading the primary superblock, but it does not attempt to read the alternate if it deems the primary corrupted.

Detection and Reaction: 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 [150]. 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” [100]. 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.

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

Detection and Reaction: Redundancy is not used. While virtually all file systems include some machinery to detect disk failures, none of them apply *redundancy* to enable recovery from such failures. The lone exception is the minimal amount of superblock redundancy found in JFS; even this redundancy is used inconsistently. Also, JFS places the copies in close proximity, making them vulnerable to spatially-local errors.

These observations can help improve failure handling in specific file systems, and can also influence the development of other techniques to tolerate partial disk failures. Indeed, these observations influence the fault-tolerance solution we have developed (Chapter 7). Specifically, since we find that (i) no single file system is capable of handling partial disk failures, (ii) file systems also contain bugs, and (iii) different file systems handle partial disk failures differently, our solution uses multiple different file systems to store data reliably.

Chapter 3

Characteristics of Partial Disk Failures

Detailed knowledge of real-world failure characteristics is essential for building systems that can tolerate failures effectively and for examining whether existing systems meet fault-tolerance goals. To date, there has been little field data available about partial disk failures, other than from small-scale studies [53]. This situation persists despite the recent influx of data on absolute disk failures [41, 102, 118, 121, 122]. Therefore, current techniques to handle partial disk failures have to be built based on anecdotal information and back-of-the-envelope calculations [119].

In this chapter, we present the first large-scale study of partial disk failures. We focus on two important types of partial disk failures: *latent sector errors* and *silent data corruptions*. These two types of failures are important since both of them could lead to data loss.

We analyze data from more than 50,000 production and development storage systems developed by NetApp™ and installed at many of their customer sites (*e.g.*, different FAS series and NearStore systems [95]). The data pertains to partial disk failures affecting the 1.53 million disks used by these storage systems. This set of disks is diverse; they were sourced from multiple vendors; there are both nearline (SATA) and enterprise (Fibre-Channel) disks; within each of the two classes, there are different disk families; within each family, there are different capacities. This diversity helps make the results of our study more generally applicable.

Our study is possible because NetApp™ storage systems use various techniques to detect and recover from various partial disk failures. The partial disk failures thus detected are reported to a central repository called the *NetApp Autosupport Database*. This repository also stores details about each disk drive and storage system. We analyze the failures reported to this repository,

starting from January 2004 for a period of 32 months for the study of latent sector errors, and for a period of 41 months for the study of silent data corruptions.

Through our study, we answer several important questions about partial disk failures, including:

- What is the magnitude of this threat? What are the average and worst-case percentages of disks affected by partial disk failures in given period of time?
- What factors impact the development of partial disk failures? For instance, do older disks have a higher probability of being affected?
- Are partial disk failures independent events? Some file systems use intra-disk redundancy to overcome partial disk failures [20, 88]; should these systems be concerned about spatial locality of partial disk failures?
- What techniques are useful for detecting partial disk failures? Is "disk scrubbing," a periodic scan of all disk blocks, useful for detecting partial disk failures?

The rest of the chapter is organized as follows. Section 3.1 describes the overall architecture of the storage systems from which the data was collected. Section 3.2 describes our data collection and analysis methodology and also outlines some limitations of the study. Section 3.3 presents the analysis of latent sector errors. Section 3.4 presents the analysis of silent data corruptions. Section 3.5 discusses lessons that we learn from the data for building systems that can tolerate partial disk failures. Section 3.6 concludes the chapter.

3.1 Storage-System Architecture

In this section, we describe the overall architecture of the storage systems used in the study, focusing on failure-handling mechanisms, and error-logging infrastructure.

3.1.1 Storage Stack

Physically, the storage system is composed of a storage controller that contains the CPU, memory, network interfaces, and storage adapters. The storage controller is connected to a set of disk

shelves via two independent Fibre-Channel loops. The disk shelves house individual disk drives. A system consists of at least 14 disks and can have as many as several hundred disks. These disks may either be enterprise (Fibre-Channel) disk drives or nearline (SATA) disks. Nearline drives use hardware adapters to convert the SATA interface to the Fibre-Channel protocol. Thus, the storage controller views all drives as being Fibre-Channel (however, for the purposes of the study, we can still identify whether a drive is nearline or enterprise using its model type). Such an architecture aids in using the same software stack for different hardware components.

The software stack of the storage system is called Data ONTAP™ [94]. Its back-end is primarily composed of three layers: the WAFL™ file system [67], the RAID layer, and the storage layer. The file-system layer processes client requests by issuing read and write operations to the RAID layer. The RAID layer transforms file-system requests into logical-block requests and issues them to the storage layer. The RAID layer also generates parity for writes and reconstructs data after failures. The storage layer includes a set of customized device drivers. This layer communicates with physical disks using the SCSI command set [135].

3.1.2 Failure-Handling Mechanisms

The system, like other commercial storage systems, is designed to handle a wide range of disk-related failures including latent sector errors, recovered errors, not-ready-condition errors, transport problems, and various forms of silent data corruption. This failure-handling helps avoid potential data loss or data unavailability due to these failures. This subsection describes the handling of latent sector errors and silent data corruptions, the proactive detection of these failures, and disk replacement decisions.

3.1.2.1 Latent Sector Errors

Latent sector errors are detected by the storage layer when the disk drive returns a *Check condition* with the sense code set to *Medium error*. As the name suggests, a latent sector error is reported at the granularity of a single disk sector (512 or 520 bytes, depending whether the disk is nearline- or enterprise-class). Error handling for latent sector errors depends on the type of disk

request and the type of disk. For enterprise disks, the storage layer re-maps the bad sector to a spare sector. If the request is a write, the storage layer re-issues the write to the re-mapped sector. If the request is a verify or a read, the RAID layer reconstructs the sector from the other disk drives and passes it to the storage layer for rewrite. For nearline disks, sector re-mapping on failed writes is automatically performed by the disk and not reported to the storage layer. The system handles read and verify errors in the same fashion for both nearline and enterprise drives.

3.1.2.2 Data Corruption

The system uses various data-integrity checks to detect corruption. The system writes a 64-byte *data-integrity segment* along with each disk block. Figure 3.1 shows two techniques for storing this extra information, and also describes its structure. For enterprise disks, the system uses 520-byte sectors. Thus, a 4-KB file-system block is stored along with 64 bytes of data-integrity segment in eight 520-byte sectors. For nearline disks, the system uses the default 512-byte sectors and stores the data-integrity segment for each set of eight sectors in the following sector. Any corruption that is detected is therefore at the granularity of a file-system block (*i.e.*, 4 KB).

One component of the data-integrity segment is a checksum of the entire 4-KB file-system block. The checksum is validated by the RAID layer whenever the data is read. Once a corruption has been detected, the original block can usually be restored through RAID reconstruction. We refer to corruptions detected by RAID-level checksum validation as *checksum mismatches*. A checksum mismatch could result from the following silent data corruptions: (i) bit corruption, (ii) a torn write, or (iii) a misdirected write.

A second component of the data-integrity segment is block-identity information. In this case, the fact that the file system is part of the storage system is utilized. The identity is the disk block's identity within the file system (*e.g.*, this block belongs to inode 5 at offset 100). This identity is cross-checked at file-read time to ensure that the block being read belongs to the file being accessed. If, on file read, the identity does not match, the data is reconstructed from parity. We refer to corruptions that are not detected by checksums, but detected through file-system identity

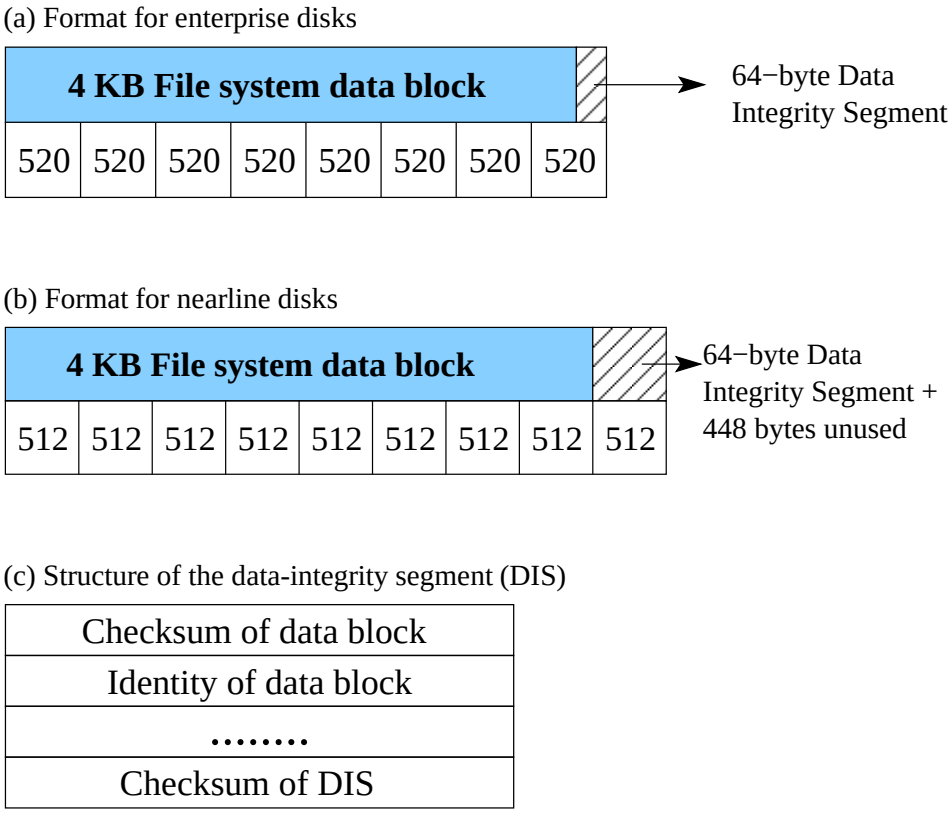

Figure 3.1 **Data-Integrity Segment.** *The figure shows the different on-disk formats used to store the data-integrity segment of a disk block on (a) enterprise drives with 520-byte sectors, and on (b) nearline drives with 512-byte sectors. The figure also shows (c) the structure of the data-integrity segment. In particular, in addition to the checksum and identity information, this structure also contains a checksum of itself.*

validation as *identity discrepancies*. An identity discrepancy could result from a lost write, or a misdirected write. Note that identity discrepancies can be detected only during file-system reads.

3.1.2.3 Proactive Failure Detection

The storage system periodically *scrubs* all disks as a proactive measure to detect latent sector errors and corruption. Two types of scrubs are performed – media scrubs and data scrubs.

Media scrubs use a SCSI VERIFY command to validate a disk sector's integrity. This command performs an ECC check of the sector's content from within the disk without transferring data to the storage layer. On failure, the command returns a latent sector error. The storage layer performs media scrubs continuously in the background, with the rate of scrub adjusted so as not to impact foreground performance. In most cases, media scrubs complete within two weeks.

Data scrubs are primarily used to detect data corruption. This scrub issues read operations for each disk block, computes a checksum over its data, compares the checksum to the checksum located in the data-integrity segment, and reconstructs the disk block from other disk blocks in the RAID stripe if the checksum comparison fails. If no reconstruction is necessary, the parity of the data blocks in the RAID stripe is generated and compared with the parity stored in the parity block. In a RAID system with single parity, if the parity does not match the verified data, the scrub process fixes the parity by regenerating it from the data blocks. In a system protected by double parity [35], it is possible to tell which of the parity or data block is corrupt, thus initiating reconstruction for the corrupt one. We refer to all of these cases of mismatch between data and parity as *parity inconsistencies*. A parity inconsistency could result from a lost write, a misdirected write, a processor miscalculation, or a software bug. Note that data scrubs are unable to validate file system identity information stored in the data-integrity segment since, by its nature, this information only has meaning to the file system and not the RAID-level scrub. Depending on system load, data scrubs are initiated on Sunday evenings. From our data, we find that an entire RAID group is data-scrubbed approximately once every two weeks on an average. However, we cannot ascertain from the data that every disk in the study has been scrubbed.

3.1.2.4 Disk Replacement

The storage system uses proprietary heuristics for determining when to replace a disk drive. These heuristics are threshold-based and take into account the time between partial failures, as well as the total number of failures encountered. Other systems use similar heuristics to predict further disk failures based on observed failures; for example, Linux systems often use SMART [2]. Our study enables the tuning of thresholds used to predict disk failures.

3.1.3 Data Collection

The storage system has a built-in, low-overhead mechanism called *Autosupport* to log important system events back to a central repository called the *NetApp Autosupport Database*. These messages can be enabled for a variety of system events including partial disk failures. These logs allow customized support based on observed events. For example, it enables proactive actions such as replacement of a disk based on the number of latent sector errors that have been observed during a time window. Not all NetApp customers enable logging, although a large percentage do. Those that do, sometimes do so only after some period of initial use. The NetApp Autosupport Database has been used for other disk failure studies as well [70, 122].

Disks undergo rigorous testing both at NetApp and by the disk vendor before they are shipped. The partial disk failures detected in in-house testing are not reported to the database and thus not reflected in our study. Sectors with errors are automatically re-mapped during this testing process. Note that this testing may even eliminate disks that would have otherwise shown up in our data as highly error-prone.

We study disk-failure data reported to the NetApp Autosupport Database, starting from January 2004 for a period of 32 months for the study of latent sector errors, and for a period of 41 months for the study of silent data corruptions.

3.2 Methodology

We first describe some terminology and our analysis methodology. Then, we outline some limitations of the study. Next, we motivate why we perform the set of analyses that we do. Finally, we present notation used to discuss our results.

3.2.1 Terminology

We use the following terms in the remaining sections.

Disk class: Enterprise or nearline disk drives with respectively Fibre-Channel and ATA interfaces.

Disk family: A particular disk drive product. The same product (and hence a disk family) may be offered in different capacities (sizes). Typically, disks in the same family only differ in the number of platters and/or read/write heads [121].

Disk model: The combination of a disk family and a particular disk size.

Disk age: The amount of time a disk has been in the field since its ship date, rather than the manufacture date. In practice these these two values are typically within a month of each other.

Error disk: A disk drive that has at least one latent sector error.

Corrupt block: A 4-KB file-system block with a checksum mismatch.

Corrupt disk: A disk drive that has at least one corrupt block.

3.2.2 Analysis Methodology

Our analysis of partial disk failures is based on a sample of 1.53 million disk drives of 2 different disk classes (nearline and enterprise), 14 different disk families and 31 distinct disk models across these families. In our analyses, we typically separate out the results by the disk model; as we shall see later, the disk model may impact the development of partial failures.

We now describe various constraints that we use in our analysis:

- We constrain the data by the age of the disk drives; that is, we typically look at the failures that occur in the first N months of use of the disk drive. This constraint is used to eliminate the impact of variations due to disk age; many of the drives in the study were shipped on different dates and have been in the field for different amounts of time. N is typically 18 months for analyses involving latent sector errors and 17 months for corruptions.

- We analyze only those disk models of which there are at least D disks in the field for the time period of a given analysis. This constraint is used since disk failures may be rare events and one needs to have a sufficient number of disk drives to arrive at conclusions with a reasonable degree of confidence. D is 1000 for the various analyses of both latent sector errors and corruptions.

- We analyze only those disk models of which there are at least E error disks (or corrupt disks) for the time period of a given analysis. This constraint is used for the same reason as the previous one. It is applied only in those cases where the properties of error disks are being analyzed (*e.g.*, the number of partial failures that an error disk develops). E is 50 for latent sector errors and 15 for corruptions.

- We analyze only those disk models for which there are total of at least F failures for the time period of an analysis. This constraint is only applied in those cases where the properties of the failures are being analyzed (*e.g.*, the fraction of failures detected by different types of disk requests). F is 1000 for latent sector errors and 500 for corruptions.

We use the specific numbers in the constraints above with a view to achieving a balance between obtaining as reliable statistics as we can for each disk model and including as many disk models in each analysis as possible.

While we usually present data for individual disk models, we sometimes also report averages (mean values) for nearline disks and enterprise disks. Since the sample size for different disk models in each disk class varies considerably, we weigh the average by the sample size of each disk model in the respective class.

3.2.3 Limitations

The study has a few limitations that mostly stem from the data collection process.

First, for a variety of reasons, disks may be removed from the system. Our study includes those disks up to the point of their removal from the system. Therefore, we may not observe errors from otherwise error prone disks after some period of time.

Second, nearline disks automatically perform sector reassignment for latent sector errors during write operations; see Section 3.1.2.1. Thus, latent sector errors encountered during writes for this class of disks are not propagated beyond the disk and nearline error rates do not reflect these write errors.

Third, since the logging infrastructure has been built with customized support as the primary purpose, the data can be used to answer most but not all questions that are interesting for a study such as ours. For example, while we can identify the exact disk when an error is detected during a scrub, we cannot verify that every disk in the study has been scrubbed periodically in the absence of errors. This limits our ability to precisely identify the extent to which scrubbing is useful.

3.2.4 Motivation

In this subsection, we preview and motivate the various analyses we perform using our data:

Factors: We examine how different factors affect the development of partial disk failures. First, we explore the impact of disk class. This analysis is important because it identifies systems that are likely to be affected. For example, if nearline disks were affected more, then personal computers that mostly use such disks would have to include mechanisms to deal with partial disk failures. Second, we explore how the age of the disk drive affects the development of partial disk failures. This analysis is useful for checking if techniques that handle partial disk failures need to be adapt as disks age. Third, we analyze whether disk capacity is a factor by comparing different disk models of the same disk family. The world is moving towards bigger capacity disks and we should analyze whether that trend decreases or increases the probability of encountering partial disk failures. Last, we perform a preliminary

examination of the impact of workload on corruptions. The goal of this analysis is to check whether we need to manage the workload on disks so that they are more reliable.

Failures per error disk: This analysis measures exactly how many latent sector errors or checksum mismatches occur in a disk that has at least one latent sector error or checksum mismatch respectively. One goal of this analysis is to check whether disks need to be replaced when the first partial disk failure is detected or whether we could continue using the disk as long as we can recover the data in the few disk blocks that were lost. Another goal is to check whether partial disk failures are independent; any dependence between failures affects data reliability (and for that reason, it is essential for analytical models of RAID reliability [17, 39, 76, 119]); in addition, it may influence techniques that can be used to handle partial disk failures.

Address space locality: This analysis measures how close partial disk failures on the same disk are in the logical disk-address space. The spatial locality of errors is often considered in the design of various existing file systems. For example, the original Fast File System (FFS) creates redundant spatially-distributed copies of the superblock, to protect against the loss of a disk head or multiple media errors on the same track or cylinder [88]. Our study of file-system robustness [104] found that JFS stores superblock copies close to each other in the logical address space, possibly exposing it to loss of both copies. Likewise, ReiserFS places its log across a contiguous set of logical blocks [108]. Multiple latent sector errors in the log area may render the file system unusable.

Today, disk drives use a block-based interface (*e.g.*, SCSI or ATA) that obfuscates physical block locations through complicated mapping schemes [115]. This limits file systems to use logical block locality unless more detailed information can be derived [103]. Since file system designers often make assumptions about spatial locality at the logical block level, we explore whether partial disk failures exhibit spatial locality at the logical level, referred to as *address space locality*.

Temporal locality: Another interesting characteristic of latent sector errors is their temporal behavior. Temporal locality is a study of how "bursty" latent sector errors are. This analysis is useful for setting various time-based thresholds used to determine when a disk should be replaced. In addition, it may influence policies on whether more active disk scrubbing should be performed when the first partial failure is detected.

Correlations: We also examine whether different types of partial disk failures correlate. This information is useful for predicting future failures and also for indicating whether different partial disk failures may have common causes.

Detection: Finally, we study the manner in which partial disk failures are detected by the system. Ideally, a storage system would proactively detect errors (*e.g.*, through periodic scrubbing) before a user-initiated request. Sector errors detected early can be recovered from RAID-style data reconstruction and re-mapped to a new sector. Proactive detection of partial disk failures reduces the likelihood of "double-failures" in a RAID system [17].

3.2.5 Notation

We denote each disk drive model as ⟨*family-size*⟩. For anonymization purposes, *family* is a single letter representing the disk family and *size* is a single number representing the disk's particular capacity. Although capacities are anonymized, relative sizes within a family are ordered by the number representing the capacity. For example, n-2 is larger than n-1, and n-3 is larger than both n-1 and n-2. The anonymized capacities do not allow comparisons across disk families. Disk families from *A* to *E* (upper case letters) are nearline disk families, while families from *f* to *o* (lower case letters) are enterprise disk families. Lines on graphs labeled *NL* and *ES* represent the weighted average for nearline and enterprise disk models respectively.

We present data as the fraction of disks in a particular sample that develop x partial failures. We use the probability notation $P(X_t \geq x)$ to denote the fraction of disks developing at least x errors within t months since the disk's first use in the field. We use $E(X_t)$ to refer to the mean number of errors developed within t months since first use.

3.3 Latent Sector Errors

This section presents the results of our analysis of latent sector errors. First, we present summary statistics on latent sector errors collected over 32 months from disk drives in the field. Second, we analyze the impact of various factors that affect the occurrence of latent sector errors, including disk class, disk model, disk age, and disk size. Third, we study various properties of latent sector errors, including the numbers of errors that occur in an error disk, the spatial locality of errors, and the temporal behavior of errors. Fourth, we discuss correlations between latent sector errors and other disk errors such as recovered errors, and not-ready-condition errors. Finally, we examine the distribution of detection of latent sector errors across the different types of disk requests: read, write, and verify.

3.3.1 Summary Statistics

In our entire sample of 1.53 million, we find 53,820 (3.45%) disks developed one or more latent sector errors. For error disks (disks with at least one error), the median number of errors per disk is three. However, the mode is one error (30% of the error disks). Only 0.2% of error disks had more than 1000 errors per disk. Ignoring these "outlier" disks, the mean number of errors per error disk is 19.7.

Observation 3.1 *Enterprise disks are less likely to develop latent sector errors than nearline disks.*

Overall, we find that nearline disks and enterprise disks exhibit different behavior with respect to latent sector errors; about 8.5% of all nearline disks are affected by latent sector errors while only 1.9% of all enterprise disks are affected. Therefore, most of our subsequent analyses break down results by disk class.

Looking at disks of the same age, we find that 3.15% of nearline disks and 1.46% of enterprise disks develop at least one latent sector error within twelve months of their ship date. This sample includes 200,408 nearline disks (56% of all nearline disks in our study) across 6 disk models and 715,033 enterprise disks (61% of all enterprise disks in our study) across 23 disk models. Using

our notation, these numbers can be represented as $P(X_{12} \geq 1)$. We present more detail about error rates as a function of time in Sections 3.3.2.1 and 3.3.3.3.

3.3.2 Factors

We now explore the impact of various factors on latent sector errors: the disk class (nearline versus enterprise), the disk model, the age of the disk drive, and its size.

3.3.2.1 Disk Class, Model, and Age

We study how the age of the disk drives affects (a) the fraction of disks that develop latent sector errors, and (b) the fraction of sectors that develop errors (ASERs).

Figure 3.2 presents the fraction of disks that develop their first latent sector error within a specific age. As described earlier, we include only disk models with at least 1000 units in the field for the entire 24-month period of this study. Using our notation, we can express the graph as $P(X_t \geq 1)$ where, t={6, 12, 18, 24} months. The same sample of disks is used for all time periods. The sample includes 68,380 nearline disks across three disk models and 264,939 enterprise disks across ten disk models.

As observed in the previous subsection (Observation 3.1), we see that nearline disks are more likely to develop latent sector errors. For example, almost 20% of E-2 disks experience latent sector errors within 24 months of their shipping. On the other hand, only 4% of k-3 disks, the enterprise disk model with the highest error rate, experience latent sector errors in the same time period.

Observation 3.2 *The fraction of disks with latent sector errors varies significantly across manufacturers and disk models.*

We see from Figure 3.2 that the fraction of disks with errors at the end of 24 months could vary from 5% to 20% for nearline disks. Enterprise disks also exhibit a significant variation.

Observation 3.3 *Over 24 months, the fraction of nearline disks developing latent sector errors grows far more rapidly than the fraction of enterprise disks with errors.*

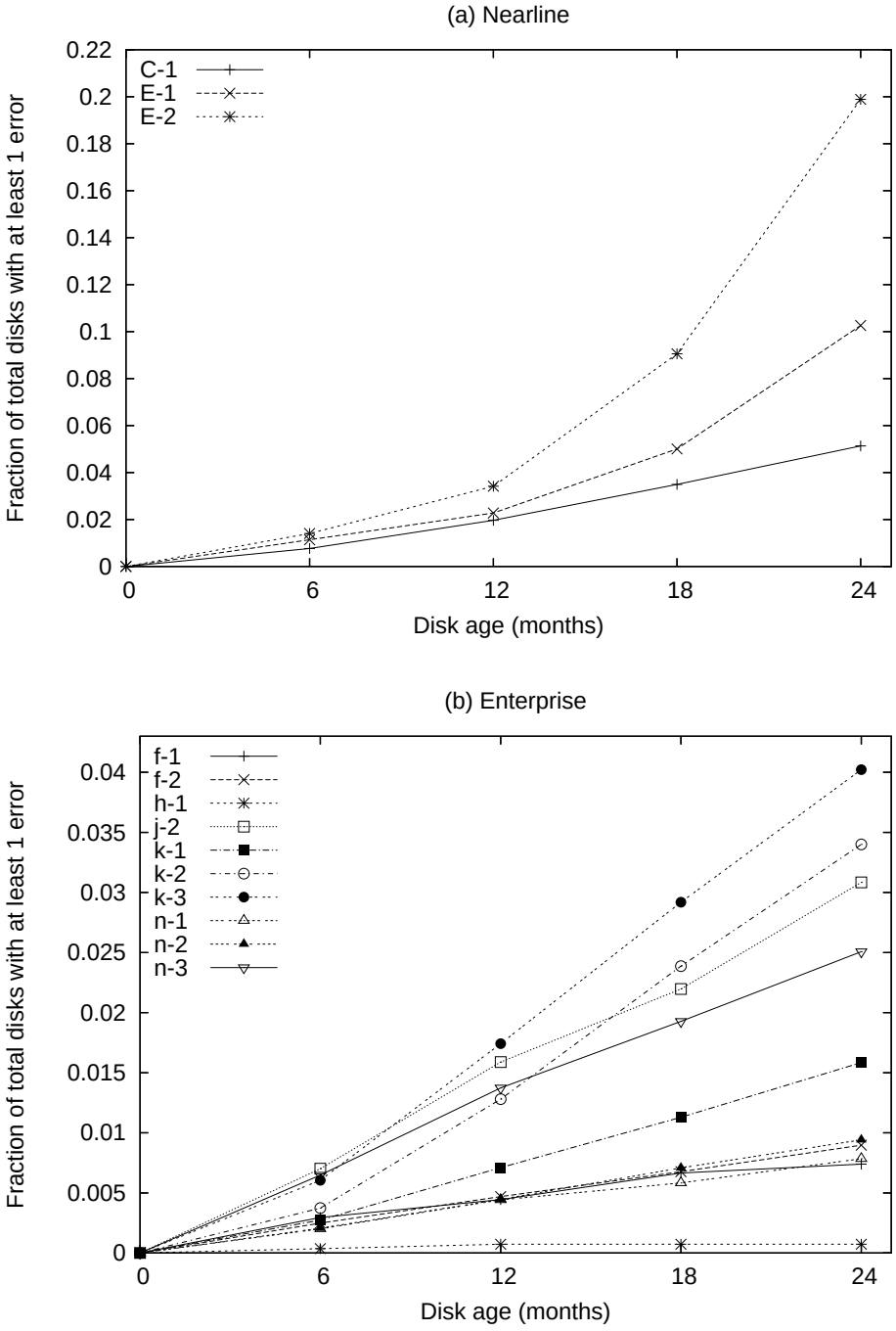

Figure 3.2 **Impact of disk age.** *The fraction of disks that develop latent sector errors as age increases is shown. Note that the fraction is cumulative. Also, note that the y-axis scale is different for the two graphs.*

In the case of enterprise disks, we observe that the fraction of disks that have latent sector errors increases almost linearly with time. Thus, the fraction of enterprise disks that develop a latent sector error in a given six month window is nearly the same within the first 24 months of use. On the other hand, this fraction for nearline disks increases super-linearly with increasing disk age. For example, the fraction of E-1 disks that develop latent sector errors in the time period between 18 and 24 months after shipping is 5.25%, while it is only 2.72% between 12 and 18 months after shipping. More generally, $(P(X_{t+6} \geq 1) - P(X_t \geq 1)) > (P(X_t \geq 1) - P(X_{t-6} \geq 1))$, where $t \leq 24$.

Observation 3.4 *Annual sector error rates vary greatly across disk models but on average are considerably worse during the second year for nearline disks.*

Figure 3.4 shows the annual sector error rates (ASERs) computed for the disk models, as well as the cumulative nearline and enterprise error rates. The error rates are for the first and second year of disk use. The sample covers all drives in the field for 24 months (the same sample as in Figure 3.2). The figure can be represented as $E(X_t - X_{t-12})/(sectors\ per\ disk)$ for t = {12, 24} months. Note that the figure does not show error bars since most disks have 0 errors. For nearline drives the sector error rates for the second year increase considerably over the first year. However, this is not the case for the enterprise drives. About half of the enterprise models show this trend, while half do not.

3.3.2.2 Disk Size

Figure 3.3(a) shows the fraction of disks with latent sector errors across the various disk families. For each disk family, the graph groups the data by disk model (disk capacity). We restrict the disk families in the graph to those for which there are at least 1000 disks in the field with an age of at least 18 months for *each* disk size. This age maximizes the number of disk models we can study. Figure 3.3(a) can be represented as $P(X_{18} \geq 1)$ for different disk models.

Observation 3.5 *We observe that as disk size increases, the fraction of disks with latent sector errors increases across all disk models.*

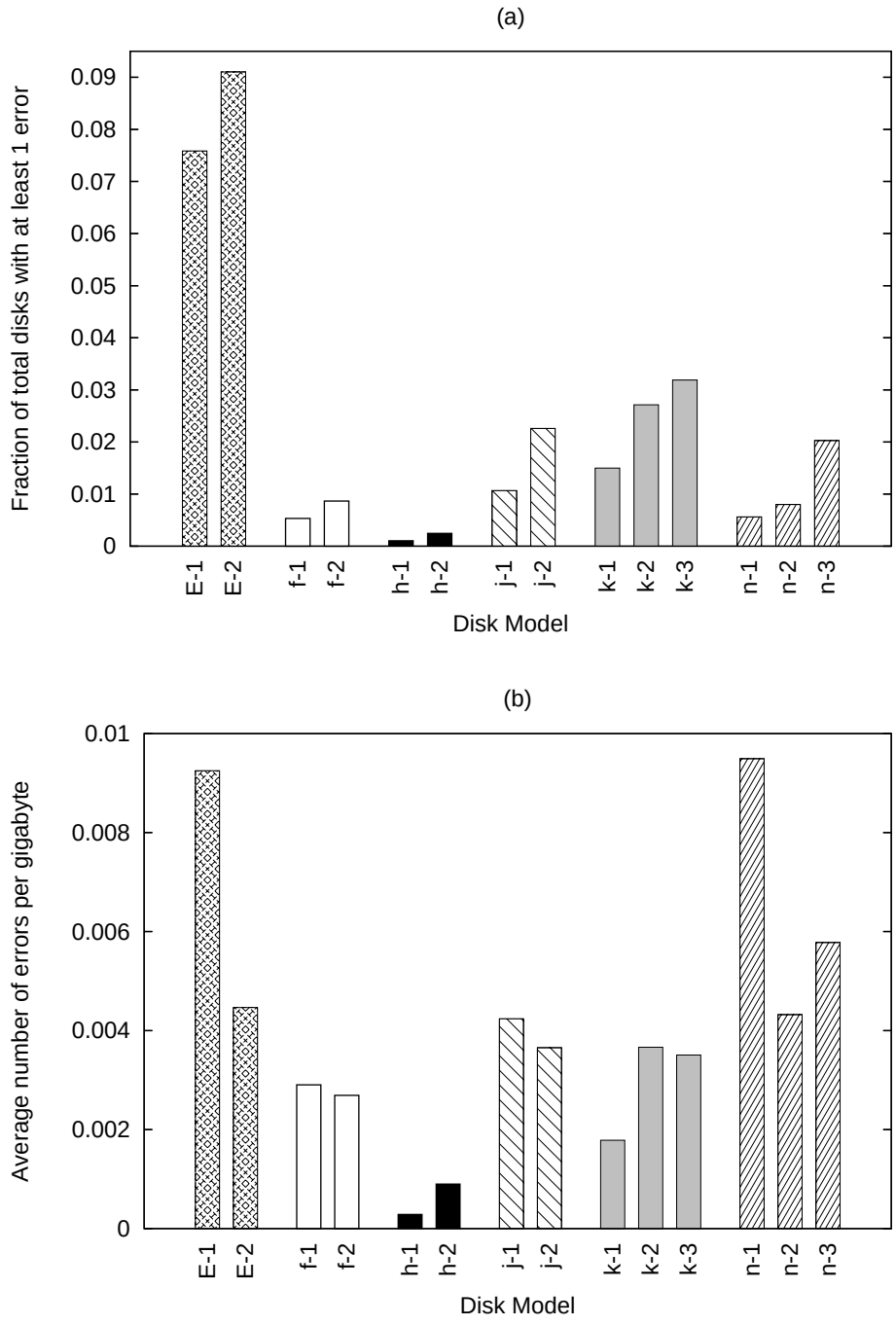

Figure 3.3 **The impact of disk size.** *(a) Fraction of disks with at least one latent sector error within 18 months of shipping to the field. (b) Average number of latent sector errors per GB observed within 18 months of shipping to the field.*

We observe the same trend even for those families that did not satisfy the 1000-disk requirement with the only exception being disk family 'l'. As disk capacity rapidly increases, storage systems will need to deal with a larger percentage of drives that develop latent sector errors. However, since many factors contribute to latent sector errors (see Section 2.2.2), we cannot draw any specific conclusion beyond the trend we observe in the data.

Observation 3.6 *The amount of probable data loss due to latent sector errors per Gigabyte does not increase or decrease consistently as disk size increases.*

Figure 3.3(b) presents the average number of latent sector errors per Gigabyte. It can be represented as $E(X_{18})/Capacity$. Interestingly, unlike Figure 3.3(a), the data does not show a consistent increase or decrease across disk size for the same disk family. Thus, we see that a higher fraction of disks with errors does not imply a greater amount of probable data loss.

3.3.3 Properties

The studies in this subsection focus on the properties of latent sector errors. We first study the actual number of errors that occur in a disk with at least one latent sector error, then whether latent sector errors within the same disk are spatially-local, and finally, the temporal behavior of latent sector errors.

3.3.3.1 Errors per Error Disk

Figure 3.5 shows the fraction of error disks that experience a given number of latent sector errors within a 18-month period after the ship date. We only include disk models that satisfy both the 1000 disk and 50 error disk limits. Thus, we can represent the figure using our notation as the conditional probability $P(X_{18} \leq x | X_{18} \geq 1)$ for x=$\{$1, 2, 3, 4, 5, 10, 20, 50$\}$.

Observation 3.7 *A large fraction of disks with latent sector errors develop fewer than 50 errors.*

The data shows that, on average, 37% of nearline error disks and 39% of enterprise error disks have only one error; that is, they do not develop any additional latent sector errors after the first

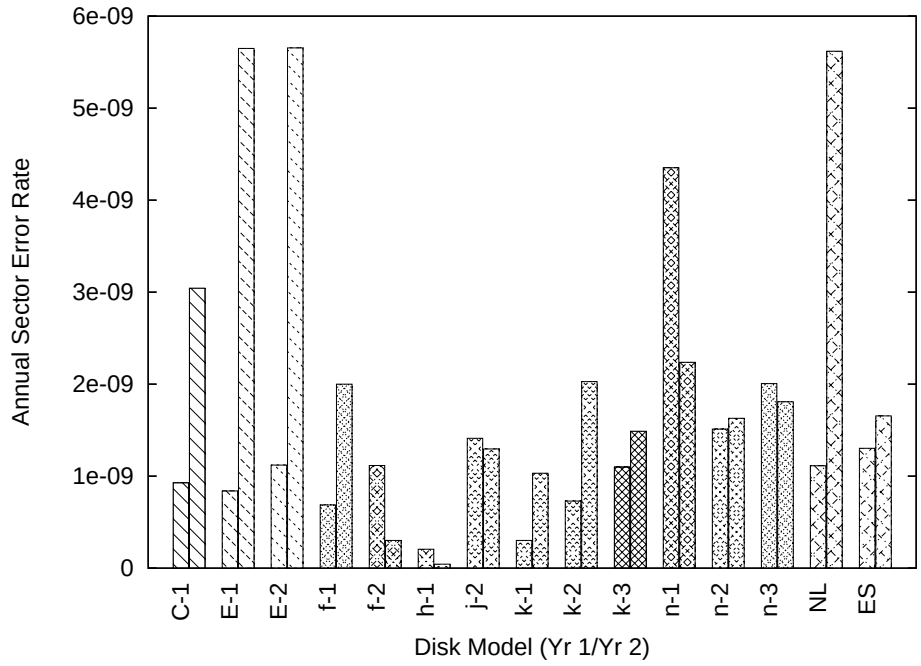

Figure 3.4 **Annual sector error rates (ASERs).** *For each disk model that has been in the field for at least two years, the first bar represents Year 1 and the second represents Year 2. The NL and ES bars represent weighted averages for nearline and enterprise drives respectively.*

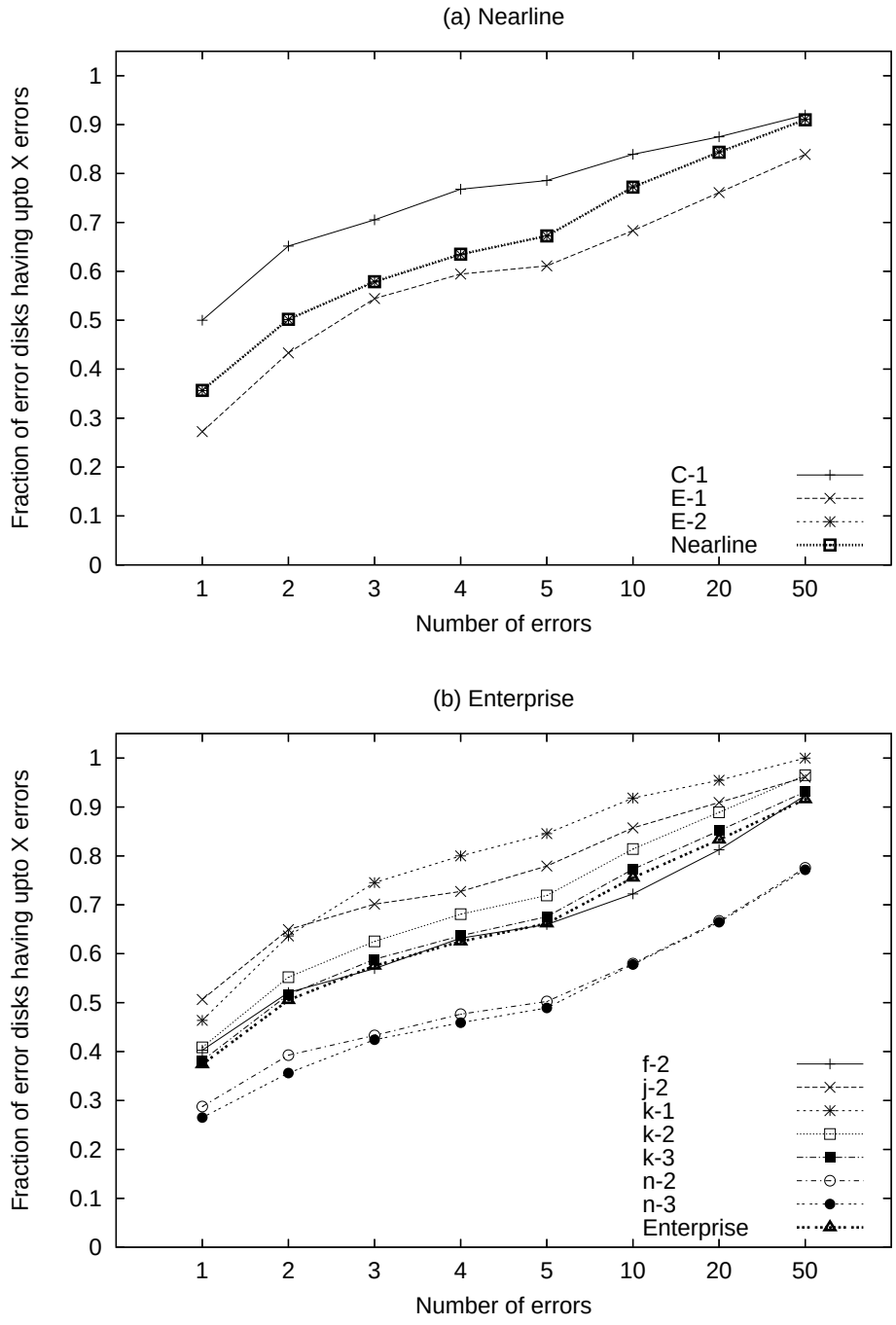

Figure 3.5 **Errors per error disk.** *The fraction of error disks as a function of the number of latent sector errors that develop within a 18 month period after the ship date for (a) nearline disk models and (b) enterprise disk models.*

one. Furthermore, over 80% of error disks have fewer than 50 errors. Since disk drives typically have thousands of spare sectors and since failed sectors can be recovered from elsewhere (*e.g.*, from RAID), it is possible to re-map bad sectors and continue operation for a large fraction of error disks.

Observation 3.8 *Enterprise and nearline disks are equally likely to develop more than one error once they develop their first error, in contrast to the very different probabilities of enterprise and nearline disks developing their first error.*

While enterprise disks seem to be more resilient to latent sector errors in general, enterprise disks and nearline disks show similar behavior once they exhibit at least one latent sector error; compare the Nearline and Enterprise lines in Figure 3.5(a) and Figure 3.5(b), respectively. Surprisingly, some enterprise disk models are worse than nearline disks – a larger fraction of enterprise error disks develop many more errors than nearline error disks. However, one should note that the actual number of latent sector errors for nearline disks could be somewhat higher (as described in Section 3.2.3).

Observation 3.9 *Latent sector errors are not independent of each other. A disk with latent sector errors is more likely to develop additional latent sector errors than a disk without a latent sector error.*

We find that the occurrence of a latent sector error depends on previous occurrences of latent sector errors on the same disk. In particular, we find that the fraction of disks developing at least 1 additional error in x amount of time given that the disk has at least 1 error, $P(X_{t+x} \geq 2|X_t \geq 1)$, is greater than the (non-conditional) fraction of disks that develop at least 1 error in x amount of time ($P(X_{t+x} \geq 1)$ - $P(X_t \geq 1)$). For example, $P(X_{18} \geq 2|X_{12} \geq 1)$ = 0.671, which is much greater than $P(X_{18} \geq 1)$ - $P(X_{12} \geq 1)$ = 0.018.

3.3.3.2 Address Space Locality

Figure 3.6 presents the fraction of latent sector errors that have at least one other latent sector error occurring within a given radius, for disks with at least 2 errors and at most 10 errors. An

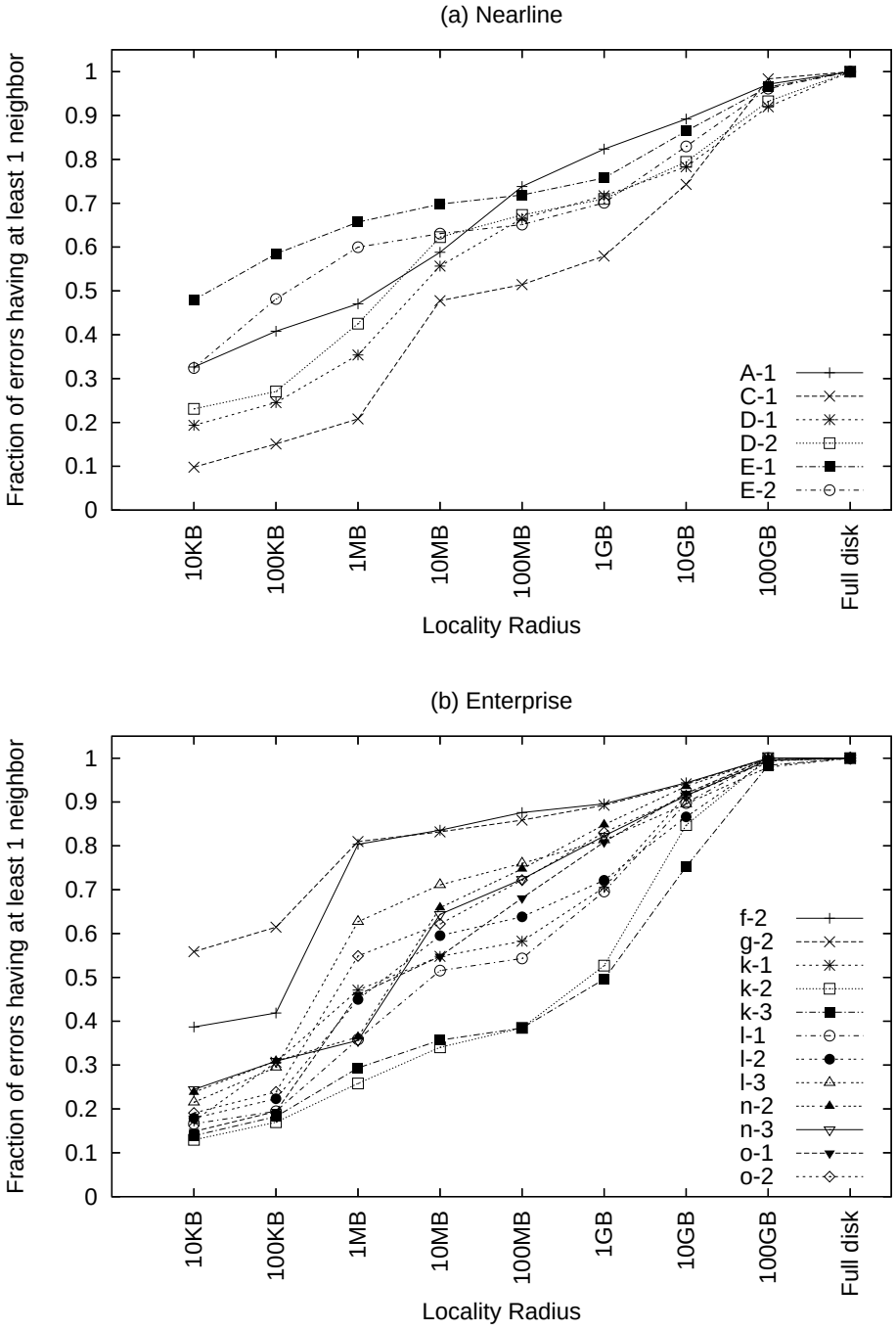

Figure 3.6 **Address space locality.** *The graphs show the fraction of latent sector errors with another latent sector error within a given radius (address range).*

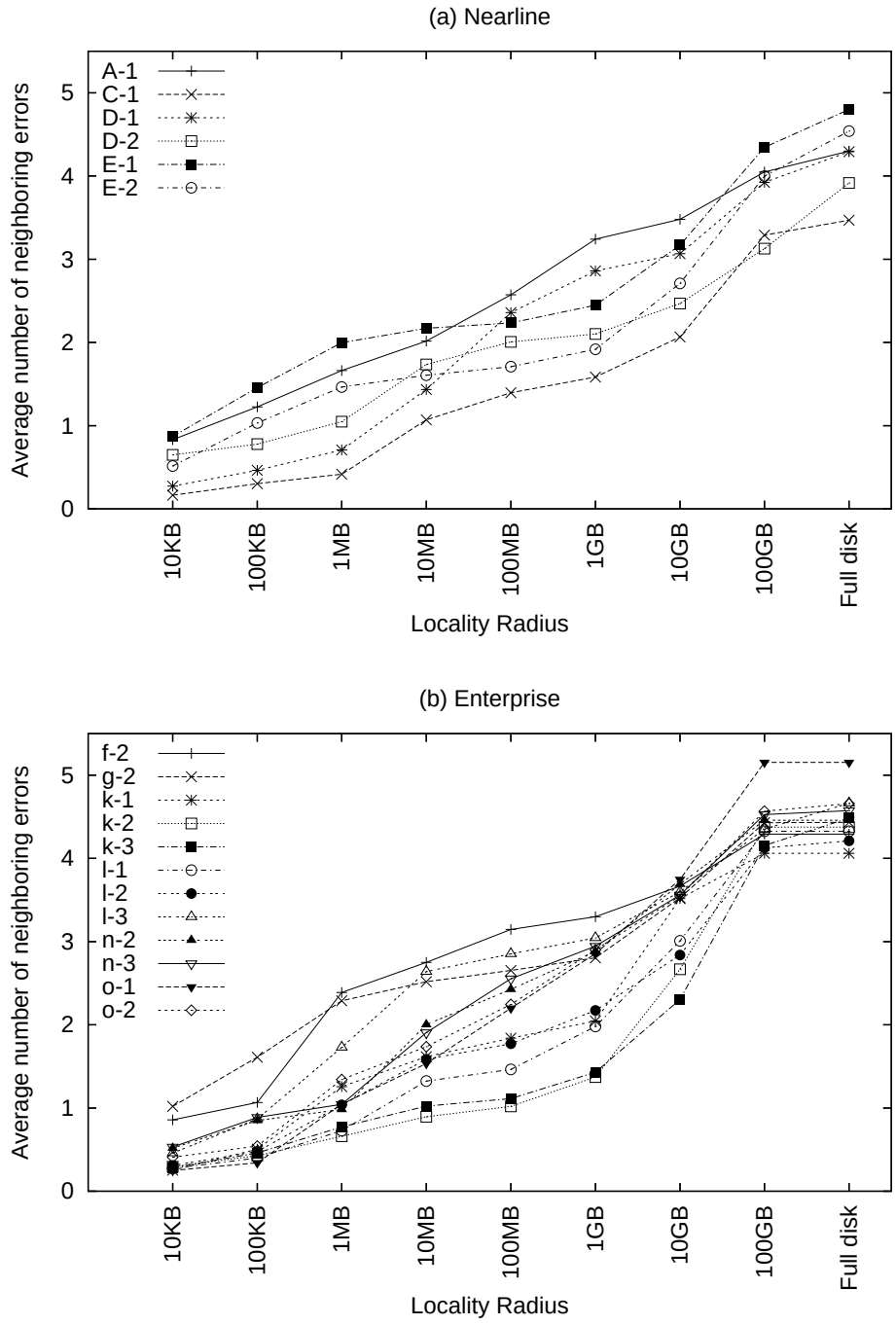

Figure 3.7 **Count of spatially-local errors.** *The figure presents the mean number of other latent sector errors within a given radius (neighbors) of an existing error. The data uses only disks with 2 to 10 latent sector errors, thus limiting the maximum value possible to 9.*

upper bound of 10 errors is used in order to avoid skew introduced by disks with a large number of errors; note, the median number of errors for error disks is 3. Since address space locality is time-invariant as long as the number of errors is bounded, the sample includes all disks irrespective of their time in the field. We only include disk models that have at least 1000 total disks and 50 error disks with between 2 and 10 errors for the entire 32 months. We can express the data in our notation as $P(X_t^r \geq 1 | 2 \leq X_t \leq 10)$ with no specific restriction on time ($0 < t < 32$), where X^r is the number of other latent sector errors in the interval $\langle a - r, a + r \rangle$ centered around sector a; sector a contains a latent sector error.

