SEMANTICALLY-SMART DISK SYSTEMS

by
Muthian Sivathanu

B.E. Computer Science (Anna University, India) 2000
M.S. Computer Science (University of Wisconsin-Madison) 2001

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy
in
Computer Sciences

University of Wisconsin - Madison
2005

Committee in charge:
Andrea C. Arpaci-Dusseau (Co-chair)
Remzi H. Arpaci-Dusseau (Co-chair)
David J. DeWitt
Mark D. Hill
Mikko H. Lipasti

©Copyright 2005 by Muthian Sivathanu.
All rights reserved.

Abstract

SEMANTICALLY-SMART DISK SYSTEMS
Muthian Sivathanu

Robust and efficient storage of data is a prerequisite of current and future computer systems. To keep pace with both rapid changes in technology as well as increasing demands from applications and users alike, storage systems must evolve in new and interesting ways.

Unfortunately, storage systems today have a problem: the range of functionality they can provide is fundamentally limited, despite the presence of significant processing power within them. The main reason for this limitation is that storage systems communicate with the outside world through a narrow block-based interface today, and therefore lack higher-level "semantic" understanding about how they are being used.

This thesis proposes a solution to this fundamental problem. It presents a new class of storage systems called "semantically-smart disk systems" (SDS's); such disk systems are capable of providing entirely new classes of functionality by exploiting information about the system above (*e.g.*, a file system or a database management system). An SDS does so by carefully monitoring the low-level stream of block reads and block writes that a storage system normally sees, and then inferring higher-level behaviors of the system above. Importantly, an SDS does so without any changes to the existing block-level storage interface, taking a pragmatic approach that enables ready deployment in existing computing environments.

In this thesis, we present a variety of techniques used by an SDS to track semantic information underneath modern file systems, demonstrating how to transform an I/O request stream into a source of useful high-level information for the underlying disk system. We also demonstrate the utility of semantic information within the disk system by presenting new improvements to the availability, security, and performance of storage. For example, we have built a storage system that exhibits much better availability under multiple failures by keeping semantically-

meaningful data available. In another case study, we show that semantic knowledge within the storage system can enable reliable secure deletion of data. Such innovations are impossible to implement in the current storage infrastructure, but become possible with the acquisition and careful use of semantic information. Finally, we present a new logic framework for reasoning about file systems and their interaction with storage systems, and use this logic to prove properties about inference within a semantically-smart disk system.

To my parents

Acknowledgements

I am indebted to my advisors Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau for making my graduate study experience both amazingly enjoyable and fruitful. Although I had no intention of pursuing a Ph.D when I joined UW, my work with Andrea and Remzi during my first year was sufficient to convince me that it would be a great experience, and in retrospect, I am very glad that I stayed on. Their invaluable guidance and constant feedback helped me mature significantly as a researcher over the last 5 years. Weekly meetings with them were always fun, thanks in part to their unflinching enthusiasm and sense of humor. If I had become a professor myself, Remzi and Andrea would have been my ideal model advisors I strive to emulate.

I would like to thank Mark Hill, David DeWitt, and Mikko Lipasti for serving on my thesis committee and providing valuable feedback and suggestions. I especially thank Mark Hill and David DeWitt for their great support during my job search. They provided wonderful feedback on my practice job talk and had plenty of useful insights while discussing various job options. I also thank Ben Liblit and Suman Banerjee for sharing their insights on the job search process.

My various summer internships during my graduate school career were enjoyable and useful in terms of providing me varied perspectives on industrial research. I would like to thank my mentors and colleagues during my various internships, mainly Mahesh Kallahalla, Ram Swaminathan, and John Wilkes in HP Labs, Honesty Young in IBM Almaden, Anurag Acharya in Google, and Madhu Talluri and Yousef Khalidi in Microsoft. They were all wonderful people to work with and I learned a lot in each of my internships. I especially thank Anurag and Yousef for their support and insights during my job search.

I was fortunate to have wonderful colleagues to work with at UW - Nitin Agrawal, Lakshmi Bairavasundaram, John Bent, Nathan Burnett, Tim Denehy, Brian Forney, Haryadi Gunawi, Todd Jones, James Nugent, Florentina Popovici, and Vijayan Prabhakaran. Our group meetings, hallway discussions and our coffee and ice-cream breaks were always enjoyable. Thanks to all of them.

My stay in Madison was made pleasant and fun-filled because of a great set of friends, especially Gogul, Lakshmi, Koushik, Madhu, Nitin, Prabu, Pranay, Ram, Ravi, Sekar, Veeve, Venkat, Venkatanand, Vijayan, and Vinod. I thank all of them for a great time.

Above all, words do not suffice to express my indebtedness and gratitude to my parents: they have been the single largest contributors to all my accomplishments, by means of their boundless love and constant support, guidance, and encouragement for all my actions. I am also deeply thankful to my brothers Gopalan and Sankaran for their love and support. I view myself profoundly lucky to have such wonderful parents and brothers, and dedicate this dissertation to them.