Observation 3.10 *There is significant locality in the occurrence of latent sector errors across logical sector addresses.*

Figure 3.6 shows that for most disk models, the fraction of latent sector errors that have at least one other latent sector error within a 10 MB radius of it is 0.5. In fact, the fraction is more than 0.6 for many models. Additionally, for many disk models, the fraction increases significantly between radii of 100 KB and 1 MB. This suggests a coarse correlation between the logical and physical block space. However, we note that the observed address space locality is not perfect and may not be as correlated as system designers believe. Finally, we note that the fraction varies considerably across disk models.

Figure 3.7 presents the mean value of X^r (X^r is the same as above) for different disk models. This figure provides an insight into how errors typically cluster together. For most models, the average number of other errors within a 10 MB radius of a latent sector error is more than 1; for some models it is as high as 2.5. When Figures 3.6 and 3.7 are compared, we see that a higher probability of a spatially local error does not necessarily imply a higher average number of spatially local errors. For example, for a 10 MB radius, g-2 has a higher probability of a spatially local error than l-3, but l-3 has more spatially local errors than g-2 on average.

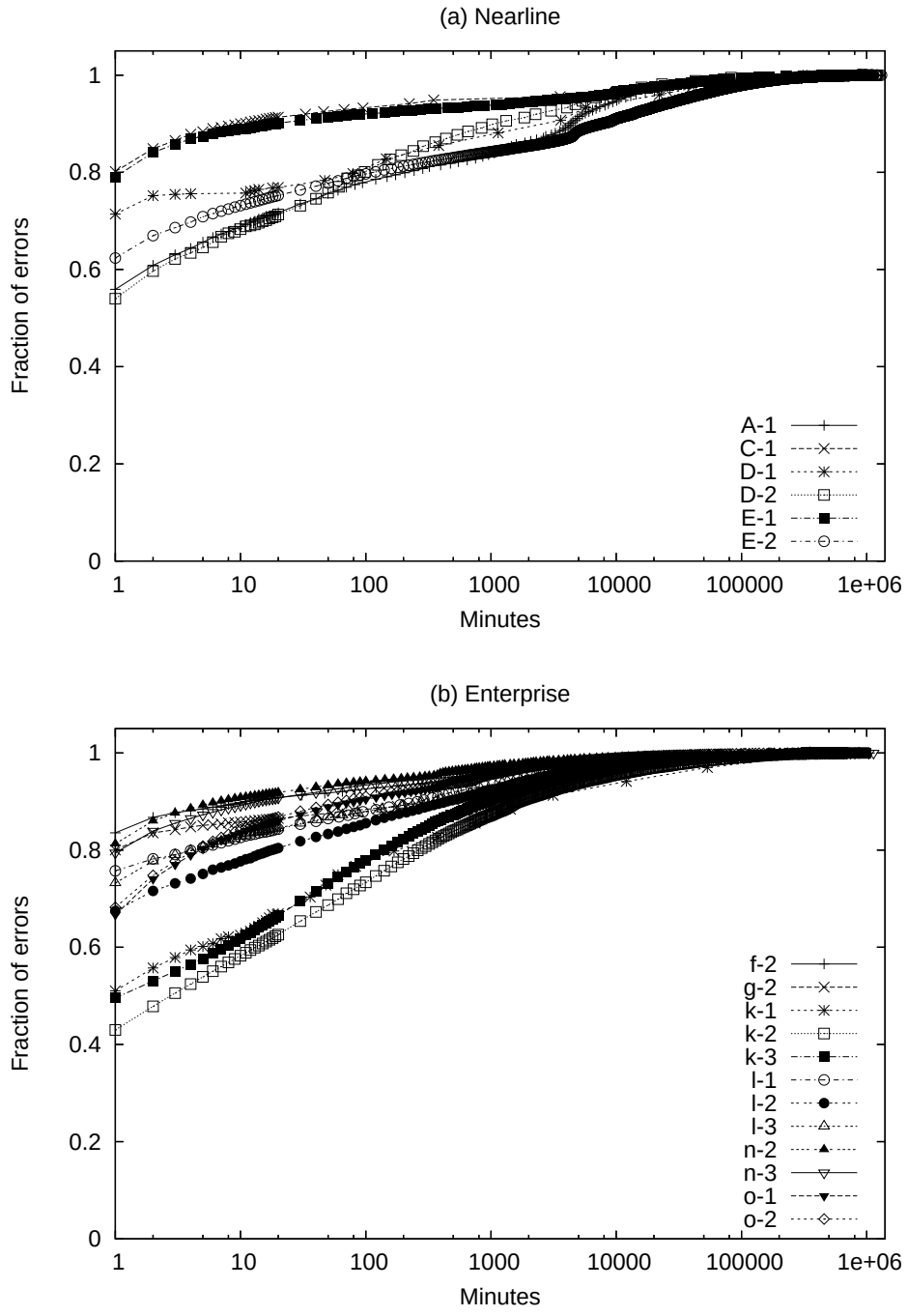

Figure 3.8 **Inter-arrival time.** *The graphs show the cumulative distribution of the inter-arrival times of latent sector errors. The fraction of errors per model is plotted against time. The arrival times are binned by minute.*

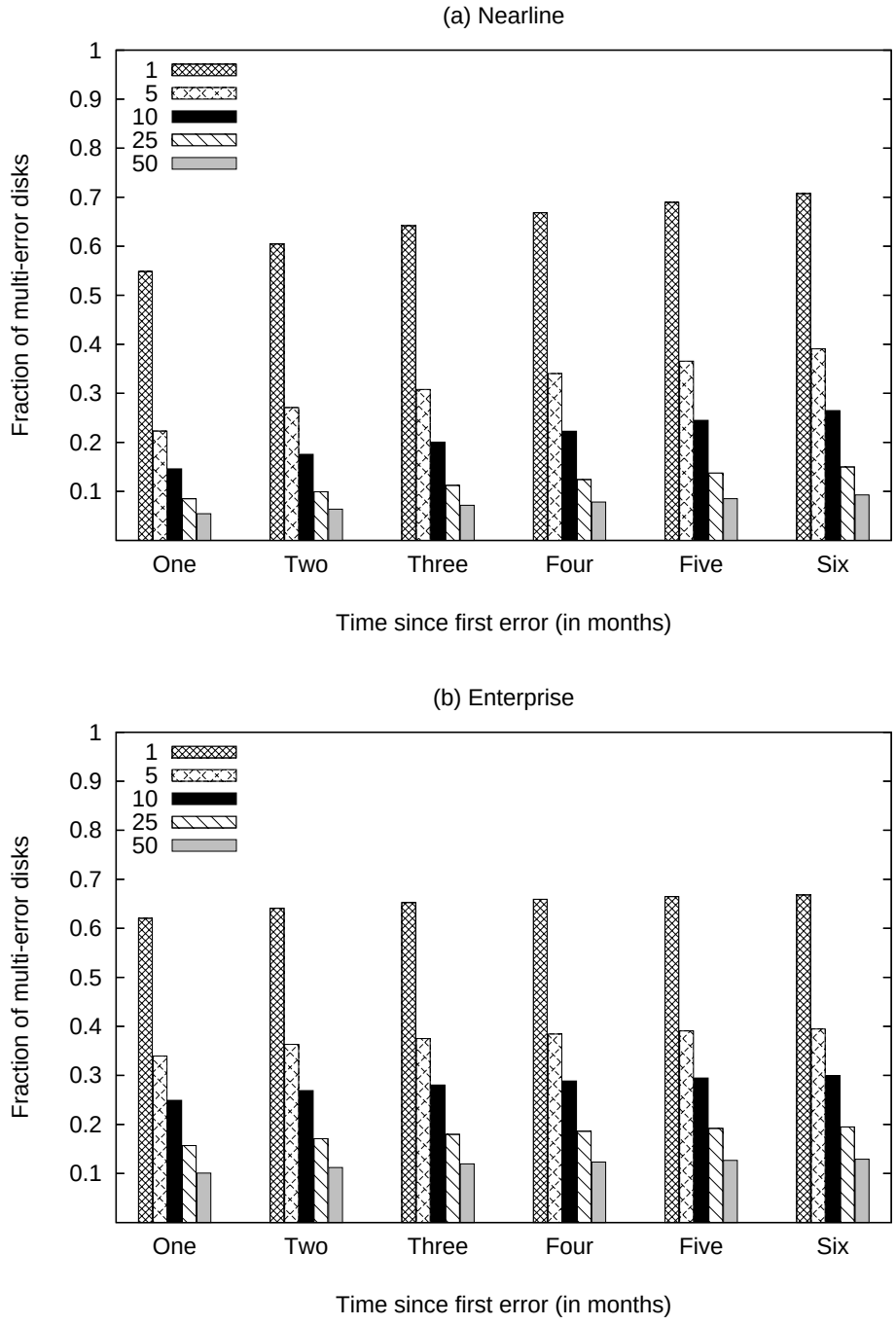

Figure 3.9 **Temporal decay.** *The figure shows the fraction of disks that experience at least 1, 5, 10, 25, and 50 additional latent sector errors within a given time period since the occurrence of the first latent sector error.*

3.3.3.3 Temporal Behavior

We study temporal behavior in two ways: *temporal locality* and *decay*. Temporal locality is a study of how "bursty" latent sector errors are. We study temporal locality by measuring the inter-arrival time of errors. Decay is a study of the time taken to develop e additional latent sector errors since the first latent sector error.

Figure 3.8 shows the fraction of latent sector errors that arrive within x minutes of the preceding error. The arrival times are binned by minute. We only include disk models that satisfy both the 1000-disk and 50 error disk limits. The figure can be represented as $P(X_{t+x} \geq k+1 | X_t = k \wedge X_T \geq k+1)$ for $0 < k \leq 1000$, and $0 < t < T < 32$ and $1 \leq x \leq$ 1e+06 minutes.

Observation 3.11 *All disk models exhibit high temporal locality of latent sector errors.*

Depending upon the model, between 40%-80% of errors arrive within one minute of the previous error. As can be seen, the arrival-time distributions have very long tails. The observed locality implies that the errors are detected close in time (even though they may have developed long before they were detected). However, due to media scrubs, there is typically only a short lag time between the occurrence and the discovery of an error. Thus, errors that develop at different times (*e.g.*, a month apart) are likely to be detected at different times. It is likely that the observed temporal locality implies actual temporal locality.

Figure 3.9 presents the fraction of disks that develop at least e additional errors within a given time period since the discovery of the first error, for nearline and enterprise disk classes. We use disks that developed the first error at least 6 months before the end of the study. Both nearline and enterprise disk classes had at least 10,000 eligible units. The figure can be represented as $P(X_{t+x} \geq e+1 | X_t = 1)$ for $x = \{1, 2, 3, 4, 5, 6\}$, $e = \{1, 5, 10, 25, 50\}$, $0 < t < 26$.

Observation 3.12 *Disks that develop errors beyond the first error see most of the additional errors within one month after the first error.*

First, we see that for 54.8% of nearline error disks and 62.0% of enterprise error disks, at least one *additional* error is developed within one month of the first ever error. Second, there is a significant

probability (nearline: 0.05, enterprise: 0.10) that a disk with one error will develop 50 additional errors within one month of the first error. Third, we observe that the fraction of disks with one error that develop at least e more errors does not increase significantly with disk age for most values of e. Most of the additional errors develop within 1 month of the first error. Interestingly, this behavior is even more pronounced for enterprise disks than for nearline disks. Finally, comparing the numbers across the two graphs, we observe that surprisingly enterprise disks in general have a higher fraction of disks with one error that develop additional errors within a given period of time, the only exception being for $e = 1$.

3.3.4 Correlations

We now explore whether disks that exhibit latent sector errors also exhibit other kinds of errors. Specifically, we consider recovered errors and not-ready-condition errors.

3.3.4.1 Recovered Errors

As discussed in Section 2.2.2, recovered errors are errors that a disk drive encounters when accessing sectors and is able to recover from them through a combination of retries and error-correcting codes (ECC). Latent sector errors occur when such disk drive-level recovery fails. Similar to latent sector errors, the storage system proactively re-maps sectors associated with recovered errors.

The error logs contain recovered errors returned by enterprise disks. We found that 52971 enterprise disks exhibited at least one recovered errors (4.5% of enterprise disks) over the period of 32 months ($P(Z_t \geq 1)$ = 0.045, where Z is the number of recovered errors returned by a disk).

Observation 3.13 *There is a high correlation between latent sector errors and recovered errors for enterprise disks.*

Interestingly, despite the fact that we observed latent sector errors in less than 2% of enterprise disks ($P(X_t \geq 1) < 0.02$), the fraction of disks that develop a latent sector error out of disks that

experienced a recovered error is 13 times higher ($P(X_t \geq 1|Z_t \geq 1)$ = 0.26). This suggests that the two kinds of errors are not independent.

3.3.4.2 Not-Ready-Condition Errors

As discussed in Section 2.2.2, a not-ready-condition error is an error during which the disk is not available to respond to requests. The storage layer handles not-ready-condition errors by retrying the operation a few times. If these efforts fail, the data is reconstructed by the RAID layer from parity.

We found that 13% of nearline disks and 1% of enterprise disks encountered not-ready-condition errors. Thus, with no specific restriction on time ($0 < t < 32$), $P(Z_t \geq 1)$ = 0.13 for nearline disks, where Z is the number of not-ready-condition errors returned by a disk.

Observation 3.14 *There is a high correlation between latent sector errors and not-ready-condition errors for nearline disks.*

The fraction of disks that develop a latent sector error out of the disks that had a not-ready-condition error, $P(X_t \geq 1|Z_t \geq 1)$, is 0.38. This value is much higher than the fraction of disks that develop a latent sector error out of all nearline disks ($P(X_t \geq 1)$ = 0.085). Thus, it is highly likely that the two kinds of errors are not independent. We did not see a similar correlation in the case of enterprise disks where $P(X_t \geq 1|Z_t \geq 1)$ = 0.014 and $P(X_t \geq 1)$ = 0.019.

3.3.5 Detection

Figure 3.10 presents the fraction of latent sector errors that are discovered by read, write and verify operations. In the system, read and write operations are issued in order to satisfy user or file system requests. Verify operations are issued by the media scrubber; see Section 3.1.2.3. We restrict models to those with at least 1000 disks in the field with at least 50 error disks in the entire 32-month study period.

Observation 3.15 *Disk scrubbing detects a large percentage of observed latent sector errors.*

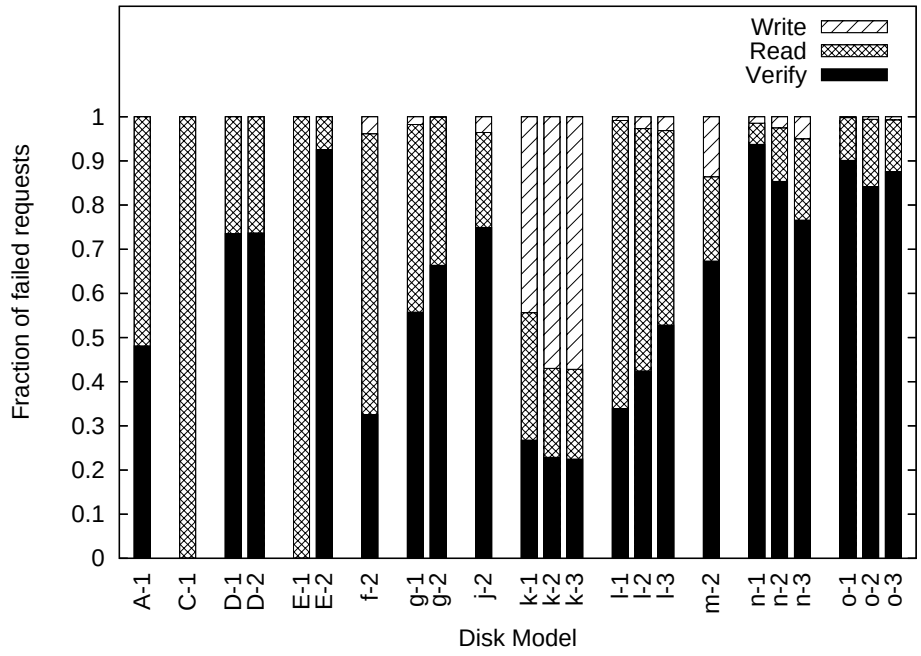

Figure 3.10 **Detection.** *The figure shows the distribution of requests that detect latent sector errors across the request types read, write and verify.*

The data shows that for many disk models, a high percentage of requests that experience a latent sector error are verify operations. On average, 86.6% of all latent sector errors in nearline disks and 61.5% of latent sector errors in enterprise disks are discovered by verify operations, while reads discover 13.4% of errors in nearline disks and 19.1% of errors in enterprise disks, and writes discover 0% of errors in nearline disks and 19.3% of errors in enterprise disks. This demonstrates that the method in which the systems perform media scrubbing is useful for discovering errors. Note, since nearline disks automatically and transparently perform sector reassignment, disk writes in these systems do not report latent sector errors (see Section 3.2.3).

While verify operations discover a widely varying proportion of latent sector errors across disk models, on average 77.4% of all errors are detected by verify requests across all disk models. We speculate that the differences we observe are in part due to the different workloads the systems with different disk models experience.

3.4 Silent Data Corruptions

This section presents the results of our analysis of silent data corruptions. We focus primarily on checksum mismatches. First, we provide basic statistics on the occurrence of checksum mismatches in the entire population of disk drives. Second, we examine various factors that affect the probability of developing checksum mismatches, including disk class, disk model, disk age, disk size and workload. Third, we analyze various properties of checksum mismatches, such as spatial locality. Fourth, we look for correlations between the occurrence of checksum mismatches and other system or disk errors. Fifth, we analyze the source of the disk requests that detect the mismatches. Sixth, we present an analysis showing that corruption may be block-number dependent. Finally, we present basic statistics on identity discrepancies and parity inconsistencies.

3.4.1 Summary Statistics

During the 41-month period covered by our data we observe a total of about 400,000 checksum mismatches. Of the total sample of 1.53 million disks, 3855 disks developed checksum mismatches – 3088 of the 0.36 million nearline disks (0.86%) and 767 of the 1.17 million enterprise

disks (0.065%). Using our probability representation, $P(Y_t \geq 1)$ = 0.0086 for nearline disks, and $P(Y_t \geq 1)$ = 0.00065 for enterprise disks where Y is the number of checksum mismatches that occur in the time since the disk was shipped, t.

This indicates that nearline disks may be more susceptible to corruption leading to checksum mismatches than enterprise disks. On average, each disk developed 0.26 checksum mismatches. Considering only corrupt disks(*i.e.*, disks that experienced at least one checksum mismatch), the mean number of mismatches per disk is 104, the median is 3 and the mode (*i.e.*, the most frequently observed value) is 1 mismatch per disk. The maximum number of mismatches observed for any single drive is 33,000.

3.4.2 Factors

We examine the dependence of checksum mismatches on various factors: disk class, disk model, disk age, disk size, and workload.

3.4.2.1 Disk Class, Model and Age

Figures 3.11 and 3.12 show the fraction of disks that develop their first checksum mismatch within a specific age for nearline and enterprise disks respectively. The graphs plot the cumulative distribution function of the time until the first checksum mismatch occurs. The figures can be represented as $P(Y_t \geq 1)$ for t = {3, 6, 9, 12, 15, 17} months, that is, the fraction of disks with at least one checksum mismatch after t months. Note the different y-axis scale for the nearline and enterprise disks. We see from the figures that checksum mismatches depend on disk class, disk model and disk age.

Observation 3.16 *Nearline disks (including the SATA/FC adapter) have an order of magnitude higher probability of developing checksum mismatches than enterprise disks.*

Figure 3.11 (line NL – Nearline average) shows that 0.66% of nearline disks develop at least one mismatch during the first 17 months in the field ($P(Y_{17} \geq 1)$ = 0.0066), while Figure 3.12(b) (line

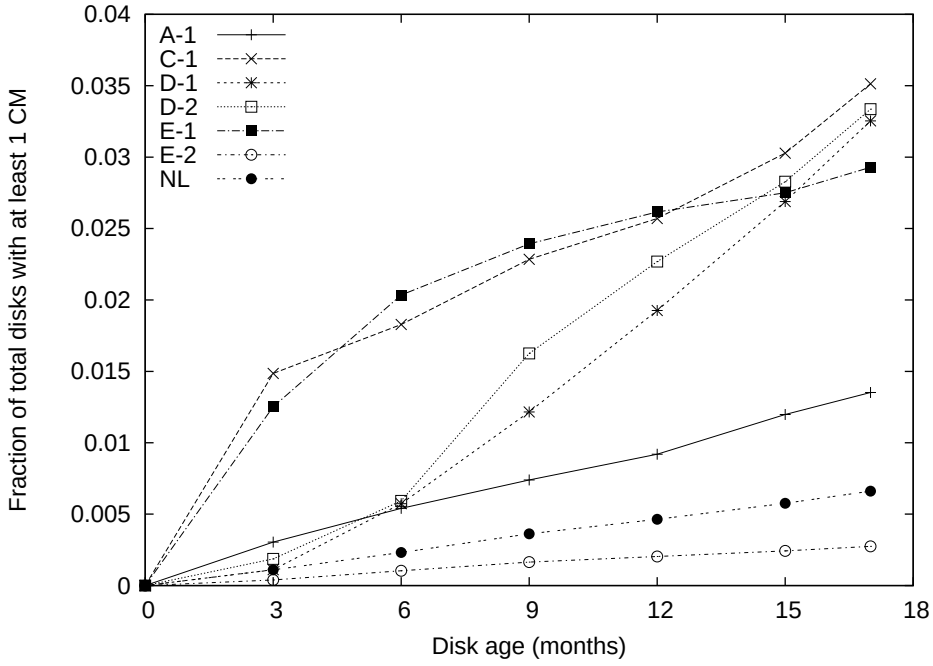

Figure 3.11 **Impact of disk age on nearline disks.** *The fraction of disks that develop checksum mismatches as age increases is shown for nearline disk models. Note that the fraction is cumulative.*

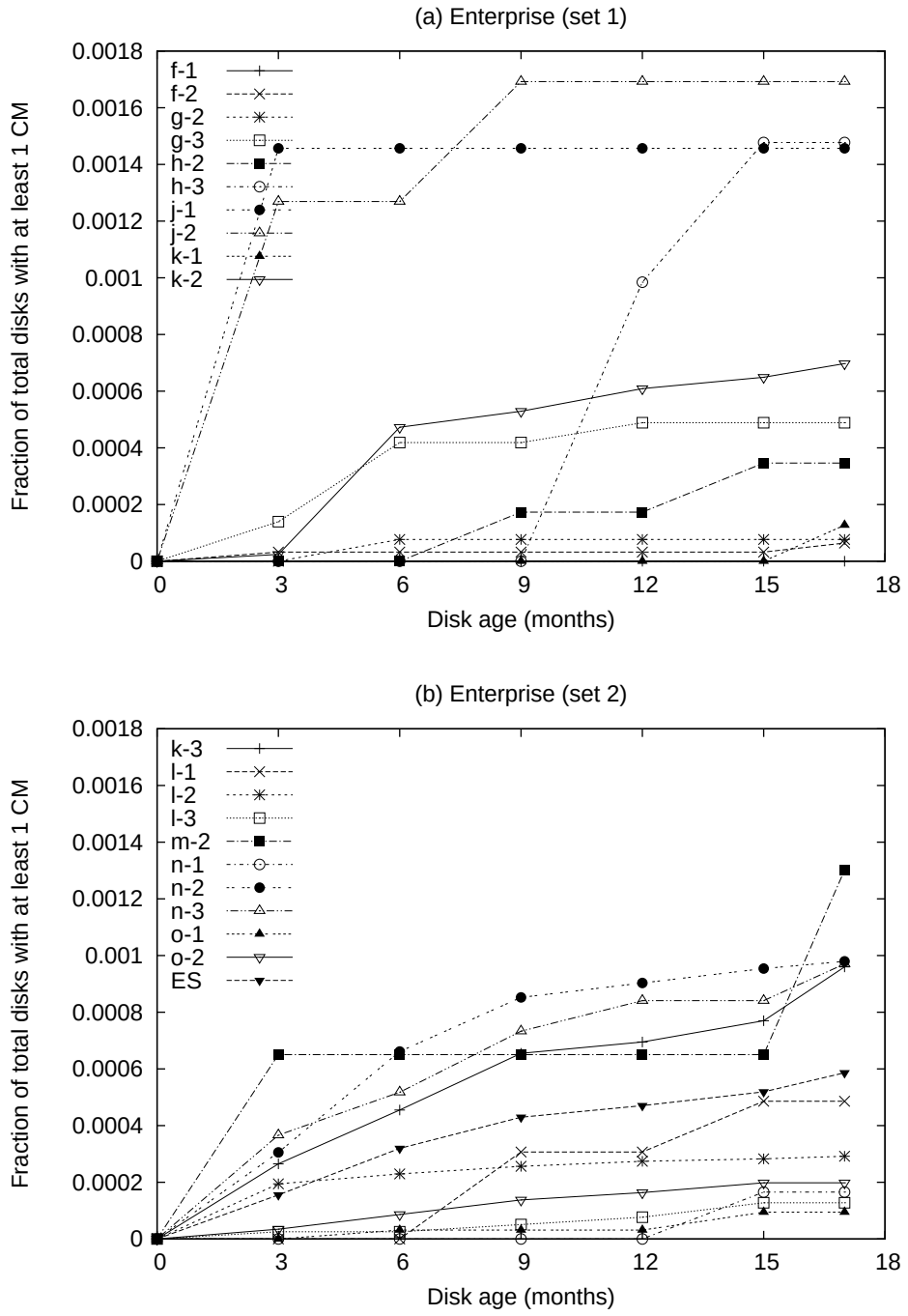

Figure 3.12 **Impact of disk age on enterprise disks.** *The fraction of disks that develop checksum mismatches as age increases is shown for enterprise disk models. Note that the fraction is cumulative.*

ES) indicates that only 0.06% of enterprise disks develop a mismatch during that time ($P(Y_{17} \geq 1)$ = 0.0006).

Observation 3.17 *The fraction of disks that develop checksum mismatches varies significantly across different disk models within the same disk class.*

We see in Figure 3.11 that there is an order of magnitude difference between models C-1 and E-2 for developing at least one checksum mismatch after 17 months; that is, $P(Y_{17} \geq 1)$ is 0.035 for C-1 and 0.0027 for E-2.)

Observation 3.18 *Age affects different disk models differently with respect to the fraction of disks that develop checksum mismatches.*

On average, as nearline disks age, the fraction of disks that develop a checksum mismatch is fairly constant, with some variation across the models. As enterprise disks age, the fraction that develop the first checksum mismatch decreases after about 6-9 months and then stabilizes.

3.4.2.2 Disk Size

Observation 3.19 *There is no clear indication that disk size affects the development of checksum mismatches.*

Figure 3.13 presents the fraction of disks that develop checksum mismatches within 17 months of their ship-date (*i.e.*, the rightmost data points from Figures 3.11 and 3.12; $P(Y_{17} \geq 1)$). The disk models are grouped within their families in increasing size. Since the impact of disk size on the fraction of disks that develop checksum mismatches is not constant across all disk families (it occurs in only 7 out of 10 families), we conclude that disk size does not necessarily impact the probability of developing checksum mismatches.

3.4.2.3 Workload

Observation 3.20 *There is no clear indication that workload affects the development of checksum mismatches.*

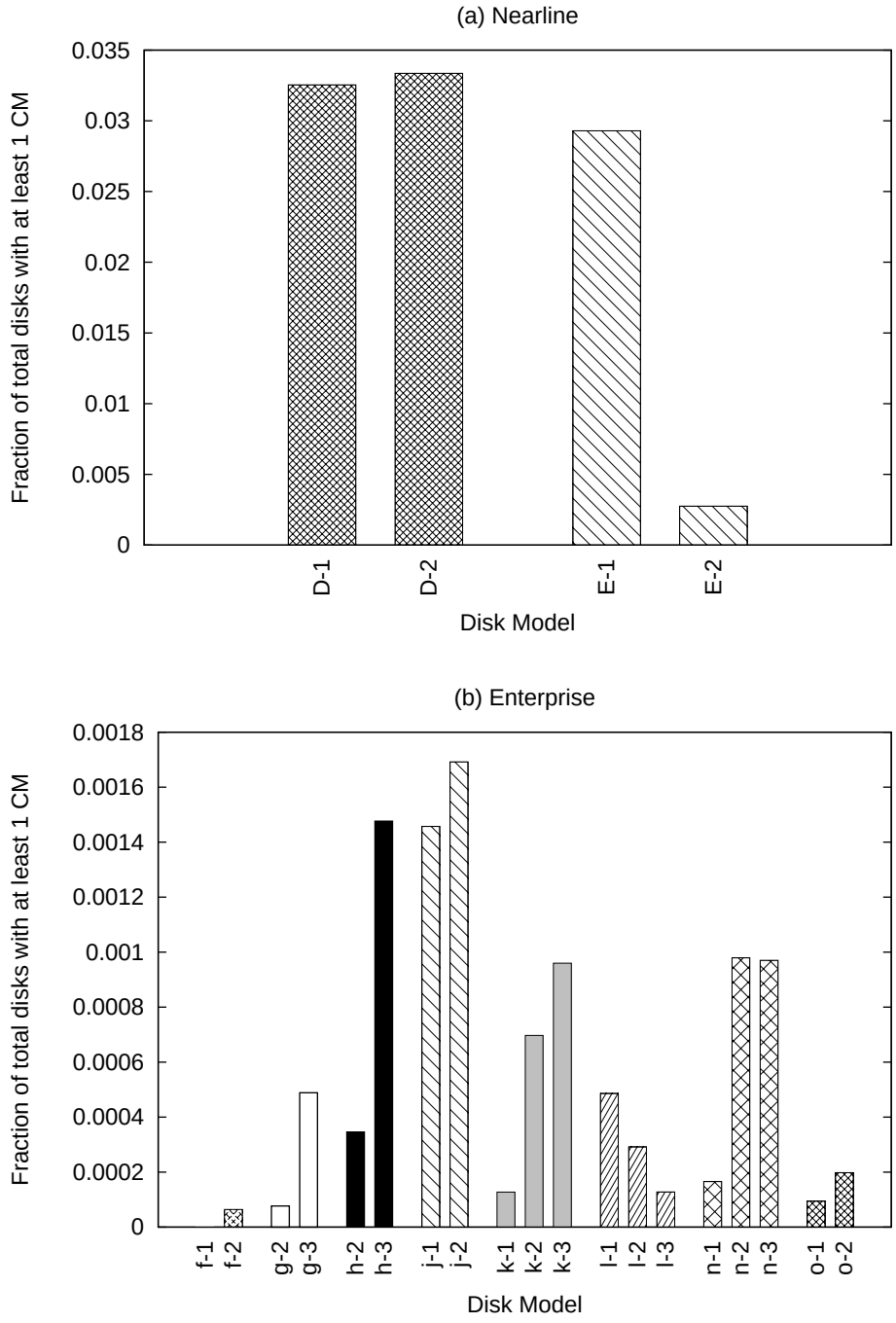

Figure 3.13 **The impact of disk size.** *The figures show the fraction of disks with at least one checksum mismatch within 17 months of shipping to the field for (a) nearline disk models, and (b) enterprise disk models.*

The systems in the study collect coarse workload data including the number of read and write operations, and the number of blocks read and written for each week of our study. To study the effect of workload on checksum mismatches, we computed the correlation coefficient between the workload data and the number of checksum mismatches observed in the system. We find that in all cases the correlation coefficient is less than 0.1 (in fact, in most cases less than 0.001), indicating no significant correlation between workload and checksum mismatches. However, these results might be due to having only coarse per-system rather than per-drive workload data. A system consists of at least 14 disks and can have as many as several hundred disks. Aggregating data across a number of disks might blur existing correlations between an individual drive's workload and corruption behavior.

3.4.3 Properties

In this subsection, we explore various characteristics of checksum mismatches. First, we analyze the number of mismatches developed by corrupt disks. Then, we examine whether mismatches are independent occurrences. Finally, we examine whether the mismatches have spatial or temporal locality.

3.4.3.1 Checksum Mismatches per Corrupt Disk

Figure 3.14 shows the cumulative distribution function of the number of checksum mismatches observed per corrupt disk (*i.e.*, the y-axis shows the fraction of corrupt disks that have fewer than or equal to y number of corrupt blocks). The figure can be represented as $P(Y_{17} \leq y | Y_{17} \geq 1)$ for $y = \{1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500, 1000\}$.

Observation 3.21 *The number of checksum mismatches per corrupt disk varies greatly across disks. Most corrupt disks develop only a few mismatches each. However, a few disks develop a large number of mismatches.*

Figure 3.14 shows that a significant fraction of corrupt disks (more than a third of all corrupt nearline disks and more than a fifth of corrupt enterprise disks) develop only one checksum mismatch.

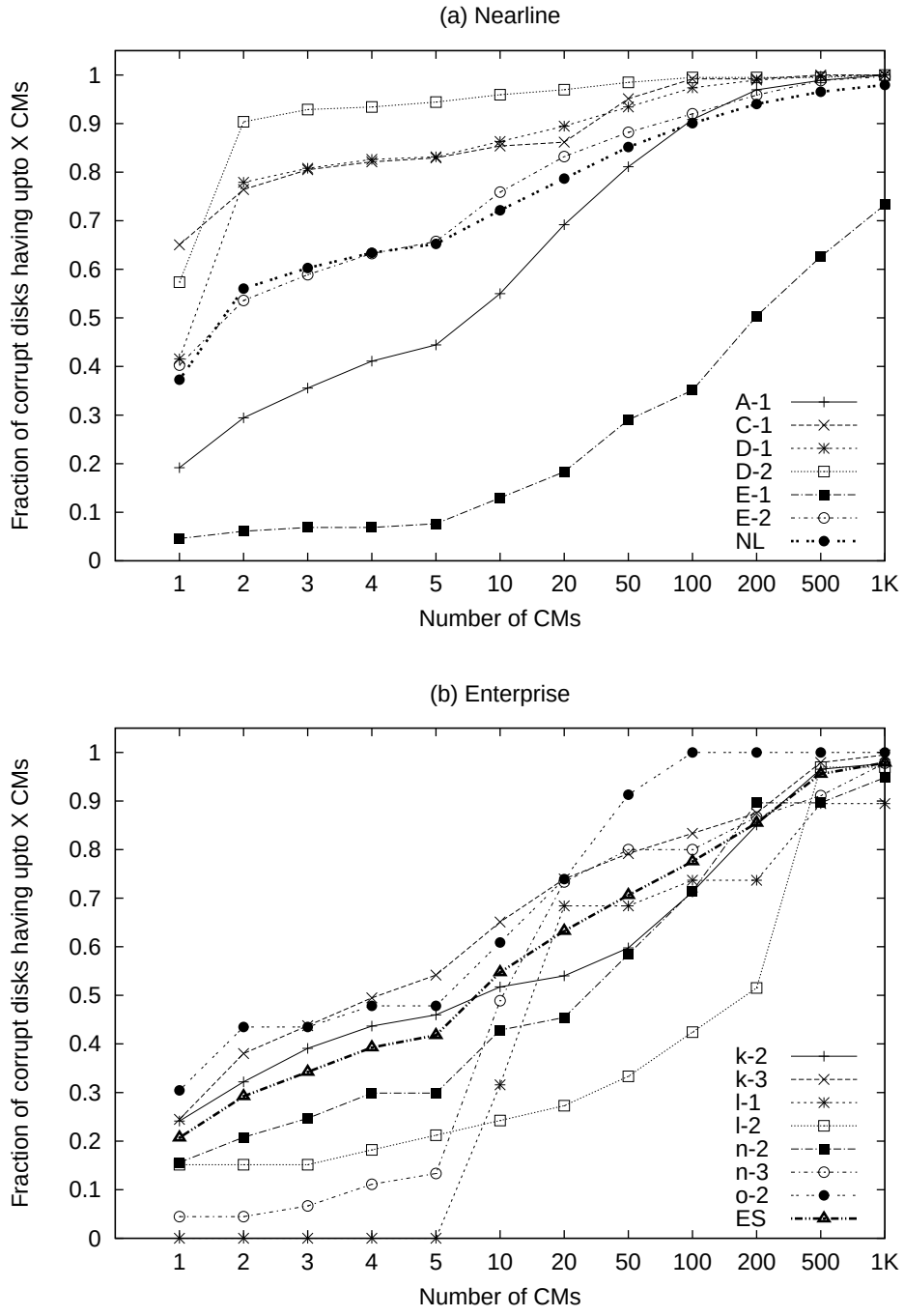

Figure 3.14 **Checksum mismatches per corrupt disk.** *The fraction of corrupt disks as a function of the number of checksum mismatches that develop within 17 months after the ship date for (a) nearline disk models and (b) enterprise disk models. Note that the x-axis is not linear in scale – the lines in the graph are used only to help distinguish the points of different disk models, and their slopes are not meaningful.*

On the other hand, a small fraction of disks develop several thousand checksum mismatches. The large variability in the number of mismatches per drive is also reflected in the great difference between the mean and median: while the median is only 3 mismatches per drive, the mean is 78.

A more detailed analysis reveals that the distributions exhibit heavy tails. A large fraction of the total number of checksum mismatches observed in our study is experienced by a very small fraction of the corrupt disks. More precisely, 1% of the corrupt disks (the top 1% corrupt disks with the largest number of mismatches) produce more than half of all mismatches recorded in the data.

Observation 3.22 *On average, corrupt enterprise disks develop many more checksum mismatches than corrupt nearline disks.*

Figure 3.14(a) (line NL) and Figure 3.14(b) (line ES) show that within 17 months 50% of corrupt disks (*i.e.*, the median) develop about 2 checksum mismatches for nearline disks, but almost 10 for enterprise disks. The trend also extends to a higher percentage of corrupt disks. For example, 80% of nearline corrupt disks have fewer than 20 mismatches, whereas 80% of enterprise disks have fewer than 100 mismatches. Given that very few enterprise disks develop checksum mismatches in the first place, in the interest of reliability and availability, it might make sense to replace the enterprise disk when the first mismatch is detected.

Observation 3.23 *Checksum mismatches within the same disk are not independent.*

We find that the fraction of disks that develop further checksum mismatches, given that a disk has at least one mismatch, is higher than the fraction of disks that develop the first mismatch in the same amount of time. For example, while the fraction of nearline disks that develop one or more checksum mismatches in 17 months is only 0.0066, the fraction developing more than 1 mismatch given that the disk already has one mismatch is as high as 0.6 (1 minus 0.4, the fraction of disks where exactly 1 block has a checksum mismatch in Figure 3.14).

Finally, it is interesting to note that nearline disk model E-1 is particularly aberrant – around 30% of its corrupt disks develop more than 1000 checksum mismatches.

3.4.3.2 System-Level Dependence

Observation 3.24 *The probability of a disk developing a checksum mismatch is not independent of that of other disks in the same storage system.*

While most systems with checksum mismatches have only one corrupt disk, we do find a considerable number of instances where multiple disks develop checksum mismatches within the same storage system. In fact, one of the systems in the study that used nearline disks had 92 disks develop checksum mismatches. Taking the maximum number of disks in the systems in the study into consideration, the probability of 92 disks developing errors independently is less than 1e-12, much less than 1e-05, the approximate fraction of systems represented by one system.

The observed dependence between disks in the same system is perhaps indicative of a common corruption-causing component, such as a shelf controller or adapter. In fact, NetApp™ engineers have observed instances of the SATA/FC adapter (a common component) causing data corruption in the case of disk models A-1, D-1 and D-2; therefore, it is also very likely that the statistics for these disk models are influenced by faulty shelf controllers.

3.4.3.3 Spatial Locality

We measure spatial locality by examining whether each corrupt block has another corrupt block (a *neighbor*) within progressively larger regions (*locality radius*) around it on the same disk. For example, if in a disk, blocks numbered 100, 200, and 500 have checksum mismatches, then blocks 100 and 200 have one neighbor at a locality radius of 100, and all blocks (100, 200, and 500) have at least one neighbor at a locality radius of 300.

Figure 3.15 shows the fraction of corrupt blocks that have at least one neighbor within different locality radii. Since a larger number of checksum mismatches will significantly skew the numbers, we consider only disks with 2 to 10 mismatches. The figure can be represented as $P(Y_t^r \geq 1 | 2 \leq Y_t \leq 10)$. Y^r is the number of corrupt blocks in block numbers $< a - r, a + r >$ around corrupt block a (but excluding a itself). The values for radius r are {1, 10, 100, ..., 100M} blocks, and $0 < t \leq 41$ months. The figure also includes a line *Random* that signifies the line that

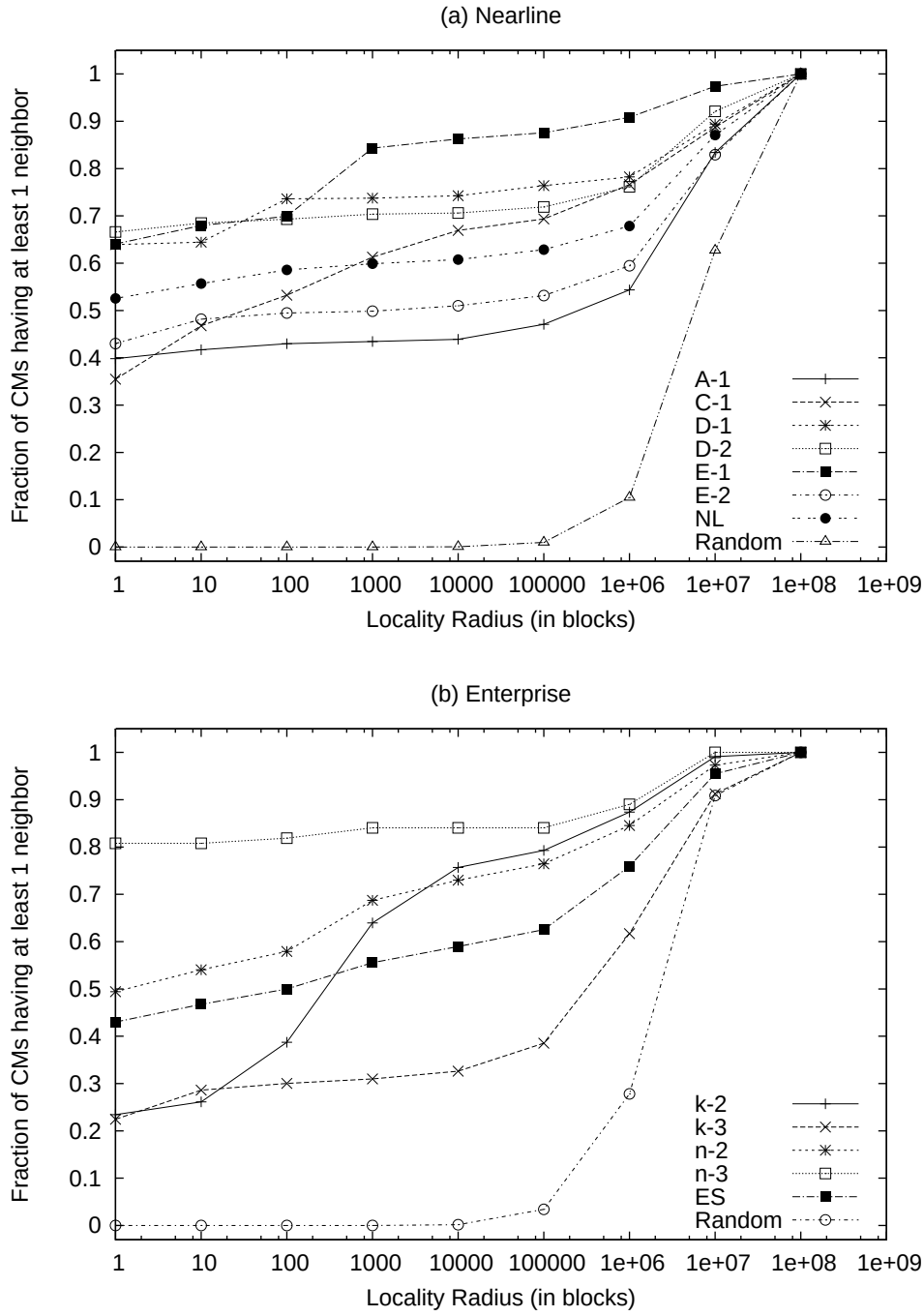

Figure 3.15 **Spatial Locality.** *The graphs show the fraction of checksum mismatches with another checksum mismatch within a given radius (disk-block range). Each figure also includes a line labeled "Random" corresponding to when the same number of mismatches (as nearline and enterprise respectively) are randomly distributed across the block address space. Only disks with between 2 and 10 mismatches are included.*

would be obtained if the checksum mismatches were randomly distributed across the block address space. This line can be used as a comparison point against the other lines. Note that this line is at 0 for most of the graph, signifying that there is no spatial locality for a random distribution.

For the actual data for the different disk models, we see that most disk models are much higher on the graph than *Random* when the x-axis value is 1; for more than 50% of the corrupt blocks in nearline disks and more than 40% of the corrupt blocks in enterprise disks, the immediate neighboring block also has a checksum mismatch (on disks with between 2 and 10 mismatches). These percentages indicate very high spatial locality.

Observation 3.25 *Checksum mismatches have very high spatial locality. Much of the observed locality is due to consecutive disk blocks developing corruption. Beyond consecutive blocks, the mismatches show very little spatial locality.*

We see from the figures that, while the lines for the disk models start at a very high value when the x-axis value is 1, they are almost flat for most of the graph, moving steeply upwards to 1 only towards the end (x-axis values more than 1e+06). This behavior shows that most of the spatial locality is due to consecutive blocks developing checksum mismatches. However, it is important to note that even when the consecutive mismatch cases are disregarded, the distribution of the mismatches still has spatial locality.

Given the strong correlation between checksum mismatches in consecutive blocks, it is interesting to examine the run length of consecutive mismatches, that is, how many consecutive blocks have mismatches. We find that, among drives with at least 2 checksum mismatches(and no upper bound on mismatches), on average 3.4 consecutive blocks are affected. In some cases, the length of consecutive runs can be much higher than the average. About 3% of drives with at least 2 mismatches see one or more runs of 100 consecutive blocks with mismatches. 0.7% of drives with at least 2 mismatches see one or more runs of 1000 consecutive mismatches.

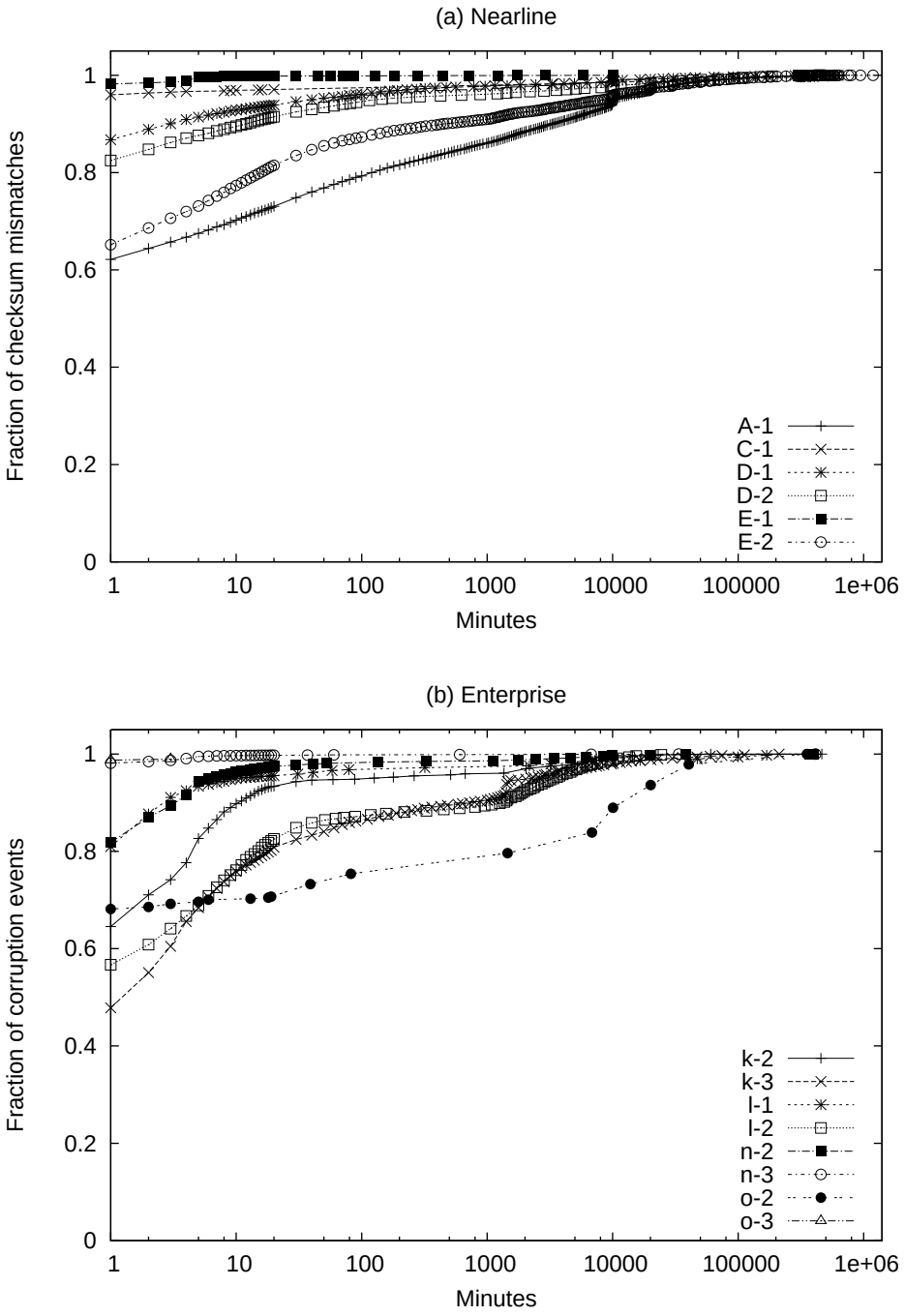

Figure 3.16 **Inter-arrival times.** *The graphs show the cumulative distribution of the inter-arrival times of checksum mismatches per minute. The fraction of mismatches per model is plotted against time. The arrival times are binned by minute.*

3.4.3.4 Temporal Locality

Figure 3.16 shows the fraction of checksum mismatches that arrive (are detected) within x minutes of a previous mismatch. The figure can be represented as $P(Y_{t+x} \geq k+1 | Y_t = k \wedge Y_T \geq k+1)$ for $k \geq 1$, $0 \leq t < T \leq$ 41 months, and $1 \leq x \leq$ 1e+06 minutes.

Observation 3.26 *Most checksum mismatches are detected within one minute of a previous detection of a mismatch.*

The figure shows that the temporal locality for detecting checksum mismatches is extremely high. This behavior may be an artifact of the manner in which the detection takes place (by scrubbing) and the fact that many mismatches are spatially local and are therefore likely to be discovered together. Further analysis shows that this is not necessarily the case.

In order to remove the impact of detection time, we examine temporal locality over larger time windows. For each drive, we first determine the number of checksum mismatches experienced in each 2-week time window that the drive was in the field and then compute the autocorrelation function on the resulting time series. The autocorrelation function (ACF) measures the correlation of a random variable with itself at different time lags l. The ACF can be used to determine whether the number of mismatches in one two-week period of our time-series is correlated with the number of mismatches observed l 2-week periods later. The autocorrelation coefficient can range between 1 (high positive correlation) and -1 (high negative correlation). A value of zero would indicate no correlation, supporting independence of checksum mismatches.

Observation 3.27 *Checksum mismatches exhibit temporal locality over larger time windows and beyond the effect of detection time as well.*

Figure 3.17 shows the resulting ACF. The graph presents the average ACF across all drives in the study that were in the field for at least 17 months and experienced checksum mismatches in at least two different 2-week windows. Since the results are nearly indistinguishable for nearline and enterprise drives, individual results are not given. If checksum mismatches in different 2-week periods were independent (no temporal locality on bi-weekly and larger time-scales) the graph

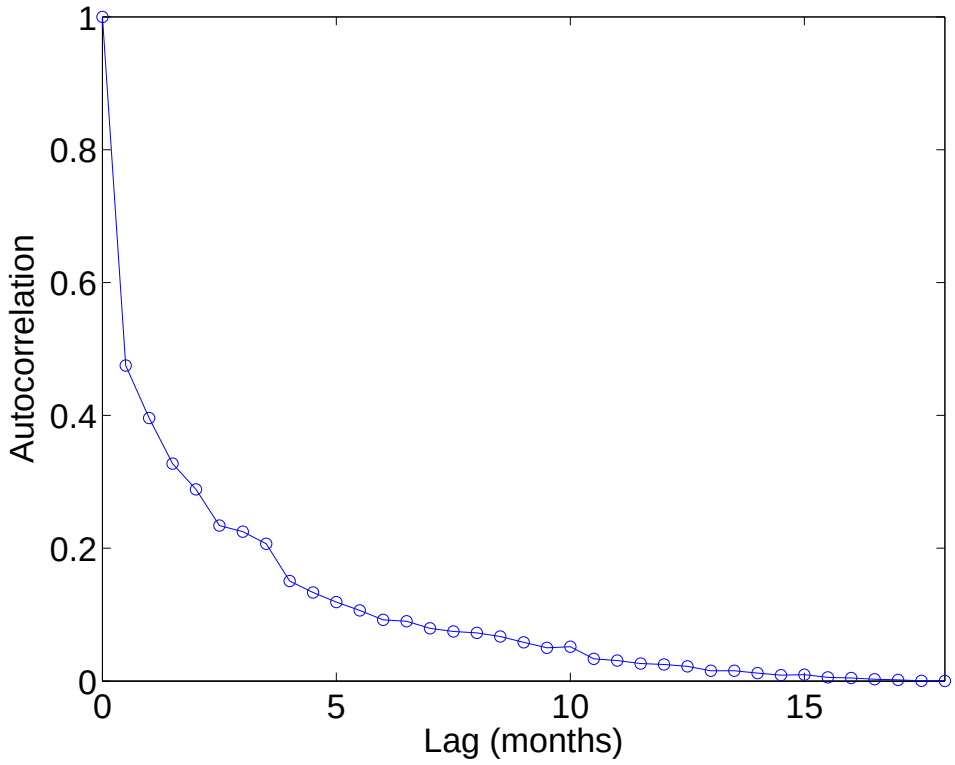

Figure 3.17 **Temporal autocorrelation.** *The graph shows the autocorrelation function for the number of checksum mismatches per 2-week time windows. This representation of the data allows us to study temporal locality of mismatches at larger time-scales without being affected by the time of detection.*

would be close to zero at all lags. Instead we observe strong autocorrelation even for large lags in the range of up to 10 months.

3.4.4 Correlations

We now establish correlations for checksum mismatches with other errors such as system resets, latent sector errors, and not-ready-condition errors.

Observation 3.28 *Checksum mismatches correlate with system resets.*

The fraction of systems that experience a system reset at some point of time, given that one of the disks in the system has a checksum mismatch, is about 3.7 times the unconditional fraction of systems that experience a system reset.

Observation 3.29 *There is a weak positive correlation between checksum mismatches and latent sector errors.*

The fraction of disks that develop latent sector errors, $P(X_t \geq 1)$, is 0.137 for nearline disks and 0.026 for enterprise disks (X is the number of latent sector errors, $0 < t \leq 41$ months). The fraction of disks that develop a latent sector error out of disks that also have a checksum mismatch, $P(X_t \geq 1|Y_t \geq 1)$, is 0.195 for nearline disks and 0.0556 for enterprise disks, which are 1.4 times and 2.2 times that of the unconditional fractions. These values indicate a weak positive correlation between the two disk errors.

In order to test the statistical significance of this correlation we performed a chi-square test for independence. We find that we can, with high confidence, reject the hypothesis that checksum mismatches and latent sector errors are independent, both in the case of nearline disks and enterprise disks (confidence level of more than 99.999%). Interestingly, the results vary if we repeat the chi-square test separately for each individual disk model (including only models that had at least 15 corrupt disks). We can reject independence with high certainty (at least 95% confidence) for only four out of seven nearline models (B-1, C-1, D-1, E-2) and two out of seven enterprise models (l-1, n-3).

Observation 3.30 *There is a weak correlation between checksum mismatches and not-ready-condition errors.*

The probability of a disk developing not-ready-condition errors, $P(Z_t \geq 1)$, is 0.18 for nearline and 0.03 for enterprise disks. $P(Z_t \geq 1|Y_t \geq 1)$ is 0.304 for nearline and 0.0155 for enterprise disks. Thus, the conditional probability of a not-ready-condition error, given that a disk has checksum mismatch, is about 1.7 times the unconditional probability of a not-ready-condition error in the case of nearline disks and about 0.5 times the unconditional probability for enterprise disks. These values indicate mixed behavior – a weak positive correlation for nearline disks and a weak negative correlation for enterprise disks.

In order to test the statistical significance of the correlation between not-ready-condition errors and checksum mismatches, we again perform a chi-square test for independence. We find that for both nearline and enterprise disks we can reject the hypothesis that not-ready-condition errors and checksum mismatches are independent with more than 96% confidence. We repeat the same test separately for each disk model (including only models that had at least 15 corrupt disks). In the case of nearline disks, we can reject the independence hypothesis for all models, except for two (A-1 and B-1) at the 95% confidence level. However, in the case of enterprise disks, we cannot reject the independence hypothesis for any of the individual models at a significant confidence level.

3.4.5 Detection

Figure 3.18 shows the distribution of requests that detect checksum mismatches into different request types. There are five types of requests that discover checksum mismatches: (i) Reads by the file system (FS Read) (ii) Partial RAID-stripe writes by the RAID layer (Write) (iii) Reads for disk-copy operations (Non-FS Read) (iv) Reads for data scrubbing (Scrub), and (v) Reads performed during RAID reconstruction (Reconstruction). Note that these request types are different (and more specific) from those for latent sector errors in Section 3.3.5 since the low-level error messages for latent sector errors do not differentiate between the source of different read operations for file system, disk copy, and reconstruction.

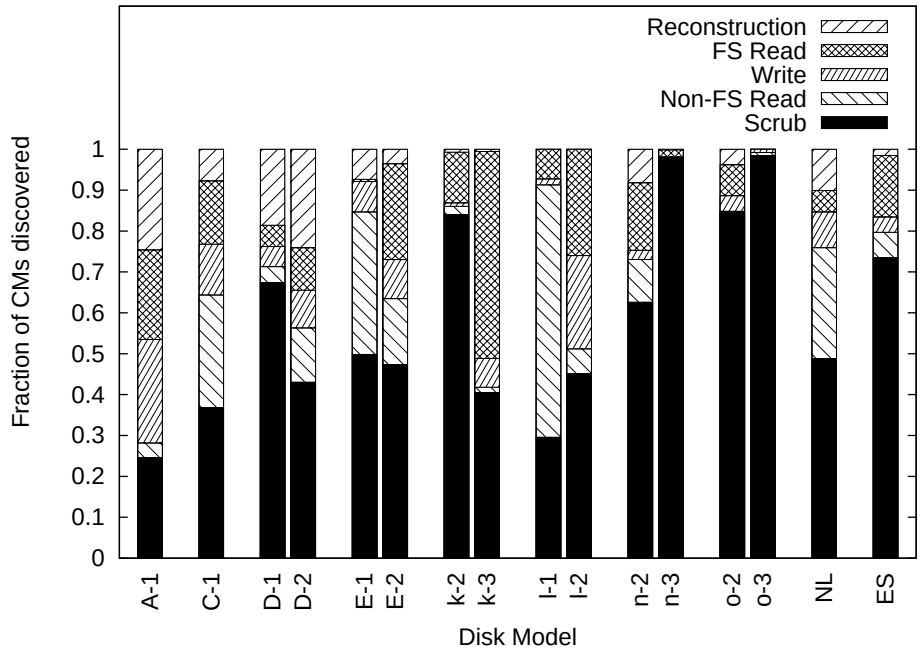

Figure 3.18 **Detection.** *The figure shows the distribution of requests that discover checksum mismatches across the request types scrub, non-file sytstem read (say, disk copy), write (of partial RAID stripe), file system read, and RAID reconstruction.*

Observation 3.31 *Data scrubbing discovers a large percentage of the checksum mismatches for many of the disk models.*

We see that on the average data scrubbing discovers about 49% of checksum mismatches in nearline disks (NL in the figure), and 73% of the checksum mismatches in enterprise disks (ES in the figure). It is quite possible that these checksum mismatches may not have been discovered in the absence of scrubbing, potentially exposing the system to double failures and data loss. We do not know the precise cause for the disparity in percentages between nearline and enterprise disks; one possibility this data suggests is that systems with nearline disks perform many more disk-copy operations (Non-FS Read), thus increasing the percentage for that request type.

Observation 3.32 *RAID reconstruction encounters a non-negligible number of checksum mismatches.*

Despite the use of data scrubbing to avoid double failures, we find that RAID reconstruction discovers about 8% of the checksum mismatches in nearline disks. For some models more than 20% of checksum mismatches were detected during RAID reconstruction. This observation implies that (a) data scrubbing should be performed more aggressively, and (b) systems should consider protection against double disk failures [3, 22, 35, 57, 59, 98].

3.4.6 Block-Specific Corruption

We find that specific block numbers could be much more likely to experience corruption than other block numbers. This behavior is observed for the disk model E-1. Figure 3.19 presents for each block number, the number of disk drives of disk model E-1 that developed a checksum mismatch at that block number. We see in the figure that many disks develop corruption for a specific set of block numbers. We also verified that (i) other disk models did not develop multiple checksum mismatches for the same set of block numbers (ii) the disks that developed mismatches at the same block numbers belong to different storage systems, and (iii) the software stack of the storage system has no specific data structure that is placed at the block numbers of interest.

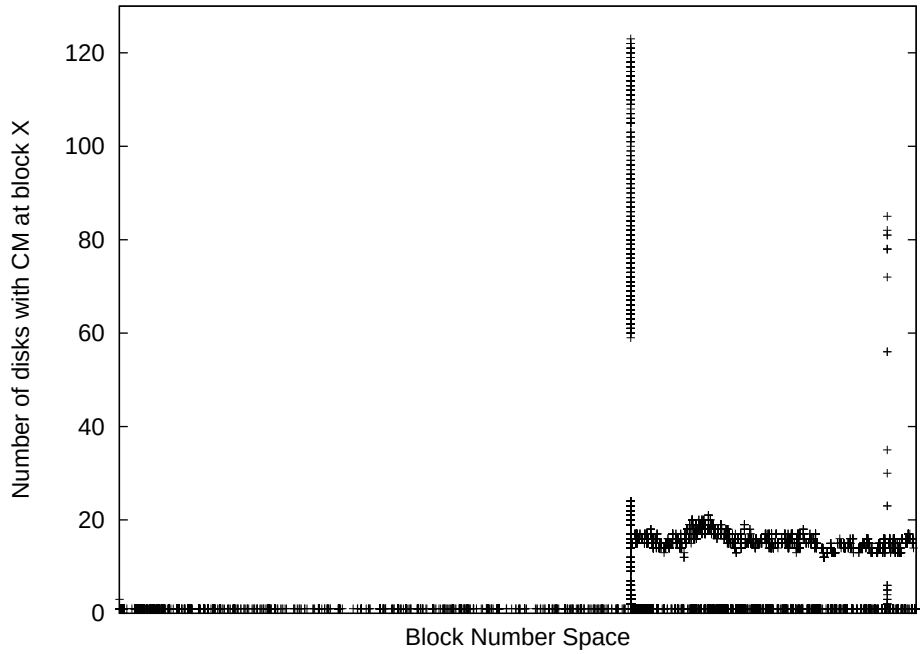

Figure 3.19 **Distribution of errors across block numbers.** *For each disk block number, the number of disks of disk model E-1 that develop checksum mismatches at that block number is shown. The units on the x-axis have been omitted in order to anonymize the disk size of disk model E-1.*

These observations indicate that hardware or firmware bugs that affect specific sets of block numbers might exist. Therefore, RAID-system designers may consider using *staggered* stripes such that the blocks that form a stripe (providing the required redundancy) are placed at different block numbers on different disks.

We also observed a number of block-specific errors on other drive models. In at least one of these instances, the block contained a heavily read and written file system metadata structure – a structure akin to the superblock. This suggests the importance of replicating important metadata structures [104, 126].

3.4.7 Identity Discrepancies

These corruptions were detected in a total of 365 disks out of the 1.53 million disks. Figure 3.20 presents the fraction of disks of each disk model that developed identity discrepancies in 17 months. We see that the fraction is more than an order of magnitude lower than that for checksum mismatches for both nearline and enterprise disks.

Since the fraction of disks that develop identity discrepancies is very low, the system recommends replacement of the disk once the first identity discrepancy is detected. It is important to note, that even though the number of identity discrepancies are small, silent data corruption would have occurred if not for the validation of the stored contextual file system information (the use of this em logical-identity information will be analyzed in Section 4.3.5).

3.4.8 Parity Inconsistencies

These corruptions are detected by data scrubbing. In the absence of a second parity disk, one cannot identify which disk is at fault. Therefore, in order to prevent potential data loss on disk failure, the system fixes the inconsistency by rewriting parity. This scenario provides further motivation for double-parity protection schemes.

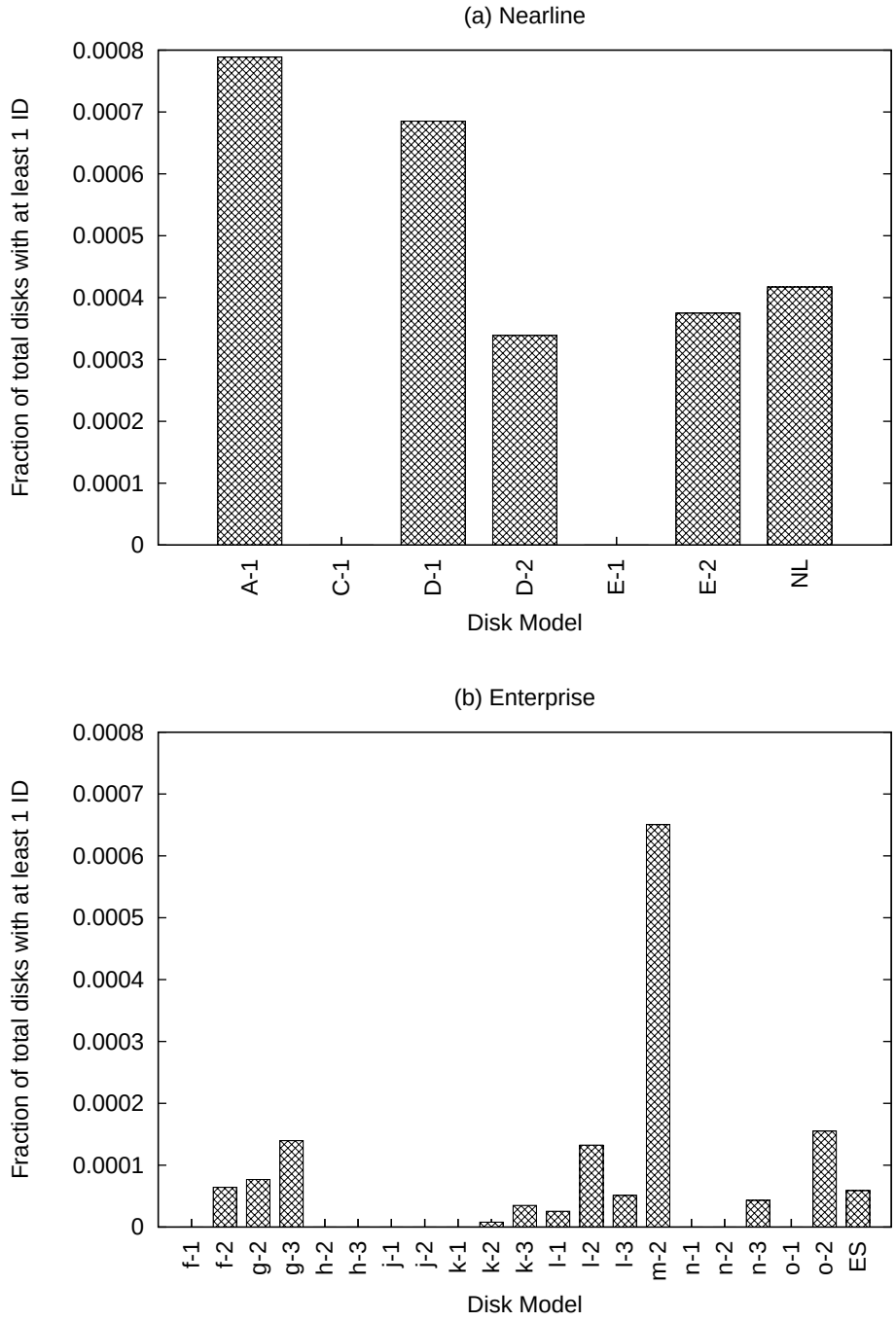

Figure 3.20 **Identity discrepancies.** *The figures show the fraction of disks with at least one identity discrepancy within 17 months of shipping to the field for (a) nearline disk models, and (b) enterprise disk models.*

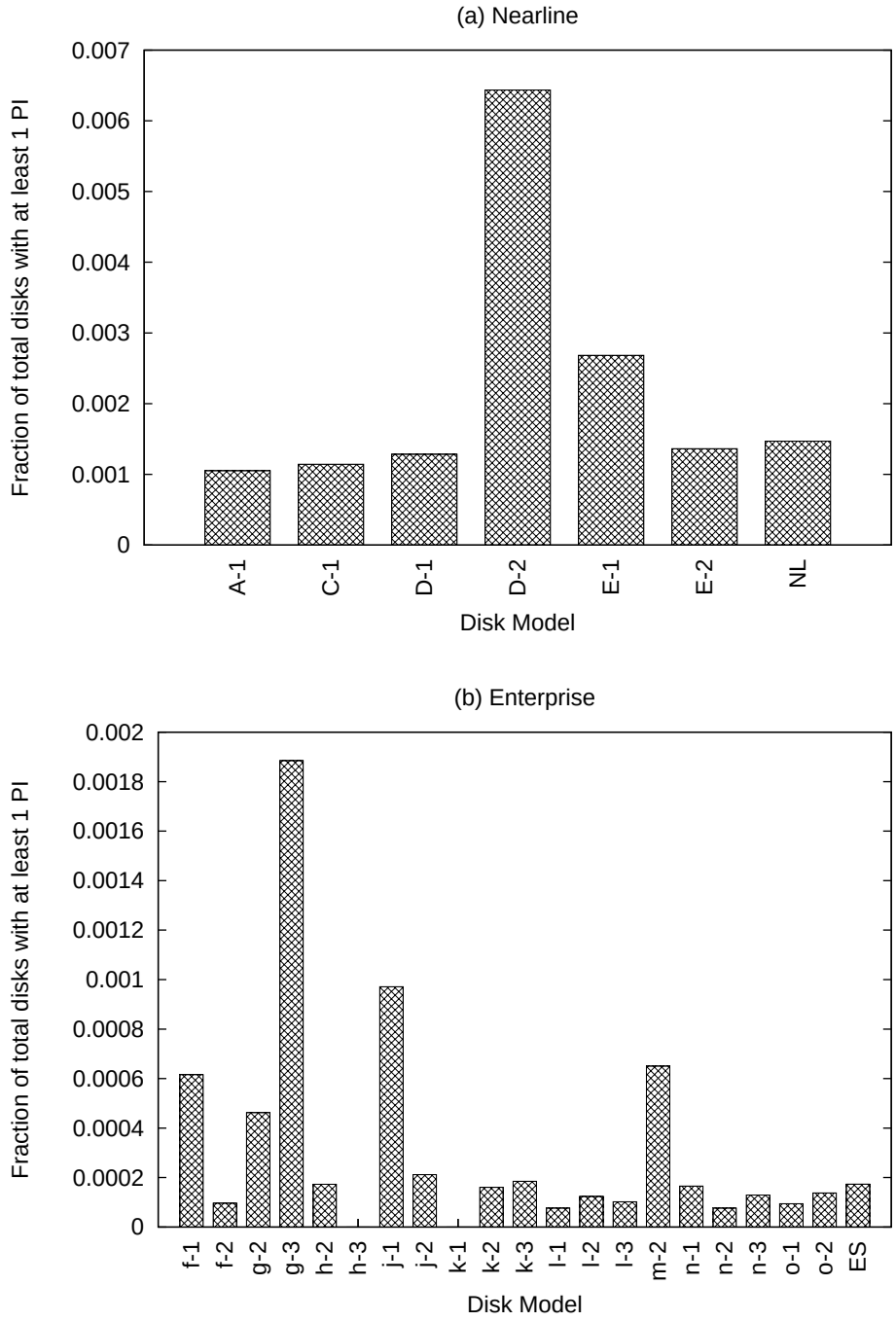

Figure 3.21 **Parity inconsistencies.** *The figures show the fraction of disks with at least one parity inconsistency within 17 months of shipping to the field for (a) nearline disk models, and (b) enterprise disk models.*

Figure 3.21 presents the fraction of disks of each disk model that caused parity inconsistencies within 17 months since ship date. The fraction is 4.4 times lower than that for checksum mismatches in the case of nearline disks and about 3.5 times lower than that for checksum mismatches for enterprise disks.

These results assume that the parity disk is at fault. We believe that counting the number of incorrect parity disks reflect the actual number of error disks since: (i) entire shelves of disks are typically of the same age and same model, (ii) the incidence of these inconsistencies is quite low; hence, it is unlikely that multiple different disks in the same RAID group would be at fault.

3.5 Discussion

In this section, we first compare the characteristics of latent sector errors and data corruption identified by checksum mismatches. Next, we use results from our analysis of latent sector errors and data corruption to develop lessons on how storage systems can be designed to deal with corruption.

3.5.1 Latent Sector Errors vs. Checksum Mismatches

Table 3.1 compares the characteristics of latent sector errors and checksum mismatches. Some of the interesting similarities and differences are as follows.

Frequency: The fraction of disks that develop checksum mismatches is about an order of magnitude smaller than that for latent sector errors. However, given that the enterprise storage world uses millions of disk drives, it is important to handle both kinds of partial disk failures. Also, latent sector errors are more likely to be detected by system software, since an actual error is reported by the disk drive; since data corruptions are silent, they may pose a bigger threat to data, especially on systems without checksumming infrastructure.

Disk model: The fraction of disks affected by both kinds of partial disk failures varies greatly by disk model. Interestingly, in comparing disk models across the two partial disk

Characteristic	Latent sector errors		Checksum mismatches	
	Nearline	**Enterprise**	**Nearline**	**Enterprise**
% disks affected per year (avg)	9.5%	1.4%	0.466%	0.042%
Disk age ⇑, P(1st error)	⇑	⇔	Varies	Varies
Disk size ⇑, P(1st error)	⇑	⇑	Varies	Varies
No. of errors per disk with errors	Low	Low	Low	Low
Are errors independent ?	No	No	No	No
Spatial locality	$\surd$	$\surd$	$\surd$	$\surd$
Temporal locality	$\surd$	$\surd$	$\surd$	$\surd$
Correlations with not-ready conditions	$+$	No	$+$	$-$
Correlations with recovered errors	No	$+$	No	Unknown
Correlations with system resets	Unknown	Unknown	$+$	$+$
Activity that detects most errors	Media scrub	Media scrub	Data scrub	Data scrub

Table 3.1 **Comparison of latent sector errors and checksum mismatches.** *This table compares our findings for latent sector errors and checksum mismatches for both nearline and enterprise disk models. We look at factors that affect error rates, characteristics of errors, how the error is detected, and correlations with other errors. In addition to these correlations, latent sector errors and checksum mismatches share a positive correlation for both nearline and enterprise disks. The symbols used are as follows.* $\Uparrow$ *for increases,* $\Leftrightarrow$ *for remaining constant,* $\surd$ *to confirm existence,* $+$ *for positive correlation, and* $-$ *for negative correlation.*

failures we find that the nearline disk model E-2 has the highest percentage of disks developing latent sector errors, but the lowest percentage of disks developing checksum mismatches within the set of nearline disk models, despite latent sector errors and checksum mismatches having a positive correlation (Section 3.4.4).

Impact of disk class: For both latent sector errors and checksum mismatches, enterprise disks are less likely to develop an error than nearline disks. Surprisingly, however, in both cases, once an error has developed, enterprise disks develop a higher number of errors than nearline disks.

Spatial locality: Both latent sector errors and checksum mismatches show high spatial locality. Interestingly, the difference in the locality radii that capture a large fraction of errors – about 10 MB for latent sector errors versus consecutive blocks for checksum mismatches – provides an insight into how the two errors could be caused very differently. Latent sector errors may be caused by media scratches that could go across tracks as opposed to consecutive sectors (hence a larger locality radius) while consecutive blocks may have checksum mismatches simply because the corruption(s) occurred when they were written together or around the same time.

3.5.2 Lessons Learned

We present some of the lessons learned from the analysis. Some of these lessons are specific to RAID systems, while others can be applied to file systems as well.

- Latent sector errors affect a significant percentage of disk drives. They affect up to 20% of the drives of one of the SATA disk models. Such a high percentage implies a need for maintaining redundant information to protect against data loss. Single-disk systems should strive for intra-disk redundancy, perhaps in the form of replicated file-system metadata, while RAID systems should consider protecting against double disk failures [3, 22, 35, 57, 59, 98]

- Albeit not as common as latent sector errors, data corruption does happen; we observed more than 400,000 cases of checksum mismatches. For some drive models as many as 4%

of drives develop checksum mismatches during the 17 months examined. Similarly, even though they are rare, identity discrepancies and parity inconsistencies do occur. Protection offered by checksums and block identity information is therefore well-worth the extra space needed to store them.

- A significant number (8% on average) of corruptions are detected during RAID reconstruction, creating the possibility of data loss. In this case, protection against double disk failures [3, 22, 35, 57, 59, 98] is necessary to prevent data loss. More aggressive scrubbing can speed the detection of errors, reducing the likelihood of an error during a reconstruction.

- Although, the fraction of disks that develop corruption is lower for enterprise drives, once they develop one corruption, many more are likely to follow. Therefore, replacing an enterprise drive on the first detection of a corruption might make sense (drive replacement cost may not be a huge factor since the probability of first corruption is low).

- Some block numbers are much more likely to be affected by corruption than others, potentially due to hardware or firmware bugs that affect specific sets of block numbers. RAID system designers might consider using *staggered* stripes such that the blocks that form the stripe are not stored at the same or nearby block number.

- Strong spatial locality for both latent sector errors and checksum mismatches suggests that redundant data structures should be stored distant from each other.

- The high degree of spatial and temporal locality for checksum mismatches also begs the question of whether many corruptions occur at the exact same time, perhaps when all blocks are written as part of the same disk request. This hypothesis suggests that important or redundant data structures that are used for recovering data on corruption should be written as part of different write requests spaced over time.

- Strong spatial and temporal locality (over long time periods) also suggests that it might be worth investigating how the locality can be leveraged for smarter scrubbing; for example, one can trigger a scrub before it's next scheduled time, when probability of latent sector

errors or corruption is high or perform *selective* scrubbing of an area of the drive that's likely to be affected.

- Failure prediction algorithms in systems should take into account the correlation of latent sector errors and corruption with other errors such as not-ready-condition errors and with each other, increasing the probability of one error when an instance of the other is found.

3.6 Conclusion

Our large-scale study of partial disk failures shows that partial disk failures do occur. In fact, latent sector errors affect an alarming number of disk drives for some disk drive models. Likewise, data corruption affects around 3% of disk drives within just a year and a half for many nearline disk models, which are the kind used in commodity systems like our desktops and laptops. Further, the use of more expensive enterprise drives does not eliminate data corruption. Therefore, it is essential that the storage stack be able to detect and recover from partial disk failures.

The analysis of the characteristics of partial disk failures provides us with insights that can be used to build and evaluate reliable storage systems. In particular, data protection techniques should factor in the spatial and temporal locality of partial disk failures. The analysis also shows that disk scrubbing techniques could play a vital role in proactively detecting partial disk failures.

There is a rich space for future work on characterizing partial disk failures. Specifically, future studies could focus on the impact of factors such as environment and workload. These avenues for future studies are discussed in Section 9.3.1.

Chapter 4

Impact on RAID Systems

RAID (Redundant Array of Independent Disks) stores data on multiple disks in a redundant fashion in order to survive the failure of one or more of the disks [101]. Since it was originally proposed, it has been employed in nearly every enterprise storage system [43, 65, 68, 94].

RAID is specifically targeted towards handling disk failures; therefore, one would expect a thorough and verifiable failure-handling scheme. With simple disk failures (*e.g.*, an entire disk failing in a fail-stop fashion), designing such protection schemes to cope with disk failures is not overly challenging. For example, early systems successfully handle the failure of a single disk through the use of mirroring or parity-based redundancy schemes [21, 81, 101]. Although getting an implementation to work correctly may be difficult (often involving hundreds of thousands of lines of code [146]), one could feel confident that the design properly handles the expected failures.

Unfortunately, as the data in Chapter 3 shows, storage systems today are confronted with a much richer landscape of disk failures involving latent sector errors and silent data corruptions. These partial disk failures, especially silent data corruptions, considerably complicate the construction of correctly-designed protection strategies.

A number of techniques have been developed and used in enterprise RAID systems to cope with silent data corruptions. For example, various forms of checksumming can be used to detect corruption [18, 129]; combined with redundancy (*e.g.*, mirrors or parity), checksumming enables both the detection of and recovery from certain classes of corruptions. However, given the broad range of techniques used (including sector checksums [18, 37, 65], block checksums [131], parental checksums [130], write-verify operations [131], identity information [107, 131], and disk

scrubbing [37, 119, 130, 131], to list a few), exactly which techniques protect against which failures is sometimes unclear; worse, combining different approaches in a single system may lead to unexpected gaps in data protection.

We propose an approach based on model checking [73] to analyze the design of protection schemes in current-day storage systems. We develop and apply a simple *model checker* to examine different data protection schemes. We first implement a simple logical version of the protection scheme under test; the model checker then applies different sequences of read, write, and failure events to exhaustively explore the state space of the system, either producing a chain of events that lead to data loss or a "proof" that the scheme works as desired.

We use the model checker to evaluate a number of different approaches found in real RAID systems, focusing on schemes that use one parity block per RAID stripe. We analyze how they detect and recover from a single partial disk failure and find holes in all of the schemes examined, where systems potentially exposes data to loss or returns corrupt data to the user. We find that many of these systems suffer from a general problem that we call *parity pollution*, wherein corruption to a disk block on a data disk can spread to the parity disk, thereby rendering the data unrecoverable. We use our analysis to construct a protection scheme to address all issues we discover including parity pollution; this scheme uses several techniques including block checksums, both logical and physical identity information, and version mirroring.

With analyses of each scheme in hand, we also show how a system designer can combine real data of failure probability (from our study of the characteristics of partial disk failures in Chapter 3) with our model checker's results to arrive upon a final estimation of data-loss probability. Doing so enables one to compare different protection approaches and determine which is best given the current environment.

The rest of the chapter is structured as follows. Section 4.1 briefly discusses the evolution of data protection in real systems. Section 4.2 describes our approach to model checking and Section 4.3 presents the results of using the model checker to deconstruct a variety of protection schemes. Section 4.4 presents the results of our analysis of data-loss probability for each scheme and Section 4.5 concludes the chapter.

4.1 Enterprise Data Protection

Protection techniques in real systems have evolved greatly over time. Early multiple disk systems focused almost solely on recovery from entire disk failures; detection was performed by the controller, and redundancy (*e.g.*, mirrors or parity) was used to reconstruct data on the failed disk [32].

As disk drives became bigger, faster, and cheaper, new and interesting failure modes began to appear, and storage-system vendors added techniques to tackle the new failures. For example, NetApp™ recently added protection against lost writes [131]. Many other systems do not (yet) have such protections, and the importance of such protection is difficult to gauge. This anecdote serves to illustrate the organic nature of data protection. While it would be optimal to simply write down a set of assumptions about the fault model and then design a system to handle the expected failures, in practice such an approach is not practical. Disks (and other storage subsystem components) provide an ever-moving target; tomorrow's disk failures may not be present today. Worse, as new problems arise, they must be incorporated into existing schemes, rather than attacked from first principles. This aspect of data protection motivates the need for a formal and rigorous approach to help understand the exact protection offered by combinations of techniques.

Table 4.1 shows the protection schemes employed by a range of systems. Although the table may be incomplete (*e.g.*, a given system may use more than the protections we list, as we only list what is readily made public via published papers, web sites, and documentation), it hints at the breadth of approaches employed as well as the on-going development of protection techniques. We discuss each of these techniques in more detail in Section 4.3, where we use our model checker to determine their efficacy in guarding against partial disk failures.

Depending on the protection techniques in place, partial disk failures may have one or more of the following outcomes:

Data recovery: The scenario where the protection scheme detects the partial disk failure and uses parity to successfully recover data.

System	RAID	Scrubbing	Sector Checksums	Block Checksums	Parent Checksums	Write-Verify	Physical Identity	Logical Identity	Version Mirroring	Other
Hardware RAID card (*e.g.*, Adaptec™ 2200 S [1])	√									
Linux software RAID [47, 97]	√	√								
Pilot [107]								√		√
Tandem NonStop™ [18]	√		√				√			
Dell™ Powervault™ [37]	√	√	√							√
Hitachi Thunder 9500™ [65, 66]	√		√			√				
NetApp Data ONTAP™ [94, 131]	√	√		√		√	√	√		
ZFS [130] with RAID-4	√	√			√					

Table 4.1 **Protection techniques in real systems.** *This table shows the known protection techniques used in real-world systems. Some systems have additional protection techniques: Pilot uses a scavenger routine to recover metadata, and Powervault uses a 1-bit "write stamp" and a timestamp value to detect data-parity mismatches. Systems may use further protection techniques whose details not made public.*

Data loss: The scenario where the protection scheme detects the partial disk failure, but is unable to successfully recover data. In this case, the storage system reports an error to the user.

Corrupt data: The scenario where the protection scheme does not detect the partial disk failure and therefore returns corrupt data to the user.

4.2 Model Checking

We have developed a simple model checker to analyze the design of various data protection schemes. The goal of the model checker is to identify all execution sequences, consisting of *user-level operations*, *protection operations*, and *a single partial disk failure*, that can lead to either data loss or corrupt data being returned to the user. The model checker exhaustively evaluates all possible states of a *single RAID stripe* by taking into account the effects of all possible operations and partial disk failures for each state.

We have chosen to build our own model checker instead of using an existing one since it is easier to build a simple model checker that is highly specific to RAID data protection; for example, the model checker assumes that the data disks are interchangeable, thereby reducing the number of unique states. However, there is no fundamental reason why our analysis cannot be performed on a different model checker.

Models for the model checker are built on top of some basic primitives. A *RAID stripe* consists of N disk blocks where the contents of each disk block is defined by the model using primitive components consisting of user data entries and protections. Since both the choice of components and their on-disk layout affect the data reliability, the model must specify each block as a series of entries (corresponding to sectors within a block). Each entry can be atomically read or written.

The model checker assumes that the desired unit of consistency is one disk block. All protection schemes are evaluated with this assumption as a basis.

4.2.1 Model-Checker Primitives

The model checker provides the following primitives for use by the protection scheme:

Disk operations: The conventional operations disk read and disk write are provided. These operations are atomic for each entry (sector) and not over multiple entries that form a disk block.

Data protection: The model checker and the model in conjunction implement various protection techniques. The model checker uses model-specified knowledge of the protections to evaluate different states. For example, the result of checksum verification is part of the system state that is maintained by the model checker. Protections like parity and checksums are modeled in such a way that "collisions" do not occur; we wish evaluate the spirit of the protection, not the choice of hash function.

The model defines operations such as user read and user write based on the model checker primitives. For instance, a user write that writes a part of the RAID stripe will be implemented by the model using disk read and disk write operations, parity calculation primitives, and protection checks.

4.2.2 Modeling Partial Disk Failures

The model checker injects exactly one partial disk failure during the analysis of the protection scheme. The model checker supports different types of partial disk failures, including latent sector errors and the different types of data corruption. We now describe how the different failures are modeled.

Latent sector errors: These failures are modeled as inaccessible data – an explicit error is returned when an attempt is made to read the disk block. Disk writes always succeed; it is assumed that if a latent sector error occurs, the disk automatically *remaps* the sectors.

Bit corruptions: These failures are modeled as a change in value of a disk sector that produces a new value (*i.e.*, no collisions).

Lost writes: These failures are modeled by not updating any of the sectors that form a disk block when a subsequent disk write is issued.

Operation	**Description**	**Notation**
User read	Read for any data disk	$\mathrm{R(X)}$
User write	Write for any combination of disks in the stripe (the model performs any disk reads needed for parity calculation)	$\mathrm{W}()$ is any write, $\mathrm{W_{ADD}}()$ is write with additive parity, $\mathrm{W_{SUB}}()$ is subractive; Parameters: X+ is "data disk X plus others", !X is "other than disk X", full is "full stripe"
Scrub	Read all disks, verify protections, recompute parity from data, and compare with on-disk parity	S
Latent sector error	Disk read to a disk returns failure	$\mathrm{F_{LSE}(X)}$, $\mathrm{F_{LSE}(P)}$ for data disk X and parity disk respectively
Bit corruption	A new value is assigned to a sector	$\mathrm{F_{CORRUPT}(X)}$, $\mathrm{F_{CORRUPT}(P)}$ for data disk X and parity disk respectively
Lost write	Disk write issued is not performed, but success is reported	$\mathrm{F_{LOST}(X)}$, $\mathrm{F_{LOST}(P)}$ for data disk X and parity disk respectively
Torn write	Only the first sector of a disk write is written, but success is reported	$\mathrm{F_{TORN}(X)}$, $\mathrm{F_{TORN}(P)}$ for data disk X and parity disk respectively
Misdirected write	A disk block is overwritten with data following the same layout as the block, but not meant for it	$\mathrm{F_{MISDIR}(X)}$, $\mathrm{F_{MISDIR}(P)}$ for data disk X and parity disk respectively

Table 4.2 **Model operations.** *This table shows the different sources of state transitions: (a) operations that are performed on the model, and (b) the different partial disk failures that are injected.*

Torn writes: These failures are modeled by updating only a portion of the sectors that form a disk block when a subsequent disk write is issued.

Misdirected writes: In a real system, these failures manifest in two ways: (i) they appear as a lost write for the block the write was intended to (the target), and (ii) they overwrite a different disk location (the victim). We assume that the target and victim are on different RAID stripes (otherwise, it would be a double failure), and therefore can be modeled separately. Thus, we need to model only the victim, since the effects of a lost write on the target is a failure we already study. A further assumption we make is that the data being written is block-aligned with the victim. Thus, a misdirected write is modeled by performing a write to a disk block (with valid entries) without an actual request from the model.

4.2.3 Model-Checker States

A state according to the model checker is defined using the following sub-states: (a) the validity of each data item stored in the data disks as maintained by the model checker, (b) the results of performing each of the protection checks of the model, and (c) whether valid data and metadata items can be regenerated from parity for each of the data disks. The data disks are considered interchangeable; for example, data disk D_0 with corrupt data is the same as data disk D_1 with corrupt data as long as all other data and parity items are valid in both cases. As with any model checker, the previously explored states are remembered to avoid re-exploration.

The output of the model checker is a state machine that starts with the RAID stripe in the clean state and contains state transitions to each of the unique states discovered by the model checker. Table 4.2 contains a list of operations and errors that cause the state transitions.

4.3 Analysis

We now analyze various protection schemes using the model checker. We add protection techniques – RAID, data scrubbing, checksums, write-verify, identity, version mirroring – one by one, and evaluate each setup.

4.3.1 Bare-bones RAID

The simplest of protection schemes is the use of parity to recover from failures. This type of scheme is traditionally available through RAID hardware cards [1]. In this scheme, failures are typically detected based on error codes returned by the disk drive.

Figure 4.1 presents the model of bare-bones RAID, specified using the primitives provided by the model checker. In this model, a user read command simply calls a RAID-level read, which in turn issues a disk read for all disks. The disk read primitive returns the "data" successfully unless a latent sector error is encountered. On a latent sector error, the RAID read routine calls the reconstruct routine, which reads the rest of the disks, and recovers data through parity calculation. At the end of a user read, in place of returning data to the user, a validity check primitive is called. This model checker primitive verifies that the data is indeed valid; if it is not valid, then the model checker has found a hole in the protection scheme that returns corrupt data to the user.

When one or more data disks are written, parity is recalculated. Unless the entire stripe is written, parity calculation requires disk reads. In order to optimize the number of disk reads, parity calculation may be performed in an additive or subtractive manner. In additive parity calculation, data disks other than the disks being written are read and the new parity is calculated as the XOR over the read blocks and the blocks being written. In subtractive parity calculation, the old data in the disks being written and the old parity are first read. Then, the new parity is the XOR of old data, old parity, and new data. Since parity calculation uses data on disk, it should verify the data read from disk. We shall see in the subsections that follow that the absence of this verification could violate data protection.