Contents

Abstract v

Acknowledgements ix

1 Introduction 1
- 1.1 Motivation: An Example . . . 2
- 1.2 Acquiring Semantic Knowledge . . . 3
- 1.3 Exploiting Semantic Knowledge . . . 4
- 1.4 Semantic Disks Underneath a DBMS . . . 5
- 1.5 Reasoning About Semantic Disks . . . 5
- 1.6 Evaluation Methodology . . . 6
- 1.7 Contributions . . . 6
- 1.8 Outline . . . 7

2 Background 9
- 2.1 Modern Storage Systems . . . 9
  - 2.1.1 RAID layout . . . 9
  - 2.1.2 NVRAM buffering . . . 10
  - 2.1.3 Block migration . . . 10
  - 2.1.4 Summary . . . 11
- 2.2 File System Background . . . 11
  - 2.2.1 Common properties . . . 11
  - 2.2.2 Linux ext2 . . . 12
  - 2.2.3 Linux ext3 . . . 12
  - 2.2.4 VFAT . . . 13
  - 2.2.5 Windows NTFS . . . 13

3 Semantic Disks: An Overview **15**
3.1 Basic Approach . . . 15
3.1.1 Benefits and concerns . . . 15
3.2 Alternative Approaches . . . 17
3.2.1 Explicit . . . 17
3.2.2 Implicit . . . 18
3.3 Evaluation Methodology . . . 19

4 Acquiring Semantic Knowledge **21**
4.1 Static Information . . . 21
4.2 Dynamic Information . . . 22
4.2.1 Classification . . . 23
4.2.2 Association . . . 25
4.2.3 Operation inferencing . . . 26
4.2.4 Accuracy of inference . . . 27
4.3 Dealing with Asynchrony . . . 27
4.3.1 Indirect classification . . . 28
4.3.2 Association . . . 29
4.3.3 Operation inferencing . . . 29
4.3.4 Impact of uncertainty . . . 29
4.4 Evaluation . . . 30
4.4.1 Time overheads . . . 30
4.4.2 Space overheads . . . 31
4.5 Summary . . . 33

5 Exploiting Semantic Knowledge **35**
5.1 File System Model . . . 35
5.2 Semantic Caching . . . 36
5.2.1 Tolerance to inaccuracy . . . 37
5.3 Journaling . . . 38
5.3.1 Design and implementation . . . 38
5.3.2 Evaluation . . . 41
5.4 Complexity Analysis . . . 43
5.5 Summary . . . 43

6 Improving Availability with D-GRAID **45**
6.1 Introduction . . . 45
6.1.1 The problem: Reduced availability due to semantic ignorance 46
6.1.2 A solution: D-GRAID . . . 46

6.1.3 Key techniques 47
6.2 Extended Motivation 48
6.3 Design: D-GRAID Expectations 49
6.3.1 Graceful degradation 49
6.3.2 Design considerations 50
6.3.3 Fast recovery 53
6.4 Exploring Graceful Degradation 53
6.4.1 Space overheads 54
6.4.2 Static availability 55
6.4.3 Dynamic availability 55
6.5 File System Model 56
6.5.1 Arbitrary ordering 57
6.5.2 Delayed update 58
6.5.3 Hidden operation 58
6.6 Implementation: Making D-GRAID 58
6.6.1 Graceful degradation 60
6.6.2 Live-block recovery 64
6.6.3 Other aspects of Alexander 65
6.6.4 Alexander the FAT 67
6.7 Evaluating Alexander 68
6.7.1 Does Alexander work correctly? 69
6.7.2 What time overheads are introduced? 71
6.7.3 How effective is access-driven diffusion? 72
6.7.4 How fast is live-block recovery? 73
6.7.5 What overall benefits can we expect from D-GRAID? . . 74
6.7.6 How complex is the implementation? 75
6.8 D-GRAID Levels 76
6.8.1 D-GRAID-0: No redundancy 77
6.8.2 D-GRAID-10: Mirroring 77
6.8.3 D-GRAID-5: Parity 79
6.8.4 Summary 80
6.9 Discussion: The Impact of Being Wrong 80
6.10 Summary 81

7 Exploiting Liveness Knowledge in FADED 83
7.1 Introduction 83
7.2 Extended Motivation 84
7.3 Liveness in Storage: A Taxonomy 86
7.3.1 Granularity of liveness 86

7.3.2 Accuracy of liveness information 87
7.3.3 Timeliness of information 88
7.4 File System Model 88
7.5 Techniques for Liveness Detection 90
7.5.1 Content liveness 91
7.5.2 Block liveness 91
7.5.3 Generation liveness 93
7.6 Case Study: Secure Delete 95
7.6.1 Goals of FADED 96
7.6.2 Basic operation 96
7.6.3 Coverage of deletes 98
7.6.4 FADED for other file systems 104
7.6.5 Evaluation 104
7.7 Implicit Detection Under NTFS 109
7.8 Explicit Liveness Notification 110
7.8.1 Granularity of `free` notification 111
7.8.2 Timeliness of `free` notification 111
7.8.3 Orphan allocations 112
7.8.4 Explicit notification in ext2 112
7.8.5 Explicit notification in ext3 112
7.8.6 Explicit secure delete 113
7.9 Discussion 114
7.10 Summary 115