When the model checker is used to evaluate this model and only one partial disk failure is injected, we obtain the state machine shown in Figure 4.2. Note that the state machine shows only those operations that result in state transitions (*i.e.*, self-loops are omitted). The model starts in the `clean` state and transitions to different states when failures occur. We now describe the state transitions with an example. A latent sector error to data disk X places the model in state $\texttt{Disk}_X$ `LSE`. The model could then transition back to `clean` state on a disk read to disk X. This disk read could be initiated due to various reasons. First, a user read to disk X (*i.e.*, $\mathrm{R(X)}$) could cause

```
UserRead(Disks[])
{
   data[] = RaidRead(Disks[]);
   if(raid read failed)
      Declare double failure and return;
   else
      CheckValid(Disks[], data[]);
}

UserWrite(Disks[], data[])
{
   if(Additive parity cost lower for num(Disks[]))
   {
      other_disks[] = RaidRead(AllDisks[] - Disks[]);
      if(raid read failed)
         Declare double failure and return;
      parity_data = Parity(data[] + other_disks[]);
   }
   else// subtractive parity
   {
      old_data[] = RaidRead(Disks[] + ParityDisk);
      if(raid read failed)
         Declare double failure and return;
      parity_data = Parity(data[] + old_data[]);
   }
   for(x = 0 to num(Disks[]))
   {
      DiskWrite(Disks[x], data[x]);
   }
   DiskWrite(ParityDisk, parity_data)
   return SUCCESS;
}

RaidRead(Disks[])
{
   for(x = 0 to num(Disks[]))
   {
      data[x] = DiskRead(Disks[x]);
      if(disk read failed) // LSE
      {
         data[x] = Reconstruct(Disks[x]);
         if(reconstruct failed) // another LSE
         {
            return FAILURE;
         }
      }
   }
   return data[];
}

Reconstruct(BadDisk)
{
   for(x = 0 to num(AllDisks[]))
   {
      if(Disks[x] is not BadDisk)
         data[x] = DiskRead(Disks[x]);
      else
         data[x] = DiskRead(ParityDisk);
      if(disk read fails) // LSE
         return FAILURE;
   }
   new_data = Parity(data[x]);
   DiskWrite(Disks[x], new_data);
   return new_data;
}
```

Figure 4.1 **Model of bare-bones RAID.** *The figure shows the model of bare-bones RAID specified using the primitives* **DiskRead**, **DiskWrite**, **Parity**, *and* **CheckValid** *provided by the model checker.* **CheckValid** *is called when returning data to the user and the model checker verifies if the data is actually valid.*

the corresponding disk read. Second, a user write to disk X could initiate a disk read to it for subtractive parity calculation ($\mathrm{W_{SUB}(X+)}$). Third, a user write to some disk may result in additive parity calculation, thereby causing disk X to be read ($\mathrm{W_{ADD}()}$). We see from the example that the model can recover from a latent sector error to data disks. The state machine shows that the model can recover from a latent sector error to the parity disk as well.

Let us now consider the state transitions that lead to corrupt data being returned to the user. We retain the names of states involved in these transitions for other data protection schemes as well, since the role they play is similar across schemes.

Any of the silent data corruptions, lost write, torn write, misdirected write, or bit corruption to data disk X when in `clean` state, places the model in state $\texttt{Disk}_X$ `Error`. In this state, disk X contains wrong data and the (correct) parity on the stripe is therefore inconsistent with the data disks. A user read to disk X will now return corrupt data to the user (`Corrupt Data`), simply because there is no means of verifying that the data is valid. If a user write to disks other than disk X triggers additive parity calculation ($\mathrm{W_{ADD}(!X)}$), the corrupt data in disk X is used for parity calculation, thereby corrupting the parity disk as well. In this scenario, both disk X and the parity disk P contain corrupt data, but they are consistent. We term this process of propagating incorrect data to the parity disk during additive parity calculation as *parity pollution* and it corresponds to the state `Polluted Parity`. Parity pollution does not impact the probability of data loss or corruption in this case since bare-bones RAID does not detect any form of corruption. However, as we shall see, parity pollution causes problems for many other protection schemes.

When in state $\texttt{Disk}_X$ `Error`, if a user write involving disk X leads to subtractive parity calculation ($\mathrm{W_{SUB}(X+)}$), the corrupt data in disk X is used for the parity calculation. Therefore, the new parity generated is corrupt (and also inconsistent with the data disks). However, since disk X is being written, disk X is no longer corrupt. This state is named as `Parity Error` in the state machine. We see that the same state can be reached from `clean` state when a silent data corruption occurs for the parity disk. This state does not lead to further data loss or corruption in the absence of a second failure (if a second failure is detected on one of the data disks, the corruption will be

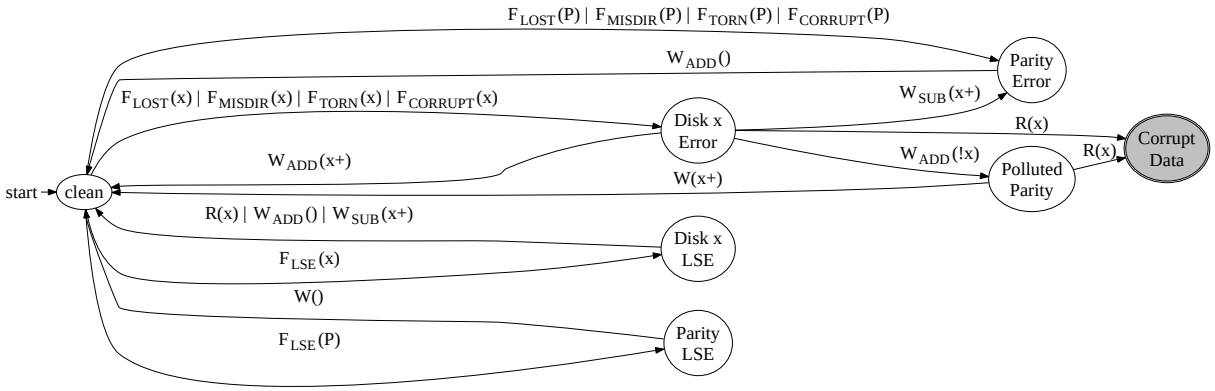

Figure 4.2 **State machine for bare-bones RAID.** *The figure shows the state machine obtained from the model checker when different RAID operations are performed on a bare-bones RAID-4 stripe and a single partial disk failure is injected.*

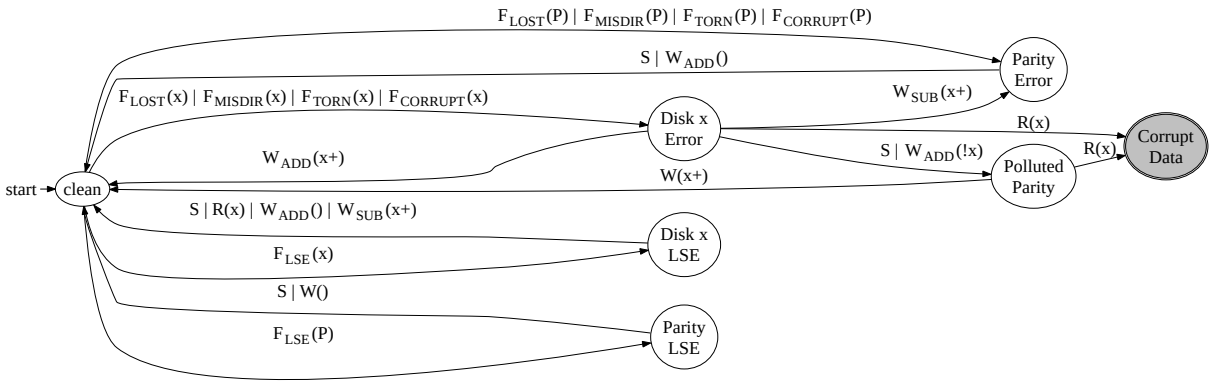

Figure 4.3 **State machine for RAID with scrubbing.** *The figure shows the state machine obtained from the model checker when different RAID operations are performed on a RAID-4 stripe, and a single partial disk failure is injected. The protection technique used in addition to RAID is scrubbing.*

propagated to that disk as well). Thus, we see that, bare-bones RAID protects against latent sector errors, but not against silent data corruptions.

4.3.2 Data Scrubbing

In this scheme, we add data scrubbing to the bare-bones RAID protection scheme. As we discussed in Section 3.1.2.3, data scrubs read all disk blocks that form the stripe and reconstruct the data if a failure is detected. The scrub also recomputes the parity of the data blocks and compares it with the parity stored on the parity disk, thereby detecting any inconsistencies. Thus, the scrubbing mechanism can convert the RAID recovery mechanism into a corruption-detection technique. Note that if an inconsistency is detected, bare-bones RAID does not offer a method to resolve it. The scrub should fix the inconsistency (by recomputing the contents of the parity disk) because inconsistent data and parity lead to further data corruption if a second failure occurs and reconstruction is performed.

When the model checker is used to examine this model, we obtain the state machine shown in Figure 4.3. We see that the state machine is very similar to that of bare-bones RAID, except that some edges include the scrub operation S. One such edge is the transition from the state $\texttt{Disk}_X$ `Error`, where data in disk X is wrong, to `Polluted Parity`, where both the data and

parity are wrong, but consistently so. This transition during a scrub is easily explained – in $\texttt{Disk}_X$ `Error`, the scrub detects a mismatch between data and parity and updates the parity to match the data moving the model to state `Polluted Parity`. We see that the addition of the scrub has not improved protection when only one failure occurs; scrubs are intended to lower the chances of double failures, not of loss from single failures. In fact, we shall see later that the tendency of scrubs to pollute parity increases the chances of data loss when only one failure occurs.

4.3.3 Checksums

Checksumming techniques have been used in numerous systems over the years to detect data corruption. Some systems store the checksum along with the data that it protects [18, 37, 131], while other systems store the checksum on the access path to the data [129, 130]. We will explore both alternatives. We also distinguish between the schemes that store per sector checksums [18, 37] and those that use per-block checksums [131].

Sector checksums: In the case of sector checksums, a separate checksum is generated for each sector and stored along with data in that sector. Figure 4.4 shows the state machine obtained for sector-level checksum protection. The obvious change from the previous state machines is the addition of two new states $\texttt{Disk}_X$ `Corrupted` and `Parity Corrupted`. The model transitions to these states from the `clean` state when a bit corruption occurs to disk X or the parity disk respectively. The use of sector checksums enables the detection of these bit corruptions whenever the corrupt block is read (including scrubs), thus initiating reconstruction and thereby returning the model to `clean` state. However, the use of sector checksums does not protect against torn writes, lost writes, and misdirected writes. For example, torn writes update a single sector, but not the rest of the block. The checksum for all sectors is therefore consistent with the data in that sector. Therefore, sector checksums do not detect these scenarios ($\mathrm{R(X)}$ from $\texttt{Disk}_X$ `Error` leads to `Corrupt Data`).

Block checksums: The goal of block checksums is to ensure that a disk block is one consistent unit, unlike with sector checksums. Therefore, a checksum is generated for each disk block (that consists of multiple sectors) and it is stored along with the data in the block. Figure 4.5 shows

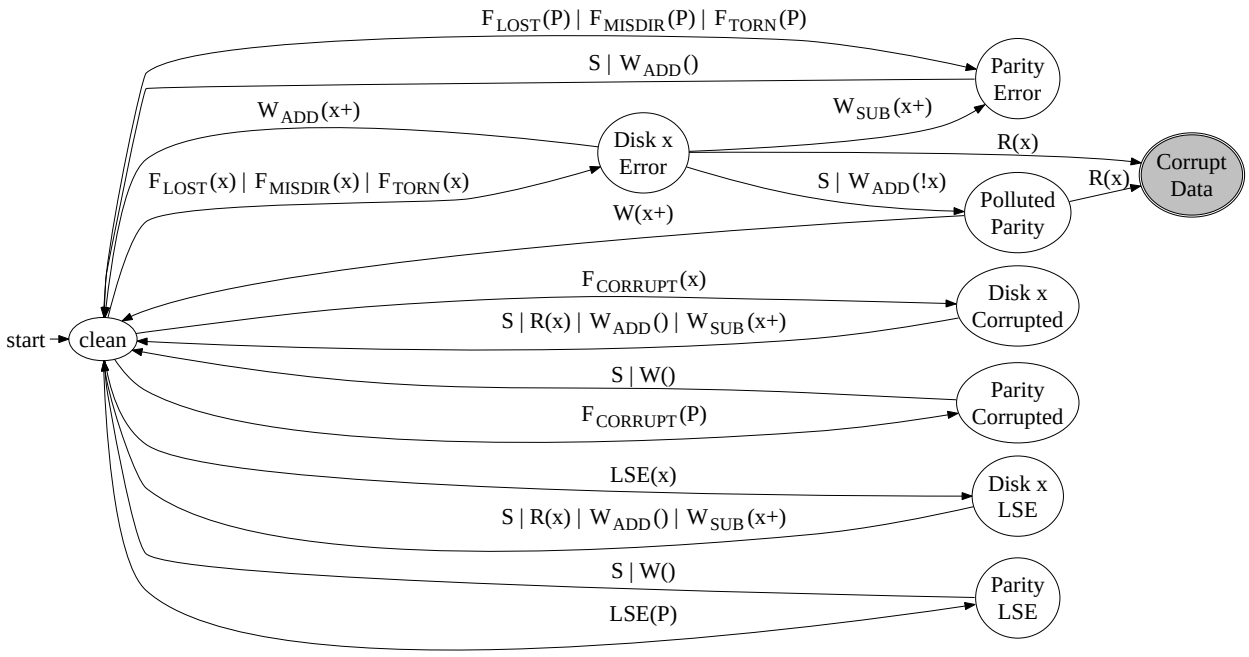

Figure 4.4 **State machine for sector checksums.** *The figure shows the state machine obtained from the model checker when different RAID operations are performed on a RAID-4 stripe, and a single partial disk failure is injected. The protection techniques used are scrubbing and sector-level checksums.*

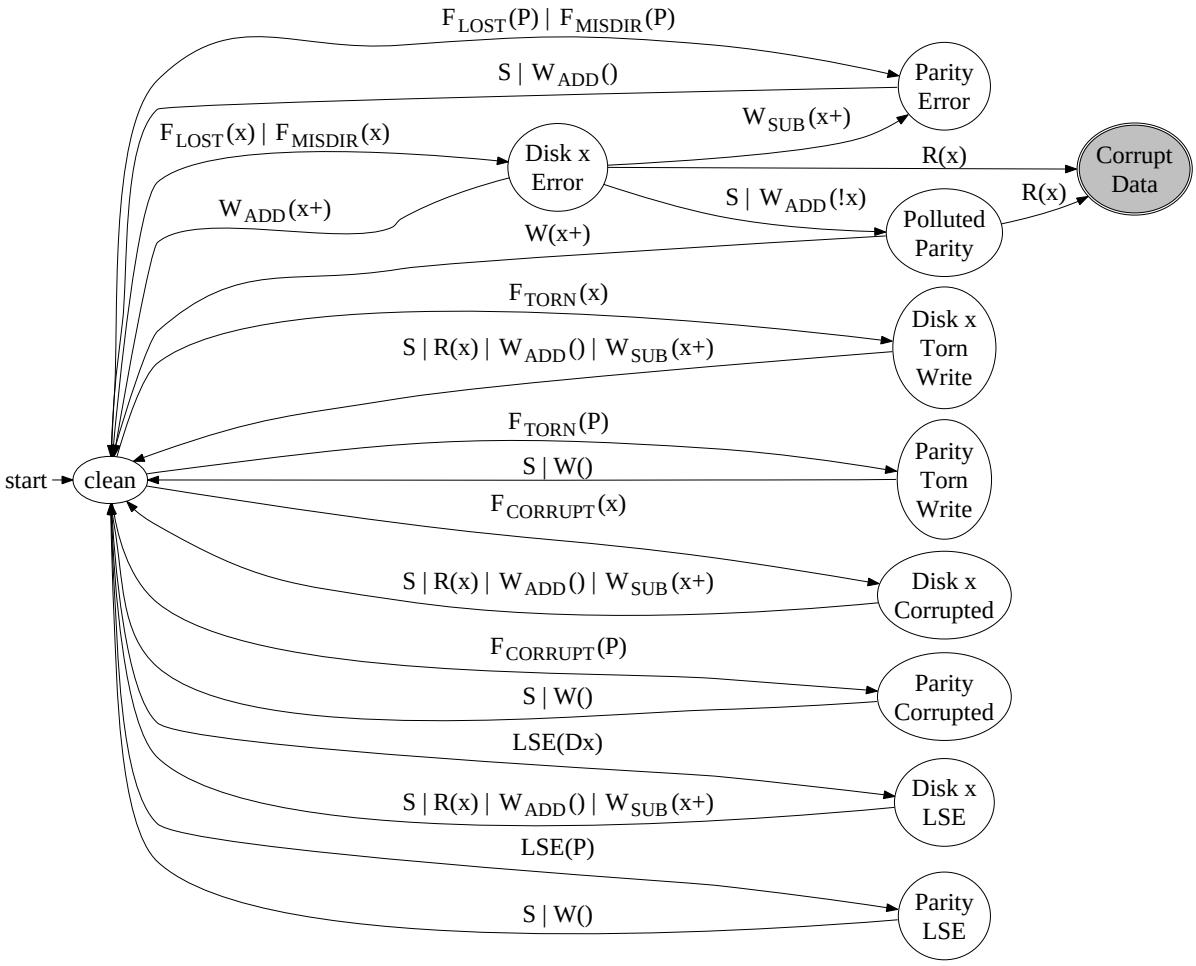

Figure 4.5 **State machine for block checksums.** *The figure shows the state machine obtained from the model checker when different RAID operations are performed on a RAID-4 stripe, and a single partial disk failure is injected. The protection techniques used are scrubbing and block-level checksums.*

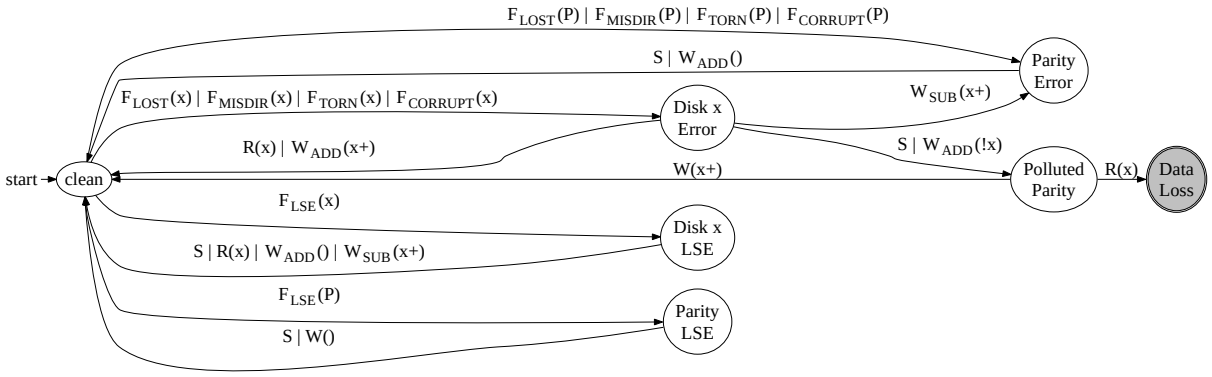

Figure 4.6 **State machine for parental checksums.** *The figure shows the state machine obtained from the model checker when different RAID operations are performed on a RAID-4 stripe and a single partial disk failure is injected. The protection techniques used are scrubbing and checksums stored in a parent block.*

the state machine obtained for block-level checksum protection. Again, the addition of new states that do not lead to `Corrupt Data` signifies an improvement in the protection. The new states added correspond to torn writes. Unlike sector-level protection, block-level protection can detect torn writes (detection denoted by transitions from states $\texttt{Disk}_X$ `Torn Write` and `Parity Torn Write` to `clean`) in exactly the same manner as detecting bit corruptions. However, we see that corrupt data could still be returned to the user. A lost write or a misdirected write transitions the model from the `clean` state to $\texttt{Disk}_X$ `Error`. When a lost write occurs, the disk block retains the data and the corresponding checksum written on a previous occasion. The data and checksum are therefore consistent. Hence, the model does not detect that the data on disk is wrong. A read to disk X now returns corrupt data to the user. The scenario is similar for misdirected writes as well.

Parental checksums: A third option for checksumming is to store the checksum of the entire disk block in a parent block that is accessed first during user reads (*e.g.*, an inode of a file is read before its data block). Parental checksums can thus be used to verify data during all user reads, but not for other operations such as data scrubs.

Figure 4.6 shows the state machine for this scheme. We notice many changes to the state machine as compared to block checksums. First, we see that the states successfully handled by

block checksums (such as $\mathrm{F_{TORN}(X)}$) do not exist. Instead, the transitions that led from `clean` to those states now place the model in $\texttt{Disk}_X$ `Error`. Second, none of the states return corrupt data to the user. Instead, a new node called `Data Loss` has been added. This change signifies that the model detects a double failure and reports data loss. Third, the only transition to `Data Loss` is due to a read of disk X when in the `Polluted Parity` state. Thus, parity pollution now leads to data loss. As before, the causes of parity pollution are data scrubs or additive parity calculations (transitions S or $\mathrm{W_{ADD}(!X)}$ lead from $\texttt{Disk}_X$ `Error` to `Polluted Parity`). Figure 4.7 presents a pictorial view of the transitions from clean state to parity pollution and data loss. At the root of the problem is the fact that parental checksums can be verified only for user reads, not other sources of disk reads such as data scrubs or parity calculations. Any protection technique that does not co-operate with RAID, allows parity recalculation to use bad data, causing irreversible data loss.

Of the three checksums techniques evaluated, we find that block checksumming has the fewest number of transitions to data loss or corruption. Therefore, we use block checksums as the starting point for adding further protection techniques.

4.3.4 Write-Verify

One primary problem with block checksums is that lost writes are not detected. Lost writes are particularly difficult to handle. If the checksum is stored along with the data and both are written as part of the same disk request, they are both lost, leaving the old data and checksum intact and valid. On later reads to disk block, checksum verification compares the old data and old checksum which are consistent, thereby not detecting the lost write.

One simple method to fix this problem is to ensure that writes are not lost in the first place. Some storage systems perform write-verify [65, 131] (also called read-after-write verify) for this purpose. This technique reads the disk block back after it is written, and uses the data contents in memory to verify that the write has indeed completed.

Figure 4.8 shows the state machine for write-verify with block checksums. Comparing this figure against Figure 4.5, we notice two differences: First, the states representing torn data or parity do not exist anymore. Second, the transitions $\mathrm{F_{TORN}(X)}$, $\mathrm{F_{TORN}(P)}$, $\mathrm{F_{LOST}(X)}$, and $\mathrm{F_{LOST}(P)}$ are

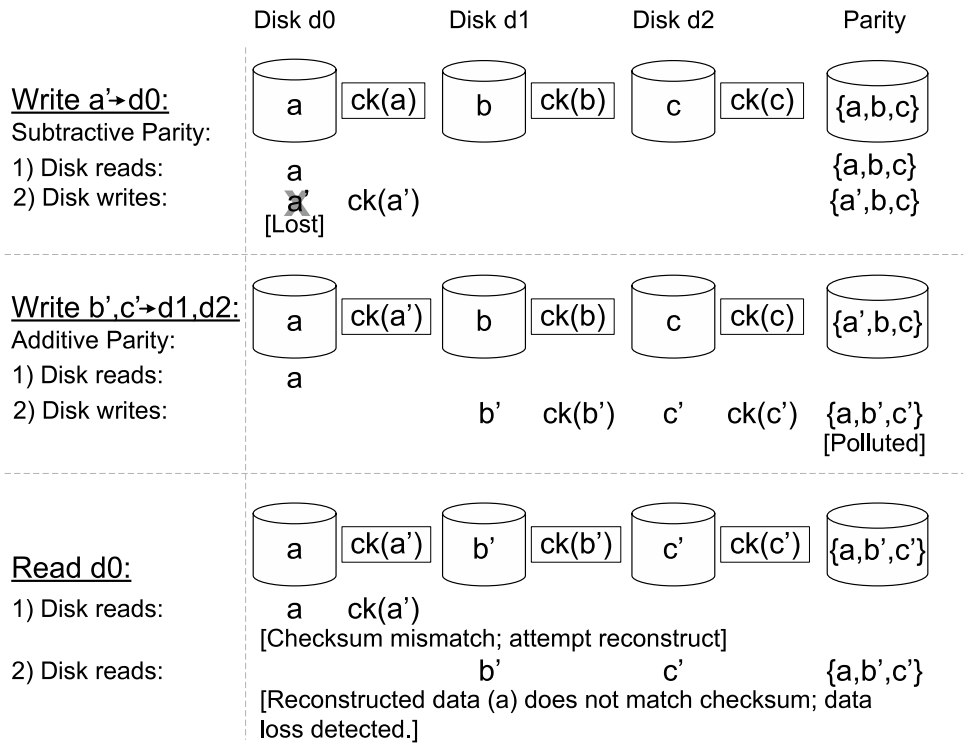

Figure 4.7 **Parity pollution sequence.** *This figure shows a sequence of operations, along with intermediate RAID states, that lead to parity pollution and subsequent data loss. Each horizontal set of disks (Data disks d0, d1 and d2 and Parity disk) form the RAID stripe. The contents of the disk blocks are shown inside the disks.* a*,* b*, etc. are data values, and* $\{a, b\}$ *denotes the parity of values* a *and* b*. The protection scheme used is parental checksums. Checksums are shown next to the corresponding data disks. At each RAID state, user read or write operations cause corresponding disk reads and writes, resulting in the next state. The first write to disk d0 is lost, while the checksum and parity are successfully updated. Next, a user write to disks d1 and d2 uses the bad data in disk d0 to calculate parity, thereby causing parity pollution. A subsequent user read to disk d0 detects a checksum mismatch, but recovery is not possible since parity is polluted.*

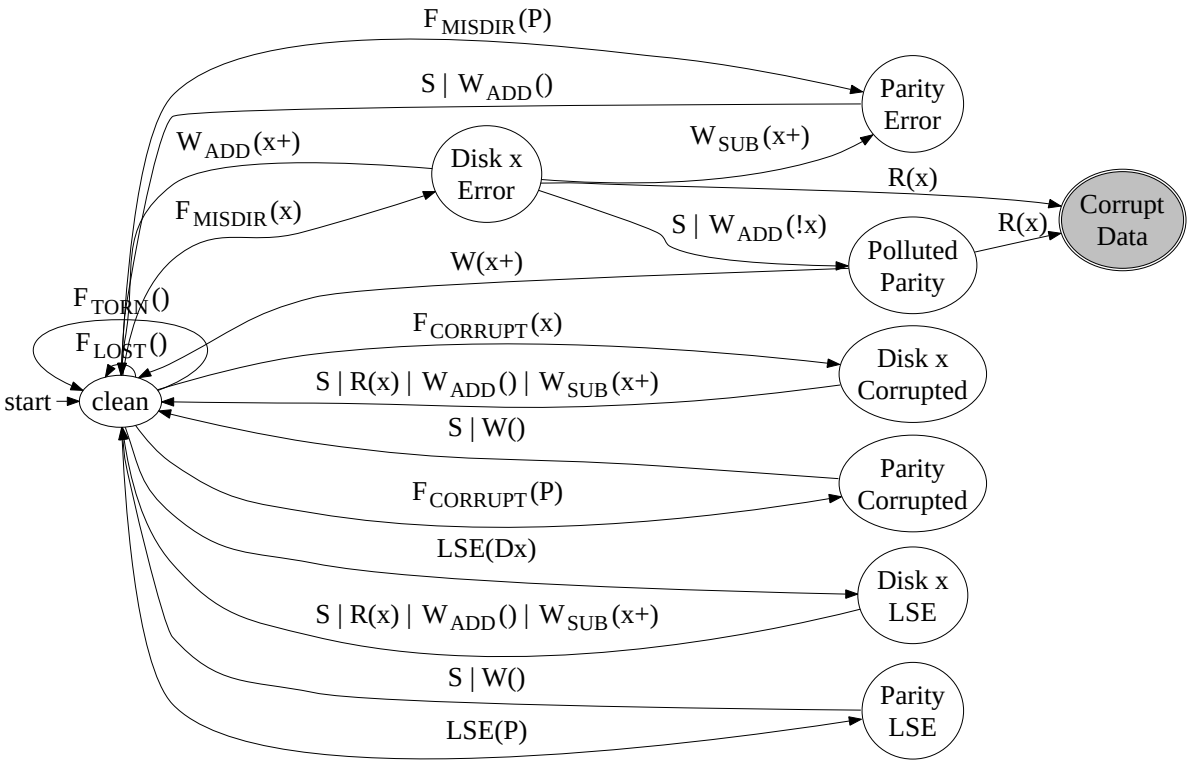

Figure 4.8 **State machine for write-verify.** *The figure shows the state machine obtained from the model checker when different RAID operations are performed on a RAID-4 stripe, and a single partial disk failure is injected. The protection techniques used are scrubbing, block-level checksums, and write-verify.*

now from `clean` to itself, instead of to other states (self-loops shown for readability). Write-verify detects lost writes and torn writes as and when they occur, keeping the RAID stripe in clean state.

Unfortunately, write-verify has two negatives. First, it does not protect against misdirected writes. When a misdirected write occurs, write-verify would detect that the original target of the write suffered a lost write, and therefore simply reissue the write. However, the victim of the misdirected write is left consistent with consistent checksums but wrong data. A later user read to the victim thus returns corrupt data to the user. Second, although write-verify improves data protection, every disk write now incurs a disk read as well, possibly leading to a loss in performance.

4.3.5 Identity

A different approach that is used to solve the problem of lost or misdirected writes without the performance penalty of write-verify is the use of identity information.

Different forms of identifying data (also called self-describing data) can be stored along with data blocks. An identity may be in one of two forms: (a) physical identity, which typically consists of the disk number and the disk block (or sector) number to which the data is written [18], and (b) logical identity, which is typically an inode number and offset within the file [107, 131].

Physical identity: Physical identity consists of the disk number and the disk block number to which the data is written. This identity is stored along with the data in the disk block. Figure 4.9 shows the state machine obtained when physical identity information is used in combination with block checksums. Compared to previous state machines, we see that there are two new states corresponding to misdirected writes, $\texttt{Disk}_X$ `Misdir Write` and `Parity Misdir Write`. These states are detected by the model when the disk block is read for any reason (scrub, user read, or parity calculation) since even non-user operations like scrub can verify physical identity. Thus, physical identity is a step towards mitigating parity pollution. However, parity pollution still occurs in state transitions involving lost writes. If a lost write occurs, the disk block contains the old data, which would still have the correct physical block number. Therefore, physical identity cannot protect against lost writes, leading to corrupt data being returned to the user.

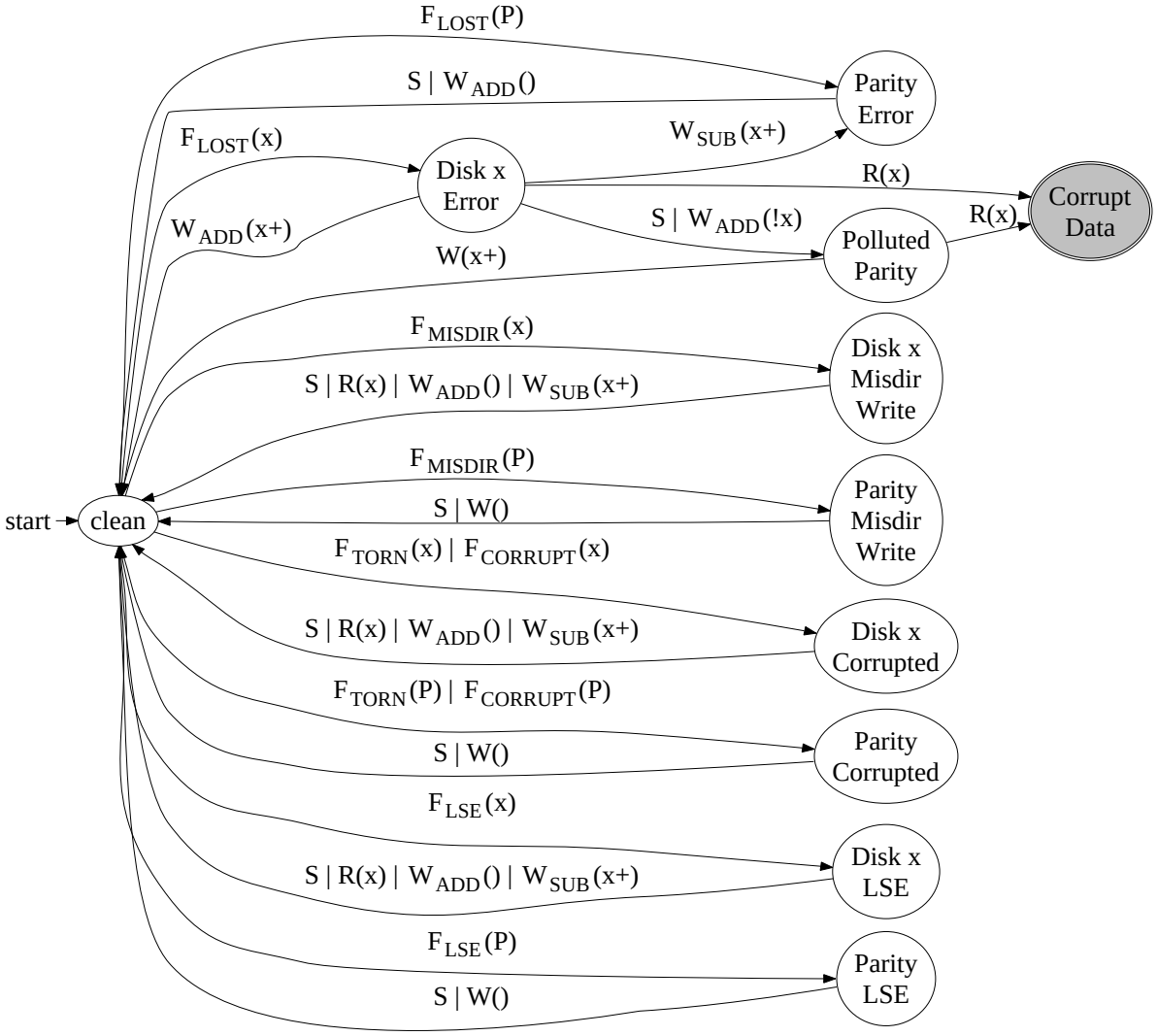

Figure 4.9 **State machine for physical identity.** *The figure shows the state machine obtained from the model checker when different RAID operations are performed on a single RAID-4 stripe, and a single partial disk failure is injected. The protection techniques used are scrubbing, block checksums, and physical identity.*

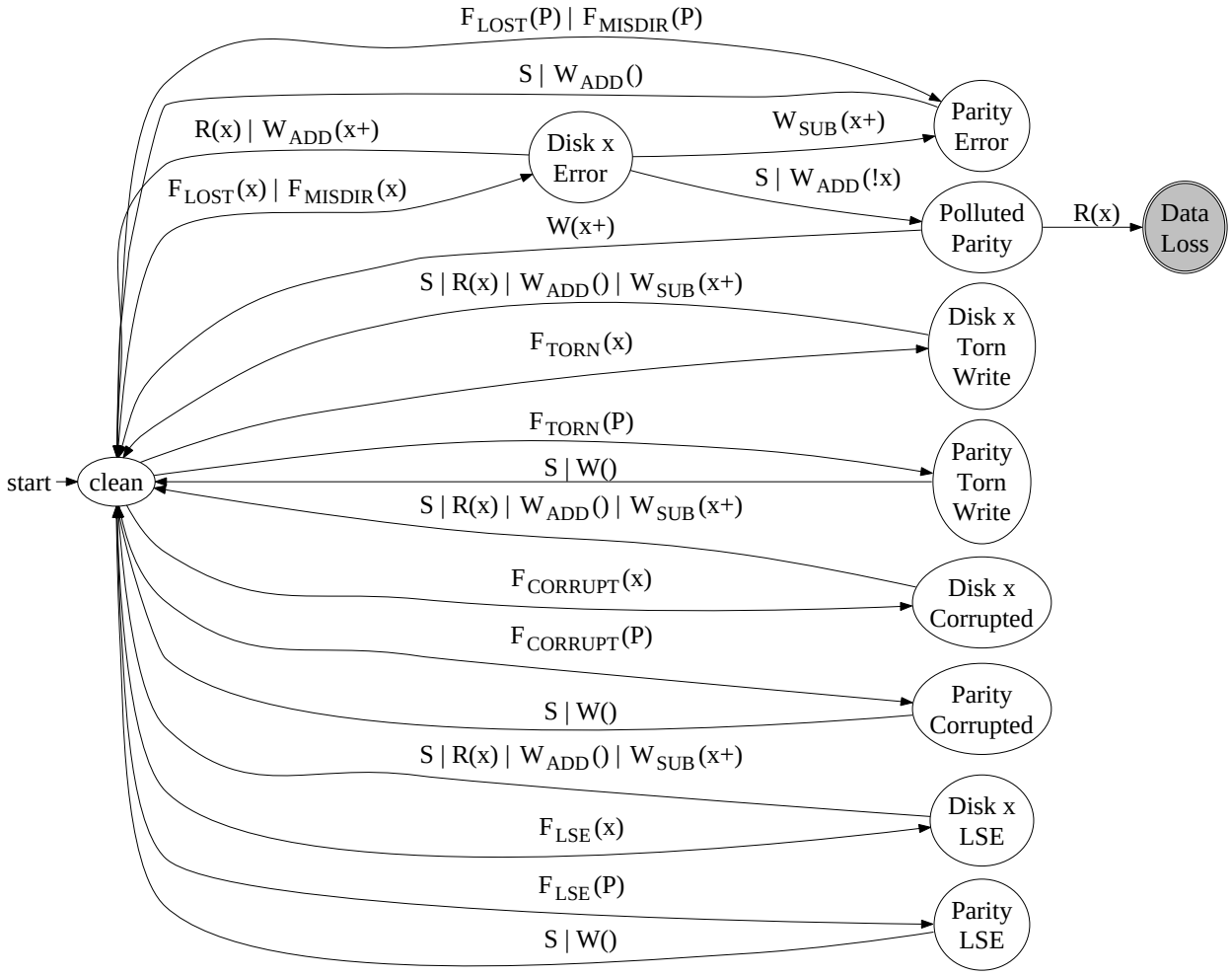

Figure 4.10 **State machine for logical identity.** *The figure shows the state machine obtained from the model checker when different RAID operations are performed on a single RAID-4 stripe, and a single partial disk failure is injected. The protection techniques used are scrubbing, block checksums, and logical identity.*

Logical identity: The logical identity is typically an inode number and offset within the file. It is stored along with the data in the disk block. The logical identity of disk blocks is defined by the block's parent and can therefore be verified only during user reads. Figure 4.10 shows the state machine obtained when logical identity protection is used in combination with block checksums. Unlike physical identity, misdirected writes do not cause new states to be created for logical identity. Both lost and misdirected writes place the model in the $\texttt{Disk}_X$ `Error` state. At this point, parity pollution due to scrubs and user writes moves the system to the `Polluted Parity` state since logical identity can be verified only on user reads, thus causing data loss. Thus, logical identity works in similar fashion to parental checksums: (i) in both cases, there is a check that uses data from outside the block being protected, and (ii) in both cases, corrupt data is not returned to the user and instead, data loss is detected.

4.3.6 Version Mirroring

The use of identity information (both physical and logical) does not protect data from exactly one scenario – parity pollution after a lost write. We now introduce version mirroring to detect lost writes during scrubs and parity calculation. Herein, each data block that belongs to the RAID stripe contains a version number. This version number is incremented with every write to the block. The parity block contains a list of version numbers of all of the data blocks that it protects. Whenever a data block is read, its version number is compared to the corresponding version number stored in the parity block. If a mismatch occurs, the newer block will have a higher version number, and can used to reconstruct the other data block.

Note that when this approach is employed during user reads, each disk block read would now incur an additional read of the parity block. To avoid this performance penalty, version numbers can be used in conjunction with logical identity. Thus, logical identity is verified during file system reads, while version numbers are verified for parity re-calculation reads and disk scrubbing. This approach incurs an extra disk read of the parity block only when additive parity calculation is performed.

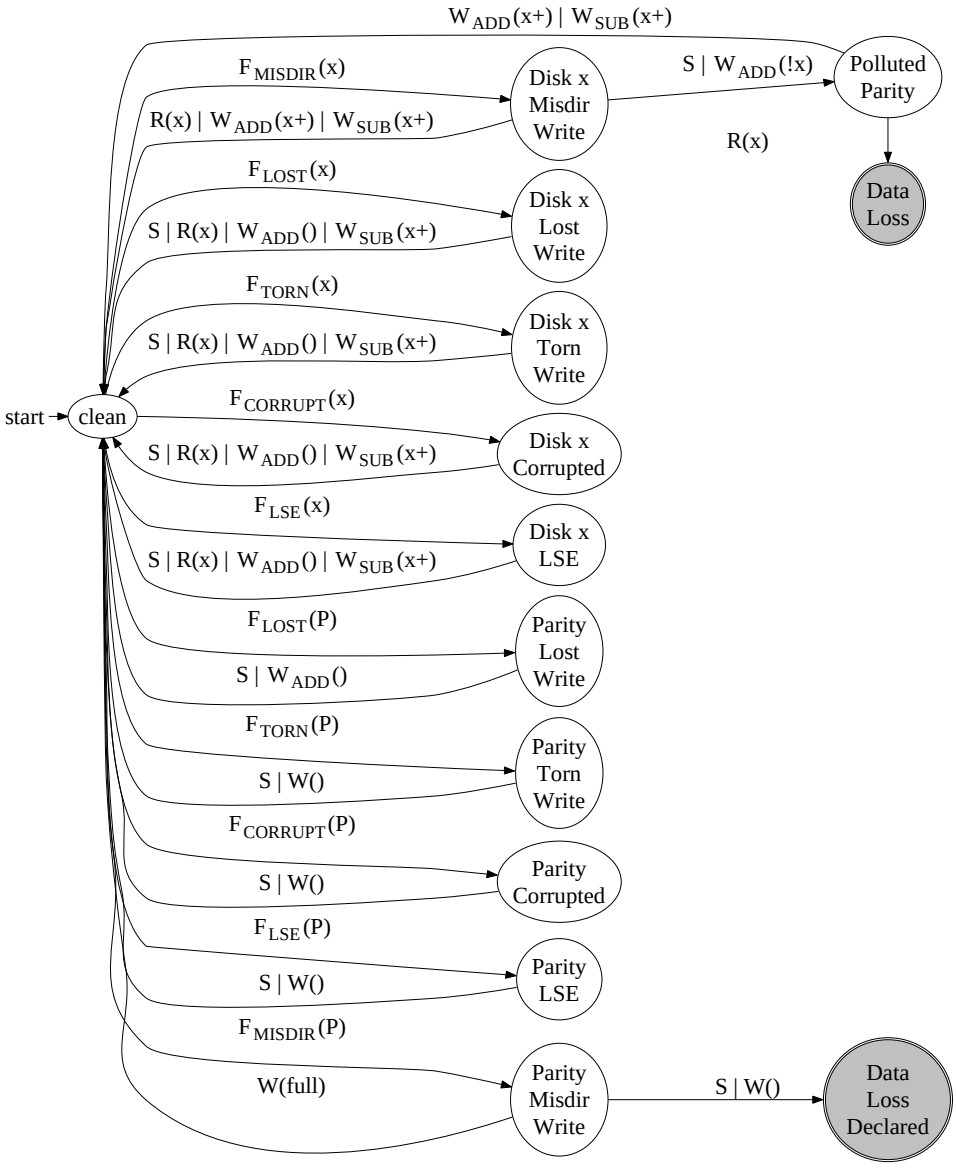

Figure 4.11 **State machine for version mirroring.** *The figure shows the state machine obtained from the model checker when different RAID operations are performed on a single RAID-4 stripe, and a single partial disk failure is injected. The protection techniques used are scrubbing, block checksums, logical identity, and version mirroring.*

A primitive form of version mirroring has been used in real systems: Dell Powervault storage arrays [37] use a 1-bit version number called a "write stamp". However, since the length of the version number is restricted to 1-bit, it can only be used to *detect* a mismatch between data and parity (which we already can achieve through parity recompute and compare). It does not provide the power to identify the wrong data (which would enable recovery). This example illustrates that the bit-length of version numbers limits the failures that can be detected and recovered from.

Figure 4.11 shows the state machine obtained when version mirroring is added to logical identity protection. We find that there are now states corresponding to lost writes ($\texttt{Disk}_X$ `Lost Write` and `Parity Lost Write`) for which all transitions lead to `clean`. However, `Data Loss` could still occur, and in addition, `Data Loss Declared` could occur as well. The only failure that causes state transitions to any of these nodes is a misdirected write.

A misdirected write to disk X places the model in $\texttt{Disk}_X$ `Misdir Write`. Now, an additive parity calculation that uses disk X will compare the version number in disk X against the one in the parity disk. The misdirected write could have written a disk block with a higher version number than the victim. Thus, the model trusts the wrong disk X and pollutes parity. A subsequent read to disk X uses logical identity to detect the corruption, but the parity has already been polluted.

A misdirected write to the parity disk causes problems as well. Interestingly, none of the protection schemes so far face this problem. The sequence of state transitions leading to `Data Loss Declared` occurs in following fashion. A misdirected write to the parity disk places new version numbers in the entire list of version numbers on the disk. When any data disk's version number is compared against its corresponding version number on this list (during a write or scrub), if the parity's (wrong) versions numbers are higher, reconstruction is initiated. Reconstruction will detect that none of the version numbers of the data disks match the version numbers stored on the parity disk. In this scenario, a multi-disk failure is detected and the model declares data loss. This state is different from `Data Loss`, since this scenario is a false positive while the other has actual data loss.

The occurrence of the `Data Loss Declared` state indicates that the policy used when multiple version numbers mismatch during reconstruction is faulty. It is indeed possible to have a policy that

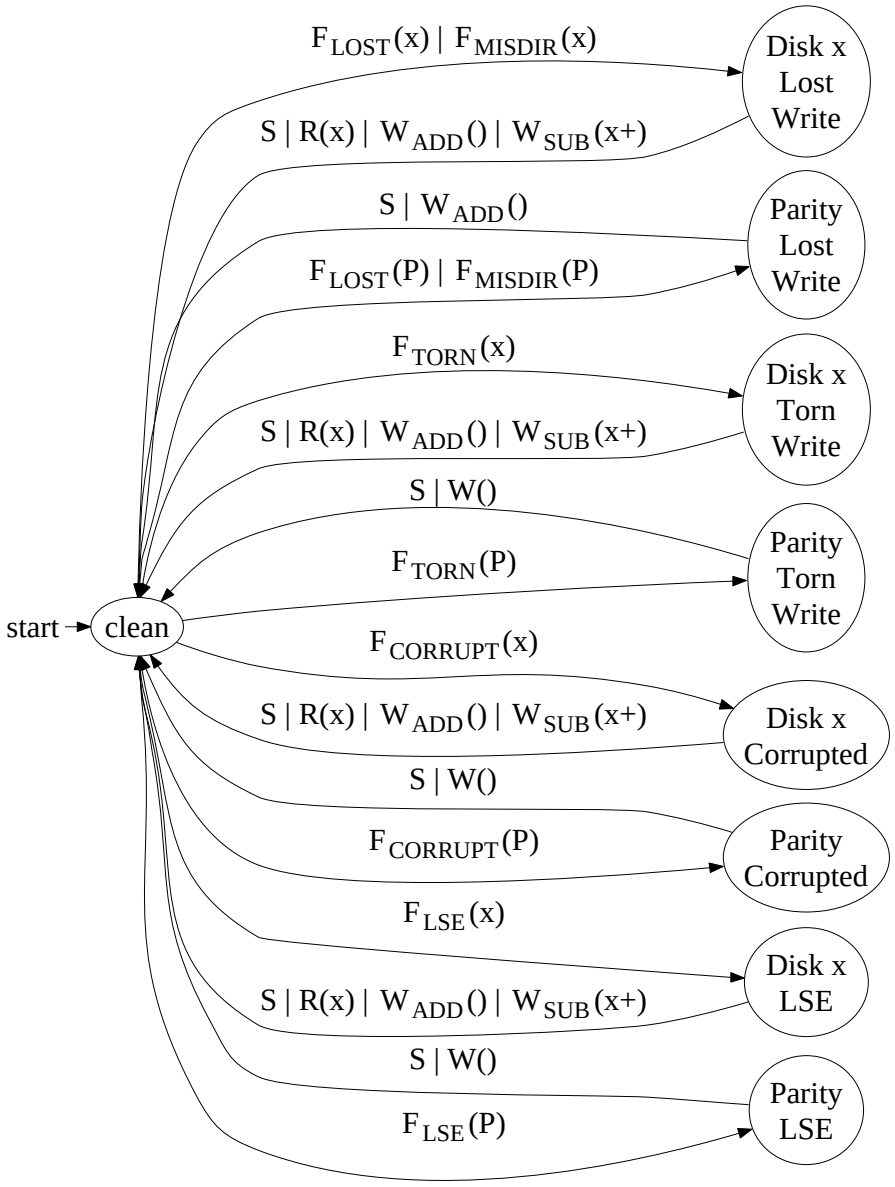

Figure 4.12 **State machine for complete data protection.** *The figure shows the state machine obtained from the model checker when different RAID operations are performed on a single RAID-4 stripe, and a single partial disk failure is injected. The protection techniques used are scrubbing, block checksums, both physical and logical identity, and version mirroring.*

fixes parity instead of data on a multiple version number mismatch. The use of a model checker thus enables identification of policy faults as well.

We know from the previous subsection that physical identity protects against misdirected writes. Therefore, if physical identity is added to version mirroring and logical identity, we could potentially eliminate all problem nodes. Figure 4.12 shows the state machine generated for this protection scheme. We see that none of the state transitions lead to data loss or data corruption. The advantage of using physical identity is that the physical identity can be verified (detecting any misdirected write) before comparing version numbers. Thus, we have identified a scheme that eliminates data loss or corruption due to a realistic range of disk failures.

We now review the techniques used in the scheme that protects against all failures:

RAID: RAID stores a parity block for each set of data blocks and thus provides the ability to recover from single failures.

Block checksums: This technique stores a checksum for each disk block along with the block. It provides the ability to detect bit corruptions whenever the disk block is read. Note that the checksum protects both the data in the block as well as other protection elements such as identity information.

Physical identity: This form of identity is typically a combination of the disk number and the disk block number; it is stored along with the disk block. It is used to detect misdirected writes whenever the disk block is read.

Version mirroring: In this technique, a version number is stored in each data block and a copy of it is stored in the parity block. This version number is incremented for each disk write. The version numbers on the data and parity blocks are compared whenever parity is updated. This technique detects lost writes during parity calculations.

Logical identity: This form of identity is typically a combination of the inode number of the file to which the disk block belongs and the offset within that file. The logical identity of a disk block is stored in the disk block. This technique is used to detect lost writes during

user read of the disk block. While version mirroring is sufficient for this purpose, logical identity does so without an extra disk read and is therefore useful.

4.3.7 Discussion

The analysis of multiple schemes has helped identify the following key data protection issues.

Parity pollution: We believe that any parity-based system that re-uses existing data to compute parity is potentially susceptible to data loss due to disk failures, in particular lost and misdirected writes. In the absence of techniques to perfectly verify the integrity of existing disk blocks used for recomputing the parity, disk scrubbing and partial-stripe writes can cause parity pollution, where the parity no longer reflects valid data.

In this context, it would be interesting to apply model checking to understand schemes with double parity [22, 35]. Another interesting scheme that could be analyzed is one with RAID-Z [24] protection (instead of RAID-4 or RAID-5), where only full-stripe writes are performed and data is protected with parental checksums.

Parental protection: Verifying the contents of a disk block against any value – either identity or checksum, written using a separate request and stored in a different disk location – is an excellent method to detect failures that are more difficult to handle such as lost writes. However, in the absence of techniques such as version mirroring, schemes that protect data by placing checksum or identity protections on the access path should use the same access path for data scrubbing, parity calculation, and reconstructing data; this approach ensures that parental protection is used to verify block contents on every read. Note that this approach could slow down these processes significantly, especially when the RAID is close to full space utilization.

Mirroring: Mirroring of any piece of data provides a distinct advantage: one can verify the correctness of data through comparison without interference from other data items (as in the case of parity). Version mirroring utilizes this advantage in conjunction with crucial knowledge about the items that are mirrored – the higher value is more recent.

Physical identity: Physical identity, like block checksums, is extremely useful since it is knowledge available at the RAID-level. We see that this knowledge is important for perfect data protection.

Recovery-integrity co-design: Finally, it is vital to integrate data integrity with RAID recovery, and do so by exhaustively exploring all possible scenarios that could occur when the protection techniques are composed.

Thus, a model-checking approach is very useful in deconstructing the exact protection offered by a protection scheme, thereby also identifying important data-protection issues. We believe that such an exhaustive approach would prove even more important in evaluating protections against double failures.

4.4 Probability of Loss or Corruption

One benefit of using a model checker is that we can assign probabilities to various state transitions in the state machine produced, and thus easily generate approximate probabilities for data loss or corruption. These probabilities help compare the different schemes quantitatively.

We use the data for nearline disks (since they are being increasingly used in enterprise storage systems) in Chapter 3 to derive per-year probabilities for the occurrence of the different partial disk failures. For instance, the probability of occurrence of $\mathrm{F_{LSE}}$ (a latent sector error) for one disk is 0.1 (derived from 0.2 for 2 years). The data does not distinguish between corruption and torn writes; therefore, we assume an equal probability of occurrence of $\mathrm{F_{CORRUPT}}$ and $\mathrm{F_{TORN}}$ (0.0022). We derive the probabilities for $\mathrm{F_{LOST}}$ and $\mathrm{F_{MISDIR}}$ based on the assumptions in Section 4.2.2 as 0.0003 and 1.88e-5 respectively.

We also compute the probability for each operation to be the first to encounter the stripe with an existing failure. For this purpose, we utilize the distribution of how often different requests detect corruption in Section 3.4.5. The distribution is as follows. P(User read): 0.2, P(User write): 0.2, P(Scrub): 0.6. We assume that partial stripe writes of varying width are equally likely.

Note that while we attempt to use as realistic probability numbers as possible, the goal is not to provide precise data loss probabilities, but to illustrate the advantage of using a model checker, and discuss potential trade-offs between different protection schemes.

Table 4.3 provides approximate probabilities of data loss derived from the state machines produced by the model checker. We consider a 4-data-disk, 1-parity-disk RAID configuration for all of the protection schemes for calculating probabilities. This table enables simple comparisons of the different protection schemes. We can see that generally, enabling protections causes an expected decrease in the chance of data loss. The use of version mirroring with logical and physical identity, block checksums and RAID produces a scheme with a theoretical chance of data loss or corruption as 0. The data in the table illustrates the following trade-offs between protection schemes:

Scrub vs. No scrub: Systems employ scrubbing to detect and fix errors and inconsistencies in order to reduce the chances of double failures. However, our analysis in the previous section shows that scrubs could potentially cause data loss due to parity pollution. The data in the table shows that it is indeed the case. In fact, since scrubs have a higher probability of encountering failures, the probability of data loss is significantly higher with scrubs than without. For example, using parental checksums with scrubs causes data loss with a probability 0.00486, while using parental checksums without scrubs causes data loss with a 3 times lesser probability 0.00153.

Data loss vs. Corrupt data: Comparing the different protection schemes, we see that some schemes cause data loss whereas others return corrupt data to the user. Interestingly, we also see that the probability of data loss is higher than the probability of corrupt data. For example, using parental checksums (with RAID and scrubbing) causes data loss with a probability 0.00486, while using block checksums causes corrupt data to be returned with a an order of magnitude lesser probability 0.00041. Thus, while in general it is better to detect corruption and incur data loss than to return corrupt data, the answer may not be obvious when the probability of loss is much higher.

RAID	Scrub	Sector Checksums	Block Checksums	Parent Checksums	Write-Verify	Physical Identity	Logical Identity	Version Mirroring	**Chance of Data Loss**
√									0.602%
√	√								0.602%
√	√	√							0.322%
√	√		√						0.041%
√	√			√					*0.486%
√				√					*0.153%
√	√		√		√				0.002%
√	√		√			√			0.038%
√	√		√				√		*0.033%
√			√				√		*0.010%
√	√		√			√	√		*0.031%
√			√			√	√		*0.010%
√	√		√				√	√	*0.004%
√			√				√	√	*0.002%
√	√		√			√	√	√	0.000%

Table 4.3 **Probability of loss or corruption.** *The table provides an approximate probability of at least 1 data loss event and of corrupt data being returned to the user at least once, when each of the protection schemes is used for storing data. It is assumed that the storage system uses 4 data disks, and 1 parity disk. A (*) indicates that the data loss is detectable given the particular scheme (and hence can be turned into unavailability, depending on system implementation).*

If the precise probability distributions of the underlying failures, and read, write, and scrub relative frequencies are known, techniques like Monte-Carlo simulation can be used to generate actual probability estimates that take multiple failures into consideration [39].

4.5 Conclusion

In this chapter, we have presented a formal approach to analyzing the design of data protection strategies. Whereas earlier designs were simple to verify by inspection (*e.g.*, a parity disk successfully adds protection against full-disk failure), today's systems employ a host of techniques, and their interactions are subtle and often non-obvious.

With our approach, we have shown that a variety of approaches found in past and current systems are successful at detecting a variety of problems but that some interesting corner-case scenarios can lead to data loss or corruption. In particular, we found that the problem of parity pollution can propagate errors from a single (bad) block to other (previously good) blocks, and thus lead to a gap in protection in many schemes. The addition of version mirroring and proper identity information, in addition to standard checksums, parity, and scrubbing, leads to a solution where no single error should (by design) lead to data loss.

In the future, as protection evolves further to cope with the next generation of disk problems, we believe approaches such as ours will be critical. Although model checking implementations is clearly important [150], the first step in building any successful storage system should begin with a correctly-specified design.

Chapter 5

Impact on Virtual-Memory Systems

This chapter explores the impact of partial disk failures on virtual-memory systems. A virtual-memory system is an integral part of most operating systems, and like file systems, is a significant user of disk storage.

The virtual-memory system uses disk space to store memory pages that are not expected to be of immediate use, thereby freeing-up physical memory for other memory pages. When a page stored on disk is accessed again, it is brought back into physical memory. Thus, the virtual-memory system is responsible for handling disk failures that affect these memory pages.

Since the virtual-memory system is an integral element of the storage stack, it is important to understand how a virtual-memory system responds to partial disk failures. We extend the type-aware fault-injection techniques presented in Section 2.5.1 to identify the failure-handling policies of the virtual-memory systems of two operating systems, Linux 2.6.13 and FreeBSD 6.0. We also perform a preliminary study of the Windows XP virtual-memory system. We characterize the policies of these systems based on the IRON taxonomy presented in Section 2.4.

From our experiments, we find that these virtual-memory systems are not well-equipped to deal with partial disk failures. Like the file systems studied in Section 2.5.2, the virtual-memory systems use policies that are illogically inconsistent and their failure-handling routines have bugs. In most cases, the failure-handling policy is simplistic, and in some cases, even absent. This disregard for partial disk failures leads to many problems, ranging for loss of physical memory abstraction, to data corruption, and even to system-security violations.

The rest of the chapter is organized as follows. Section 5.1 provides a background on virtual-memory systems. Section 5.2 describes our fault-injection and analysis methodology. Section 5.3

presents experimental results, Section 5.4 analyzes the failure-handling approaches of the systems, and Section 5.5 concludes the chapter.

5.1 Virtual-Memory Systems

A virtual-memory system uses disk storage to provide applications with an address space larger than available physical memory. This helps the system execute multiple processes with large address spaces simultaneously. The disk area used by the virtual-memory system is called *swap space*. The virtual-memory system uses swap space to store memory pages that are not expected to be of immediate use. Typically, systems tend to remove pages that have not been accessed recently or that are not accessed frequently from memory and store them on disk (called *page-out*). When a page stored on disk is accessed again, it is brought back into physical memory (called *page-in*). The page-out/page-in process is transparent to applications (except for performance effects). Thus, the virtual-memory system is responsible for handling partial disk failures and maintaining the illusion that the page is actually in physical memory.

Virtual-memory systems make use of file systems in two scenarios. First, instead of directly using on-disk space, swap space can also be maintained as a file in a file system. Second, virtual-memory systems allow applications to memory-map file data (*e.g.* using the *mmap* system call). When a file (or a portion of a file) is memory-mapped, applications can operate on file data as if they were memory locations. User code pages are also memory-mapped from the executable file when a program is executed. In such scenarios involving a file system, the virtual-memory system depends on the file system to recover from or report partial disk failures.

The following subsections outline the features of two virtual-memory systems, Linux 2.6.13 and FreeBSD 6.0 whose failure-handling policies have been studied in this chapter. The features of the Windows XP virtual-memory system will be discussed with its evaluation in Section 5.3.4.

5.1.1 Linux 2.6.13

The Linux 2.6.13 virtual-memory system has largely been derived from the previous Linux versions. It performs swapping only for user-mode pages [25]. User-mode pages are the data, stack,

and code pages that form the user process. In order to keep the virtual-memory system simple, pages that belong to the kernel are not paged out. This simplification is not highly restrictive as kernel pages occupy only a small portion of main memory. The page replacement algorithm used is similar to the "2Q" algorithm [71]. When paged-out pages are accessed, space is created for the pages and they are read from disk. The system also issues reads in advance (*i.e.*, *read-ahead*) based on application accesses to improve performance. The swap area can either be a separate disk partition or a file in a file system. It contains a *swap header* that has information about the swap area like number of blocks, a list of faulty blocks and so on.

5.1.2 FreeBSD 6.0

The design of the virtual-memory system in FreeBSD is based on the Mach 2.0 virtual-memory system, with considerable updates over the years. The FreeBSD 6.0 virtual-memory system allocates pages when requested from a free list of pages and it maintains sufficient free pages by paging out less frequently used (inactive) pages [89]. The FreeBSD virtual-memory system also provides for paging out entire processes. This implies that in addition to user-mode pages, the kernel thread stacks of processes can be paged out and page tables can be freed when the system is under extreme memory pressure [89]. Unlike Linux, the FreeBSD virtual-memory system does not perform extra read-ahead; that is, it does not issue separate block read commands, although it tries to read as many as 8 blocks as part of one read command for a block that is needed. Like in Linux, the FreeBSD swap area can either be a disk partition or a file. The FreeBSD swap area does not have any data structures like the Linux swap header.

5.2 Methodology

In this section, we describe our fault injection and analysis methodology. As in the file-system study in the previous chapter, the methodology is primarily derived from type-aware fault injection described in Section 2.5.1. In this section, we first describe the failure model used, then describe our fault-injection framework, and finally discuss type-awareness for a virtual-memory system.

5.2.1 Failure Model

The different types of partial disk failures injected are read errors, write errors, and bit corruptions. In the case of read and write errors (latent sector errors experienced during read and write respectively), an error code (EIO) is returned to the virtual-memory system. We also zero-out the page in memory (ensuring that valid data is not placed in memory) if the read is failed with an error code. This zeroing-out is needed because the virtual-memory system may ignore an error code returned; in such a case, if valid data is placed in the respective memory page, the system may seem to work just fine. For bit corruption, the block contents are altered; we zero-out the block in our experiments and in case the corruption is detected, we perform a more detailed analysis, corrupting each field of the data structure with field-specific values in separate experiments. All partial disk failures are permanent; no amount of retrying of the disk operation will yield correct data.

5.2.2 Fault-Injection Framework

Our fault-injection framework consists of two components, the *harness* and the *injector*. The harness sets the system up for exposure to disk faults. This layer consists of three types of user processes: a *coordinator* for managing the benchmarking and fault injection, *victims* that allocate a large memory region, sleep for a while and then read the memory region, and *aggressors* that allocate large memory regions to force out the victims' pages to the swap area or to the file system. Partial disk failures are injected either when the victims' pages are paged out to disk or when they are read back by the victims.

The fault injection is performed by the injector, which interposes between the virtual-memory system and the hard disk. Specifically, the injector has been built as a pseudo-device driver for Linux 2.6.13, as a geom layer [89] for FreeBSD 6.0, and as an upper filter driver for Windows XP.

The failure-handling policy of the system is identified by a manual observation of the results of fault injection. Specifically, we use the following sources of information:

- The injector logs all I/O operations in detail, enabling us to determine some failure-handling policies; for instance, the logs show whether the virtual-memory system is performing retries

(read or write is repeated with the same disk block number) or remapping (disk write is repeated for a different disk block, but with the same memory page).

- The harness records all return values and signals received. This helps in determining whether an error is reported. The harness also checks (and reports) the validity of data read back. This helps in checking whether there is data corruption.

- We manually examine the system message log for any error messages recorded by the virtual-memory system.

We characterize the different failure-handling policies using the IRON taxonomy described in Section 2.4. The techniques described in this subsection are primarily used to determine detection and reaction policies. We discuss experiments to determine prevention policies in Section 5.3.3.

5.2.3 Type Awareness

We perform *type*-aware and *context*-aware fault injection by injecting partial disk failures for specific disk blocks at specific times. An example of a block type in a virtual-memory system is a user-level private data segment (*user data*). Therefore, a fault injected for a disk block that holds a private user data page is type-aware. A context is a basic function performed by the virtual-memory system or an interface offered by the virtual-memory system to applications. An example of a context is the `swapoff` system call. Therefore, a fault injected for a disk block when `swapoff` is in progress is context-aware. Table 5.1 presents various block types for which failures are injected and indicates which virtual-memory systems use them, and Table 5.2 presents different contexts when fault injection can be performed. The different types and contexts that can be explored are dependent on the particular system under study.

As discussed in Section 2.5.1, in order to perform type-aware fault injection, the injector should be able to detect the type of blocks being read or written. This detection is accomplished in a variety of ways. The harness communicates type information regarding data pages to the injector. For example, the harness allocates user data pages and initializes those pages to contain specific values and conveys the values to the injector. Thus, in such cases, the injector uses block *content*

Block Type	**Description**	**Detection**	**System**
swap header	Describes the swap space	Disk location	Linux
user data	Page from private user data segment	Content	Linux, FreeBSD
user stack	Page from user stack segment	Content	Linux, FreeBSD
shared	Shared memory page used by many processes	Content	Linux, FreeBSD
mmapped	Memory-mapped file data	Content	Linux, FreeBSD
user code	Page from user code segment	Disk location	Linux, FreeBSD
kernel stack	Page from kernel thread stack of a user process	Kernel information	FreeBSD

Table 5.1 **Block types.** *The table describes the different types of blocks that are failed and gives the detection method and applicable virtual-memory system for each type. In order to detect kernel thread stack pages, we made a simple modification to the FreeBSD kernel to obtain the memory addresses of these pages.*

Context	**Workload**	**Virtual-memory system actions**
`swapon`	Makes swap space available for swapping	Read swap header if any, initialize in-core structures
`swapoff`	Removes swap space from use	Page-in valid blocks and free the swap space
`pagetouch`	Page is accessed by the victim	Read page from disk
`readahead`	Workload induces readahead by reading nearby pages	Perform read-ahead by reading blocks from disk
`madvise`	Victim issues madvise (MADV_WILLNEED) to hint possible future reads	May or may not page-in the blocks specified in hint
`pageout`	Aggressors create memory pressure causing page-out	Write inactive memory pages to disk
`umount`	The file system is unmounted	May have to write of "dirty" mmaped file data
`complete`	Process scheduled again after complete page-out	Page-in essential data structures of process

Table 5.2 **Contexts.** *The table shows the workload for the different contexts that are used in the experiments and the actions performed by the virtual-memory system for each context.*

to determine the block type. Another method employed to determine the type is to use the *disk location* of the block. For example, the Linux *swap header* is always located at block 0 in the disk partition. Table 5.1 also provides the detection method for each block type.

Thus, we use fault injection to determine the failure-handling policies adopted by virtual-memory systems for different combinations of block type, context, and type of partial disk failure.

5.3 Experimental Results

In this section, we present the results of our type-aware fault-injection experiments on three virtual-memory systems, Linux 2.6.13, FreeBSD 6.0, and Windows XP. We have performed a detailed analysis of the Linux 2.6.13 and FreeBSD 6.0 virtual-memory systems, and a preliminary analysis of the Windows XP virtual-memory system. We first focus on IRON detection and reaction techniques of Linux and FreeBSD, then discuss prevention techniques of those systems, and finally evaluate Windows XP. The different levels of the IRON taxonomy are described in Section 2.4.

We present about 30 different scenarios (combinations of block type, context, and type of partial disk failure) for Linux and FreeBSD. All experiments involving swap space are performed using a separate disk partition as swap space (except for Windows XP), while experiments involving memory-mapped files or user code pages use the ext3 file system [141] in Linux 2.6.13 and the Unix File System (UFS2) [89] in FreeBSD 6.0. The observed failure-handling policy for experiments involving a file system is a combination of the policies of the virtual-memory system and the file system.

5.3.1 Linux 2.6.13

Tables 5.3 and 5.4 present the results of fault injection on the Linux 2.6.13 virtual-memory system.

Detection: We find that most read errors are detected using $D_{ErrorCode}$, which is checking of return codes. The exceptions occur during `swapoff` (when the virtual-memory system pages valid blocks into memory); the error is not detected (D_{Zero}) and the application to which the data

		swapon	swapoff	pagetouch	readahead	madvise	pageout	umount
Read Errors	*user data*	—	**Z**	**E**	**E**	—	—	—
	user stack	—	**Z**	**E**	**E**	—	—	—
	shared	—	**Z**	**E**	**E**	—	—	—
	mmapped	—	—	**E**	**E**	**E**	—	—
	user code	—	—	**E**	—	—	—	—
	swap header	**E**	—	—	—	—	—	—
Write Errors	*user data*	—	—	—	—	—	**Z**	—
	user stack	—	—	—	—	—	**Z**	—
	shared	—	—	—	—	—	**Z**	—
	mmapped	—	—	—	—	—	**Z**	**Z**
	user code	—	—	—	—	—	—	—
	swap header	—	—	—	—	—	—	—
Corruption	*user data*	—	**Z**	**Z**	**Z**	—	—	—
	user stack	—	**Z**	**Z**	**Z**	—	—	—
	shared	—	**Z**	**Z**	**Z**	—	—	—
	mmapped	—	—	**Z**	**Z**	**Z**	—	—
	user code	—	—	**Z**	—	—	—	—
	swap header	**Y**[1]	—	—	—	—	—	—

Symbols: **Z** Zero **E** Errorcode **Y** Sanity **—** Not applicable

Comments (1) Sanity checks for swap space signature, version number and bad block count

Table 5.3 **Linux 2.6.13 detection techniques.** *This table presents the Linux 2.6.13 detection techniques for read errors, write errors, and corruptions for combinations of block type (rows) and context (columns). Comments, if any, are provided below the tables.*

		swapon	swapoff	pagetouch	readahead	madvise	pageout	umount
Read Errors	*user data*	—	**Z**	**P**[1]	**D**[5]	—	—	—
	user stack	—	**Z**	**P**[1]	**D**[5]	—	—	—
	shared	—	**Z**	**P**[1,6]	**D**[5]	—	—	—
	mmapped	—	—	**R**[2],**P**[1]	**D**[5]	**D**[5]	—	—
	user code	—	—	**R**[2],**P**[1]	—	—	—	—
	swap header	† **R**[3,4]	—	—	—	—	—	—
Write Errors	*user data*	—	—	—	—	—	**Z**	—
	user stack	—	—	—	—	—	**Z**	—
	shared	—	—	—	—	—	**Z**	—
	mmapped	—	—	—	—	—	**Z**	**Z**
	user code	—	—	—	—	—	—	—
	swap header	—	—	—	—	—	—	—
Corruption	*user data*	—	**Z**	**Z**	**Z**	—	—	—
	user stack	—	**Z**	**Z**	**Z**	—	—	—
	shared	—	**Z**	**Z**	**Z**	—	—	—
	mmapped	—	—	**Z**	**Z**	**Z**	—	—
	user code	—	—	**Z**	—	—	—	—
	swap header	**P**	—	—	—	—	—	—

Symbols: **Z** Zero **P** Propagate **R** Retry **D** Record — Not applicable

Comments (1) `SIGBUS` signal (2) One separate retry for every block needed in the original request (3) Retry is not actually used (4) Operation fails but success is returned (error is not reported) (5) This operation is remembered when page is actually touched (6) Error is reported to all processes that touch the page after the read error occurs

Table 5.4 **Linux 2.6.13 reaction techniques.** *This table presents the techniques used by Linux 2.6.13 to react to read errors, write errors and corruptions for combinations of block type (rows) and context (columns). † indicates a possible bug in the implementation. Comments, if any, are provided below the tables.*

belongs is given junk data on a future memory access. This could lead to application crashes or data corruption.

None of the write errors are detected (D_{Zero}). A read of the page after an ignored write error causes the virtual-memory system to page-in the disk block with its previous contents. Missing these errors can lead to application crashes or application data corruption (because of bad data) or even system security problems since the application could possibly read data that belongs to another process.

Almost all corruptions are not detected and the corrupted data is returned to the application. One exception is the use of D_{Sanity} for the swap header during `swapon`. The checks are for (a) the correct swap space signature (*i.e.*, a type check) (b) the correct version number, and (c) the number of bad blocks being less than the maximum allowable. When we used a zeroed-out block as corrupted data, the check for the swap space signature occurred; we then modified specific fields in the swap header without modifying the signature to identify the other sanity checks.

Reaction: For cases where the partial disk failure is detected, Linux uses basic reaction mechanisms. On a read error for an application-accessed page, the `SIGBUS` signal is used to inform the application of an error (R_{Report}). In the case of a shared memory page, all processes that touch the page *after* the read error occurs receive the `SIGBUS` signal – in other words, the virtual-memory system does not retry the read when each process accesses the page. Another use of R_{Report} is when the swap header is corrupted, in which case an error is returned for the `swapon` call.

In the experiments with memory-mapped file data and user code, a retry is observed (R_{Retry}) for the specific disk block that the system actually needs; even if the original operation involved many disk blocks, the retry is performed for only one block. This retry may have been initiated by the file system and not the virtual-memory system. When a read to the swap header fails during `swapon`, a retry is performed (R_{Retry}), but perhaps due to implementation bugs, the results of the retry are not actually used. Also, `swapon` returns success during read errors even though the call fails internally (*i.e.*, it does not report the error).

R_{Record} is used to handle read errors for `readahead` and `madvise`. By using R_{Record}, the system records the failure of the read for future reference. In both `readahead` and `madvise`, the data is

not required immediately – read-ahead is only an optimization by the virtual-memory system and madvise is only a hint that the block will likely be accessed. In the `readahead` case, the error is reported when the page is actually touched and for `madvise`, a retry is performed when the page is touched – both actions use the fact that the first read was unsuccessful.

5.3.2 FreeBSD 6.0

Tables 5.5 and 5.6 present the results of fault injection on the FreeBSD 6.0 virtual-memory system.

Detection: $D_{ErrorCode}$ is used in every single case for detecting both read and write errors – the FreeBSD 6.0 virtual-memory system always checks the error code returned. FreeBSD does not detect block corruption (D_{Zero}). While this leads to application crash or data corruption in most cases, it leads to a kernel crash when corruption of kernel thread stack blocks is not detected; in this case serious errors like system becoming unbootable are also possible.

Reaction: Various reaction mechanisms are used in FreeBSD 6.0 to deal with detected errors. R_{Retry} is used when memory-mapped data is written during a file system unmount. In fact, the system retries as many as 6 times for each `umount` call. We believe that these retries are performed by the file system and not the virtual-memory system (we still document the behavior here since it is the behavior observed by an application using memory-mapped file data, a feature supported by the virtual-memory system).

Read errors during page accesses cause the virtual-memory system to deliver a `SIGSEGV` (segmentation fault) to the application, an instance of R_{Report}. Experiments showed that in the case of shared memory, unlike in Linux, processes sharing the memory region operate independently; that is, even if the error has been reported to one of the processes that accessed the page, the disk access is retried when a second process accesses the page. R_{Report} is also used when all write retries are failed during `umount`; an I/O error is returned to the application.

R_{Stop} is used for read errors during `swapoff` and for read errors during a page-in of the kernel thread stack. In both cases, the result is a kernel `panic`, a conservative action. During `pageout`, the virtual-memory system attempts to free memory pages by writing them to swap space. If write

		swapon	swapoff	pagetouch	madvise	complete	pageout	umount
Read Errors	*user data*	—	**E**	**E**	—	—	—	—
	user stack	—	**E**	**E**	—	—	—	—
	shared	—	**E**	**E**	—	—	—	—
	mmapped	—	—	**E**	—	—	—	—
	user code	—	—	**E**	—	—	—	—
	kernel stack	—	**E**	—	—	**E**	—	—
Write Errors	*user data*	—	—	—	—	—	**E**	—
	user stack	—	—	—	—	—	**E**	—
	shared	—	—	—	—	—	**E**	—
	mmapped	—	—	—	—	—	**E**	**E**
	user code	—	—	—	—	—	—	—
	kernel stack	—	—	—	—	—	**E**	—
Corruption	*user data*	—	**Z**	**Z**	—	—	—	—
	user stack	—	**Z**	**Z**	—	—	—	—
	shared	—	**Z**	**Z**	—	—	—	—
	mmapped	—	—	**Z**	—	—	—	—
	user code	—	—	**Z**	—	—	—	—
	kernel stack	—	**Z**	—	—	**Z**	—	—

Symbols: **Z** Zero **E** Errorcode — Not applicable

Table 5.5 **FreeBSD 6.0 detection techniques.** *This table presents the FreeBSD 6.0 detection techniques for read errors, write errors, and corruptions for combinations of block type (rows) and context (columns). FreeBSD does not read any block during* `swapon` *and does not read pages in for* `madvise` *(— in the table).*

		swapon	swapoff	pagetouch	madvise	complete	pageout	umount
Read Errors	*user data*	—	**S**[3]	**P**[2]	—	—	—	—
	user stack	—	**S**[3]	**P**[2]	—	—	—	—
	shared	—	**S**[3]	**P**[2]	—	—	—	—
	mmapped	—	—	**P**[2]	—	—	—	—
	user code	—	—	**P**[2]	—	—	—	—
	kernel stack	—	**S**[3]	—	—	**S**[3]	—	—
Write Errors	*user data*	—	—	—	—	—	**D**[4]	—
	user stack	—	—	—	—	—	**D**[4]	—
	shared	—	—	—	—	—	**D**[4]	—
	mmapped	—	—	—	—	—	**D**[4]	**R**[5],**P**[6]
	user code	—	—	—	—	—	—	—
	kernel stack	—	—	—	—	—	**D**[4]	—
Corruption	*user data*	—	**Z**	**Z**	—	—	—	—
	user stack	—	**Z**	**Z**	—	—	—	—
	shared	—	**Z**	**Z**	—	—	—	—
	mmapped	—	—	**Z**	—	—	—	—
	user code	—	—	**Z**	—	—	—	—
	kernel stack	—	**Z**[1]	—	—	**Z**[1]	—	—

Symbols: **Z** Zero **P** Propagate **R** Retry **D** Record **S** Stop **—** Not applicable

Comments: (1) Kernel crash when the stack is used (2) `SIGSEGV` signal (3) Kernel panic (4) Memory page not freed; alternate victim chosen for page-out (5) Upto 6 retries of the disk write (for all blocks) (6) I/O error returned

Table 5.6 **FreeBSD 6.0 reaction techniques.** *This table presents the techniques used by FreeBSD 6.0 to react to read errors, write errors, and corruptions for combinations of block type (rows) and context (columns). Comments, if any, are provided below the tables. FreeBSD does not read any block during* `swapon` *and does not read pages in for* `madvise` *(— in the table).*

errors occur during this page-out process, the FreeBSD virtual-memory system reaction is R_{Record}. In this case, the virtual-memory system remembers that the write operation has not been performed successfully, so that the memory page is not freed. Since the virtual-memory system is not able to successfully free the memory page, it proceeds to select an alternate victim for page-out.

5.3.3 Prevention Techniques

Determining prevention policies is more difficult than determining detection and reaction policies since the prevention policy may not be triggered by a particular disk fault. Therefore, our methodology for uncovering the prevention policy is to use a specific test for each prevention technique.

$P_{Remember}$ is the only prevention technique that may be triggered by faults. We test for use of $P_{Remember}$ by injecting a "sticky" error repeatedly for the same disk block and checking whether the virtual-memory system stops using the disk block. The workload performs 10 iterations of a page-out/page-in of victim pages (*user data*). For both Linux 2.6.13 and FreeBSD 6.0 we find that the "bad" disk block is used repeatedly, in spite of returning an error each time. The same results are obtained for both read and write errors. This indicates that Linux and FreeBSD likely do not keep track of bad blocks (*i.e.*, $P_{Remember}$ is not used).

We test for $P_{LoadBalance}$ by causing the virtual-memory system to page-out many user data pages numerous times and checking whether all blocks in the swap area are used fairly evenly. This workload performs 10 iterations of a page-out/page-in of victim pages. In both Linux and FreeBSD, the same disk blocks are reused repeatedly, even though many other blocks in the swap area have not been written to even once. This indicates that the systems likely do not perform wear-leveling (*i.e.*, $P_{LoadBalance}$ is not used).

Finally, to detect P_{Reboot} and P_{Scan} we simply observe whether these activities occur over an interval of using the virtual-memory system. Given that we did not observe any instance of P_{Reboot} or P_{Scan} during any of our experiments, we infer that it is likely that neither Linux nor FreeBSD use these techniques. In summary, our experiments indicate that neither Linux 2.6.13 nor FreeBSD 6.0 appear to use any of the prevention techniques.

5.3.4 Windows XP

This subsection first outlines particular features of the Windows XP virtual-memory system, then discusses its failure-handling policies. Windows XP uses a file in an NTFS partition to store memory pages that get paged-out. Therefore, the failure-handling policy we extract is a combination of policies of NTFS and the virtual-memory system. Windows XP allows for paging out of both user and kernel memory. We inject faults only for *user data pages*. Read errors and corruption are injected during `pagetouch` and write errors are injected during `pageout`. We use the error code STATUS_DEVICE_DATA_ERROR for read and write errors.

Detection: Windows XP uses the error code returned by the disk to detect both read and write errors ($D_{ErrorCode}$). Corruptions are not detected (D_{Zero}).

Reaction: Reaction to read errors is terminating the user application, reporting the error *InPageError* (R_{Report}). The reaction to write errors is more involved. It primarily uses R_{Record}: the memory pages for which the error occurs are written elsewhere when they are selected for paging out again. As for the disk block with the error, it is first read back. If this read succeeds, a half-block write is performed. If the read fails, a half-block read is performed. Irrespective of the success or failure of the half-block operations, the block is used for future writes, using R_{Record} to deal with any errors to these writes. We have not been able to identify the purpose of the half-block operations. Also, after a transient write error, although the disk blocks are subsequently successfully written, they are not read back even when the application accesses the data, thereby leading to the application receiving junk data. This indicates a possible bug in handling write errors. Further investigation is required to ascertain this behavior.

Prevention: Fault-injection experiments demonstrated that a given disk block is not re-used after about 6 errors for the block ($P_{Remember}$). The block is added to a *bad cluster file* and is never used again unless the disk is re-formatted. We did not observe the use of any of the other prevention techniques.

5.4 Discussion

In this section, we first discuss the failure-handling approach of the virtual-memory systems, and then discuss our experience with the fault injection techniques used.

5.4.1 Failure-Handling Approaches

In this section, we discuss the approaches that current virtual-memory systems adopt to handle disk failures, contrasting the techniques used and identifying the deficiencies of the systems. We also compare the approach of virtual-memory systems to that of file systems (discussed in Section 2.5.2). We start by summarizing the different approaches of the virtual-memory systems:

Linux: Linux fails to detect many partial disk failures (even ones where error codes are returned) and follows simple reaction schemes to deal with detected errors. With respect to corruption, only *swap header* corruption is detected.

FreeBSD: FreeBSD correctly detects all disk errors with error codes, but ignores corruptions. It uses simple reaction schemes to deal with errors, although it is more conservative than Linux for some cases – the kernel calls `panic` to stop the entire system when a read fails during `swapoff`, even if the read affects only a single application.

Windows XP: Windows XP detects disk errors with error codes but ignores corruptions. It uses simple reaction schemes. It is the only system for which we observed a prevention technique ($P_{Remember}$).

In general, the systems suffer from the following deficiencies:

Simple reaction techniques: The virtual-memory systems studied use only simple reaction techniques to deal with partial disk failures. There is no attempt to use techniques like redundancy to completely recover from partial disk failures.

Ignore data corruption: Of all our data corruption experiments, only one case (Linux *swap header*) is detected. Virtual-memory systems assume that disks store data reliably, which may not be true for commodity hardware.

Under-developed mechanisms: A prime example of an under-developed mechanism is remembering bad blocks. The Linux swap header has a provision to store a list of bad blocks. This list can be used effectively to prevent data loss ($P_{Remember}$). However, the list is initialized during `mkswap` and not updated afterward when new errors occur (on the other hand, Windows XP actively uses and updates a bad cluster file to avoid using error-prone blocks).

Memory abstraction mismatch: Applications expect all their pages to behave as if they are always in memory. The virtual-memory system should maintain this memory abstraction even when partial disk failures occur. An important part of maintaining the abstraction is error reporting. If an failure cannot be handled by the system, it should be reported in a manner that fits the memory abstraction. For example, Linux uses the `SIGBUS` signal to report page read errors (by definition, hardware failures can cause `SIGBUS` to be generated). However, FreeBSD uses the `SIGSEGV` signal (which almost always is intended to indicate a programming error) to report read errors, which is not appropriate.

Very few retries: There were very few instances of retrying an operation when an error occurs. Retrying can solve the problem in the case of a transient error and systems would benefit greatly by employing retries [53].

Illogical inconsistency: The reaction techniques employed are inconsistent for cases which are not very different. For example, in FreeBSD, a read error for a user data page may result in a report in one case (`pagetouch`), while it results in kernel panic in another (`swapoff`).

Buggy implementation: It is observed in Linux that failure-handling code is buggy. For example, the result of a retry is ignored, making it useless. We suspect that failure-handling code is rarely tested and is thus likely to have bugs, as seen elsewhere [93].

Security issues: A system that is fairly secure during normal operation could become insecure when there is a partial failure. In Linux, when data is read back after a failed write, the disk block's previous contents are returned to the application, possibly delivering data

that the application is not authorized to read. Such failures need to be dealt with given that there is an increasing awareness towards exploiting even transient hardware errors to attack systems [50].

Kernel exposure: Systems should take special care when kernel-mode data is stored on disk. In FreeBSD, corruption of the kernel thread stack is not detected. This may result in undesirable crashes or severe data corruption.

5.5 Conclusion

The virtual-memory system is an important component of the storage stack in nearly every operating system. Therefore, virtual-memory systems should be designed to handle partial disk failures. From our fault-injection experiments, we find that current virtual-memory systems do not employ consistent failure policies that provide complete recovery from partial disk failures. Improving the failure-awareness of these systems would enable them to truly virtualize memory, providing applications with a robust memory abstraction.

Chapter 6

Impact on File Systems

A file system is a crucial component of the storage stack; most applications use file systems to store data. In commodity systems, such as desktops and laptops, file systems are also tasked with the responsibility of ensuring that the data is stored reliably. Since partial disk failures could have a huge adverse impact on data reliability, it is extremely important to understand how current file systems react to partial disk failures.

In previous work [104] (described in Section 2.5), we examined how various file systems handle latent sector errors and completely-corrupt disk blocks. In this chapter, we develop a more thorough understanding of how file systems react to corruption. In particular, we perform targeted non-random corruption of on-disk pointers of file systems.

File systems today use a variety of techniques to protect against corruption. ReiserFS, JFS, and Windows NTFS perform lightweight corruption checks like type checking [104]; that is, ensuring that the disk block being read contains the expected data type. These file systems also employ sanity checking (verifying that particular values in data structures follow certain constraints) to detect corruption [104]. ZFS checksums both data and metadata blocks to protect against corruption [130]. The techniques above are useful for detecting corruption. In order to recover from corruption, most systems rely on replicated data structures. For example, JFS and NTFS replicate key data structures, giving them the potential to recover from corruption of these structures [20, 128].

We seek to evaluate how a set of corruption-handling techniques work in reality. While conceptually simple, there may be design or implementation details that preclude a file system from reaping the full reliability benefit of these techniques. We evaluate file systems using software fault injection.

One difficulty with a pointer-corruption study is the potentially huge exploration space for corruption experiments. To deal with this problem, we develop a fault-injection technique called *type-aware pointer corruption*, an extension of type-aware fault injection. Type-aware pointer corruption (TAPC) reduces the search space by systematically changing the values of only one disk pointer of each type in the file system, then exercising the file system and observing its behavior. We further narrow the large search space by corrupting the disk pointers to refer to each type of data structure, instead of to random disk blocks. The technique is successful because different block types are used differently by the file system, thereby implying that the blocks and the pointers that point to them might be protected in different ways. An important advantage of TAPC is that it helps understand the underlying causes for observed system behavior. TAPC works outside the file system, obviating the need for source code.

We use TAPC to evaluate two widely-used file systems, Windows NTFS [128] and Linux ext3 [141]. We examine their use of type checking, sanity checking, and replication to deal with corrupt pointers. We ask the simple question: *do these techniques work well in reality?* We focus on NTFS in this study; NTFS is a closed-source system for which little information is available about exact failure policies, thus making its study very interesting. Our analysis of ext3 is less-detailed, primarily aimed at demonstrating the general utility of our approach.

From our pointer-corruption experiments, we find that both file systems fail to recover from many pointer corruptions despite the availability of redundant information. This failure to recover is due to poor use of techniques like type checking and replication. Our observations help us identify several lessons and pitfalls for building corruption-proof file systems.

The rest of this chapter is organized as follows. Section 6.1 discusses why we explore pointer corruption. Section 6.2 describes type-aware pointer corruption. Section 6.3 presents a brief overview of NTFS and how we have applied TAPC to study NTFS. Section 6.4 presents the results of our analysis of both NTFS and ext3, and Section 6.5 concludes the chapter.

6.1 Why Pointer Corruption?

Although any block on disk may become corrupt, some corruptions are more damaging than others. If a data block of a file is corrupt, then only the application that reads the file is impacted. However, if a disk block belonging to file-system metadata is corrupt, then the entire file system can be affected; for example, if the boot sector is corrupt, the file system may not be mountable. In other cases, a corrupt on-disk pointer incorrectly referring to data belonging to a different data structure can cause the data to be overwritten and corrupted. Therefore, an integral part of ensuring the long-term availability of data is ensuring the reliability and availability of pointers, the *access paths* to data.

Pointers are fundamental to the construction of nearly all data structures. This observation is especially true for file systems, which rely on pointers located in on-disk metadata to access data. This reliance necessitates that pointers should be managed with care. File systems researchers have long recognized the salience of metadata and especially of pointers in metadata with a view to *on-disk consistency management*. For example, many early UNIX file systems [88, 111] carefully order writes to prevent the creation of bad on-disk pointers. Subsequent work on soft updates [46] and journaling file systems [60, 120] also treat pointers with care to maintain metadata consistency in the presence of crashes.

On-disk pointers could become corrupt due to the any of the silent data corruptions we study. Since pointers are fundamental to data access, it is important to understand how file systems behave when their on-disk pointers are corrupt. Therefore, in this chapter, we explore this facet of file-system behavior.

6.2 Type-Aware Pointer Corruption

To identify the behavior of file systems when disk pointers are corrupted, we develop and apply *type-aware pointer corruption* (TAPC). TAPC is an extension of type-aware fault injection described in Section 2.5.1. We observe how the file system responds after we modify different types of on-disk pointers to refer to disk blocks containing different types of data.

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 [123]. 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.

We use type-awareness to address both problems. 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 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 within the file system that exercise the same types of pointers are the same, and disk blocks of the same type of data structure contain similar contents. Thus, TAPC 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.

6.2.1 Terminology

The following terms are used to describe methodology and discuss results.

$Container$**:** disk block in which the disk pointer is present. Corrupting the pointer involves modifying the contents of the $container$.

$Target_{original}$**:** disk block that the disk pointer should point to, that is, the block pointed to on no corruption.

$Target_{corrupt}$**:** disk block being pointed to by a corrupt disk pointer.

6.2.2 Corruption Model

Any of the sources of corruption discussed in Section 2.2.1 could produce a corrupt file system image on disk. Our corruption 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.
- The pointer corruption is *not sticky*. Future writes to the pointer by the file system can potentially correct the corruption. Reads performed after a write will be returned the newly written data and not the corrupt data.

6.2.3 Corruption Framework

Our TAPC framework has been designed to work without file system source code. It consists of a *corrupter* layer that injects pointer corruption and a *test harness* that controls the experiments. The corrupter resides between the file system and the disk drivers; the layer has been implemented as a Windows filter driver for NTFS and as a pseudo-device for ext3. This layer corrupts disk pointers and observes disk traffic. Thus, the corrupter has knowledge of the file system's on disk data structures [127]. The test harness is a user-level program that executes file system operations and controls the corrupter. The experiments involve the following steps:

- The test harness creates a file system on disk with a few files and directories. It then instructs the corrupter to corrupt a specific pointer to a specific value and performs file operations (*e.g.*, mount, `CreateFile`, etc. for NTFS and mount, `creat`, etc. for ext3) to exercise the pointer under consideration. We execute the file operations from a user with limited permissions (non-administrator).

- The corrupter intercepts the disk accesses performed by the file system and scans the requests for the $container$ (the disk block containing the pointer). When that disk block is read, exactly one pointer in the data structure is modified to a specific value.
- The corrupter continues to monitor disk accesses. The same corruption is performed on future reads to the $container$. Disk writes to the $container$ may overwrite any corruption and therefore further reads to the disk block are returned the newly-written data.
- All disk accesses, system call return values, and the system event log are examined in order to identify the behavior of the file system. This holistic view of system behavior in co-ordination with type-awareness is essential to understanding the underlying design or implementation flaws that lead to any system failures.

Our experiments are performed on an installation of Windows XP (Professional Edition without Service Pack 2) for NTFS and Linux 2.6.12 for ext3. We run them both on top of VMWare Workstation for ease of experimentation. The experiments use a separate 2GB IDE virtual disk. We believe that the use of VMWare does not change the results; since the corrupter layer is between the file system and the virtual disk, we observe all disk requests and responses, and we did not detect any anomaly. In addition, we have verified our ext3 results by reading through source code.

6.3 NTFS Details

Although TAPC can be applied to any file system, the specific pointers to be corrupted and the interesting corruption values depend upon the file system under test. As an example, we now describe how we have applied TAPC to NTFS.

6.3.1 NTFS Data Structures

We provide a brief introduction to NTFS. A detailed description can be found elsewhere [128]. NTFS, the Windows NT File System, is the standard file system for Windows NT, 2000, XP and Vista. It is a journaling file system that guarantees the integrity of its metadata structures on a crash. All user data and metadata structures in an NTFS volume are contained in files, allowing

Term	Description
Cluster	The fundamental unit of disk storage; it consists of a fixed number of sectors, similar to a UNIX disk block.
LCN	A Logical Cluster Number (LCN) is assigned to each disk cluster. This is the same as a physical block number in UNIX-based systems. On-disk pointers contain the LCN of the cluster they point to.
VCN	A Virtual Cluster Number is the same as a file offset (in number of blocks) in UNIX.
Data run	The format of NTFS on-disk pointers, consisting of a base LCN and length, and a series of <offset,length> fields. E.g., if base LCN is X, length is a, and the first <offset,length> combination is $< b, c >$, the data is located at LCNs X to $X + a$ and then from $X + b$ to $X + b + c$. In our experiments we corrupt the base LCN.
Boot sector	The boot sector is the sector read first by NTFS when the file system is mounted. It is the starting point for discovering the LCNs of all other data structures. The last cluster of the file system contains a copy of the boot sector.
MFT	Master File Table contains an entry for each file (both user and system). First 24 entries are reserved for system files.
MFT entry	Equivalent of a UNIX inode. Most pointers that are corrupted are located in different MFT entries in form of data runs.
MFT VCN 0	This is the first cluster of the MFT. Its LCN is present in the boot sector. The first entry of this cluster is a file that contains LCNs of itself and the rest of the MFT.
MFT mirror	This is a replica of MFT VCN 0. Its LCN is also present in the boot sector.
Index buffer	An index buffer consists of a series of index entries that provide information for indexing into any data structure.
Directory	A directory in NTFS consists of index buffers. The entries in these buffers point to MFT entries of the directory's files.
MFT bitmap	This is a bitmap that tracks whether MFT entries are allocated or not.
Volume bitmap	This is a bitmap that tracks whether disk clusters are allocated or not.
Log file	NTFS implements ordered journaling: when a user writes data to disk, the data cluster is flushed first, followed by log updates, and finally the metadata clusters. The log file is organized as a restart area, a copy of the restart area, and a "logging area", which consists of log records that each denote a disk action.
$Secure	NTFS stores information about the owner of the file and the permissions granted to other users by the owner (in form of ACLs) in a security descriptor. Each unique descriptor is stored in $Secure along with its hash and given a *security id*. This security id is stored in the MFT entry of the file for looking up the correct descriptor from $Secure. The descriptors in $Secure are indexed on the hash of the security descriptor and the security id.
Upcase table	This is an upper case - lower case character conversion table essential for directory path name traversal.