8 Semantic Disks for Database Systems 117
8.1 Introduction 117
8.2 Extracting Semantic Information 118
8.2.1 Static information 119
8.2.2 Dynamic information 119
8.3 Partial Availability with D-GRAID 122
8.3.1 Design 122
8.3.2 Transactions and Recovery 125
8.3.3 Evaluation 126
8.3.4 Discussion 131
8.4 Secure Delete with FADED 131
8.4.1 Table-level deletes 131
8.4.2 Record-level deletes 132
8.4.3 Performance 133
8.4.4 Discussion 134

8.5 Towards a Semantic Disk-Friendly DBMS 134
8.5.1 Information required 134
8.5.2 How DBMSes can help semantic disks 136
8.6 Summary 137

9 A Logic of File Systems and Semantic Disks 139
9.1 Introduction 140
9.2 Extended Motivation 141
9.2.1 Reasoning about existing file systems 142
9.2.2 Building new file system functionality 142
9.2.3 Designing semantically-smart disks 143
9.3 Background 143
9.3.1 File system metadata 143
9.3.2 File system consistency 144
9.3.3 File system asynchrony 144
9.4 The Formalism 145
9.4.1 Basic entities 145
9.4.2 Beliefs and actions 146
9.4.3 Ordering of beliefs and actions 147
9.4.4 Proof system 148
9.4.5 Attributes of containers 148
9.4.6 Logical postulates 149
9.5 File System Properties 152
9.5.1 Container exclusivity 152
9.5.2 Reuse ordering 152
9.5.3 Pointer ordering 153
9.6 Modeling Existing Systems 153
9.6.1 Data consistency 153
9.6.2 Modeling file system journaling 157
9.7 Redundant Synchrony in Ext3 162
9.8 Support for Consistent Undelete 164
9.8.1 Undelete in existing systems 165
9.8.2 Undelete with generation pointers 166
9.8.3 Implementation of undelete in ext3 167
9.9 Application to Semantic Disks 167
9.9.1 Block typing 168
9.9.2 Utility of generation pointers 171
9.10 Summary 171

10 Related Work **173**
10.1 Smarter Storage 173
10.1.1 Fixed interfaces 173
10.1.2 More expressive interfaces 174
10.1.3 New programming environments 175
10.1.4 Smarter file systems 175
10.2 Implicit Systems 176
10.3 Partial Availability 176
10.3.1 Distributed file systems 177
10.3.2 Traditional RAID systems 177
10.4 Logical Modeling of Systems 178

11 Conclusions and Future Work **181**
11.1 Lessons Learned 182
11.2 Future Work 184
11.2.1 Implicit inference in other domains 184
11.2.2 Integrating logic into implementation checkers 185
11.2.3 More semantic disk functionality 186
11.2.4 Making semantic disks more semantic 186
11.3 Summary 187

Chapter 1

Introduction

Storage systems form the backbone of modern computing, and innovation in storage is crucial to improving present and future computing environments. Improving storage systems along various dimensions such as availability, security, and performance is of paramount importance to keep pace with ever-increasing modes of usage and new requirements on storage systems.

Unfortunately, the range of innovation possible in storage today is limited due to the *narrow interface* that exists between the storage system itself and the software layer (*e.g.*, the file system or DBMS) that uses the storage system. Storage systems today export a simple block-based interface (*e.g.*, SCSI) that abstracts the storage system as a linear array of blocks; file systems perform block reads and block writes into this linear address space. This interface was designed at a time when storage systems were simple, passive disks and thus fit well into the simple abstraction of a linear address space.

However, storage systems have since evolved into massively complex, powerful systems incorporating a wide range of optimizations. Today, storage systems are composed of multiple disks with different forms of redundancy to tolerate disk failures [12, 16, 22, 40, 49, 73, 76, 77, 95, 117], perform migration of blocks across disks for load balancing [29, 117], transparently buffer writes in non-volatile RAM before writing them out to disk [117], and perform transparent remapping of blocks to hide failure. As a result of such sophistication, a significant amount of low-level information and control is available within the storage system, such as the failure boundaries across disks and the exact mapping of logical blocks to physical disk blocks.

While storage systems have become more intelligent and complex, the narrow block-based interface has remained unchanged, mainly due to the massive industry-

wide effort required for such a change, and legacy issues. As a result, file systems no longer *understand* the storage system, but instead, simplistically continue to view the storage system as a “dumb” disk. Thus, the file system cannot implement any functionality that requires low-level knowledge or control over the low-level details of block layout; the storage system is the only locale that has the information required to provide such functionality.

Unfortunately, placing functionality within storage systems is limited as well, again due to the narrow interface. Storage systems simply observe a raw stream of block reads and block writes that have no semantic meaning. Semantic knowledge about the logical grouping of blocks into files, the type of a block (*e.g.*, data vs. metadata), liveness of blocks, and so on, is unavailable within the storage system. Thus, research efforts have been limited to applying disk-system intelligence in a manner that is oblivious to the nature and meaning of file system traffic, *e.g.*, improving write performance by writing data to the closest block on disk [30, 115].