Table 6.1 **NTFS terminology.** *This table provides brief descriptions of NTFS terminology and data structures. The descriptions offer a simplified view of NTFS, eliminating details that are not essential for understanding the experiments.*

NTFS to flexibly allocate disk space for its metadata. Table 6.1 defines important NTFS terms and data structures that we use in our descriptions and results. For example, a *cluster* is the NTFS term for a disk block. We recommend a quick pass through the table for easier understanding of the rest of the chapter.

6.3.2 NTFS Pointer Corruption

We corrupt 14 of the 15 different pointer types that NTFS uses on disk. Table 6.2 summarizes these pointers. We give each pointer a unique name based on its $Target_{original}$, and resolving name conflicts by prefixing those names with its $container$. Note that NTFS replicates important data structures like Boot and MFT VCN 0. Thus, the pointers `Boot-MFT0`, `Boot-MFTM`, `MFT0-MFT`, `MFTBitmap`, `MFT0-MFTM`, and `LogFile` are replicated. Security descriptors are also replicated and their indexes can be rebuilt; thus, some form of redundancy exists for the pointers `SDS`, `SDH`, and `SII`. Also note that the security descriptor indexes `SDH` and `SII`, and directories use the same index data structure format.

The pointers are corrupted to 27 different types of values. In addition to using disk locations that belong to all the different NTFS data types (*e.g.*, directory index buffer and MFT cluster), we also include clusters of a certain type that serve a special purpose (*e.g.*, MFT VCN 0, MFT mirror), unallocated clusters, and out-of-range values. Table 6.3 lists the different types of values used as $Target_{corrupt}$. In most cases, the data structure used as $Target_{corrupt}$ is at a specific location, while for FileData, we create a file and use the location of its data block as the numerical value for corruption. Thus, we perform 360 experiments on NTFS, corrupting 14 different pointers with 27 different values.

In each experiment, we run a specialized workload to exercise the corrupt pointer. Table 6.4 indicates the workload used for each of the pointers. Most workloads involves modifications to $Target_{original}$, potentially creating the worst case scenario in case the corruption is not detected. We now describe the disk accesses that take place during the mount workload, as this is our workload for exercising most of the disk pointers. When an NTFS volume is mounted, first the boot sector is read. The boot sector is used by NTFS to discover the on-disk location of MFT VCN 0

Pointer	**Container**	$Target_{original}$
`Boot-MFT0`	Boot	MFT VCN 0
`Boot-MFTM`	Boot	MFT mirror
`MFT0-MFT`	MFT VCN 0	The MFT clusters (to itself)
`MFTBitmap`	MFT VCN 0	MFT bitmap
`MFT0-MFTM`	MFT VCN 0	MFT mirror
`LogFile`	MFT VCN 0	Log file
`RootSecDesc`	MFT VCN 1	Root directory security descriptor
`RootIndxBuf`	MFT VCN 1	Root directory index buffers
`SDS`	MFT VCN 2	$Secure security descriptors
`SDH`	MFT VCN 2	Index of security descriptors' hash
`SII`	MFT VCN 2	Index of security descriptors' ids
`UpCase`	MFT VCN 2	Upcase table
`DirIndxBuf`	MFT any VCN	A directory's index buffer
`FileData`	MFT any VCN	A file's data cluster

Table 6.2 **NTFS disk pointers.** *This table presents the different on-disk pointers used by NTFS.*

Value	Description
Boot	The boot sector (LCN 0)
LogRes	Log restart area
LogResDup	Copy of Log restart area
LogData	Log data cluster
MFTBitmap	The MFT bitmap
MFT0	MFT VCN 0
MFT1	MFT VCN 1
MFT2	MFT VCN 2
MFTRes	Contains unused, reserved MFT entries
MFTFree	Unallocated MFT entries
MFT6	MFT VCN 6
MFTOthers	Contains user file MFT entries
SDS	Security descriptors
AttrDef	File with definitions of file attributes
SDH	Index of security descriptor hash
SII	Index of security descriptor ids
MFTMirror	The MFT mirror
RootIndxBuf	Root directory index buffer
RootSecDesc	Root dir security descriptor
VolBitmap	Volume bitmap
UpCase	Upcase table
DirIndxBuf	Any directory index buffer
FileData	Any user file data cluster
Unalloc	Unallocated clusters
Last-Size+1	Data Run ends at last cluster
LastCluster	Boot sector copy
Out-of-Bounds	Data Run exceeds disk partition

Table 6.3 **NTFS pointer corruption values.** *This table presents the different values used for corrupting disk pointers used by NTFS, sorted in the order of typical disk location. In total, 27 different values are used. Note that the value Last-Size+1 is applicable only for pointers that point to data runs of length* > 1.

Workload	Pointer
mount	Boot-MFT0, Boot-MFTM, MFT0-MFT, MFT0-MFTM, LogFile, RootSecDesc, SDS, SII
mount then CreateFile	MFTBitmap, RootIndxBuf, SDH, DirIndxBuf
mount then ReadFile	UpCase
mount then WriteFile	FileData

Table 6.4 **NTFS workloads.** *This table presents the workloads that exercise the disk pointers. mount enables the file system volume for use; it consists of a* `DeviceIoControl` *system call with the control code FSCTL_UNLOCK_VOLUME performed on a previously "locked" volume.* `CreateFile` *creates a new file of size 0,* `ReadFile` *reads the first cluster of a file, and* `WriteFile` *writes the first cluster of a file.*

and its replica, MFT Mirror, and both clusters are read. MFT's VCN 0 is itself a $container$ for four more pointers, including ones to the MFT bitmap (`MFTBitmap`) and the logfile (`LogFile`). The logfile pointer is then used to perform a series of log operations. MFT's VCN 0 also contains its self pointer (`MFT0-MFT`). The cluster indicated by this self pointer is read. In the absence of corruption, this leads to MFT VCN 0 being read again. This is followed by reads to the rest of the system file records in the MFT. NTFS now has the locations of various system data structures. The $UpCase table is read in next, followed by other data structures, namely, the volume bitmap, MFT bitmap, root directory's security descriptor, root directory's index allocation structure and clusters of the $Secure file. Finally, a series of log operations mark the termination of a successful mount.

6.4 Results

This section discusses the results of our analysis. First, we describe some terminology, then our visual representation of the results. Then, we discuss NTFS behavior as observed by the experimenter. Our discussion focuses on how NTFS deals with pointer corruption, whether NTFS misses opportunities to improve fault tolerance, and what design principles are useful with respect to dealing with pointer corruption. Next, we discuss the user-visible results of NTFS pointer corruption. This view is important since the primary concern of end users is the observed data and system reliability. Finally, we present results for ext3. We organize our results into *observations* (facets of system behavior uncovered by TAPC), *lessons* for corruption-handling techniques, and potential design *pitfalls*.

6.4.1 Terminology for System Behavior

We use a subset of the detection and reaction policies of IRON taxonomy described in Section 2.5 to characterize system behavior. Specifically, we use the following terms:

Detection: The file system identifies that either the pointer or the disk block pointed to is corrupt. The IRON detection level is typically D_{Sanity} (that includes sanity checks and type checks).

Recovery: The file system is able to regenerate the data lost due to pointer corruption using redundant information, thereby continuing execution without errors. The corresponding IRON reaction level is $R_{Redundancy}$.

Report: The file system informs the application or user that it has encountered an error (IRON reaction level R_{Report}).

Retry: The file system repeats the set of disk accesses needed for the mount operation (IRON reaction level R_{Retry}).

Repair: The file system modifies corrupt data structures in order to continue execution. The modification does not necessarily lead to error-free execution (IRON reaction level R_{Repair}).

Detection is essential for the rest of the actions to occur. Recovery is the ideal action the file system can perform. If recovery is not possible, repair is an alternative approach for continuing execution. If a file operation fails due to corruption, the file system is expected to report an error.

6.4.2 Visualization of Results

Presenting the data from our experiments is a difficult task, as the data represents the results of hundreds of experimental runs, and the outputs are not readily quantified. We divide the observations into two: the behavior of NTFS as observed by the experimenter (Figure 6.1, Tables 6.5 and 6.6), and the user-visible results (Figure 6.2).

We now describe the visualization in Figures 6.1 and 6.2. In the two figures, each row presents the results of corrupting one pointer (*e.g.*, `Boot-MFT0`). Every row is divided into 27 columns, each corresponding to different $Target_{corrupt}$ values used to corrupt the pointer (*e.g.*, LogData). Each cell is marked with a symbol representing our observations when the pointer for its row is corrupted with the column value. A dot before pointer name indicates that some form of redundancy exists for the pointer or for $Target_{original}$. In the ideal case, NTFS would be able to recover from corruption to these pointers.

We provide an example from Figure 6.1 to illustrate the interpretation of the figures. The results of corrupting `Boot-MFT0` is presented in the first row. The first cell corresponds to the boot sector

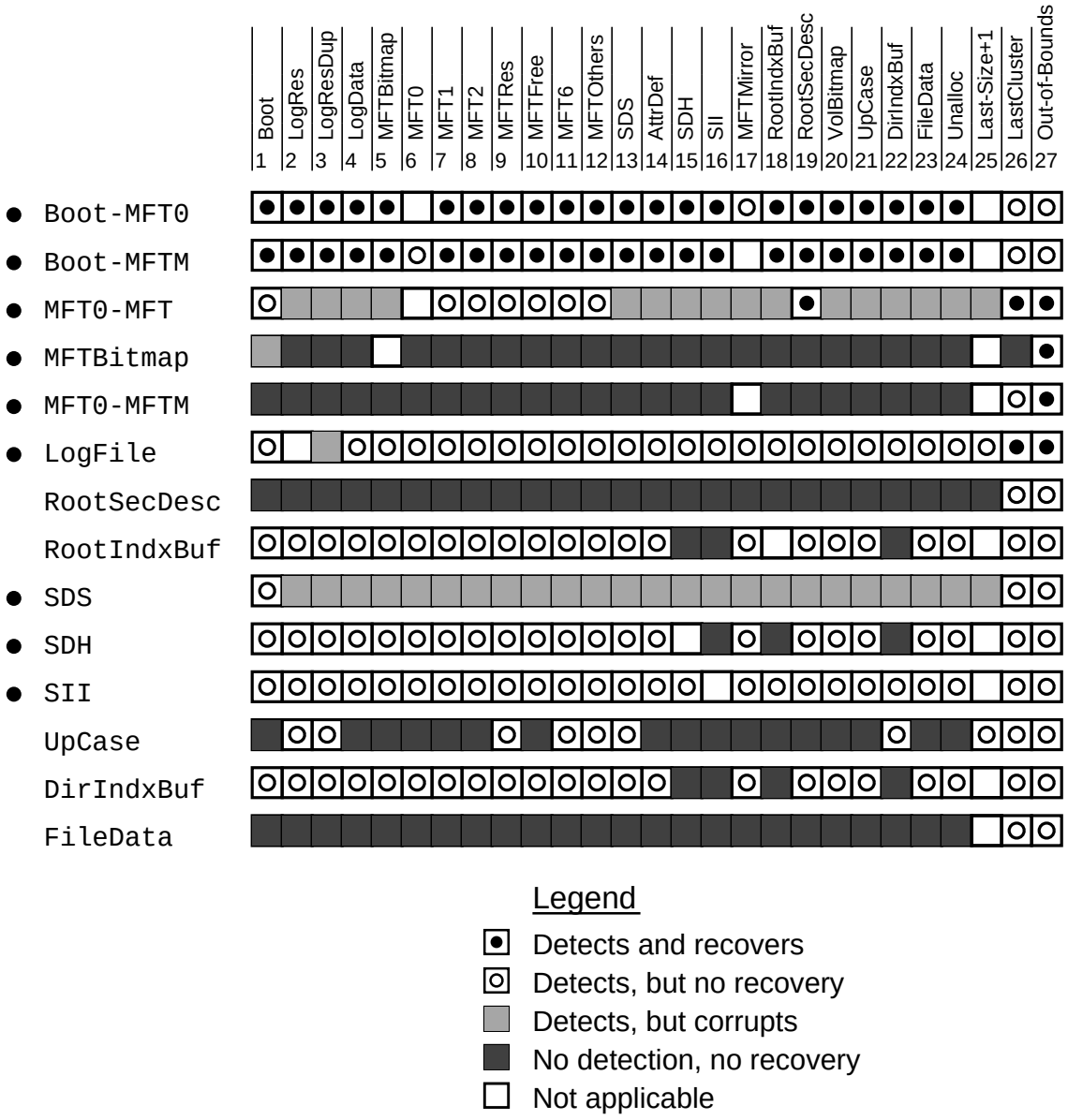

Figure 6.1 **NTFS behavior.** *Ths figure shows how NTFS responds to corruption Each row characterizes the behavior for the given pointer. Each cell in a row is marked with the behavior observed for the given pointer when it is corrupted with the value of that column. Of the values, Last-Size+1 denotes Last Cluster - Size of data run + 1 and is applicable only for data runs of length more than 1. A dot next to a pointer name for any row implies that some form of redundancy exists; in the ideal case NTFS would recover from corruption to these pointers. Note that for unallocated clusters, further corruption just implies that the cluster is overwritten since, by definition, the cluster cannot be "corrupted".*

Pointer	NTFS Behavior Details
Boot-MFT0	**Reports** error and **retries** mount for values MFTMirror, LastCluster and Out-of-bounds; **recovers** using replica for others.
Boot-MFTM	**Reports** error and **retries** mount for values MFT0, LastCluster and Out-of-bounds; **recovers** using replica for others.
MFT0-MFT	**Recovers** using MFT mirror for values RootSecDesc, LastCluster and Out-of-bounds; **reports** error and **retries** mount for others – however, both $Target_{corrupt}$ and the replica (MFT Mirror) are corrupted if the value is *not* an MFT entry or Boot.
MFTBitmap	**Recovers** only for an out-of-bounds value; **reports** error for the value Boot (however, NTFS corrupts Boot); does not detect all other cases corrupting $Target_{corrupt}$ and possibly an MFT entry.
MFT0-MFTM	**Recovers** for an out-of-bounds value; **reports** error for LastCluster; does not detect all other cases and corrupts $Target_{corrupt}$. Interestingly, this corruption of $Target_{corrupt}$ is reversed for LogRes and LogResDup due to the order of disk operations.
LogFile	**Recovers** for an out-of-bounds value or LastCluster; attempts **repair** but corrupts clusters for LogResDup; reports error and retries for others but corrupts the replica of the pointer in MFT mirror.
RootSecDesc	**Reports** error and **retries** mount for values LastCluster and Out-of-bounds; other cases are undetected.
RootIndxBuf	**Reports** error and **retries** mount for all values except for other index buffers (SDH, SII or DirIndxBuf) which go undetected thus corrupting $Target_{corrupt}$.
SDS	**Reports** error and **retries** for Boot, LastCluster Last-Size+1 and out-of-bounds (For Last-Size+1, report and retry occur after corrupting it); attempts to **repair** data structure for other cases, resulting in corruption of $Target_{corrupt}$.
SDH	**Reports** and **retries** during mount for an out-of-bounds value; **reports** error during CreateFile for other values except for index buffers (SII, RootIndxBuf and DirIndxBuf) which go undetected thus corrupting $Target_{corrupt}$.
SII	**Reports** and **retries** mount for all values.
UpCase	**Reports** error and **retries** mount for the 10 detected cases (refer Figure 6.1); undetected cases do not cause further corruption.
DirIndxBuf	**Reports** an error for all values except for other index buffers (these go undetected, thus corrupting $Target_{corrupt}$).
FileData	**Reports** an error for values Last Cluster and out-of-bounds; others are not detected leading to corruption of $Target_{corrupt}$. The corruption is reversed for LogRes, LogResDup, MFT0, and MFTMirror due to the order of disk operations.

Table 6.5 **NTFS behavior details.** *The table presents the details of NTFS behavior when its pointers are corrupted.*

(Boot). The symbol in the cell corresponds to "Detects and recovers." This indicates that when the pointer `Boot-MFT0` is corrupted to the value Boot, NTFS detects the corruption and fully recovers from it, thus continuing normal operation. The same behavior is observed for most of the cells in this row. When the $Target_{corrupt}$ is MFTMirror (column number 17), the symbol indicates that NTFS "detects, but does not recover" from the corruption. The same behavior is observed for values LastCluster and Out-of-bounds at the end of row. The value MFT0 (column 6) is the correct value for the pointer and hence the "Not applicable" symbol is used. Note that there is no similar correct value for pointers like `FileData` since we can use data locations of a *different* file to corrupt the pointer. Finally, the value "Last-Size+1" is not applicable for pointers with a data-run length of 1.

6.4.3 NTFS Behavior

We discuss the behavior of NTFS when each of its pointers are corrupted. The detailed results are presented in Figure 6.1 and Table 6.5. Table 6.6 summarizes these results. This subsection distills the results into higher-level observations on system behavior and lessons to be learned. The goal is to analyze whether NTFS effectively uses its type information and redundancy, and to understand why NTFS is or is not able to detect and recover from pointer corruption.

Out of 360 corruption experiments, NTFS detects corruption in 238 cases (66%) and recovers in only 51 cases (14%). Despite the availability of redundant information for recovery for most cases, NTFS either simply reports an error to the user or retries the mount operation. Also, despite detecting the corruption, NTFS itself causes further corruption in 42 cases (12%).

6.4.3.1 Detection

From our experiments, we find that NTFS uses type checking and sanity checking to detect pointer corruption. Both techniques belong to the D_{Sanity} level of detection in the IRON taxonomy (Section 2.4). We review each of these techniques below.

Type checking verifies that a disk cluster conforms to the requirements for a data type. Typically, type information for a cluster is encoded in the form of a "magic" number and stored in the cluster. In order to perform type checking, the cluster pointed to should be read.

Sanity checking verifies that certain values in data structures follow constraints. A pointer can be compared with well-known values, such as locations of metadata like the boot sector or disk partition size, to ensure that the pointer is not corrupt. This technique enables the system to detect corruption before the cluster pointed to is read. Sanity checks are especially important when type information cannot be stored along with the cluster (like for file data clusters).

We determine whether type checking or sanity checking based on whether the detection occurs before or after the $Target_{corrupt}$ (cluster pointed to) is read.

Observation 6.1 *NTFS detects corruption errors primarily through type checking.*

We observe that NTFS detects corruption errors *after reading* $Target_{corrupt}$ for many pointers, including `Boot-MFT0`, `MFT0-MFT`, `LogFile`, `RootIndxBuf`, `SII`, and `DirIndxBuf`. An examination of the corresponding data structures shows that they contain "magic" numbers ("FILE" for MFT clusters, "RSTR" for log restart area, "INDX" for index buffers) that identify the clusters as a certain data type.

Lesson 6.1 *Type checking is useful for detecting pointer corruption. However, systems that use type checking should not overload the data types.*

NTFS does not detect corruption when one index buffer pointer (`RootIndxBuf`, `SDH`, `SII`, or `DirIndxBuf`) points to a wrong index buffer. In this case, the type "INDX" is *overloaded*; it is used to represent different data structures used for different purposes. Not detecting corruption in these cases leads to further corruption by NTFS. Thus, when a data type is used for different purposes in different places, it must be assigned a different type identifier to prevent corruption across uses.

Pitfall 6.1 *Inadequate or inconsistent use of sanity checks.*

We observe that NTFS detects corruption to any pointer with an out-of-bounds value without reading $Target_{corrupt}$. Similarly, the corruption is detected immediately when `Boot-MFTM` is assigned the value MFT0 (Row 2, column 6 in Figure 6.1). These immediate detections indicate the use of sanity checks. However, while NTFS detects the above corruption scenario where pointers `Boot-MFTM` and `Boot-MFT0` are equal, it allows `MFT0-MFTM` and `MFT0-MFT` to be equal (Row 5, column 6 in Figure 6.1), although the $Target_{original}$ for each pointer is the same as before. This difference in behavior points to the lack of a consistent approach to sanity checking. There are more examples of inconsistencies – pointers for which some corruptions are recovered from, while others are not even detected.

Lesson 6.2 *Type checks do not work for all pointers. Therefore, detailed sanity checks should be performed.*

Type checking is not useful for pointers like `FileData` since a type identifier cannot be stored in a user data cluster. In these cases, sanity checking assumes greater significance. However, NTFS does not perform many simple sanity checks that can determine whether a pointer is corrupt. For example, NTFS does not check whether a pointer is pointing to the boot sector (Boot). Another example sanity check that NTFS could but does not perform is checking the "in-use" flag of an MFT entry before allocating it. The absence of such a check prevents NTFS from detecting corruption to `MFTBitmap`, causing further corruption.

We note that not all NTFS behavior can be explained based on sanity or type checking. NTFS detects corruption of `UpCase` after reading $Target_{corrupt}$ for some experiments but does not detect for others. It is not clear what kind of check is used for this pointer.

6.4.3.2 Reactions

NTFS reacts in various ways on detecting corruption. It either recovers from corruption, or reports an error to the application, or retries the mount operation, or attempts to repair a seemingly corrupt data structure.

Observation 6.2 *NTFS typically uses replication to recover from corruption.*

Pointer	Redundancy?	Detects & Recovers	Detects, but no recovery	Detects, but corrupts	No detection	Further corruption	Replica destroyed
Boot-MFT0	√	22	3				
Boot-MFTM	√	22	3				
MFT0-MFT	√	3	7	16		16	16
MFTBitmap	√	1		1	23	24	24
MFT0-MFTM	√	1	1		23	20	
LogFile	√	2	23	1		1	24
RootSecDesc			2		25		
RootIndxBuf			22		3	3	
SDS	√		3	24		24	
SDH	√		22		3	3	
SII	√		25				
UpCase			10		17		
DirIndxBuf			22		4	4	
FileData			2		24	20	
Total		51	145	42	122	115	64
Total recoverable	√	**51**	**87**	**42**	**49**	**88**	**64**

Table 6.6 **NTFS behavior summary.** *The table summarizes observed NTFS behavior on corruption for the different pointers. The first column indicates whether some form of redundancy exists for either the pointer or $Target_{original}$. Columns 2 to 5 summarize the number of cases for which NTFS behaves in a certain manner (from Figure 6.1). The last two columns indicate the total number of cases for which further corruption occurs and for which the replica of the pointer is destroyed. The penultimate row is the sum of all rows and the last row is the sum of rows that have a √ for the "Redundancy?" column.*

We observe that NTFS uses replication of MFT VCN 0 to recover from corruption to the pointer `Boot-MFT0`. In this case, it uses the MFT mirror to obtain the required information. Similarly, NTFS uses redundant information in MFT VCN 0 to recover from corruption to `Boot-MFTM`. Interestingly, for both pointers, this recovery is *temporary*; that is, NTFS does not overwrite the corrupt pointer with the correct value. Thus, the same recovery has to be performed for each mount. This approach could lead to unrecoverable data loss in the event of a second failure (loss or corruption). When an out-of-bounds value is used for the pointers `MFT0-MFT`, `MFTBitmap`, `MFT0-MFTM`, and `LogFile`, NTFS performs *permanent* recovery; that is, the pointer value is overwritten with the correct value, thus completely healing the file system image.

Observation 6.3 *NTFS uses error reporting and retries in response to corruption when it is unable to recover.*

As described in Table 6.5, typically, NTFS reports an error to the application when corruption is detected. For a subset of cases, NTFS also retries the mount operation, perhaps hoping that the corruption is transient and mount will succeed the second time. These retries do not succeed since the corruption is persistent. Examples of pointers for which this behavior is observed include `MFT0-MFT` and `LogFile`, and `RootIndxBuf`.

Observation 6.4 *NTFS attempts to repair certain data structures that it believes to be corrupt.*

When the pointer `SDS` is corrupted, NTFS assumes that the security descriptors pointed to by `SDS` are corrupt and attempts to reinitialize the data structure, thus corrupting $Target_{corrupt}$. Similar behavior occurs when `LogFile` points to LogResDup instead of LogRes (the log restart area). In this case, the first cluster of the data region of the log is corrupted.

Pitfall 6.2 *Detecting that a pointer target is corrupt instead of detecting that the pointer is corrupt.*

The instances under Observation 6.4 above show that NTFS trusts the pointer to be correct, while not trusting the cluster pointed to. Thus, attempting to repair a seemingly corrupt target causes more harm than good if the corruption is actually to the pointer.

In general, we observe that there are multiple instances where NTFS does not detect the corruption or detects the corruption but does not recover from it despite possessing type information to detect corruption and redundancy to recover from corruption. Table 6.6 shows that despite possessing redundant information, NTFS detects an error but does not recover from it in 87 cases, and in fact, causes further corruption in 88 cases. From these failures, we derive more potential pitfalls when handling pointer corruption.

Pitfall 6.3 *Ineffective replica management: (a) not using replicas when available, (b) destroying secondary replicas without verifying the primary, and (c) not maintaining independent access paths for replicas.*

(a) When pointers in MFT VCN 0 are corrupted, NTFS does not use the copy of pointers available in the MFT mirror for most scenarios. For some pointers, NTFS could but does not use the replica for comparing and detecting that the pointer is possibly corrupt. An example is `MFTBitmap`. For other pointers, NTFS detects corruption through different means (type or sanity checking). However, NTFS does not use the replica for recovering from the corruption. Example pointers are `MFT0-MFT` and `LogFile`. Thus, the advantage of replication is completely lost for these pointers. (b) There are 64 instances where the replica of the pointer is overwritten by NTFS with the corrupt value (the last column of Table 6.6). In particular, in the cases where the primary MFT (MFT VCN 0) is corrupt, but the MFT mirror is correct, NTFS erroneously synchronizes the two copies by overwriting the MFT mirror with data in the corrupt MFT. (c) For some of the data structures in NTFS, the replica is placed at a fixed virtual offset from the regular copy, thus often using a single pointer value to access both. The security descriptors are an example. Corruption to the pointer `SDS` will thus make both the regular copy and the replica inaccessible (Figure 6.1 shows that NTFS does not recover when `SDS` is corrupted).

Pitfall 6.4 *Not realizing that most indexes are simply performance improvements and that their unavailability should not cause complete failure.*

NTFS uses two indexes SDH and SII for its security descriptors in $Secure. The security descriptors contain all information necessary to rebuild both the indexes. However, when either SDH or SII is corrupted, NTFS does not recover despite detecting the corruption.

Thus, using type-aware corruption to characterize NTFS behavior yields many lessons for handling corruption in addition to providing an insight into the inner workings of NTFS. If NTFS follows the lessons discussed herein, it can completely recover from corruption in 229 scenarios (that is, for pointers Boot-MFT0, Boot-MFTM, MFT0-MFT, MFTBitmap, MFT0-MFTM, LogFile, SDS, SDH, and SII).

6.4.4 User-Visible NTFS Results

The previous subsection detailed NTFS behavior in response to pointer corruption. However, understanding these actions does not imply an understanding of how they manifest to users or applications. The primary concern for users is data and system reliability. Hence, in this subsection, we discuss user-visible results of NTFS behavior. Figure 6.2 presents the user-visible results.

Observation 6.5 *The system works correctly when NTFS recovers from corruption.*

The system works without problems in 61 scenarios (17%), primarily because NTFS detects and recovers from corruption. For example, corruption of any one pointer field (MFT, MFTMirror) in the boot sector does not affect normal operation. In 10 other cases, even though NTFS does not recover, pointer corruption does not cause problems due to the order of disk operations or due to non-use of $Target_{corrupt}$.

Observation 6.6 *The most frequent user-visible result is an unmountable file system.*

The file system becomes unmountable when NTFS detects corruption to a pointer used during mount, but is unable to recover. This situation applies to many pointers across the range of $Target_{corrupt}$ values used. An example of such a pointer is LogFile. The file system could also become unmountable when undetected pointer corruption (*e.g.*, for FileData) causes key data structures to be corrupted. The file system is rendered unmountable in 133 scenarios (37%).

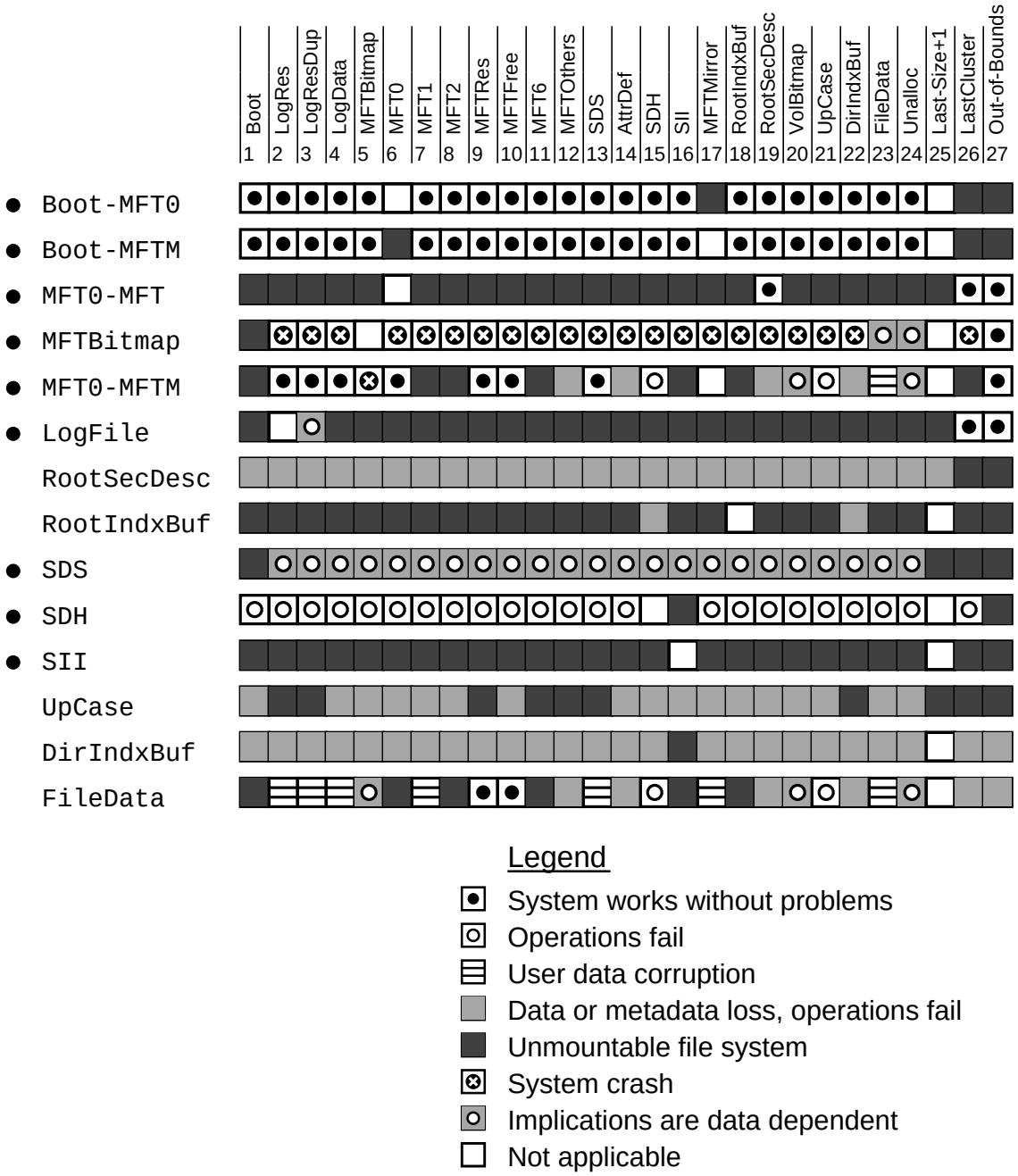

Figure 6.2 **User-visible results for NTFS.** *These figures presents the user-visible results of corrupting NTFS pointers. Each row characterizes the results observed for the given pointer. Each cell in a row is marked with the result observed for the given pointer when it is corrupted with the value of that column. Of the values, Last-Size+1 denotes Last Cluster - Size of data run + 1 and is applicable only for data runs of length more than 1. A dot next to a pointer name for any row implies that some form of redundancy exists; in the ideal case, normal operation would occur when these pointers are corrupted.*

Observation 6.7 *The second most frequent user-visible result is loss of data or user-visible metadata*

Data or metadata loss occurs in 102 scenarios (28%). Data is rendered inaccessible when the pointers `DirIndxBuf`, `RootSecDesc`, `SDS`, and `UpCase` are corrupted. In the case of `DirIndxBuf`, the pointer simply points elsewhere and therefore the directory's contents are lost and inaccessible. In the case of `RootSecDesc` and `SDS`, security descriptors (metadata) are lost due to corruption. NTFS therefore *restricts access* to directories and files allowing access only to users with administrator privileges; other users cannot access even their own files. From this experience, we also learn that pointer corruption experiments should be performed using user accounts with different access privileges. Note that we are able to identify that corruption to `RootSecDesc` or `SDS` leads to inaccessible data because the file operations are performed on a user account with non-administrator privileges.

Observation 6.8 *File operations can be significantly affected by pointer corruption.*

For some corruption scenarios, file operations fail since NTFS does not recover from the pointer corruption. An example is corruption to `SDH`; attempts to create files fail while files already created can be accessed. Note that operations also fail when data or metadata is lost. In total, file operations fail in 127 scenarios (35%). In some of the scenarios, the error code returned by NTFS correctly identifies the corruption. In others, the error code returned is misleading; it has no relation to the data structure that is corrupt. One example occurs when `UpCase` is corrupted to point to Boot.

Observation 6.9 *In addition to data loss, users could also observe data corruption.*

User data corruption is said to occur when user file data is overwritten with other data or metadata thereby corrupting it. An instance of this situation occurs when a file data pointer points to another file's data clusters. User data is corrupted in 8 scenarios (2%).

Lesson 6.3 *Undetected pointer corruption can pose a significant security risk.*

One would expect that pointer corruption might affect data on a particular disk. However, it could be worse; most experiments involving the pointer `MFTBitmap` result in a system crash (22 cases), thus affecting the entire system. By systematically setting bits contained in $Target_{corrupt}$ (the disk block being pointed to after corruption), we observe that the system crash happens whenever the allocation status bits corresponding to the system files $Quota, $ObjId and $Reparse happen to be zero (instead of one), resulting in their MFT entries getting re-used (and hence corrupted). Thus, a particular series of operations (mount, `CreateFile`) can be performed on specifically corrupted file system images to cause crashes. Such malicious disk images [149] could become a security threat with the widespread use of portable flash drives and disk image downloads.

In general, we observe that for many scenarios, the user-visible results follow readily from the reactions of NTFS (or lack thereof). However, it is not as straightforward for some scenarios. An interesting case in point occurs when `FileData` is corrupted to the value SDS (Last row, column 13 in Figure 6.1). Figure 6.1 shows that the observed NTFS behavior is that of no detection and the $Target_{corrupt}$ cluster is therefore corrupted. However, the result of this pointer corruption is user data corruption and not metadata loss (as seen in the last row, column 13 in Figure 6.2). This result occurs because the clusters of the SDS file are replicated; after an unmount, on a subsequent mount, NTFS detects that the SDS cluster is corrupt, reads the replica, and restores the corrupted cluster. This action causes file data to be overwritten and hence leads to data corruption. Similar behavior is observed for the pointer `MFT0-MFTM` with $Target_{corrupt}$ as SDS. Once again, although SDS information is lost when MFT mirror updates occur, the information is recovered from the redundant copy of the SDS cluster and the system works fine.

In certain pointer corruption scenarios, the user-visible results depend on the actual data present in various clusters. Corrupting `MFTBitmap` with the location of a file data cluster (FileData) is an example. In this case, user data is corrupted. In addition, depending on the exact values of bits in the file data cluster, there may be a system crash, or data might be lost.

Pointer	Redundancy?	Detects & Recovers	Detects, but no recovery	Detects, but corrupts	No detection	Further corruption	Replica destroyed
Block bitmap	√		1		12	12	
Inode bitmap	√		5		8	8	
Inode table	√		13				
Journal superblock			13				
Root directory			11		2	2	
Directory data			11		3	3	
File data			1		13	13	
Total		0	55	0	38	38	0
Total recoverable	√	**0**	**19**	**0**	**20**	**20**	**0**

Table 6.7 **Ext3 behavior summary.** *The table summarizes observed ext3 behavior on corruption. The columns are the same as in Table 6.6. The first column indicates whether some form of redundancy exists for either the pointer or $Target_{original}$. Columns 2 to 5 summarize the number of cases for which the system behaves in a certain manner. The last two columns indicate the total number of cases for which further corruption occurs and for which the replica of the pointer is destroyed. The penultimate row is the sum of all rows and the last row is the sum of rows that have a √ for the "Redundancy?" column.*

6.4.5 Ext3 Results

We corrupt 7 primary ext3 pointers with 14 $Target_{corrupt}$ values each, chosen in similar fashion to NTFS. Table 6.7 presents a summary of ext3 results. We describe the key ext3 observations below, focusing on how it is similar to or different from NTFS.

- Unlike NTFS, ext3 relies more on sanity checks than on type checks. For example, it verifies that bitmap and inode table pointers point within the block group. Also, when allocating inodes ext3 verifies that the inode bitmap has marked "reserved" inodes as allocated, unlike the NTFS mishandling of `MFTBitmap`. However, lack of type checks causes ext3 to use the superblock as directory data.

- Like NTFS, ext3 typically assumes that the cluster pointed to (rather than the pointer) is corrupt.

- Even though ext3 replicates the group descriptors, it never uses these replicas even when a pointer in the primary copy is detected as corrupt.

- The typical reaction on detecting corruption is to report an error and remount the file system as read-only. Ext3 does not recover even in one corruption scenario.

In summary, our analysis of ext3 shows that it is no better than NTFS in pointer protection. Our analysis also demonstrates that TAPC can be applied to very different file systems. One advantage with ext3 is that we have verified our results by reading ext3 source code.

6.4.6 Discussion

Using TAPC to characterize system behavior yields many lessons for handling corruption, in addition to providing an insight into pitfalls for real file system implementations. If NTFS and ext3 follow these lessons, they can completely recover from over 63% and 40% of the corruption scenarios respectively. In this subsection, we discuss general issues related to TAPC and corruption handling.

First, TAPC does not consider the likelihood of different values used for corruption. This likelihood depends on the source of corruption. For example, if the corruption values are arbitrary, more than 99% of the values will be out-of-bounds, while corruption due to bit flips will imply that the corrupt value is "closer" to the correct value. While our likelihood-agnostic approach does not provide probabilities for file system failures due to corruption, it provides interesting insights into how a file system handles corruption.

Second, a question that arises from the results is whether type and sanity checks are the right techniques to use, especially when there are many pitfalls involved. While it is true that the use of checksums (like in ZFS [130]) might significantly improve corruption handling, it does not subsume the protection offered by type and sanity checks. For example, checksums cannot protect against file-system bugs that place the wrong pointer value and checksum it as well.

Third, it is non-trivial to add checksums and other protection to a file system without changing the on-disk format. Type-aware pointer corruption helps identify potential sanity checks that can be used without format changes.

6.5 Conclusion

Preserving data access paths is integral to any system that wishes to preserve data as illustrated by the following anecdote:

> One of the terms of the settlement of Control Data Corporation v. IBM, the first antitrust suit against IBM, was that the 'CDC database' should be destroyed. This database was prepared by Control Data legal staff as a means of organizing the evaluation of the enormous quantity of documents subpoenaed from IBM. IBM could not legally destroy the documents themselves, but through this settlement they could destroy the *index* to the documents, making the millions of documents virtually useless [62].

File systems rely on on-disk pointers to access data. As file systems employ different and newer techniques to protect against corrupt pointers, we need to understand how these techniques perform

in reality. We have developed type-aware pointer corruption as a way to rapidly and systematically analyze the corruption-handling capability of file systems. We have applied type-aware pointer corruption to NTFS and ext3, and find that despite their potential to recover from many pointer-corruption scenarios, they do not, causing data loss, unmountable file systems, and system crashes. We use this study to learn important lessons on how to handle corrupt pointers.

We believe that future file systems should be more careful in implementing pointer protection techniques. A first step would be to develop a consistent corruption-handling policy and the corresponding machinery that can be used by all file system components.

Chapter 7

N-Version File Systems

We learn from our study of partial disk failures that these failures affect a high percentage of inexpensive SATA disk drives. These drives are the kind used in our desktops and laptops. Unfortunately, as the previous chapter shows, the commodity file systems that manage these drives are not effective at handling partial disk failures. Therefore, we need to develop a solution for tolerating partial disk failures in personal computers.

Typical approaches to solving this problem have been to either fix one of the commodity file systems, or develop a better file system, or ensure that partial disk failures are handled in some layer just beneath the file system. Indeed, over the years, many research efforts and real systems have adopted one of these options [18, 55, 60, 104, 107, 129, 130, 131]. The main problem with any of these approaches is the dependence on the file system itself to maintain the reliability and integrity of data.

The file system is a complex piece of software that not only handles partial disk failures poorly (despite the availability of techniques to tolerate them), but also contains bugs. The bugs and poor use of failure-handling techniques exist despite the file systems being widely-used and potentially well-tested. Indeed, recent research efforts, including ours, have uncovered numerous bugs in file-system code [104, 148, 149, 150]. File-system bugs could cause data corruption that may not be caught by techniques within or beneath the file system.[1]

[1] Techniques within the file system may be useful to a limited extent. For instance, Hagmann [60] states "A bug in the file system will often show up as an error in comparing the computed label with the disk label." while describing the Cedar File System.

Our solution for handling partial disk failures is an *N-version file system*. An N-version file system is an instance of N-version software [6]. In such a file system, data is stored in N *different* file systems. All file operations performed by the user are received by a simple software layer that issues the operation to all *child* file systems. This layer then determines the majority result from those returned by the child file systems and delivers it to the user. Thus, we eliminate the reliance on a single complex file system, and place it on a simpler software layer.

One major issue in building an N-version-software system is the high development costs associated with formulating a common specification for the system, and creating N different versions of the system. In order to reduce these costs, we hypothesize that for an N-version file system, (i) we can use an existing specification, such as POSIX, as the common specification, and (ii) we can use existing file systems, such as ext3, JFS, etc., as the N different file-system versions. In building an N-version file system using an unmodified specification and existing file systems, we verify these hypotheses.

A second issue in using an N-version file system is the high performance and disk-space overheads introduced by storing and retrieving data from N file systems instead of one. Our solution to this issue is to use a block-level single-instance store underneath the file systems. A block-level single-instance store uses content hashing to identify disk blocks with the same content; it then stores a single copy of these blocks on disk. In an N-version file system, user data stored in the different file systems will have the same content and will therefore be coalesced into a single block, while file-system metadata of different file systems will have different contents and will not be coalesced. Therefore, a single-instance store protects against partial disk failures that affect metadata (thereby protecting the important access paths to data), but not against failures that affect data blocks. A single-instance store is especially useful in cases where file-system bugs are the main contributors to partial disk failures.

We evaluate the reliability of a 3-version file system (that uses ext3, JFS, and ReiserFS as child file systems) through fault-injection experiments. We find that the 3-version file system can recover from a partial disk failure or a file system with incorrect contents in almost all scenarios; many of

these scenarios cause irreparable data loss or non-mountable file systems in the child file systems that the 3-version file system is composed of.

In addition to the reliability benefits of using an N-version file system, the detailed information it maintains about the child file systems makes it attractive for use as a diagnostic tool. As an example, one of our reliability experiments triggered a bug in ext3 that led to a system crash. When ext3 was used as part of a 3-version file system, the 3-version file system not only avoided the bug, but also helped identify the location of the bug.

The rest of the chapter is organized as follows. Section 7.1 describes the N-version programming approach and how it can be applied to file systems. Section 7.2 presents the design of the N-version file system and Section 7.3 discusses how the N-version file system has been built to use the existing POSIX specification and available file systems. Section 7.4 describes the single-instance store layer we have developed to address performance and disk-space overheads. Section 7.5 evaluates the reliability of a 3-version file system and Section 7.6 concludes the chapter.

7.1 An N-Version Approach

This section provides a background on N-version programming and then motivates why it is particularly suitable for file systems.

7.1.1 N-Version Programming

N-version programming [6, 7, 31] has been used over the years to build reliable systems that can tolerate software bugs. A system based on N-version programming uses N different versions of the same software and determines a majority result from the ones produced by the different versions. These different versions of the software are created by N different developers or development teams for the same software specification. It is assumed (and encouraged using the specification) that different developers will design and implement the specification differently, lowering the chances that the versions will contain the same bugs or will fail in similar fashion.

The benefits of N-version programming have been validated by various experiments [9, 10]. In one such experiment, Avižienis and Kelly [9] study the results of using three different specification

languages to develop 18 different versions of an airport scheduler program. They perform 100 demanding transactions with different sets of 3-version units and determine that while at least one version failed in 55.1% of the tests, a collective failure occurred only in 19.9% of the cases.

The N-version approach has been used primarily in computerized control systems where safety is critical, such as for train switching and flight control operations [144]. More recently, with the increase in both the threat of bug-induced failures and the cost of such failures, many recent research efforts have explored the use of N-version software in various other systems, including network file systems [112], database systems [142], and for security [36].

N-version programming has three important aspects: (a) producing the initial specification for the software, (b) developing the N different versions of the software, and (c) creating the environment that supports the execution of the different versions and also contains algorithms to determine a consensus result from the ones produced by the different versions [6].

7.1.2 N-Version Programming in File Systems

The aim of our work is to explore the use of N-version programming in file systems for the purpose of reliably storing and retrieving data. An N-version file system receives user operations and issues them to multiple *child* file systems. It then determines the majority result from the results produced by the different file systems, and delivers it to the user.

The advantages of an N-version file system can be broken down as follows.

Diversity: This advantage is the traditional advantage of an N-version software system. File systems that store data using different data structures may be used to provide diversity. This diversity would reduce the chances of common bugs.

Storage redundancy: Since data is stored in multiple file systems, an N-version file system provides the benefits of replication. Data lost due to a partial disk failure (including corruption due to a file-system bug) in one file system can still be accessed through the other file systems.

Operational Redundancy: Each file operation is performed multiple times, once in each file system. Each of the file systems also issue disk operations that cause the the rest of the (possibly buggy) storage stack to perform multiple operations that have the same purpose. The redundancy in operations lowers the chances that a given file operation will fail in all file systems.

In addition to these advantages, the use of N-version programming in file systems is particularly attractive since the design and development effort required for the first two aspects (*i.e.*, specification and version development) of creating N-version software could be much lower than the typical case:

Specification: Many existing file systems answer to a common interface; user file operations use the POSIX interface, which internally translates to the Virtual File System (VFS) interface for all Linux file systems. Thus, by using the POSIX/VFS interface, the effort needed in developing a common specification could be minimized. By building an N-version file system using the POSIX/VFS interface, we evaluate the suitability of the interface for N-versioning; we have addressed various issues in order to use this existing specification (Section 7.3.1).

N **file systems:** There are many diverse file systems available today, such as ext3, JFS, and ReiserFS, that are built for the POSIX/VFS interface. These different file systems have drastically different data structures, both on disk and in memory. This diversity reduces the chances of common file-system bugs. In addition to a smaller chance of common bugs, we find from our experiments that different file systems behave differently when they encounter partial disk failures. Figure 7.1 shows the results from our analysis of whether commodity file systems detect corruption (from previous work [104]). Each row in the figure corresponds to a file-system data structure, each column corresponds to a workload, and the symbol in each cell denotes whether or not corruption to the data structure for that row is detected when the workload for that column is executed. When we compare file-system behavior on corruption to similar data structures, we see that there are cases where a subset

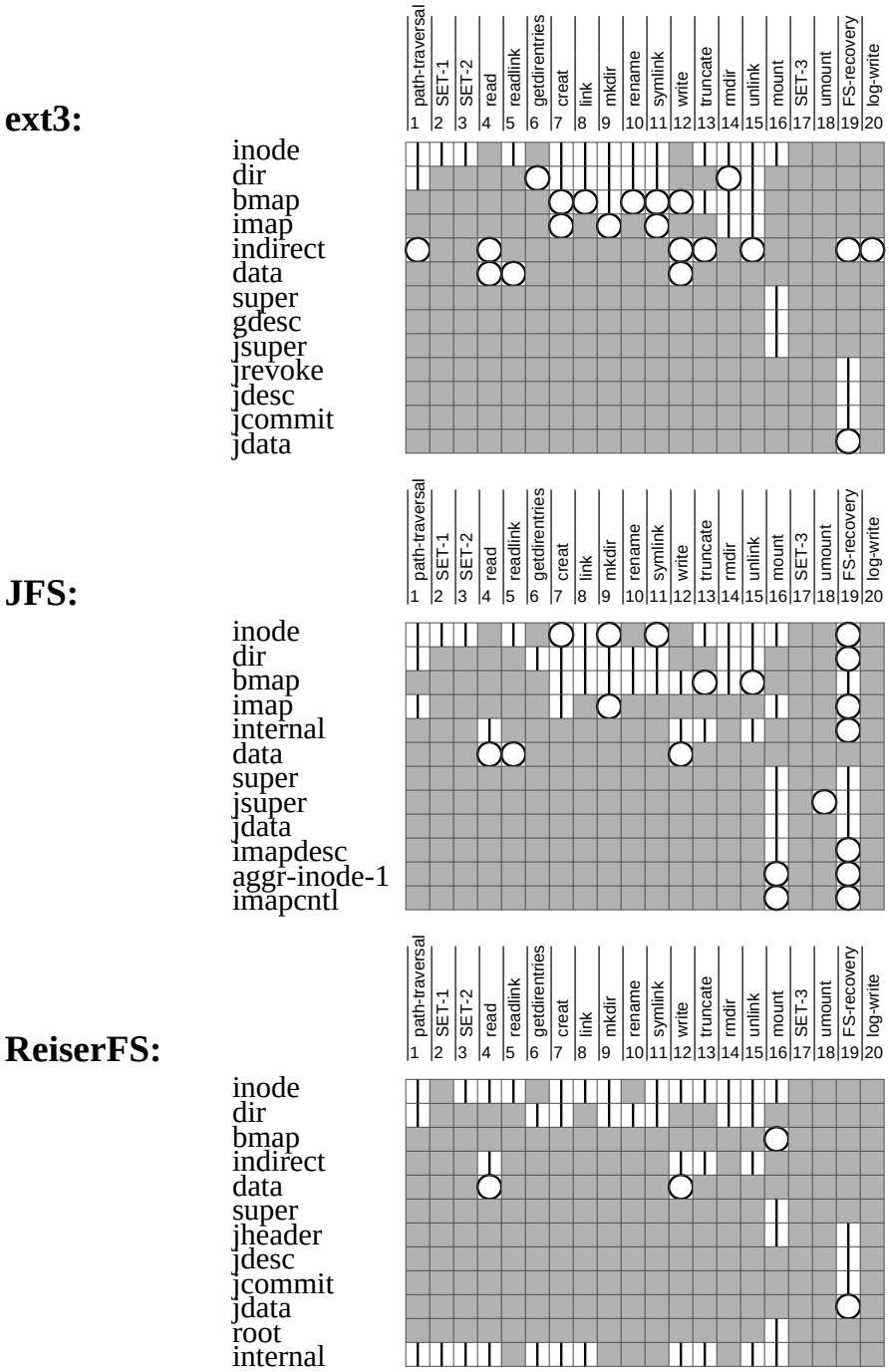

Figure 7.1 **Comparison of corruption detection.** *The figures indicate whether or not the file systems ext3, JFS, and ReiserFS detect corruptions to their data structures when the data structures are read in response to different workloads. Each row corresponds to a data structure and each column corresponds to a file operation. The symbol* [|] *denotes that the corruption is detected and* [◯] *denotes that it is not. A gray box indicates that the workload is not applicable for the block type.*

(but not all) of the file systems detect the corruption. For example, consider directory data blocks (row "dir") and the workload getdirentries (column "6"): ReiserFS and JFS detect the corruption while ext3 does not. The availability of such diversity motivates the opportunistic use of existing file systems as opposed to building new ones.

It is our hypothesis that we can leverage an existing specification and file systems that have already been developed to this specification to build an N-version file system. Such opportunistic approach of using an existing specification and an existing set of systems has previously been employed successfully by Rodrigues *et al.* [112]; they have used the NFS specification to build Byzantine-fault-tolerant NFS servers using different off-the-shelf file systems. More recently, Vandiver *et al.* [142] have developed a Byzantine-fault-tolerant transaction processing system using heterogeneous replicas.

7.2 An N-Version File System

This section describes the design of an N-version file system. We focus on the third aspect of building N-version software, that is, the execution environment. We first outline the goals that influence the design of this environment. Next, we present the basic architecture of the environment, and finally discuss various details of the design of this environment.

7.2.1 Assumptions and Goals

Overall, the design of the N-version file system is influenced by the following goals and assumptions:

Simplicity: As systems have shown time and again, complexity is the source of many bugs. Therefore, the N-version file system should be as simple as possible. This goal primarily translates to avoiding persistent metadata for the N-version file system, thereby avoiding issues such as disk-block allocation and protection of metadata against partial disk failures.

Single disk: The N-version file system is intended for use in a commodity system. Therefore, it will replicate data across multiple local file systems that use the same disk drive. This

goal translates to a need for reducing disk-space overheads; we develop a single-instance store to address this goal (Section 7.4).

Non-malicious file systems: The N-version file system should protect against partial disk failures and file-system bugs that lead to errors in the persistent state of file systems. The N-version file system does not protect against malicious behavior by file systems or file-system bugs that corrupt the rest of the kernel.

No application modifications: Applications should not need modifications to use an N-version file system instead of a single local file system.

7.2.2 Basic Architecture

An N-version file system receives application file operations, issues the operations to multiple *child file systems*, compares the results of the operation on all file systems, and returns the majority result to the application. Each child file system stores its data and metadata in its own disk partition.

We have built the N-version file system for Linux. Figure 7.2 shows the basic architecture. The N-version file system consists of a software layer NVFS that operates underneath the virtual file system (VFS) layer. NVFS operates underneath VFS because VFS provides core functionality (like ordering of file operations) that is hard to replicate without modifying applications. The VFS layer has been heavily tested over the years and, hence, is likely to have fewer bugs that the file systems themselves; in a study of file systems, Yang *et al.* [150] find 2 bugs in the VFS layer while they find 2 bugs in ext2, 5 in ext3, 2 in ReiserFS and 21 in JFS.

The NVFS layer executes file operations that it receives on multiple child file systems. We have used ext3 [141], IBM's JFS [20], and ReiserFS [108] for this purpose. We have chosen these file systems due to their popularity, our experience in analyzing these file systems, and the differences in the handling of partial disk failures across these file systems (as shown in Figure 7.1).

Similar to stackable file systems [63], NVFS interposes transparently on file operations; it acts as a normal file system to the VFS layer and as the VFS layer to the child file systems. It thus presents file-system data structures and interfaces that the VFS layer operates with and in turn

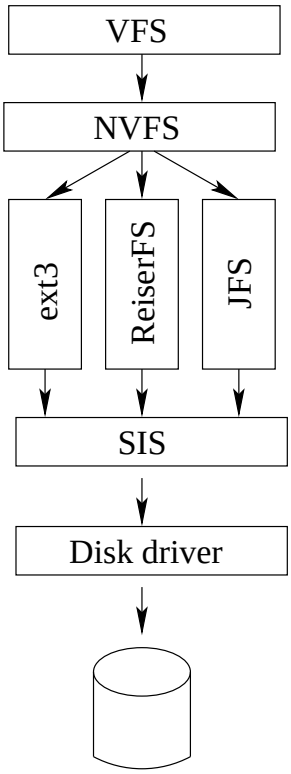

Figure 7.2 **N-version file system in Linux.** *The figure presents the architecture of a 3-version file system with ext3, ReiserFS and JFS as the 3 child file systems. The core layer is the NVFS layer; it is responsible for issuing file operations to all 3 file systems, determining a majority result from the ones returned by the file systems, and returning it to the VFS layer. The SIS layer beneath the file systems is a single-instance store built to work in an N-version setting; it coalesces user data stored by the different file systems in order to reduce performance and space overheads.*

manages the data structures of the child file systems; it has its own in-memory inodes, dentry structures, etc., to interact with the VFS layer and in turn manages the allocation and deallocation of such structures for child file systems. This management of data structures of child file systems includes tracking the status of each data structure, that is, whether it matches with the majority and whether it needs to be deallocated. In keeping with our simplicity goal, we have designed the N-version file system so that it does not maintain any persistent data structures of its own. This decision affects various parts of the design, from handling faulty file systems (Section 7.2.3.2), to handling system crashes (Section 7.2.3.4), and to management of inode numbers (Section 7.3.1).

We have implemented wrappers for file and directory operations.[2] These wrappers first verify the status of necessary objects in the child file system before issuing the operation to it. For example, NVFS verifies whether the status of both the file and its parent directory are valid in the case of an unlink operation. Each operation is issued in series to the child file systems. Issuing an operation in parallel to all file systems will increase the complexity of the NVFS layer and it is not clear that such an approach will have much of a performance benefit considering that the child file systems likely share the same disk drive. When the operations complete, the results are compared to determine the majority result; this result is then returned to the user. When no majority result is obtained, NVFS returns an I/O error to the user; future implementations can consider using the response of a "primary" file system in such cases. The next subsection describes the result comparisons and actions taken when a file system does not agree with the majority in more detail.

7.2.3 Design Details

This subsection describes the details of various design choices that we made in building the N-version file system, including the comparison of results, handling cases when file systems disagree, ordering of file operations, and the implications of system crashes.

[2] Our current implementation of NVFS does not include the `mmap` operation.

7.2.3.1 Result Comparison

The NVFS layer compares the results of all operations across the different child file systems. For example, for a file read operation, NVFS compares (a) the size of data read (or the error code returned), (b) the actual content read, and (c) the file position at the end of the read. For all file operations where inodes may be updated, NVFS compares (and copies to the NVFS-level inode) the contents of the individual inodes. We have developed comparators for different file-system data types like superblocks, inodes, and directories. For example, an inode comparator checks whether the fields `i_nlink`, `i_mode`, `i_uid`, etc. in the different inodes are the same.

In performing read operations, we would like to avoid the performance overhead of allocating memory to store the results returned by all of the file systems. Therefore, the NVFS layer uses the memory provided by the application as part of the `read` system call. This choice influences two decisions: (i) NVFS calculates a checksum on the data returned and compares the checksums for different file systems, since a more thorough byte-by-byte comparison would require memory for all copies of data, and (ii) NVFS issues the read operation in series to child file systems only until a majority opinion is reached (the read is not issued to the remaining child file systems); this choice eliminates the problem of issuing reads again in case the latest file system to perform the read returns incorrect data; in addition, in the common case, when file systems agree, we save on extra reads.[3]

In choosing the checksum algorithm used to compare data, we have to remember that the cost of checksumming can be significant for reads that hit in the file-system buffer cache. We have measured using microbenchmarks that this cost is especially high for cryptographic checksums such as MD-5 and SHA-1. Therefore, in keeping with our goal of protecting against bugs, but not potentially malicious behavior of child file systems, we use a simple TCP-like checksum for comparisons.

[3]We choose not to take the same issue-only-until-majority approach with other VFS operations like lookup since the limited performance gain for such operations is not worth the complexity involved, say in tracking and issuing a sequence of lookups for the entire path when a lookup returns erroneous results in one file system.

7.2.3.2 Handling Disagreement

An important part of an N-version file system is the handling of cases where a child file system disagrees with the majority result. This part is specifically important for local file systems since the ability to perform successive operations may depend on the result of the current operation (*e.g.*, a file read cannot be issued when the open operation for a file fails).

If the result produced by a child file system disagrees with the majority result across all child file systems, the N-version file system operates in *degraded-mode* for the associated object; that is, it does not perform future operations on the object for the file system with the error. The N-version file system continues to perform operations on other objects for that file system. As an example, if a child file system's file inode is declared faulty, then read operations for that file are not issued to that file system. As another example, if a lookup operation completes successfully for only one file system, its corresponding in-memory *dentry* data structure is deallocated, and any future file create operation for that dentry is not issued to that file system.

The validity information for objects is not maintained persistently. In the absence of an option to repair the child file system, this choice maintains the simplicity goal. Note that permanent errors will likely be detected again during future operations, while we allow the child file systems to return to normal operation for transient errors as long as the object is not modified in the interim.

When an error is detected, in order to restore the N-version file system to full replication, the erroneous child file system should be repaired. Our N-version file system currently does not repair child file systems. As an example of why logical file-system repair is difficult, consider the following scenario. A file with two hard links to it may have incorrect contents. If the N-version file system detects the corruption through one of the links, it may create a new file in the file system to replace the erroneous one. However, there is no simple way to identify the directory where the other link is located, so that it can be fixed as well (except through an expensive scan of the entire file system).

7.2.3.3 Operation Ordering

As in many replication-based fault tolerance schemes, determining an ordering of operations is extremely important; in fact, recent work in managing heterogeneous database replicas focuses primarily on operation ordering [142]. In the context of a file system, consider the scenario where multiple file operations are issued for the same object. If an ordering is not predetermined for these operations, their execution may be interleaved such that the different child file systems perform the operations in a different order and therefore produce different results even in the absence of bugs.

Unlike for databases, the dependence between operations can be predetermined in the case of file systems. In our NVFS implementation, we rely on the locking provided by the Linux VFS layer to order metadata operations. As explained earlier, this reliance cannot be avoided without modifying applications (to issue operations to multiple replicas of VFS that execute an agreement algorithm). In addition to the VFS-level locking, we perform file locking at the NVFS layer for reads and writes to the same file. This locking is necessary since the VFS layer does not (and has no need to) order file reads and writes.

7.2.3.4 System Crashes

When a system crash occurs, file-system recovery in an N-version file system consists of performing file-system recovery for all child file systems before the N-version file system is mounted again. This approach leads to consistent states for each of the child file systems (assuming that they use techniques like journaling to maintain consistency on crashes). However, it is possible that the different file systems recover to different states. Specifically, when a crash occurs in the middle of a file operation, NVFS could have issued (and completed) the operation for only a subset of the file systems, thereby causing the file systems to recover to different states. In addition, file systems like ext3 maintain their journal in memory, flushing the blocks to disk periodically; in this case, journaling provides consistency and not durability. For the N-version file system, the state modifications that occur durably for a majority of file systems before the crash are considered to have completed. The differences in the minority set can be detected when the corresponding objects are read, either during user file operations or during a proactive file system scan. There are corner

cases where a majority result will not be obtained when a system crash occurs. In these cases, choosing the result of any one file system will not affect file-system semantics; future N-version file system implementations could choose to use the result from a designated "primary" child file system.

7.3 Achieving Opportunistic N-Versioning

We now discuss how well it works to opportunistically use an existing specification and available file systems for an N-version file system. We first focus on the ways in which the POSIX specification is imprecise for use in an N-version setting and how we address the issues that arise, and then discuss the implications of using a shared address space for all child file systems.

7.3.1 Imprecise Specification

One of the problems that we encounter in building an N-version file system is the fact that file systems do not use the same specification. While most file systems support the POSIX/VFS interface (which serves as the interface exported by the N-version file system), they differ in various user-visible aspects that are not a part of the POSIX interface. For example, the POSIX specification does not specify the order in which directory entries are to be returned when a directory is read. Thus, different child file systems return directory entries in a different order. As another example, the inode number of files is available to users and applications through the `stat` system call, and different file systems issue different inode numbers to the same file.

One approach to addressing this problem would be to make the specification more precise and change the file systems to adhere to the new specification. The first problem with this approach is that it discourages diversity across the different file systems. For example, in the inode number case, all file systems will be forced to use the same algorithm to allocate inode numbers, perhaps causing them to also use the same data structures, thereby inviting common bugs. The second problem with the approach is the development effort needed to change each file system in order to use it as a child file system. The third problem is that non-determinism and difference in operation ordering could cause different behavior even if the same file system is used as all N "versions."

Our approach is to address the differences in specification at the NVFS layer. In the directory entry example, the NVFS layer reads all directory entries from all file systems, and then returns results that occur in a majority of file systems (as opposed to reading exactly as many entries as the user provides space for). This approach increases the overhead for the `getdirentries` system call for very large directories. We handle the inode-number issue by having the N-version file system assign inode numbers as and when a new object is encountered. In keeping with our simplicity goal, inode numbers so assigned are not persistent; that is, an object has a specific inode number only between mount and the corresponding unmount. This decision impacts only a few applications, such as NFS servers (pre-NFSv4) and rsync, that depend on the persistence of local file system inode numbers.

7.3.1.1 Shared Environment

One problem with using multiple local file systems for replication is that the different file systems execute within the same address space, thus exposing the N-version file system to two problems: (a) kernel panics called or caused by any file system, and (b) memory bugs in the file systems that corrupt the rest of the kernel. A solution to both problems would be to completely isolate the child file systems using a technique such as Nooks [133]. However, due to the numerous interactions between the VFS layer and the file systems, such isolation comes at a very high performance cost. Therefore, we explore a more limited solution to this problem.

We find the current practice of file systems issuing a call to `panic` when they encounter errors to be too drastic[4] [104]. This scenario is one instance where using existing file systems for N-versioning causes problems. In the case of ext3 and JFS, a mount option *errors* can be used to specify the action to be taken when a problem is encountered; we could specify *errors=continue* to ensure that panic is not called by the file systems. However, this option is not available on all file systems. Therefore, our solution is to replace calls to `panic`, `BUG`, and `BUG_ON` by child file

[4]File system developers seem to agree, as evidenced by the following comment in ext3 code: "Give ourselves just enough room to cope with inodes in which i_blocks is corrupt: we've seen disk corruptions in the past which resulted in random data in an inode which looked enough like a regular file for ext3 to try to delete it. Things will go a bit crazy if that happens, but at least we should try not to panic the whole kernel."

systems with a call to a `nvfs_child_panic` routine in NVFS layer. This simple replacement is performed in file-system source code. The `nvfs_child_panic` routine disables issuing of further file operations to the failed file system. However, since the intent of the child file system is to cause the system to crash, one cannot guarantee that crash-free progress can be made in all cases when execution is allowed to continue.

7.4 Single-Instance Store

Two issues that arise in using an N-version file system are the disk-space and performance overheads. Since data is stored in N file systems, there is an N-fold increase (approximately) in disk space used. Since each file operation is performed on all file systems (except for file reads), the likely disk traffic is N times that for a single file system.

Our solution is to these problems is to apply single-instance storage technology [23, 42, 105]. Our block-level single-instance store trades-off some data reliability for disk space and performance, while still maintaining metadata reliability for the different child file systems. We have designed this single-instance store specifically for use in an N-version file system setting.

In our system, the disk operations of the multiple child file systems pass through a block-level single-instance store layer. This layer computes a content hash (SHA-1[5]) for all disk blocks being written and uses the content hash to detect duplicate data; the single-instance store writes out only unique disk blocks.

The single-instance store layer ensures that only one copy of each user data block will be stored since data blocks produced by different file systems will likely have the same content[6]. Thus, disk-space usage is reduced. However, any disk failure that affects a data block in one file system will also affect the data block in the other file systems. The question that arises now is: *why use a single-instance store underneath an N-version file system if all file systems will be affected?*

[5] Using SHA-1 does not impact performance greatly when used in disk operations.

[6] We only require that each of the file systems use a minimum block size of 4 KB

There are two reasons why a single-instance store is applicable. First, the reliability of file-system metadata is not affected by the use of a single-instance store; since the format of file-system metadata is different across different file systems, metadata blocks of different file systems will have different hash values and will therefore be stored separately. Thus, the single-instance store layer can distinguish between data and metadata blocks *without* any knowledge of file-system data structures. Since metadata form the access path to multiple units of data, their reliability may be considered more important than that of a data block. Second, file-system bugs that cause in-memory corruption of data blocks will result in the data stored by different file systems having different content hashes, thus maintaining the isolation between different file systems' data. Therefore, while disk failures will affect multiple file systems, they are still protected against each other's file-data corruptions. Thus, the single-instance store is especially applicable when file-system bugs are the primary contributors to partial disk failures.

The design of a single-instance store specifically for an N-version file system should satisfy slightly different requirements than a conventional single-instance store. At the same time, these requirements provide new opportunities for optimizations. The new requirements and opportunities are as follows.

- The ability of child file systems to recover from failures to their metadata blocks should be retained. This ability may depend on the availability of replicas for these disk blocks. For example, JFS replicates its superblock and uses the replica to recover from a latent sector error to the primary. Thus, our single-instance store does not coalesce disk blocks with the same content if they belong to the same file system. This feature has the additional benefit of maintaining any file-system remapping of disk blocks when a disk error occurs.

- In order to use unmodified file systems (that have no knowledge of content addressing), the single-instance store also virtualizes the disk address space; it exports a virtual disk to the file system, and maintains a mapping from each file system's virtual disk address to the corresponding physical disk address, as well as the reference counts for each physical disk block.

The single-instance store uses file-system virtual addresses as hints for assigning physical disk blocks in order to maintain as much sequentiality and spatial locality as possible.

- Since the goal of the single-instance store is to coalesce common data from different file systems, we can take advantage of the fact that in an N-version file system, this common data is always written around the same time. Therefore, in our single-instance store, the content hash information for each disk block is not stored persistently; the content hashes are maintained in memory and deleted after some time has elapsed (or after N file systems have written the same content). This ephemeral nature of content hashes also reduces the probability of data loss or corruption due hash collisions [64].

- In an N-version file system, reads of the same data blocks will occur around the same time as well. Thus, the single-instance store layer services reads from different file systems by maintaining a very small read cache, thus reducing the number of disk reads. This read cache holds only those disk blocks whose reference count (number of file systems that use the block) is more than 1. It also tracks the number of file systems that have read a block and removes a block from cache as soon as this number reaches the reference count for the block.

Thus, the single-instance store data structures include: (i) virtual-to-physical mappings, (ii) allocation information for each physical disk block in the form of reference count maps, (iii) a content-hash cache of recent writes and the identities of the file systems that performed the write, and (iv) a small read cache.

We have built the single-instance store as a pseudo-device driver in Linux. It exports virtual disks that are used by the file systems. Our current implementation of the single-instance store does not store virtual-to-physical mappings and reference-count maps persistently; future implementation efforts could focus on a reliable persistent store for these data structures.

7.5 Reliability Evaluation

We evaluate the reliability improvements of a 3-version file system that uses ext3, JFS, and ReiserFS (version 3) as the child file systems. All our experiments use the versions of these file systems that are available as part of the Linux 2.6.12 kernel.

We evaluate the reliability of the 3-version file system in two ways: First, we examine whether the 3-version file system recovers from scenarios where file-system content is different in one out of the three child file systems. Second, we examine whether it can recover from partial disk failures that affect one of the child file systems.

7.5.1 Non-Matching File System Content

The first set of experiments is intended to mimic the scenario where one of the file systems has a consistent but incorrect disk image. Such a scenario might occur either when (i) a system crash occurs and one the child file systems has written either more or less to disk than the other file systems, and (ii) a bug causes one of the file systems to corrupt file data, say by performing a misdirected write of data belonging to one file to another file.

We experiment by modifying the contents of one child file system and executing a set of file operations on the 3-version file system. We have explored various file-system content differences, including extra or missing files or directories, and differences in file or directory content. The different file operations performed include all possible file operations for the object. Our file operations include those that are expected to succeed as well as those that are expected to fail with a specific error code. Note that the ability to perform some file operations depends on whether the object exists. For example, a file read can be attempted only on a file that exists, while a file open can be attempted for both existing and absent files. These experiments are intended to verify whether the 3-version file system detects differences correctly, and responds to requests with the majority opinion.

Table 7.1 presents the results of the file-system content experiments. We find that the 3-version file system correctly detects (and reports to the system log) all differences. In addition, it returns

Difference in content	**Number of experiments**	**Correct success**	**Correct error code**
None	28	17 / 17	11 / 11
Different directory contents in one file system	13	6 / 6	7 / 7
Directory is present in only two file systems	13	6 / 6	7 / 7
Directory is present in only one file system	9	4 / 4	5 / 5
Different file contents in one file system	15	11 / 11	4 / 4
Different file metadata in one file system	45	33 / 33	12 / 12
File is present in only two file systems	15	11 / 11	4 / 4
File is present in only one file system	9	3 / 3	6 / 6
Total	**147**	**91 / 91**	**56 / 56**

Table 7.1 **File-system content experiments.** *This table presents the results of issuing file operations to 3-version file system objects that differ in data or metadata content across the different child file systems. The first column describes the difference in file-system content. The second column presents the total number of experiments performed for this content difference; this is simply the number of applicable file operations for the file or directory object. The third column is the fraction of operations that return correct data and/or successfully complete. The fourth column is the fraction of operations that correctly return an error code (and it is the expected error code) (e.g., -ENOENT when an unlink operation is performed for a non-existent file). We see that the 3-version file system successfully uses the majority result in all 147 experiments.*

the majority result to the user in all cases, either returning the expected data or returning the expected error code. The N-version file system can also be successfully mounted and unmounted in all cases. We find that the results are the same irrespective of which file system (ext3, JFS, or ReiserFS) has incorrect contents.

7.5.2 Partial Disk Failures

The second set of experiments is intended to analyze whether a 3-version file system recovers when a child file system is affected by a partial disk failure. We experiment by injecting partial disk failures for JFS and ext3. We use type-aware fault injection for our experiments (Section 2.5.1). In each experiment, we inject a fault when a specific file-system data structure is read by the child file system in response to a specific file operation. Tables 7.2 and 7.3 briefly describe the JFS and ext3 data structures for which faults are injected. The tables also provide the approximate fraction of disk blocks of each type in a typical file system on a 40-GB disk partition. We inject two types of partial disk failures: read errors and corruption. For read errors, the error code EIO is returned to the file system. In addition, the read buffer is zeroed out.[7] For corruption, the entire buffer is zeroed out, but no error code is returned. We first present a reliability comparison of JFS and 3-VFS, then a comparison of ext3 and 3-VFS. Next, we derive approximate probabilities for different user-visible results such as data loss, and finally discuss some key observations that arise from our evaluation.

7.5.2.1 Partial Disk Failures in JFS

Figures 7.3 and 7.4 compare the user-visible results of injecting read errors and corruptions respectively for JFS data structures. Each figure compares the results between (a) JFS used stand-alone and (b) JFS used as one of the child file systems in a 3-version file system that uses ext3 and ReiserFS as the other child file systems.

[7] We find that it is important to not have correct data in the buffer since JFS often ignores error codes and uses the data in the buffer.

Data structure	Description	Fraction
INODE	Inode block containing 8 inodes	0.01
DIR	Directory data block	1.5e-03
BMAP	Block-allocation bitmap block; these blocks are organized as a tree	1.2e-04
IMAP	Inode map block that contains pointers to inode blocks	2e-05
INTERNAL	Indirect block of a file, containing pointers to data blocks	1e-04
DATA	File data block	0.5
SUPER	Superblock of the file system	1e-07
JSUPER	Superblock of the journal	1e-07
JDATA	Journal data block	1.2e-03
AGGR-INODE-1	First block of the aggregate inode table	1e-07
IMAPDESC	Third block of the aggregate inode table; contains pointers to imap blocks	1e-07
IMAPCNTL	Inode map control block with summary information about the inode map	1e-07

Table 7.2 **JFS data structures.** *The table describes the different JFS data structures and the fraction of disk blocks of that type in a typical file system.*

Data structure	Description	Fraction
INODE	Inode block	0.01
DIR	Directory data block	1.5e-03
BMAP	Block-allocation bitmap block	3e-05
IMAP	Inode-allocation bitmap block	3e-05
INDIRECT	Indirect block of a file, containing pointers to data blocks	5e-04
DATA	File data block	0.5
SUPER	Superblock of the file system	1e-07
JSUPER	Superblock of the journal	1e-07
GDESC	Group descriptor block	3e-07

Table 7.3 **Ext3 data structures.** *The table describes the different ext3 data structures and the fraction of disk blocks of that type in a typical file system.*

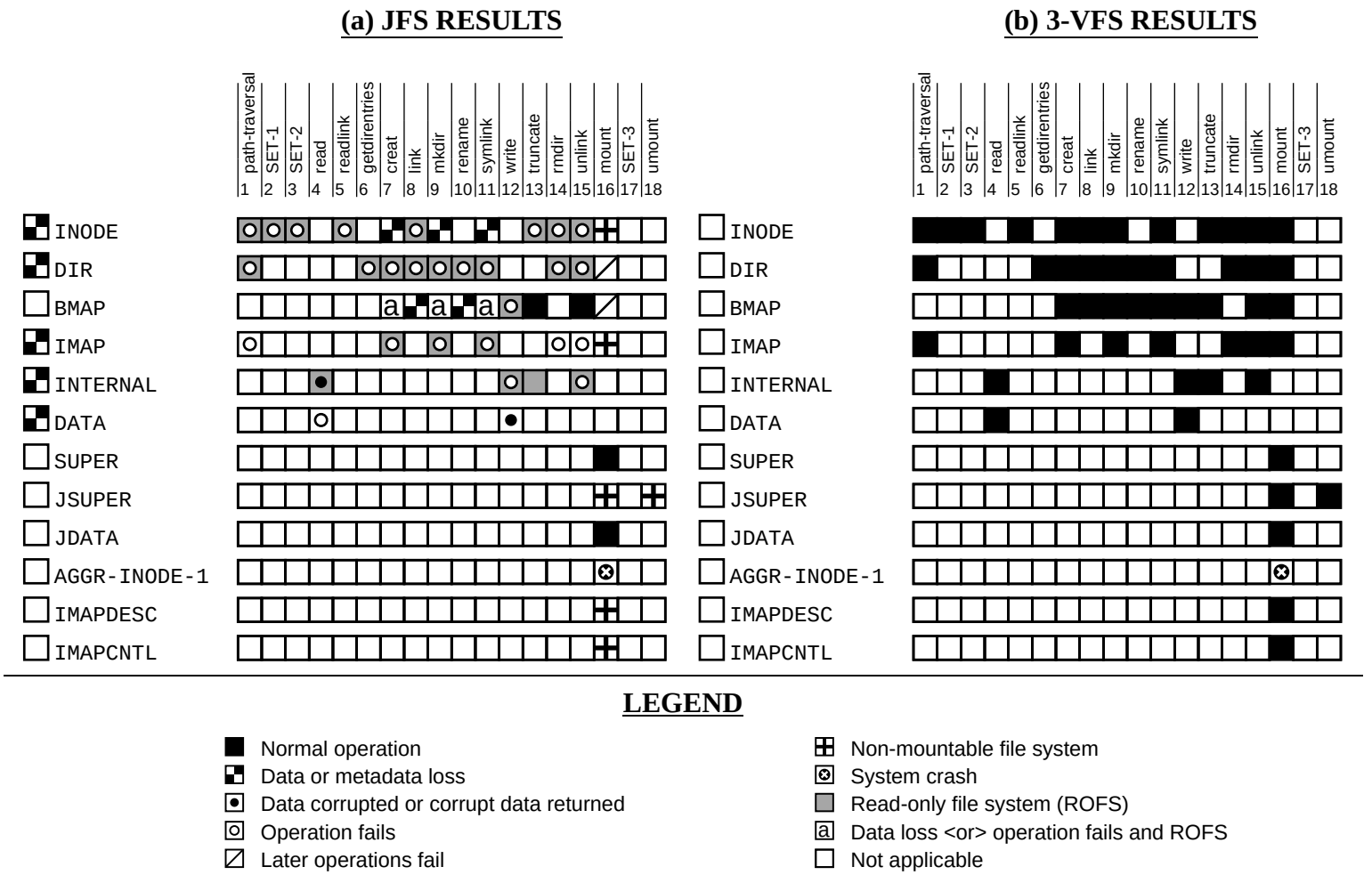

Figure 7.3 **Read error experiments for JFS.** *The figures show the results of injecting read errors for JFS data structures when (a) JFS is used stand-alone and (b) when JFS is one of the child file systems in a 3-version file system. Each row in the figures corresponds to the data structure for which the fault is injected; each column corresponds to a file operation; each symbol represents the user-visible result of the fault injection. Note that (i) the column SET-1 denotes file operations access, chdir, chroot, stat, statfs, lstat, and open; SET-2 denotes chmod, chown, and utimes; SET-3 denotes fsync and sync, (ii) some symbols are a combination of two symbols, one of which is the light-gray square for "read-only file system," (iii) [a] denotes cases where one of two possibilities could occur depending on disk state.*

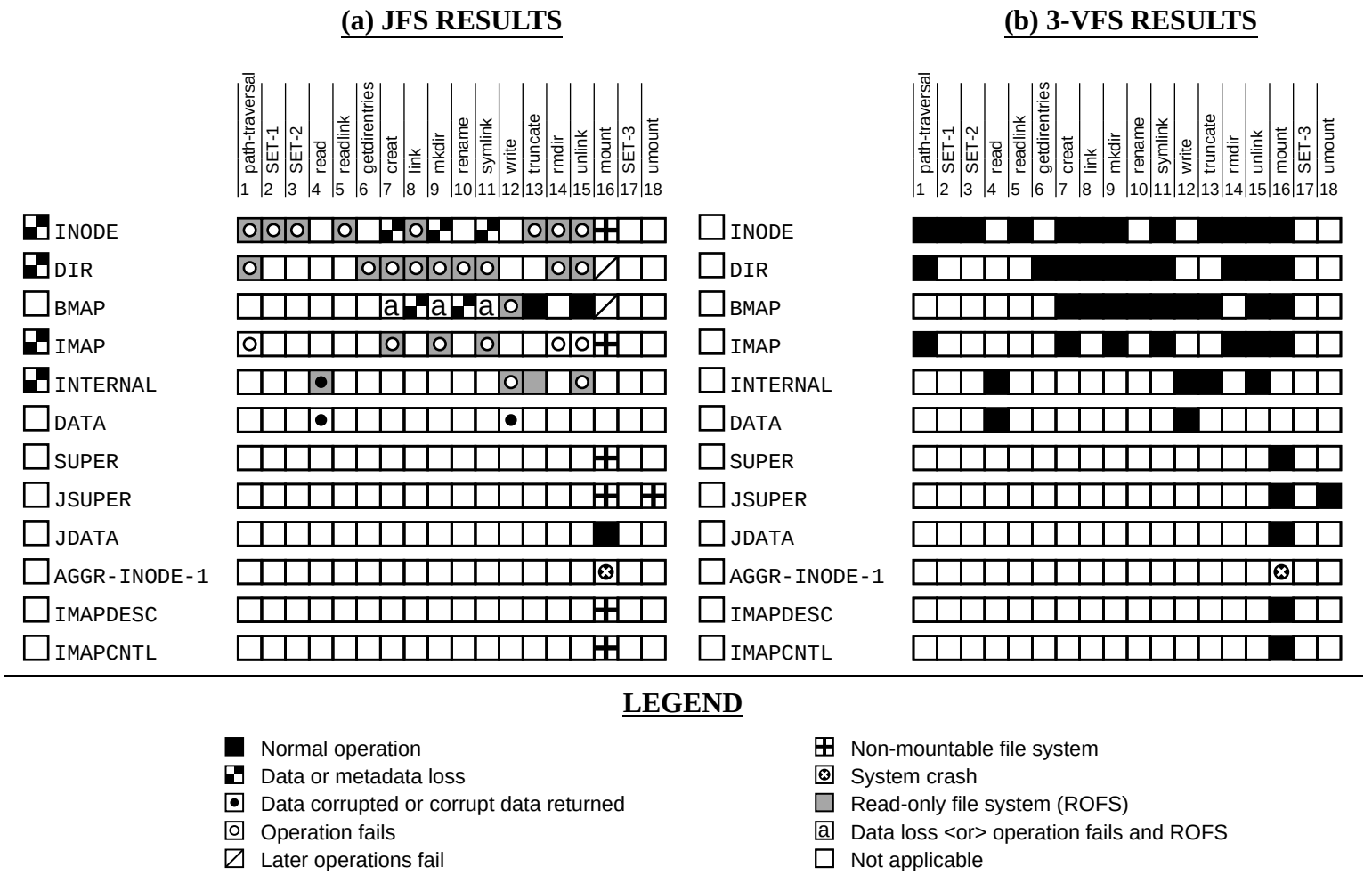

Figure 7.4 **Corruption experiments for JFS.** *The figures show the results of injecting corruption into JFS data structures when (a) JFS is used stand-alone and (b) when JFS is one of the child file systems in a 3-version file system. Each row in the figures corresponds to the data structure for which the fault is injected; each column corresponds to a file operation; each symbol represents the user-visible result of the fault injection. Note that (i) the column SET-1 denotes file operations access, chdir, chroot, stat, statfs, lstat, and open; SET-2 denotes chmod, chown, and utimes; SET-3 denotes fsync and sync, (ii) some symbols are a combination of two symbols, one of which is the light-gray square for "read-only file system," (iii) [a] denotes cases where one of two possibilities could occur depending on disk state.*

Each row in the figures corresponds to the data structure for which the fault is injected. Each column in the figures corresponds to different file operations. The exact instance of the data structure might be different for the different columns in a row. For example, while the first row is for inode blocks, in some cases it is a file inode (*e.g.*, unlink), while in some others it is a directory inode (*e.g.*, path traversal) and so on. The different symbols represent the user-visible results of the fault; examples of user-visible results include data loss, and a non-mountable file system. For example, in Figure 7.3a, when an inode block has an error during path traversal (column 1), the symbol indicates that (i) the operation fails, and (ii) the file system is remounted in read-only mode.

In most cases, there is only one user-visible result that could occur. Sometimes, there could be more than one possible result depending on disk state; for example, the symbol [a] denotes that either *data loss* or a combination of *operation failure* and *read-only file system* occurs. In addition to the symbols for each column, the symbol next to the data structure name for all the rows indicates whether or not the loss of the disk block causes irreparable data or metadata loss.

As shown in Figure 7.3a, JFS is able to recover from the read error and continue normal operation in very few of the cases; it uses a copy of the superblock to continue normal operation when a read to the superblock fails; it can continue normal operation when the read to the block-allocation bitmap fails during truncate and unlink (although disk blocks that should have been freed are now no longer available for allocation).

In most cases, partial disk failures result in undesirable results. Data loss is indicated for many of the rows; the loss of these data structures cannot be recovered from (there is no redundancy). Often, the operation fails and JFS remounts the file system in read-only mode. The loss of some data structures also results in a file system that cannot be mounted. In one interesting case, JFS detects the read error to an internal (indirect) block of a file and remounts the file system in read-only mode, but still returns corrupt data to the user.

In comparison to stand-alone JFS, the 3-version file system recovers from all except one of the read errors (Figure 7.3b). The 3-version file system detects errors reported by JFS and also detects corrupt data returned by JFS when the internal block fails during file read. In all these cases, the

3-version file system uses the two other file systems to continue normal operation. Therefore, no data loss occurs when any of the data structures is failed.

In one fault-injection experiment, a system crash occurs both when using JFS stand-alone and when using it in a 3-version file system. In this experiment, the first aggregate inode block (`AGGR-INODE-1`) is failed, and the actions of JFS lead to a kernel panic during paging. Since this call to `panic` is not in JFS code, it cannot been replaced as described in Section 7.3.1.1. Therefore, the kernel panic occurs both when using JFS stand-alone and when using a 3-version file system.

The results for corruption (Figure 7.4) are nearly-identical to the results for read errors. This similarity arises because JFS uses sanity checks (and not error codes) to detect both kinds of errors. Interestingly, JFS does not use the superblock copy to recover from corruption to the superblock while it uses the copy when a read error occurs. As in the case of read errors, the 3-version file system recovers and continues normal operation for all corruptions but one.

7.5.2.2 Partial Disk Failures in Ext3

Figures 7.5 and 7.6 show the results of injecting read errors and corruption respectively for ext3 data structures. As in the case of JFS, each figure compares ext3 against a 3-version file system.

We first examine the results of injecting read errors (Figure 7.5). Since ext3 does not utilize the available redundancy in data structures, none of the fault-injection scenarios lead to normal operation. In most cases, there is unrecoverable data loss (as denoted by the symbol next to each data structure name in the figures), and either the operation fails (ext3 reports an error) or the file system is remounted in read-only mode or both. In the remaining cases, the file system cannot even be mounted. We see from Figure 7.5b that the 3-version file system is able to continue normal operation in every single case.

We now examine the results of injecting corruption (Figure 7.6). Ext3 detects the corruption in various cases but cannot restore normal operation (the file operation fails and data loss occurs). In other cases, ext3 fails to detect corruption (*e.g.*, `IMAP`, `INDIRECT`), thereby either causing data loss (`IMAP`) or returning corrupt data to the user (`INDIRECT`). Finally, in one scenario (corrupt `INODE` during `unlink`), the failure to handle corruption leads to a system crash when the file system is

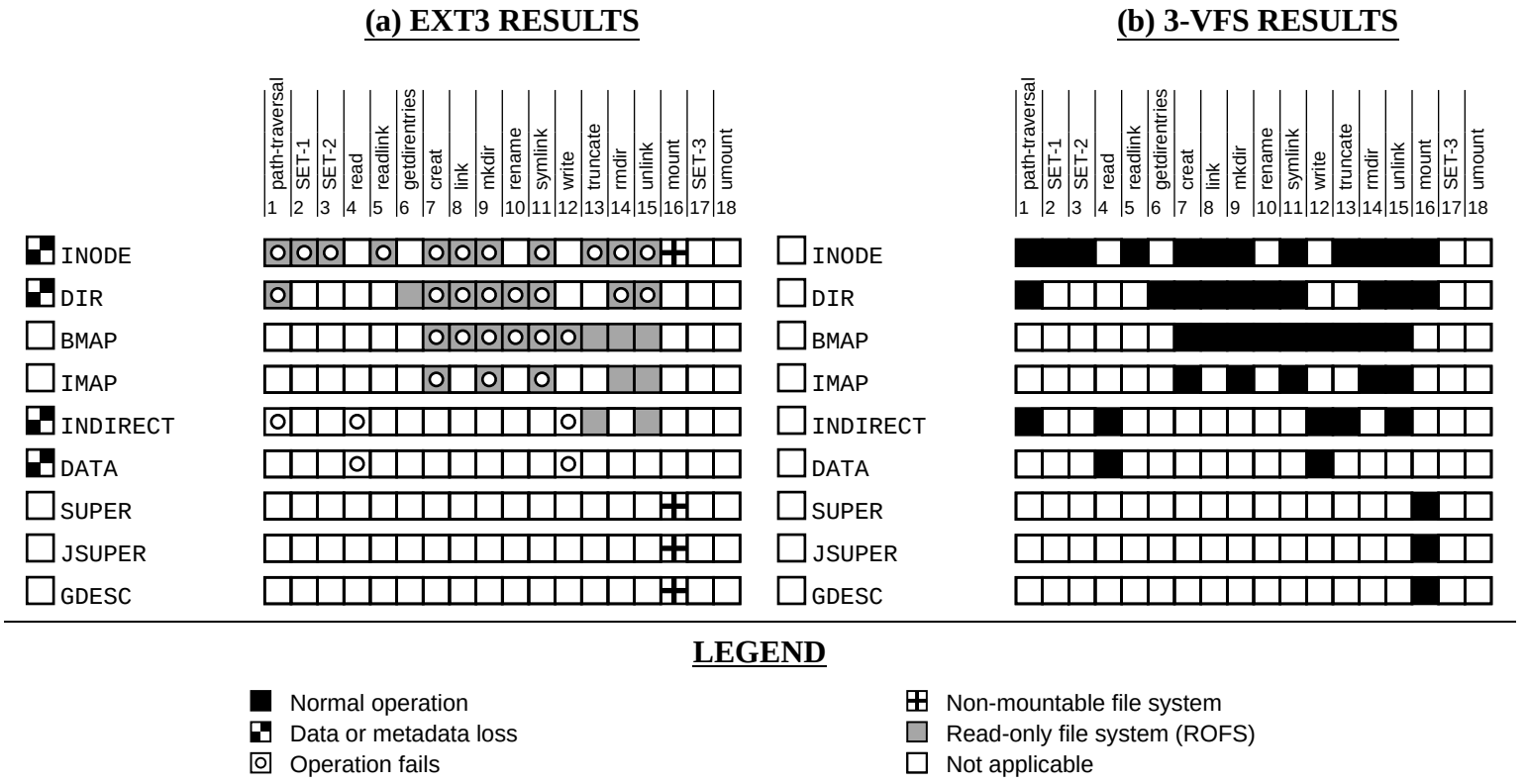

Figure 7.5 **Read error experiments for ext3.** *The figures show the results of injecting read errors for ext3 data structures when (a) ext3 is used stand-alone and (b) when ext3 is one of the child file systems in a 3-version file system. Each row in the figures corresponds to the data structure for which the fault is injected; each column corresponds to a file operation; each symbol represents the user-visible result of the fault injection. Note that (i) the column SET-1 denotes file operations access, chdir, chroot, stat, statfs, lstat, and open; SET-2 denotes chmod, chown, and utimes; SET-3 denotes fsync and sync, (ii) some symbols are a combination of two symbols, one of which is the light-gray square for "read-only file system."*

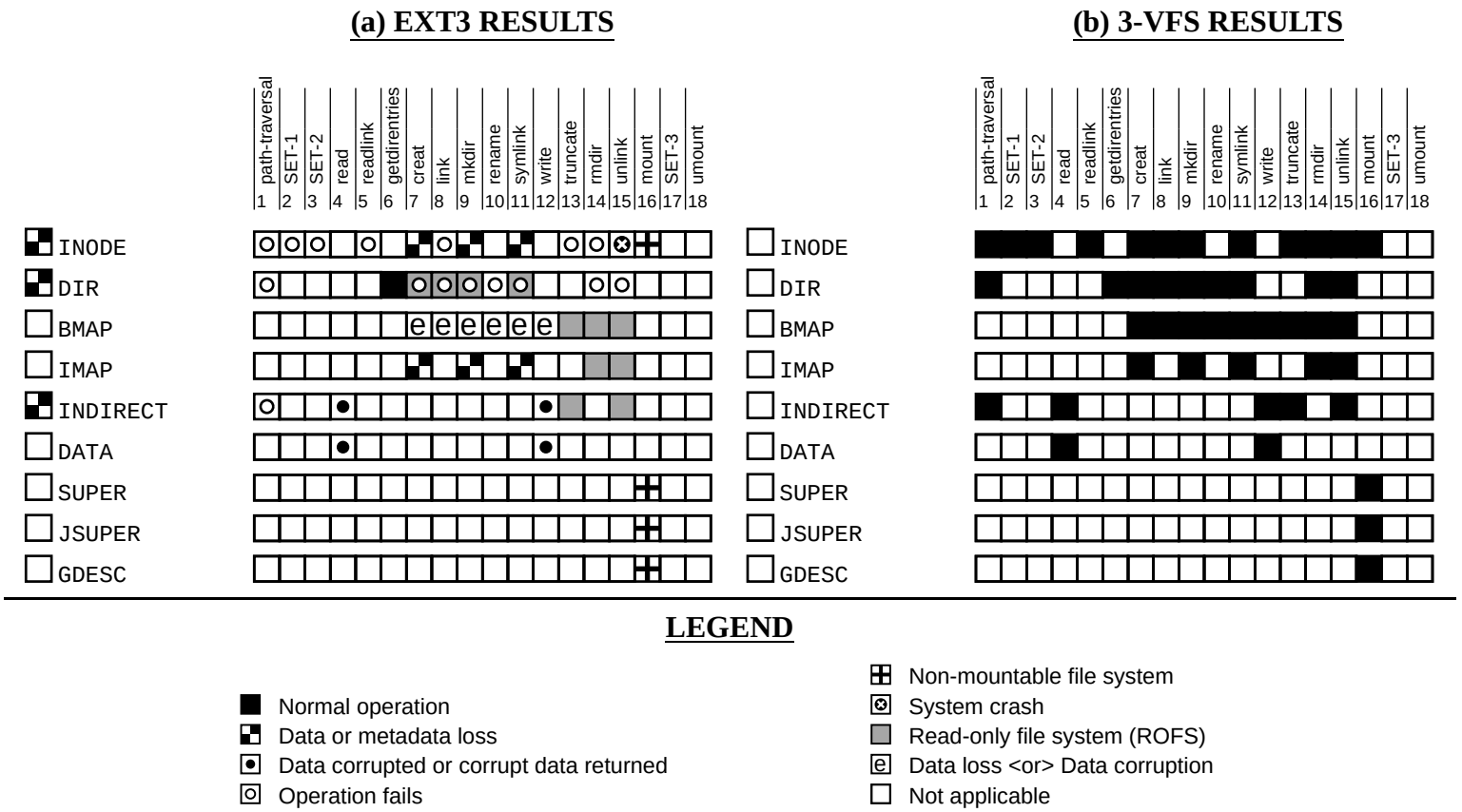

Figure 7.6 **Corruption experiments for ext3.** *The figures show the results of injecting corruption into ext3 data structures when (a) ext3 is used stand-alone and (b) when ext3 is one of the child file systems in a 3-version file system. Each row in the figures corresponds to the data structure for which the fault is injected; each column corresponds to a file operation; each symbol represents the user-visible result of the fault injection. Note that (i) the column SET-1 denotes file operations access, chdir, chroot, stat, statfs, lstat, and open; SET-2 denotes chmod, chown, and utimes; SET-3 denotes fsync and sync, (ii) some symbols are a combination of two symbols, one of which is the light-gray square for "read-only file system."*

unmounted. In comparison, Figure 7.6b shows that the 3-version file system can continue normal operation in every single experiment, including in the system-crash case. We discuss this case in greater detail in Section 7.5.2.4.