Thus, the modern storage stack precludes an entire class of functionality that requires information both from the file system and the storage system. This thesis proposes a solution to this fundamental limitation.

In this thesis, we propose a new class of storage systems that bridges the information gap between file systems and storage *without* requiring any change to the existing block-based interface. Operating underneath an unmodified SCSI interface, such storage systems automatically track higher-level semantic information about the file system or database system running above, and utilize this information to provide new classes of functionality that are impossible to provide in existing systems. We call such a storage system that is aware of the semantics of the higher layer, a *semantically-smart disk system* (SDS). By not requiring any changes to the existing storage interface and in many cases, to the file system above, semantically-smart disk systems present a pragmatic solution, and such storage systems are extremely easy to deploy and adopt.

The thesis of this dissertation is that it is possible, feasible, and useful for a block-based storage system to track higher level semantic information about the file system. As we show later, such information can be tracked to a high level of accuracy that enables new classes of innovation within storage systems.

1.1 Motivation: An Example

To motivate the need for semantic intelligence within storage systems, we provide an example of a functionality that cannot be provided today, but becomes possible within a semantically-smart disk system. The example relates to improving the

availability of storage in the face of certain kinds of failures. Through the rest of this thesis, we present various examples of semantic disk functionality that improve security, availability, and performance of storage in new ways.

Modern storage systems tolerate disk failures by employing various forms of RAID [77]; by storing redundant information in a small number of disks, the storage system can recover from a small, fixed number of failures without losing any data. The availability guarantee provided by existing RAID systems is quite simple: if D or fewer disks fail, the RAID continues to operate correctly, but if more than D disks fail, the RAID is entirely unavailable. In most RAID schemes, D is small (often 1); thus even when most disks are working, users observe a failed disk system. For example, even 2 failures in a 10 disk RAID system would result in complete unavailability, despite the fact that 80% of the disks are still working; ideally, availability loss in this scenario should be at most 20%.

This "availability cliff" behavior in existing RAID systems is because the storage system lays out blocks oblivious of their semantic importance or relationship; most files become corrupted or inaccessible after just one extra disk failure. For example, the storage system has no information on the semantic importance of blocks, and therefore treats a root directory block in the same way it does a regular file block. Thus, if the extra failed disk happened to contain an important block, a large fraction of the file system is rendered unavailable. The file system cannot address this problem either, because it views the storage system as a single large disk, and thus has no information about the failure boundaries between the disks, nor can it control the exact physical disk to which a logical block is mapped. In Chapter 6, we present a system called D-GRAID that provides much better availability under multiple failures by exploiting semantic knowledge within the storage system. Specifically, D-GRAID enables graceful degradation of availability under multiple failures by selective replication of key metadata structures of the file system and fault-isolated placement of semantically-related data.

1.2 Acquiring Semantic Knowledge

An important constraint in a semantically-smart disk system is that the existing block-based interface to storage cannot be modified. Thus, an SDS *infers* semantic information by carefully monitoring the block-level read and write traffic. In order to bootstrap its inference, the storage system relies on some minimal understanding of the file system. Specifically, the SDS is embedded with *static* knowledge about the key on-disk structures of the file system. Once the SDS has this static knowledge, it can build upon it to extract more sophisticated kinds of information;

by monitoring the write traffic to some specific on-disk structures and observing how those structures change, the SDS can correlate those changes to higher level file system activities that should have led to the change.

A key challenge in tracking semantic information based on observing changes to on-disk structure is the *asynchrony* exhibited by modern file systems. File systems typically buffer data in memory and sometimes arbitrarily re-order block writes. While crucial for file system performance, such asynchrony effectively obfuscates information from the storage system, because the writes made to disk would reflect the effect of multiple file system operations, some of which may cancel each other. As we show later, such asynchrony imposes basic limits on the extent and accuracy of semantic information that can be tracked within an SDS.

In this thesis, we present detailed techniques to extract various kinds of semantic information within an SDS, and bring out the limits to how accurately such information can be tracked. We find that the dynamic behavior of the file system significantly affects the effectiveness and simplicity of dynamic inference, and identify various file system properties that can simplify our techniques or improve their accuracy. We experiment with a variety of file systems, namely Linux ext2, Linux ext3, VFAT, and to a smaller extent, Windows NTFS, in order to explore the generality of our techniques, and to analyze their conformance to the various file system properties that we identify. As we go through the thesis, we successively refine our model of the file system, starting from a very simple synchronous file system to more complex file systems with various dynamic properties.

1.3 Exploiting Semantic Knowledge

Given that the disk system has semantic knowledge about the file system, the next relevant question is its utility. To demonstrate the utility of semantic disks, we have built prototype implementations of several semantic disk case studies that show that semantic knowledge can improve storage systems in fundamental ways. Specifically, we present case studies targeted at improving the availability, security, and performance of storage.