7.5.2.3 Probability Estimates

We now derive approximate probabilities of (at least 1) occurrence of each of the user-visible results in one year due to a read error. These estimates can be used to quantify the reliability impact of using a 3-version file system.

For this calculation, we use: (i) the probability of occurrence of read errors (latent sector errors) on a single disk in one year (from Section 3.3.2.1), (ii) the approximate fraction of disk blocks that belong to each block type (from Tables 7.2 and 7.3), and (iii) the results of injecting read errors (Figures 7.3, and 7.5),

We assume that (i) each disk block has an equal probability of being affected by a partial disk failure irrespective of the data structure it holds, (ii) the different file operations in the columns of fault-injection result figures will definitely occur at least once in a year, thereby triggering the corresponding user-visible result, (iii) multiple partial disk failures will not affect more than one of the replicas of the 3-version file system (less likely since the replicas do not have spatial locality), and (iv) the 3-version file system has JFS as one child file system and ext3 as two child file systems (since we have not characterized behavior with ReiserFS).

Table 7.4 presents the probability that each user-visible result will occur at least once in a year, comparing JFS, ext3, and a 3-version file system. We see that the 3-version file system is more reliable than either JFS or ext3, with a zero data-loss probability under one read error. The only non-zero probability for the 3-version file system is that of a system crash (1e-08).

7.5.2.4 Discussion

In this subsection, we discuss some key observations from our fault-injection experiments.

Bug localization: We find that corruption to an ext3 inode block read during `unlink` results in a system crash when the file system is subsequently unmounted. The system crash does not occur

User-visible result	JFS	Ext3	3-VFS
Data loss	0.0512	0.0512	0
Data corruption	0.050	0	0
Operation failure	0.0512	0.0512	0
Non-mountable file system	0.001	0.001	0
Read-only file system	0.0012	0.0012	0
System crash	1e-08	0	1e-08

Table 7.4 **Probability of undesirable results.** *This table presents the probability of each user-visible result occurring at least once in a given year due to read errors for: JFS, ext3, and a 3-version file system.*

Linux 2.6.12

```
static struct dentry *
ext3_lookup(struct inode * dir, struct dentry *dentry, struct nameidata *nd)
{
        struct inode * inode;
        ...
        if (bh) {
                ...
                inode = iget(dir->i_sb, ino);
                if (!inode)
                        return ERR_PTR(-EACCES);
        }
        if (inode)
                return d_splice_alias(inode, dentry);
        d_add(dentry, inode);
        return NULL;
}
```

Linux 2.6.23

```
static struct dentry *
ext3_lookup(struct inode * dir, struct dentry *dentry, struct nameidata *nd)
{
        struct inode * inode;
        ...
        if (bh) {
                ...
                        inode = iget(dir->i_sb, ino);
                if (!inode)
                        return ERR_PTR(-EACCES);
                if (is_bad_inode(inode)) {
                        iput(inode);
                        return ERR_PTR(-ENOENT);
                }
        }
        return d_splice_alias(inode, dentry);
}
```

Figure 7.7 **Bug in ext3_lookup.** *The figure compares the code for* `ext3_lookup` *in Linux kernel versions 2.6.12 and 2.6.23. The check* if(is_bad_inode(inode)) *in 2.6.23 is missing in 2.6.12 and its absence causes a system crash when a corrupt inode is unlinked.*

in the case of the 3-version file system, not because ext3 is modified to call nvfs_child_panic, but because the code paths that cause the panic are avoided. In particular, the 3-version file system detects that the inode returned by ext3 in response to a lookup (that is performed by VFS prior to the actual unlink) is faulty. Therefore, it does not issue the subsequent unlink operation to ext3, hence avoiding actions that cause the panic altogether. Interestingly, the bug that causes the crash is actually in the lookup operation, the first point where the 3-version file system detects a problem. Note that in the absence of a 3-version file system, one would find that the system crashed on an unmount, but will not have information linking the crash to the unlink system call or the bug in `ext3_lookup`. Figure 7.7 shows the code for `ext3_lookup` in Linux kernel versions 2.6.12 and 2.6.23. The check *if(is_bad_inode(inode))* and corresponding actions in 2.6.23 is missing in 2.6.12 and its absence causes the system crash. This experience highlights the potential for using an N-version file system to localize bugs in file systems.

Error codes: We find that different file systems use different error codes to report errors. The error codes vary from "Input/output error" to "Permission denied" to "Read-only file system" and so on. However, the system log shows that in most of these cases the file system detects a read error or corruption. A consistent error code to represent these scenarios would enable the N-version file system take further action than just the measures discussed in Section 7.2.3.2; for instance, any repair of file systems could retain a (dummy) file whose data block has a latent sector error so that the file system does not re-use this disk block for other files. In addition, if multiple file systems are affected by disk errors, in the absence of consistent error codes, the NVFS layer only detects the lack of a majority error code and reports an I/O error, but cannot detect a more pervasive disk problem. We find that this issue is one of the limitations of using existing file systems, one that cannot be solved by simple replacement of function calls as in handling kernel panic.

7.6 Conclusion

We have proposed the use of an N-version file system to tolerate all partial disk failures, including file system bugs. Our approach includes techniques that enable the use of existing file systems, thereby significantly reducing the cost of development. We have also proposed the use

of a single-instance store to reduce the performance and disk-space overheads of an N-version file system. The single-instance store is especially applicable in cases where file-system bugs are the primary contributors to partial disk failures. We have built an N-version file system for Linux file systems and show that it is significantly more reliable than file systems that it is composed of. We also show that the N-version file system can be used to localize bugs in file systems.

Modern file systems are becoming more complex by the day; mechanisms to achieve data structure consistency [140], scalability and flexible allocation of disk blocks [20, 132], and the capability to snapshot the file system [67, 130] significantly increase the amount of code and complexity in a file system. Such complexity could lead to bugs in the file system that render any data protection further down the storage stack useless. Thus, the use of an N-version file system would prove to be particularly relevant as file systems evolve.

Chapter 8

Related Work

This chapter discusses various research efforts and real systems that are related to this dissertation. We first discuss literature on characterization of system and disk failures, then summarize research on analyzing the failure behavior of systems. Next, we outline techniques that have been used to handle disk failures, and finally we discuss research related to N-version programming.

8.1 Failure Characteristics

This section discusses research efforts that have analyzed real-world data on failures. Such data often provides great insights into failure handling, both in terms of identifying techniques that can be used to tackle failures and in terms of fine-tuning the policies that trigger the use of different techniques. We first summarize research on the characteristics of system failures and then discuss research specifically targeted towards disk and storage subsystem failures.

8.1.1 System Failures

Various research efforts have studied real-world system failures. We focus on research on the root causes of failures in our discussion, in order to check whether disk failures play a major role in causing system failures.

One of the very first and very influential studies of system failure was by Gray [51]. Gray analyzes the cause of system failures reported to Tandem over a 7-month period for a sample set that covered more than 2000 systems. A total of 166 failures were reported, of which 42% of the failures were due to system administration. Software bugs caused 25% of the failures and hardware

problems caused 18% of the failures (disk drives 7%). These results spurred further research on system failures, especially with a view to verifying the impact of system administration on other systems. Murphy and Gent [91] analyze failures that occurred in VAX systems and find that from 1985 to 1993, the primary cause of failure shifted from software to system administration. Oppenheimer *et al.* [96] study internet systems and find that, again, operator error is the main cause of failure in two out of three services. All of these findings have sparked research in examining the effect of operator errors, as well as into techniques that can deal with the consequences [27, 79, 126, 143].

Other studies of system failure have laid the blame on software. In following up on his first study of failures, Gray [52] finds that from 1985 to 1989 the primary cause of failures shifted from system administration to software bugs (the cause of 62% of the failures in 1989); he surmises that as other components of the system become more reliable, software became the primary cause of failure. More recently, Murphy and Levidow [92] analyze the causes of Windows NT 4 failures and find that the core of NT caused failures 43% of the time, while drivers and hardware caused failures 32% and 13% of the time respectively.

Finally, hardware has been the main root cause of failure in various systems. Schroeder and Gibson [117] study failures in high-performance computing systems and find that hardware is the main contributor to system failure, causing about 60% of all failures and about 60% of system downtime. In a subsequent study focusing on disk failures, Schroeder and Gibson [118] find that disk drives were the hardware components to be replaced most frequently in two out of three of the data sets that they examined. They also find that disk drives were the third-most frequent hardware cause of node outages, while the most frequent was the CPU.

Most of the above studies do not identify disk drive or storage subsystem failures as the primary causes of system failure. However, the increasing trend of using large numbers of inexpensive hard disks [48] are expected make disk failures larger contributors to system failure. From the point of view of data loss, disk and storage failures are certainly the primary threats.

8.1.2 Disk and Storage Failures

This subsection discusses research that analyzes disk and storage subsystem failures. Unfortunately, very little data has been published on real-world disk and storage subsystem failures; there have been no large-scale studies of partial disk failures prior to our work.

In one of the first studies to focus on disk failures, Talagala and Patterson [137] analyze failures and errors that occurred in a 3.2 TB storage system and find that (i) disk drives are more reliable than other storage subsystem components, (ii) IDE disks are less reliable than SCSI disks, and (iii) medium errors (latent sector errors) occur, albeit much less often than network errors. A more recent study of storage subsystem failures by Jiang *et al.* [70], which uses the same error database as we do, agrees with Talagala and Patterson's observation that disk drives fail less often than other storage subsystem components. However, one must remember that although disk drives experience failures less often, these failures may lead to actual data loss, while other subsystem components typically lead to data unavailability.

Elerath and Shah have studied various aspects of disk-drive reliability in detail [40, 41, 121, 122]. These studies often do not provide actual numbers on disk failures, but use relative numbers to highlight important trends. For instance, Shah and Elerath show that the reliability of disk drives changes with their *vintage*, that is, the batches in which they are produced [121]. This behavior is explained by the maturing of manufacturing processes and changes to firmware code over time. In another study, Shah and Elerath show that the cause of disk failure changes with disk age [122]. Indeed, the observation in our study that nearline disks have a greater probability of developing latent sector errors as they age may contribute to this general phenomenon.

Recently, two large-scale studies of disk failure were published [102, 118]. Both studies focus on "complete" disk failures, wherein it is known that the disk failed or was replaced, but the exact cause of failure is not known or not presented. Schroeder and Gibson [118] study the disk replacement logs of systems that use a total of around 100,000 disk drives. They find that (i) disks fail more often than manufacturer-reported MTTF (Mean Time to Failure) numbers suggest, (ii) rather than have a significant infant-mortality effect, disk-replacement rates grow with age, and (iii) there is little difference between the SCSI and SATA replacement rates if the rejection of a

particularly bad batch of SATA disks is ignored. Pinheiro *et al.* [102] present numbers from a study of disk failures that occurred at Google in a population of more than 100,000 disks. They find that (i) there is very little correlation between disk failures and temperature, or activity levels, and (ii) some SMART [2] parameters like scan errors and reallocation counts have a large impact on failure probability.

In the study most related to ours, Gray and van Ingen [53] present a small-scale analysis of partial disk failures. They moved 2 PB of data through about 17 disk drives over a period of 6 months, and detected 4 read error events. This number is lower in comparison to manufacturer-reported bit-error rates. As we have observed in studying partial disk failures, they found a case where the likely culprit was the disk controller and not the disk drive.

8.2 Analysis of Failure Behavior

This section presents research efforts that analyze how systems respond to failures, with a focus on file systems and storage systems. We first discusses efforts that use software fault injection to evaluate systems, and then discuss efforts that use other techniques such as model checking and static analysis to examine systems.

8.2.1 Software Fault Injection

A multitude of software fault-injection techniques and frameworks have been developed over the years [26, 33, 19, 29, 61, 74, 75, 90, 123, 139]. These frameworks differ in various ways: the types of faults they can inject, the ease of use of the framework, the monitoring capability they provide to track the propagation of faults, and so on. Some of the types of faults that can be injected using these frameworks include processor, memory, and bus faults [19, 29, 61, 74, 75, 139], disk faults [26, 139], communication faults [19, 61, 74], software faults [33, 75], and faulty user input [90].

The FTAPE [139] framework 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). Unlike our approach, the FTAPE fault injector does not inject type-aware faults. The framework was

implemented for studying the Tandem Integrity S2 fault-tolerant computer. It performs stress-based injection, in which faults are injected at times and locations of greatest workload activity. The authors show that this approach leads to higher errors to faults ratio, an indication that fault-tolerant mechanisms are being well-exercised.

A second related fault-injection study is an analysis by Siewiorek *et al.* [123] of how a file system reacts when various fields of file pointers are corrupt. Unlike our approach to pointer corruption, they do not corrupt pointers in other metadata structures. Also, unlike us, they perform the corruption in memory once a file has been opened and do not use type-aware corruption values. As a result of these differences, their approach does not provide information about why the file system is able or unable to handle the corruptions.

Also closely related to our work is the disk-fault injector developed by Brown and Patterson [26]. More than the fault-injection technique, it is their goal of unearthing the *design philosophy* of the system, as opposed to simply reporting the fraction of experiments that the system handled, that has inspired our own fault-injection methodology. For their experiments, they use a PC as a SCSI "target"; code running on the PC emulates disk drives and their faults. Using the framework, they measure the availability of software RAID systems in various operating systems, and 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. It is interesting to see that some of these design-philosophy findings are applicable to file systems as well; we find that Windows NTFS performs many more retries of disk operations than Linux file systems do, thereby helping it tolerate transient errors better.

8.2.2 Other Approaches

In addition to fault injection, various other approaches have been used to examine the fault-tolerance capability of systems.

Analytical modeling has primarily been used to examine the reliability of RAID systems. Gibson [49] developed a model for RAID systems that focused on absolute disk failures. Over the

years, the analytical models have been refined to include the impact of latent sector errors and data corruption, and additional techniques such as disk scrubbing [17, 39, 76, 119]. The results of our analysis of the characteristics of partial disk failures, such as the fact that these failures are not independent, can be used to refine the different models.

Model checking is a formal technique that has been used over the years to analyze a variety of systems [73]. Recently, model checking has been adapted to work well with real operating system code [93], and subsequently employed to find bugs in file systems [150]. Interestingly, our type-aware fault-injection technique identifies a different set of bugs than the ones triggered by model checking [104].

Static analysis is another formal technique that has been used to study file systems. Yang *et al.* [149] use static analysis to analyze the disk mounting code of ext2, ext3, and JFS and find bugs in all of them. These bugs could potentially cause a kernel panic or allow buffer overflow attacks when a malicious disk image is mounted. We have observed a similar bug in NTFS, where a corrupt on-disk pointer could cause a system crash (Chapter 6). Recently, Gunawi *et al.* [56] use static analysis to investigate the propagation of errors through file system code and find that many functions drop error codes that should have been propagated up.

Some research efforts have manually inspected and analyzed systems as well. In recent work, Hafner *et al.* [58] manually analyze the impact of various forms of data corruption on RAID data protection. They present an example of the parity pollution problem that we identified in Chapter 4 and show that such a problem could occur in a RAID system with double-disk parity protection as well. Carrier [30] analyzes the importance of various fields in on-disk data structures in different file systems from a forensic viewpoint. He also presents methods that can be used to recover data in cases where one cannot use the file system itself to obtain the data (*e.g.*, from deleted files).

8.3 Handling Partial Disk Failures

This section describes research efforts that focus on handling partial disk failures. We focus on systems and techniques not discussed previously as background (Section 2.4) or in our evaluation of RAID data protection (Chapter 4).

Sivathanu *et al.* [124] develop a technique called *type-safe disks*, in which disk drives with knowledge of file-system data structures prevent file systems from accessing data that it shouldn't. For example, if a file system does not read the inode containing the pointer to a data block prior to accessing the data block itself, the disk drive returns an error. This approach is particularly useful in handling pointer-related file-system bugs. However, the approach relies on a disk drive's bug-free operation.

Gunawi *et al.* [55] present a technique called *I/O shepherding*, using which a variety of data protection techniques for file systems can be easily composed. The I/O shepherd co-operates with the file system for various purposes including identifying the type (*e.g.*, inode) of disk blocks, performing disk block allocating, and journaling replicas used for fault-tolerance. While this approach provides more flexibility over previous data protection approaches [104, 129, 130], it does not protect against file-system bugs.

8.4 N-Version Programming

Over the years, N-version programming has been used in various real systems and research prototypes to reduce the impact of software bugs on system reliability. As noted by Avižienis [6], N-version computing has very old roots:

> "The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods." – Dionysius Lardner, *Edinburgh Review*, 1834 [6]

> "When the formula to be computed is very complicated, it may be algebraically arranged for computation in two or more totally distinct ways, and two or more sets of cards may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the accuracy of them all." – Charles Babbage, *Unpublished Manuscript*, 1837 [6]

The concept was (re)introduced in computer systems by Avižienis and Chen in 1977 [7]. Since then, various other efforts, many from the same research group, have explored the process as well as the efficacy of N-version programming [8, 10, 9, 31, 80].

Avižienis and Kelly [9] study the results of using different specification languages; they use 3 different specification languages to develop 18 different versions of an airport scheduler program. They perform 100 demanding transactions with different sets of 3-version units and determined that while at least one version failed in 55.1% of the tests, a collective failure occurred only in 19.9% of the cases. This demonstrates that the N-version approach reduces the chances of failure. Avižienis *et al.*also determine the usefulness of developing the different software versions in different languages like Pascal, C etc. [10]. As in the earlier study, the different versions developed had faults, but only very few of these faults were common and the source of the common faults were traced to ambiguities in the initial specification.

N-version computing has been employed in many systems. For many years, such uses have primarily been in mission- or safety-critical systems [144, 151]. More recently, with the increasing cost of system failures and the rising impact of software bugs, many research efforts have focused on solutions that use N-version programming for improving system security and for handling failures [36, 72, 112, 142]. Joukov *et al.* [72] store data across different local file systems with different options for storing the data redundantly. However, unlike our approach, they do not protect against file-system bugs, and inherently rely on each individual file system to report any errors, so that data recovery may be initiated in RAID-like fashion. Rodrigues *et al.* [112] develop a framework to allow the use heterogeneous network file systems as replicas for Byzantine-fault tolerance. Vandiver *et al.* [142] explore the use of heterogeneous database systems for Byzantine-fault tolerance. They specifically address the issue of ordering of operations using *commit barriers*. In our N-version file system, this issue is made simpler due to two reasons: (i) in the absence of transactions, file systems are not expected to provide atomicity across multiple operations on the same file, and (ii) the VFS layer can easily identify conflicts through locking of file system data structures.

Chapter 9

Conclusions and Future Work

> "There is much to be said for failure. It is much more interesting than success."
>
> – Max Beerbohm, *Mainly on the Air*, 1946.

Most disk-drive failures are partial failures, wherein only a few blocks on disk are inaccessible or a few blocks are silently corrupted. These partial disk failures could lead to permanent data loss, or worse, cause unwitting users and applications to use corrupt data.

We have adopted a complete and detailed approach to addressing the threat posed by partial disk failures; we have examined the characteristics of these failures, analyzed how the failures impact various storage-stack components, and developed a solution for tolerating these failures.

In this chapter, we first summarize each of the three components of our work, then discuss general lessons learned over the course of this work, and finally outline directions for future research.

9.1 Summary

To summarize our work, we review our study of the characteristics of latent sector errors and data corruption, then discuss our analysis of how partial disk failures impact RAID systems, virtual-memory systems and file systems, and finally recap N-version file systems.

9.1.1 Characteristics

We have performed the first large-scale study of the characteristics of partial disk failures; our sample of 1.53 million disk drives far exceeds the sample size of previous disk-failure studies [53,

102, 118, 137]. The disk drives used in our study are also diverse: they were sourced from multiple vendors; there were both nearline (SATA) and enterprise (FC) disks; within each class, there were different disk families, and within each family, there were different capacities. Such diversity helps derive better conclusions.

Our study was made possible because NetApp™ storage systems use various techniques to detect and recover from various partial disk failures. Our results show that partial disk failures affect a large number of disks; therefore these techniques are extremely important and well-worth the disk-space or performance overheads they impose.

We have studied two classes of partial disk failures, latent sector errors and data corruption, both of which could lead to permanent data loss. Our important observations include:

Magnitude of threat: Latent sector errors affect a large percentage of disks. They affect 20% of the disks belonging to a specific nearline disk model in 2 years. While silent data corruption affects fewer disks that latent sector errors, they are not as rare as one would hope. They affect more than 3% of disks of multiple nearline disk models in less than 1.5 years.

Factors: *Disk class* and *disk model* are important factors that influence the development of partial disk failures. The more expensive enterprise disks are more reliable than nearline disks with respect to partial disk failures; a smaller percentage of enterprise disks are affected than the nearline ones, across all of the disk models in the study. On average, nearly an order of magnitude fewer enterprise disks are affected by partial disk failures. *Disk age* and *size* are factors as well. In the case of nearline disk drives, the probability that a disk will develop a latent sector error within a given period of time increases as the age of the disk drive increases. Higher-capacity disk drives are more likely to develop latent sector errors than the lower-capacity ones of the same disk family.

Independence and locality: Partial disk failures are *not independent* within the same disk drive. They also show significant *spatial and temporal locality*. Data corruptions are not independent even across different disks in the same storage system, indicating that the cause

of corruption is perhaps a common component in the storage subsystem, such as a storage adapter.

Correlations: Latent sector errors and silent data corruptions correlate with each other and with various other disk errors, such as not-ready-condition errors.

Detection: A large percentage of partial disk failures are discovered through different forms of *disk scrubbing*. Also, a significant number of data corruptions are detected during *RAID reconstruction*; in the absence of protection against double disk failures, these cases would lead to data loss.

We have used these observations to derive important lessons for data protection including the importance of using disk scrubbing and studying its benefits in greater detail, and the need to place redundant copies of data far away from each other when stored on the same disk.

9.1.2 Impact

We have examined the impact of partial disk failures on a variety of important storage-stack components: RAID systems, virtual-memory systems, and file systems. We first summarize the analyses, then compare different systems and discuss our experience with the methodology used.

Of the different systems that use disk drives, RAID is specifically targeted towards handling disk failures. Therefore, one would expect a thorough and verifiable failure-handling scheme. We tested this premise by developing and using a simple model checker to examine data protection in single-parity RAID systems. We have analyzed a range of data protection techniques used in real systems, including disk scrubbing, various kinds of checksums, and two types of identity information. We found that composing these techniques does not provide the expected fault-tolerance; in each scheme, there are one or more scenarios that result in data loss or corrupt data being returned to the user. Many schemes suffer from parity pollution, where corruption can propagate from a bad data block to the parity block, thereby causing the data to become unrecoverable. We also showed that the use of version mirroring along with block checksums, physical identity and logical identity

provides an efficient way to tolerate all partial disk failures.[1] In the future, as protection evolves further to cope with the next generation of disk problems, we believe that a formal approach such as ours will be extremely useful in verifying the protection.

We have extended type-aware fault injection to analyze virtual-memory systems, a crucial component of operating systems. We analyzed the virtual-memory systems of Linux, FreeBSD, and Windows XP, and found that (i) the virtual-memory systems use only simple techniques to deal with partial disk failures; there is no attempt to use techniques like redundancy to completely recover from the failures, (ii) tolerance mechanisms are under-developed; for example, the Linux swap header has a provision to store a list of bad blocks, but the list is not updated as and when errors are detected, and (iii) virtual-memory systems do not detect most corruptions; this neglect may have severe consequences as demonstrated by the system crash that occurs when FreeBSD does not detect corruption of the kernel thread stack.

Our final analysis was motivated by the fact that any system that wishes to preserve data needs to preserve access paths to the data. In the case of file systems, the pointers located in on-disk data structures serve as the access paths. We have developed type-aware pointer corruption, an extension of type-aware fault injection, to observe how file systems respond to corrupt on-disk pointers. We analyzed two widely-used file systems, NTFS and ext3, and found that while these file systems use fault-tolerance techniques such as type checking, sanity checking, and replication, these techniques are poorly-designed and implemented in the systems. As a result, the file systems do not even detect some pointer corruptions that they could have otherwise recovered from.

Thanks to our analysis of both file systems and virtual-memory systems we can now compare the failure-handling policies of the two kinds of systems:

- Both kinds of systems share problems like illogical inconsistency and implementation bugs in failure-handling code. This points to a general disregard for partial disk failures, thus exposing commodity computer systems to data loss, data corruption and inexplicable crashes.

[1] Specifically, version mirroring is storing version numbers on a data block and its corresponding parity block; block checksum is a checksum of the block stored in the block; physical identity is the disk number and disk block number stored in the block; logical identity is the inode number and offset within a file for that block.

- The Linux virtual-memory system, like some file systems [104], misses a large number of write errors.

- Both virtual-memory systems and file systems do not deal with corruption errors in an elegant manner. We see that file systems perform some type and sanity checking to deal with corruption to file system data structures, but there is no protection for user data (which is the only data handled by virtual-memory systems for the most part). Recent file systems such as ZFS [130] are targeted at addressing this issue; it would be interesting to examine how they well they work in practice.

- The FreeBSD and Windows XP virtual-memory systems leverage an important difference between file systems and virtual-memory systems in that writes are required to succeed in file systems, while virtual-memory systems have alternatives like choosing an other page as victim and writing it elsewhere on disk.

Experimenting with multiple systems not only helps us compare these systems, but also provides an insight into the advantages and limitations of our methodology. Our experience is that the techniques are simple to use and can be applied to many different systems. In the case of type-aware fault injection, while the tool has to be rewritten for each environment, we found that the task was not onerous.

We observed one particular limitation: there is no easy way to identify the source of disk accesses and the accesses may be attributed to error recovery while it may be unrelated. An example of this problem occurred when a read error was injected for a user data page in FreeBSD or Windows XP. We observed what seemed to be a "retry" of the read. Even if this "retry" succeeded, the application was terminated, indicating a possible bug in the retry code. Only closer examination revealed that the second read was performed to create a core dump and not to recover from the error. It would therefore be interesting to explore techniques to identify the exact source of disk accesses in future work.

9.1.3 Tolerance

Our solution to the problem of partial disk failures was motivated by the lessons that we learned from the above studies: partial disk failures affect a high percentage of SATA disks, the kind used in our personal computers; commodity file systems that use these drives are poor at handling partial disk failures, despite years of development and testing; the file systems also have numerous bugs. Therefore, we cannot rely on a single complex file system to handle partial disk failures.

We have developed an N-version file system, in which data is stored and retrieved from N different file systems. We started with the hypothesis that the N-version file system can be built to use the existing POSIX specification and existing file systems, thereby reducing the high development costs typically associated with building N-version software. We also made it a goal to keep the software entity that replicates user operations across file systems as simple as possible. From our design and development effort, we found that it is indeed possible to build a simple entity that can use existing file systems.

We have addressed the disk-space and performance overheads of an N-version file system by developing a block-level single-instance store that is specifically designed for an N-version file system. This single-instance store uses content hashes to coalesce disk blocks with the same content. Due to differences in data structures across different file systems, the layer coalesces only user data and not file-system metadata, thereby providing metadata replication to protect against file-system bugs and partial disk failures that affect metadata.

We have evaluated the reliability of the N-version file system using two types of experiments: experiments where one of the file systems has wrong contents, and type-aware fault-injection experiments where one of the file systems suffers a partial disk failure. We found that the N-version file system recovers successfully from almost all of the scenarios, thus significantly reducing the probability of data loss or file-system unavailability due to partial disk failures.

Modern file systems are becoming more complex by the day. As a result, bugs in file systems are becoming the norm rather than the exception. The use of N-version file systems could prove to be even more relevant as file systems evolve.

9.2 Lessons Learned

In this section, we briefly discuss general lessons we learned while working on our thesis.

Information and control for failures: In current systems, applications receive little information about failures other than a single error code, and have little say in how failures are handled. For example, disk errors could cause the virtual-memory system to deliver a SIGBUS signal to the application; in the absence of more information about what memory page(s) were lost, the application has no option but to terminate. Even worse, in some cases, the error code delivered to applications does not even reflect the fact that a partial disk failure was detected (*e.g.*, in many cases, JFS returns errors like "Permission denied" and "No space left on device" instead of "Input/output error"). We believe that future system interfaces should provide for much richer information and control mechanisms for failures (*e.g.*, applications could be allowed to register to receive a notification if an asynchronous write fails).

Software bugs: One cannot expect that the complex software systems of today will be bug-free. Software bugs and imperfections are the norm rather than the exception. While it is important to focus on identifying and eliminating bugs, it is equally important to build systems that can tolerate these bugs. A few recent systems are motivated by this theme, such as Nooks [133, 134] and our N-version file system. We believe that the design of future systems will be strongly influenced by the need to handle bugs.

9.3 Future Work

In this section, we outline various avenues for future research. We first discuss possible future studies of failure characteristics, then outline potential analyses of failure-handling techniques, and finally discuss extensions to N-version file systems.

9.3.1 Characteristics of Partial Disk Failures

Future efforts to characterize partial disk failures could focus on various questions on latent sector errors and data corruption that our current study does not answer. We discuss some such questions below.

Our study looks at data corruption across different disk models. We find that the numbers vary significantly across disk models, suggesting that disks (and their adapters) may directly or indirectly cause corruption most of the time. However, disks are only one of the storage-stack components that could potentially cause corruption. A recent study shows that other storage subsystem components do have a significant impact on storage failures [70]. Future studies could examine corruption numbers across different models or versions of all hardware and software components. Such a study may help pinpoint the exact sources of data corruption.

In our study, we examine various factors that affect the development of partial disk failures, such as the age or capacity of the disk drive. Future studies could analyze the impact of various other factors, including operating conditions. One important operating condition is the workload that the disk is subject to. In the results we obtained, the impact of workload on the development of latent sector errors or data corruption is unclear, especially due to the lack of fine-grained disk-level workload information. Examples of such fine-grained information include the number of disk seeks that are performed and how "bursty" the workload is. Future studies could thus examine on correlations between fine-grained workload information and partial disk failures. In addition to disk-level workload, a future study could analyze the impact of logical-level workload; for example, one could study whether a data structure that is written frequently is the one that is corrupted frequently (we have seen indications that this may be so in Section 3.4.6). Thus, future studies may focus on obtaining both logical- and disk-level workload information along with recording latent sector errors and silent data corruptions in order to explore the impact of workload. A second important operating condition is temperature. Although a recent study of absolute disk failures found that temperature does not affect failures significantly [102], a future study could examine this trend in the context of partial disk failures.

In addition to the above studies involving hard disks, future research could focus on partial failures that affect emerging storage technologies such as solid-state drives [113, 114].

9.3.2 System Analysis

We have used two different techniques, model checking and type-aware fault injection, to explore how partial disk failures impact storage-stack components. We believe that both techniques could be extended to study other systems.

We have used our RAID model checker to study a variety of data protection techniques in RAID systems, focusing on systems with a single parity disk and injecting only one failure. Avenues for future work would include extending our model checker to study the data protection offered by storage systems that use two parity disks [35, 43]. Hafner *et al.* [58] point out that there cases where the problems we discovered in single-parity schemes may extend to double-parity schemes as well. In the case of double-parity schemes, it would also be interesting to explore the impact of two partial failures. Finally, our analysis could be extended to analyze non-traditional RAID schemes such as RAID-Z [24].

We have used type-aware fault injection to analyze numerous file systems and virtual-memory systems. We found that the failure-handling in many of these systems was simplistic; for example, none of the systems used checksums to detect corruption. A newer file system such as ZFS [130] is more corruption-aware; it uses parental checksums to detect corruption. An interesting future project could look at how well checksumming works in practice by studying ZFS. Another avenue for future work is to extend type-aware fault injection to study other data management systems such as database systems, which are often configured to directly use disk drives.

9.3.3 N-Version File Systems

Future work on N-version file systems could focus on various enhancements, including file-system repair, proactive error detection, and single-instance caching.

File-system repair: Our N-version file system currently does not repair child file systems. When the N-version file system detects that one of its child file systems is faulty (*e.g.*, a file has

different contents in one file system), in order to restore the N-version file system to full replication, the erroneous child file system should be repaired. Future work in N-version file systems could focus on different ways in which such repair can be accomplished.

A simple approach would be to treat the "partial file-system failure" as an absolute file-system failure – completely erase the faulty file system and recreate it using data from the other child file systems. This approach has the benefit that any unknown corruptions in the faulty file system will be eliminated. However, recreating the entire file system would take time, and similar to RAID reconstruction, reading the entire contents from the other two file systems may yield errors that cannot be resolved.

A different approach would be to fix (create, replace, or delete) only the object that has wrong contents. However, consider the following scenario. A file with two hard links to it may have the wrong contents. If the N-version file system detects an error through one of the links, it may create a new file in the file system to replace the erroneous one, but there is no way to identify the directory where the other link is located (except through a scan of the entire file system). The issues and trade-offs across different approaches make file-system repair an interesting problem to address in future work. We believe that such "logical" file-system repair is applicable beyond N-version file systems as well.

Proactive error detection: In an N-version file system that is capable of performing file-system repair, system reliability could be further improved by detecting and fixing errors proactively. We have seen from our study of partial disk failures that disk scrubbing is very useful for detecting failures, thereby reducing the chances of double failures; "file-system scrubbing" that scans file-system contents in the background may be equally useful in detecting partial file-system failures.

Single-instance caching: In the current system, the block-level single-instance store interposes on disk requests and coalesces disk blocks with the same content. This helps reduce disk-space overheads and to avoid performance overheads that arise from disk accesses to the same data by N file systems. However, in this system, for each file data block, N copies of the data are maintained in the file-system cache, thereby wasting memory space. Future work could focus on

eliminating this overhead by enhancing the block-level single-instance store such that it interposes on file-system cache accesses as well; then, only one copy of each data block will be cached, thus reducing the memory space overhead of an N-version file system.

In addition to these specific enhancements, future work may explore the utility of N-version file systems in locating bugs in file systems. An N-version file system has a detailed view of both file operations that are performed as well as the manner in which one file system disagrees with other file systems. This information could potentially be leveraged to locate the bug in a file system that causes its content or response to differ from the other file systems. Section 7.5.2.4 details one such instance where our N-version file system helped locate a bug in ext3. This instance serves as proof of the potential that an N-version file system holds for locating bugs.

"*Prediction is very difficult, especially about the future.*" – Niels Bohr

LIST OF REFERENCES

[1] Adaptec, Inc. Adaptec SCSI RAID 2200S. http://www.adaptec.com/en-US/support/raid/scsi_raid/ASR-2200S/, 2007.

[2] Bruce Allen. Monitoring hard disks with S.M.A.R.T. *Linux Journal*, 2004(117):9, 2004.

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

[4] 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.

[5] 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.

[6] Algirdas A. Avižienis. The Methodology of N-Version Programming. In Michael R. Lyu, editor, *Software Fault Tolerance*, chapter 2. John Wiley & Sons Ltd., 1995.

[7] Algirdas A. Avižienis and Liming Chen. On the Implementation of N-Version Programming for Software Fault Tolerance During Execution. In *Proceedings of 1st Annual International Computer Software and Applications Conference (COMPSAC'77)*, Chicago, USA, 1977.

[8] Algirdas A. Avižienis, P. Gunningberg, John P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges. The UCLA DEDIX system: A Distributed Testbed for Multiple-version Software. In *Digest of 15th International Symposium on Fault-Tolerant Computing (FTCS'85)*, pages 126–134, Ann Arbor, MI, June 1985.

[9] Algirdas A. Avižienis and John P. J. Kelly. Fault Tolerance by Design Diversity: Concepts and Experiments. *IEEE Computer*, 17(8), August 1984.

[10] Algirdas A. Avižienis, Michael R. Lyu, and Werner Schütz. In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software. In *Digest of 18th International Symposium on Fault-Tolerant Computing (FTCS '88)*, Tokyo, Japan, June 1988.

[11] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Dependability Analysis of Virtual Memory Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*, Philadelphia, Pennsylvania, June 2006.

[12] 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, CA, June 2007.

[13] 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)*, San Jose, California, February 2008.

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

[15] Lakshmi N. Bairavasundaram, Meenali Rungta, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Limiting Trust in the Storage Stack. In *The 2nd International Workshop on Storage Security and Survivability (StorageSS '06)*, Alexandria, Virginia, November 2006.

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

[17] 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 1st EuroSys Conference (Eurosys '06)*, Leuven, Belgium, April 2006.

[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. http://jfs.sourceforge.net/project/pub/jfs.pdf, 2000.

[21] 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.

[22] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA '94)*, pages 245–254, Chicago, Illinois, April 1994.

[23] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle, Washington, August 2000.

[24] Jeff Bonwick. RAID-Z. http://blogs.sun.com/bonwick/entry/raid_z, November 2005.

[25] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel (Second Edition)*. O'Reilly, December 2002.

[26] 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.

[27] Aaron B. Brown and David A. Patterson. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.

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

[29] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998.

[30] Brian Carrier. *File System Forensic Analysis*. Addison Wesley Professional, March 2005.

[31] Liming Chen and Algirdas A. Avižienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Digest of 8th International Symposium on Fault-Tolerant Computing (FTCS'78)*, Toulouse, France, 1978.

[32] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[33] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[34] 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.

[35] 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.

[36] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davdison, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-Variant Systems - A Secretless Framework for Security through Diversity. In *Proceedings of the 15th USENIX Security Symposium (Sec '06)*, Vancouver, British Columbia, August 2006.

[37] Michael H. Darden. Data Integrity: The Dell—EMC Distinction. http://www.dell.com/content/topics/global.aspx/power/en/ps2q02_darden?c=us&cs=555&l=en&s=biz, May 2002.

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

[39] Jon G. Elerath and Michael Pecht. Enhanced Reliability Modeling of RAID Storage Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2007)*, Edinburgh, United Kingdom, June 2007.

[40] Jon G. Elerath and Sandeep Shah. Disk Drive Reliability Case Study: Dependence upon Head Fly-Height and Quantity of Heads. In *Proceedings of the IEEE Reliability and Maintainability Symposium*, pages 608–612, 2003.

[41] Jon G. Elerath and Sandeep Shah. Server Class Disk Drives: How Reliable Are They. In *Proceedings of the IEEE Reliability and Maintainability Symposium*, pages 151–156, January 2004.

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

[43] EMC Corporation. EMC Clariion RAID 6 Technology – A Detailed Review. http://www.emc.com/techlib/pdf/H2891_clariion_raid_6.pdf, July 2007.

[44] 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.*

[45] 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.

[46] 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.

[47] Gentoo HOWTO. HOWTO Install on Software RAID. http://gentoo-wiki.com/HOWTO_Gentoo_Install_on_Software_RAID, September 2007.

[48] 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.

[49] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, University of California at Berkeley, 1991.

[50] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 154, Washington, DC, USA, 2003.

[51] Jim Gray. Why Do Computers Stop and What Can We Do About It? In *6th International Conference on Reliability and Distributed Databases*, June 1987.

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

[53] Jim Gray and Catharine van Ingen. Empirical Measurements of Disk Failure Rates and Error Rates. Technical Report MSR-TR-2005-166, Microsoft Research, December 2005.

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

[55] 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.

[56] 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 Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.

[57] James L. Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, California, December 2005.

[58] James L. Hafner, Veera W. Deenadhayalan, Wendy Belluomini, and Krishnakumar Rao. Undetected Disk Errors in RAID Arrays. *IBM Journal of Research and Development*, 52(4/5):413–425, 2008.

[59] James L. Hafner, Veera W. Deenadhayalan, Krishnakumar Rao, and John A. Tomlin. Matrix Methods for Lost Data Reconstruction in Erasure Codes. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, California, December 2005.

[60] 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.

[61] Seungjae Han, Kang G. Shin, and Harold A. Rosenberg. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems. In *Proceedings of the International Computer Performance and Dependability Symposium (IPDS'95)*, 1995.

[62] David M. Hart. From "Ward of the State" to "Revolutionary Without a Movement": The Political Development of William C. Norris and Control Data Corporation, 19571986, 2005.

[63] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994.

[64] Val Henson. An Analysis of Compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS'03)*, Lihue, Hawaii, May 2003.

[65] Hitachi Data Systems. Hitachi Thunder 9500 V Series with Serial ATA: Revolutionizing Low-cost Archival Storage. www.hds.com/assets/pdf/wp_157_sata.pdf, May 2004.

[66] Hitachi Data Systems. Data Security Solutions. http://www.hds.com/solutions/storage-strategies/data-security/solutions.html, September 2007.

[67] 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.

[68] IBM Corp. IBM System Storage DS8000 Turbo. http://www.ibm.com/systems/storage/disk/ds8000/index.html, 2008.

[69] Hiroo Ishikawa, Tatsuo Nakajima, Shuichi Oikawa, and Toshio Hirotsu. Proactive Operating System Recovery. In *Poster Session of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[70] Weihang Jiang, Chongfeng Hu, Arkady Kanevsky, and Yuanyuan Zhou. Is Disk the Dominant Contributor for Storage Subsystem Failures? A Comprehensive Study of Failure Characteristics. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[71] Theodore Johnson and Dennis Shasha. 2Q: A Low-Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 439–450, Santiago, Chile, September 1994.

[72] Nikolai Joukov, Abhishek Rai, and Erez Zadok. Increasing Distributed Storage Survivability with a Stackable RAID-like File System. In *Proceedings of the 1st International Workshop on Cluster Security (Cluster-Sec'05)*, Cardiff, UK, 2005.

[73] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[74] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Computing*, 44(2), 1995.

[75] 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.

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

[77] 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.

[78] 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.

[79] Lorenzo Keller, Prasang Upadhyaya, and George Candea. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'08)*, Anchorage, Alaska, June 2008.

[80] John P. J. Kelly and Algirdas A. Avižienis. A Specification-Oriented Multi-version Software Experiment. In *Digest of 13th International Symposium on Fault-Tolerant Computing (FTCS '83)*, Milano, Italy, June 1983.

[81] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[82] 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.

[83] 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)*, San Jose, California, February 2008.

[84] 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.

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

[86] C. Lumb, J. Schindler, G.R. Ganger, D.F. Nagle, and E. Riedel. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 87–102, San Diego, California, October 2000.

[87] Peter Lyman and Hal R. Varian. How Much Information ? http://www.sims.berkeley.edu/how-much-info-2003, 2003.

[88] 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.

[89] Marshall K. McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, August 2004.

[90] Barton P. Miller, Lars Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12), 1990.

[91] Brendan Murphy and Ted Gent. Measuring System and Software Reliability using an Automated Data Collection Process. *Quality and Reliability Engineering International*, 11(5), 1995.

[92] Brendan Murphy and Bjorn Levidow. Windows 2000 Dependability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '00)*, New York, New York, June 2000.

[93] 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.

[94] NetApp, Inc. Introduction to Data ONTAP 7G. Technical Report TR 3356, NetApp, Inc., October 2005.

[95] NetApp, Inc. NetApp Storage Systems. http://www.netapp.com/us/products/storage-systems/, 2008.

[96] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)*, 2003.

[97] Jakob Ostergaard and Emilio Bueso. The Software-RAID HOWTO. http://tldp.org/HOWTO/html_single/Software-RAID-HOWTO/, June 2004.

[98] Chan-Ik Park. Efficient Placement of Parity and Data to Tolerate Two Disk Failures in Disk Array Systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(11):1177–1184, November 1995.

[99] 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.

[100] 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.

[101] 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.

[102] 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.

[103] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 297–310, San Antonio, Texas, June 2003.

[104] 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.

[105] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[106] Red Hat, Inc. The Journalling Flash File System, version 2. http://sourceware.org/jffs2/.

[107] 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.

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

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

[110] Martin Rinard, Christian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[111] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[112] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[113] Samsung Electronics Co., Ltd. Samsung Solid State Drives. http://www.samsungssd.com, 2008.

[114] SanDisk Corp. SanDisk SSD Solid State Drives. http://www.sandisk.com/OEM/ProductCatalog(1274)-SanDisk_SSD_Solid_State_Drives.aspx, 2008.

[115] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R.Ganger. On Multidimensional Data and Modern Disks. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 225–238, San Francisco, California, December 2005.

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

[117] Bianca Schroeder and Garth Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*, Philadelphia, Pennsylvania, June 2006.

[118] 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.

[119] 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.

[120] 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.

[121] Sandeep Shah and Jon G. Elerath. Disk Drive Vintage and its Effect on Reliability. In *Proceedings of the IEEE Reliability and Maintainability Symposium*, pages 163–167, January 2004.

[122] Sandeep Shah and Jon G. Elerath. Reliability Analysis of Disk Drive Failure Mechanisms. In *Proceedings of the IEEE Reliability and Maintainability Symposium*, pages 226–231, January 2005.

[123] 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.

[124] 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.

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

[126] 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.

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

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

[129] 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.

[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. http://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] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 1–16, San Francisco, California, December 2004.

[135] T10 Technical Committee. Information Technology: SCSI Primary Commands (SPC-2). Technical Report Project T10/1236-D Revision 5, September 1998.

[136] T13 Technical Committee. Information Technology – AT Attachment 8 – ATA/ATAPI Command Set (ATA-8/ACS). Technical Report T13/1699-D Revision 3f, December 2006.

[137] 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.

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

[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] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

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

[142] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine Faults in Transaction Processing Systems using Commit Barrier Scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

[143] Marco Vieira and Henrique Madeira. Recovery and Performance Balance of a COTS DBMS in the Presence ofOperator Faults. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '02)*, Bethesda, Maryland, June 2002.

[144] Udo Voges, editor. *Software Diversity in Computerized Control Systems*. Springer, Wien, New York, December 1988.

[145] John Wehman and Peter den Haan. The Enhanced IDE/Fast-ATA FAQ. http://burks.brighton.ac.uk/burks/pcinfo/hardware/atafaq/atafq.htm, 1998.

[146] 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.

[147] David Woodhouse. JFFS: The Journalling Flash File System. In *The Ottawa Linux Symposium*, July 2001.

[148] 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.

[149] 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.

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

[151] Y. C. Yeh. Triple-Triple Redundant 777 Primary Flight Computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, 1996.