One of the case studies we present is D-GRAID, a storage system that provides graceful degradation of availability under multiple failures, by ensuring that semantically-meaningful data is available even after catastrophic failures. D-GRAID also enables faster recovery from failure by exploiting semantic knowledge to preferentially recover only those blocks that are useful to the file system, *i.e.*, only blocks that are live at the file system. We show that D-GRAID significantly improves storage availability at a modest performance cost.

To demonstrate that semantic disks can also provide functionality that has extreme correctness requirements, we have designed and implemented a semantic disk system called FADED that performs *secure deletion* (*i.e.*, makes deleted data irrecoverable by performing repeated overwrites), by inferring logical deletes occurring within the file system. This case study utilizes semantic information in a way that directly impacts correctness; an error in detecting a delete could result in irrevocable deletion of valid data, or a missed secure deletion of really deleted data. By showing that reliable secure deletion can be implemented despite inherently uncertain information, we show that even functionality that is correctness-sensitive can be implemented within semantic disks.

From our various case studies, we find that despite limits on the accuracy of semantic inference, complex functionality can be implemented within a semantically-smart disk system. The key to implementing complex functionality in an SDS is *conservatism*. By identifying conservative techniques and abstractions, one can circumvent the inherently inaccurate information and still provide guarantees about the functionality implemented based on the information. We also find that the performance cost of such conservatism is quite small; in one of our case studies, it results in a 10-12% overhead. Besides the cost of conservatism, we find that there is also a modest CPU cost to tracking semantic information. We quantify these costs during our discussion of the case studies.

1.4 Semantic Disks Underneath a DBMS

Given that database management systems (DBMS) are another prime client of storage besides file systems, we have also investigated techniques for semantic inference underneath a DBMS. We have implemented two of our major case studies, D-GRAID and FADED, in the DBMS case, and bring out key differences. Overall, we find that semantic inference underneath a DBMS is easier because of the write-ahead logging performed by a DBMS; the log thus communicates to the semantic disk a complete time-ordered list of operations that the DBMS does. However, we also find that database systems track fewer general purpose statistics than file systems and this limits the effectiveness of some of our case studies. We identify and propose minor changes to database systems that will address this limitation.

1.5 Reasoning About Semantic Disks

Developing functionality within semantic disks entails careful reasoning about the accuracy of different pieces of semantic information that the functionality relies

on, and the accuracy of information in turn depends on the specific properties of the file system running above. In the process of developing our techniques and case studies, we found this reasoning to be quite challenging, and recognized the need for a more systematic formal framework to model semantic disks and their interaction with file systems.

Towards the end of this thesis, we present a formal logic for representing and proving properties about file systems and semantic disks. Although the intended initial goal of this logic was to model semantic disks, we identified that reasoning about information available to a semantic disk has a strong parallel to reasoning about file system consistency management, since in both cases, the information purely pertains to what can be "known" from on-disk state. Thus, we present this logic as a way to model file systems in general and reason about their correctness properties, and then show how we can use it to reason about semantic disks.

1.6 Evaluation Methodology

We evaluate our techniques for semantic inference and our various case studies through prototype implementations. To prototype an SDS, we employ a software-based infrastructure. Our implementation inserts a pseudo-device driver into the kernel interposing between the file system and the disk. Similar to a software RAID, our prototype appears to file systems above as a device upon which a file system can be mounted. The prototype observes the exact block-level information that the disk controller would, and is thus functionally identical to a hardware prototype.

1.7 Contributions

The key contributions of this dissertation are as follows:

- The formulation and design of techniques by which a block-based storage system can infer various pieces of semantic information underneath modern file systems, despite the uncertainty caused due to file system asynchrony.

- The design, implementation, and evaluation of a variety of prototype case studies that demonstrate that semantic disks can significantly improve storage systems along various axes such as availability, security, and performance. The case studies also serve to explore the costs of providing the functionality within a semantic disk, in terms of performance and implementation complexity.

- The design of various *conservative* techniques to circumvent fundamental uncertainty in semantic inference. These techniques ensure that an SDS functionality can still provide correctness guarantees (such as in secure delete) despite being based on inherently inaccurate information.

- The identification of various dynamic file system properties that impact the effectiveness and accuracy of semantic inference and exploration of the limits of semantic inference under such properties.

- The formulation of a logic framework and proof system for reasoning about file systems and semantic disks, and demonstration of the effectiveness of the framework in representing file system properties and proving high-level correctness guarantees about file systems, and their interaction with semantic disks.

1.8 Outline

The rest of this dissertation is organized as follows. Chapter 2 provides background information on modern storage systems and file systems. In Chapter 3, we present an overview of the basic SDS approach and compare it with alternative approaches to addressing the information gap in storage. We describe the techniques that an SDS uses to track semantic information in Chapter 4, and present some simple case studies that exploit this information to improve functionality in Chapter 5. In Chapter 6, we present D-GRAID, the system that improves storage availability by exploiting semantic knowledge. Chapter 7 presents FADED, a secure deleting disk that shreds deleted data by inferring logical deletes. In Chapter 8, we extend our semantic disk technology to work underneath database systems. Chapter 9 presents our formal logic framework to reason about file systems and semantic disks. We discuss related work in Chapter 10, and we conclude and discuss future directions in Chapter 11.

Chapter 2

Background

In this chapter, we provide background information on modern storage systems and file systems. First, we describe the range of functionality that modern storage systems already provide, hinting at the complexity and the extent of low-level knowledge available in such storage systems. We then provide a background on the various file systems we consider for semantic inference.

2.1 Modern Storage Systems

Storage systems today constitute a huge industry, ranging from desktop hard drives to mid-range and high-end storage servers. Due to availability of cheap processing power and memory, the level of intelligence in storage has been on the rise. This is exemplified by the high-end storage systems available today; for example, the EMC Symmetrix storage system has hundreds of processors and about 256 GB of RAM [29]. Storage systems use this processing power they have to implement a wide range of optimizations transparent to higher layers of the system. In this section, we briefly describe some common examples of functionality that modern storage systems provide.

2.1.1 RAID layout

A very common feature available in most enterprise storage systems today is tolerance to a small number of disk failures. By spreading data across multiple disks with some redundant information, the storage system can tolerate a fixed number of failures without losing data [77]. Spreading data across multiple disks also improves performance, since the storage system can exploit parallelism across several

disks.

There are various levels of RAID, each varying in the exact layout strategy employed for redundancy. The two most common RAID levels are RAID-1 (*i.e.*, mirroring), and RAID-5 (*i.e.*, parity). In RAID-1, each disk block is mirrored across two disks, resulting in a 100% space overhead. RAID-5 achieves better space efficiency by computing a parity block for each row of data blocks across the various disks. For example, the ith parity block will be the XOR of the ith block in each of the disks. A common variant of RAID-1 extended for the case of more than two disks is RAID-10, where data is striped across mirrored pairs; for example, in an 8-disk RAID-10 system, there are 4 mirrored pairs, and data is striped across the pairs.

Given the complex performance and space trade-offs between the various RAID schemes, certain storage systems adaptively choose the ideal RAID level for a given workload, and migrate data across RAID levels based on access patterns [117]. Such optimizations result in dynamic mappings between logical and physical blocks that the storage system explicitly keeps track of.

2.1.2 NVRAM buffering

Storage systems today perform buffering of writes in non-volatile RAM for better performance [29]. When a write request arrives from the host system, the storage system simply writes it to NVRAM and returns success on the write; the actual propagation of the write to disk occurs at a later time when the NVRAM gets filled or after a certain delay threshold. This enables better queueing of writes to disk resulting in better scheduling performance.

2.1.3 Block migration

Traditional RAID follows a systematic pattern in choosing which disk (and which physical block) a given logical block is mapped to. For example, in a N disk system, logical block k will be mapped to the disk k modulo N. However, storage systems today sometimes break this mapping by migrating blocks across disks for reasons such as load balancing to eliminate specific hotspots, avoiding bad blocks on disk, etc. To keep track of the logical-to-physical mapping, such storage systems maintain an internal indirection table that tracks the location of the blocks that have migrated [117].

2.1.4 Summary

While storage systems have been getting smarter at providing a wide range of optimizations as described above, higher layers of the system still view them as simple disks, as they were a few decades ago. The linear address space abstraction provided by SCSI hides all the complexity of modern storage. As a result, modern storage systems exclusively have a rich amount of low-level knowledge (*e.g.*, number of disks in RAID array, RAID layout strategy, logical-to-physical block mapping, etc.) and control (*e.g.*, choosing which blocks are buffered in NVRAM).

2.2 File System Background

Techniques for tracking semantic information from within the storage system are dependent on the characteristics of the file system above. We therefore study the range of techniques required for such inference by experimenting underneath three different file systems: ext2, ext3, and VFAT. Given that ext2 has two modes of operation (synchronous and asynchronous modes) and ext3 has three modes (writeback, ordered, and data journaling modes), all with different update behaviors, we believe these form a rich set of file systems. In Chapter 7, we also report on some limited experience underneath the Windows NTFS file system.

In this section, we provide some background information on the various file systems we study. We discuss both key on-disk data structures and the update behavior.

2.2.1 Common properties

We begin with some properties common to all the file systems we consider, from the viewpoint of tracking semantic information. At a basic level, all file systems track at least three kinds of on-disk metadata: a structure that tracks allocation of blocks (*e.g.*, bitmap, freelist), index structures (*e.g.*, inodes) that map each logical file to groups of blocks, and directories that map human-readable path names to logical files.

File systems manage data across two domains: main memory and disk. At any given time, the file system caches a subset of blocks in memory. Before modifying a block, the block is read from memory, and is written back to disk some time after the modification.

A common aspect of the update behavior of all modern file systems is *asynchrony*. When a data or metadata block is updated, the contents of the block is

not immediately flushed to disk, but instead, buffered in memory for a certain interval (*i.e.*, the *delayed write interval*). Blocks that have been "dirty" longer than the delayed write interval are periodically flushed to disk. The order in which such delayed writes are committed can be potentially arbitrary, although certain file systems enforce ordering constraints [32].

2.2.2 Linux ext2

The ext2 file system is an intellectual descendant of the Berkeley Fast File System (FFS) [65]. The disk is split into a set of *block groups*, akin to cylinder groups in FFS, each of which contains inode and data blocks. The allocation status (live or dead) of data blocks is tracked through *bitmap blocks*. Most information about a file, including size and block pointers, is found in the file's inode. To accommodate large files, a few pointers in the inode point to *indirect blocks*, which in turn contain block pointers.

While committing delayed writes, ext2 enforces no ordering whatsoever; crash recovery therefore requires running a tool like *fsck* to restore metadata integrity (data inconsistency may still persist). Ext2 also has a synchronous mode of operation where metadata updates are synchronously flushed to disk, similar to early FFS [65].

2.2.3 Linux ext3

The Linux ext3 file system is a *journaling* file system that evolved from ext2, and uses the same basic on-disk structures. Ext3 ensures metadata consistency by write-ahead logging of metadata updates, thus avoiding the need to perform an fsck-like scan after a crash. Ext3 employs a coarse-grained model of transactions; all operations performed during a certain *epoch* are grouped into a single transaction. When ext3 decides to commit the transaction, it takes an in-memory copy-on-write snapshot of dirty metadata blocks that belonged to that transaction; subsequent updates to any of those metadata blocks result in a new in-memory copy, and go into the next transaction.

Ext3 supports three modes of operation. In *ordered data* mode, ext3 ensures that before a transaction commits, all data blocks dirtied in that transaction are written to disk. In *data journaling* mode, ext3 journals data blocks together with metadata. Both these modes ensure data integrity after a crash. The third mode, *data writeback*, does not order data writes; data integrity is not guaranteed in this mode.

2.2.4 VFAT

The VFAT file system descends from the world of PC operating systems. In our work, we consider the Linux implementation of VFAT. VFAT operations are centered around the *file allocation table (FAT)*, which contains an entry for each allocatable block in the file system. These entries are used to locate the blocks of a file, in a linked-list fashion. For example, if a file's first block is at address b, one can look in entry b of the FAT to find the next block of the file, and so forth. An entry can also hold an end-of-file marker or a setting that indicates the block is free. Unlike UNIX file systems, where most information about a file is found in its inode, a VFAT file system spreads this information across the FAT itself and the directory entries; the FAT is used to track which blocks belong to the file, whereas the directory entry contains information like size, type information and a pointer to the start block of the file. Similar to ext2, VFAT does not preserve any ordering in its delayed updates.

2.2.5 Windows NTFS

NTFS is the default file system used in Windows today. Like ext3, NTFS is a journaling file system. The journaling mode that NTFS employs is metadata journaling where only metadata operations are journaled; there is no equivalent of the data journaling mode of ext3.

The fundamental piece of metadata in NTFS is the Master File Table (MFT); each record in the MFT contains information about a unique file. Every piece of metadata in NTFS is treated as a regular file; file 0 is the MFT itself, file 2 is the recovery log (similar to the ext3 journal), and so on. The allocation status of all blocks in the volume is maintained in a file called the cluster bitmap, which is similar to the block bitmap tracked by ext2. In addition, NTFS contains extensible metadata in the form of attribute lists for each logical file.

Chapter 3

Semantic Disks: An Overview

Before getting into the details of how a semantically-smart disk system extracts and uses knowledge about the file system, we first present an overview of our approach, and discuss its merits and demerits in comparison to alternative approaches.

3.1 Basic Approach

The basic idea presented in this thesis is to build a storage system that understands higher-level semantic information about the file system (or a database system), but does so without requiring any changes to the existing block-based SCSI-like interface to storage. Such a *semantically-smart disk system* infers semantic knowledge about the file system by carefully observing the block-level read and write traffic, and combining this observation with a minimal amount of embedded knowledge about the file system. Since the embedded knowledge and the techniques are somewhat specific to the file system running on top, there is a level of dependency created between the file system and the storage system.

3.1.1 Benefits and concerns

Implementing new functionality in a semantically-smart disk system has the key benefit of enabling wide-scale deployment underneath an unmodified SCSI interface without any OS modification, thus working smoothly with existing file systems and software base. Although there is some desire to evolve the interface between file systems and storage [36], the reality is that current interfaces will likely survive much longer than anticipated. As Bill Joy once said, “Systems may come and go, but protocols live forever”. Similar to modern processors that innovate beneath

unchanged instruction sets, a semantic disk-level implementation is non-intrusive on existing infrastructure. An individual storage vendor can decide to provide a new functionality and can just sell the enhanced storage system without having to interact with other layers of the system or achieve industry consensus.

However, because semantically-smart storage systems require more detailed knowledge of the file system that is using them, a few concerns arise on the commercial feasibility of such systems. We consider three main concerns.

The first concern that arises is that placing semantic knowledge within the disk system ties the disk system too intimately to the file system above. For example, if the on-disk structure of the file system changes, the storage system may have to change as well. We believe this issue is not likely to be problematic. On-disk formats evolve slowly, for reasons of backwards compatibility. For example, the basic structure of FFS-based file systems has not changed since its introduction in 1984, a period of almost twenty years [65]; the Linux ext2 file system, introduced in roughly 1994, has had the exact same layout for its lifetime. Finally, the ext3 journaling file system [111] is backwards compatible with ext2 on-disk layout and the new extensions to the FreeBSD file system [27] are backwards compatible as well. We also have evidence that storage vendors are already willing to maintain and support software specific to a file system; for example, the EMC Symmetrix storage system [29] comes with client-side software that can understand the format of most common file systems. Similarly, storage systems from EMC can recognize an Oracle data block and provide an extra checksum to assure that a block write (comprised of multiple sector writes) reaches disk atomically [15], again illustrating that storage vendors are already willing to put in minimal amount of knowledge about specific higher layers.

The second concern is that the storage system needs semantic knowledge for each file system with which it interacts. Fortunately, there are not a large number of file systems that would need to be supported to cover a large fraction of the usage population. If such a semantic storage system is used with a file system that it does not support, the storage system could detect it and turn off its special functionality (*e.g.*, revert revert to a normal storage system). Such detection can be done by simple techniques such as observing the file system identifier in the partition table or looking for a magic number in the super block, similar to how the host operating system would detect the file system that a disk contains when it starts up.

One final concern that arises is that too much processing will be required within the disk system. We do not believe this to be a major issue, because of the general trend of increasing disk system intelligence [1, 88]; as processing power increases, disk systems are likely to contain substantial computational abilities. Indeed, mod-

ern storage arrays already exhibit the fruits of Moore's Law; for example, the EMC Symmetrix storage server can be configured with 100s processors and upto 256 GB of RAM [29].

3.2 Alternative Approaches

In this section, we discuss alternative approaches to addressing the problem of information divide between the file system and the storage system, and compare them to the SDS approach. First, we describe an approach of explicitly changing the interface to storage to convey richer information. Second, we look at other implicit forms of information extraction besides the SDS approach.

3.2.1 Explicit

The explicit approach involves changing the interface between file systems and storage, to convey richer information across both layers. For instance, the storage system could expose low-level information about its layout (*e.g.*, failure boundaries) to the file system [24], and then the file system could utilize this knowledge for better layout. Alternatively, the file system could explicitly communicate semantic information to the storage system (*e.g.*, notify the storage system on logical operations such as deletes), which can then be used by the storage system to implement new functionality. These techniques, while being conceivably less complex than our approach, have a few major drawbacks. First, changing the interface to storage raises legacy issues in terms of the huge existing investment on the block-based interface. Second, adopting a new interface to storage requires broad industry consensus, which is often extremely slow to occur. Finally, a demand for such a new interface often requires industry-wide agreement on the clear benefits of the interface, which is difficult to achieve without actually having the interface deployed; this chicken-and-egg problem is another key weakness of this approach.

Although the explicit approach has problems with regard to practicality of immediate deployment, it also has some benefits. The first benefit is along the axis of system complexity; the explicit approach conceivably results in simpler systems than those built on semantic inference. Second, the semantic disk approach incurs performance costs in inferring file system information; explicit communication of the information can be potentially more efficient. In Chapter 7, we quantify the costs of the SDS approach relative to the explicit approach, in the context of a specific case study.

3.2.2 Implicit

Implicit approaches are intended to address the bootstrapping issue with the explicit approach; the implicit approach requires no changes to the existing storage interface, and involves inference of additional information while adhering to existing interfaces. Semantically-smart disks are an example of the implicit approach. In this subsection, we discuss other alternative implicit approaches besides the SDS approach.

The first alternative is to have the file system infer information about the storage system, in contrast to the SDS approach where the storage system infers information about the file system. The main problem with this approach is the inadequacy of implicit observation channels that the file system can use to infer information. Implicit techniques rely on careful observations on an implicit channel, and the efficacy of the inference depends on the richness of this implicit channel. From the viewpoint of the storage system, this channel is rich because the file system has to inevitably store data in the storage system; the contents of the data written by the file system constitutes a rich information channel for the storage system, and semantic disks use this channel to make inferences. In contrast, the information channel from the viewpoint of the file system is very fragile. All that the file system can observe are specific timing characteristics of certain requests [25, 120]. This limited channel is often insufficient given the range of optimizations modern storage systems perform.

Another implicit approach that is pertinent is a *black-box* approach, where the storage system simply uses the logical block stream to make inferences such as correlating related blocks based on observing sequences of blocks that are accessed together [61]. This approach has the advantage of requiring no information about the file system. The main disadvantage with such a black-box approach, however, is that its applicability is limited to a very small class of functionality; for example, it cannot be used for implementing functionality where correctness is paramount. Since the black-box approach is fundamentally heuristic and approximate, such techniques cannot provide any guarantees in terms of the accuracy of information. As we show in the rest of this thesis, the SDS approach enables implementing more aggressive classes of functionality that utilize semantic knowledge in ways that can impact correctness. Further, such black-box techniques are fragile to concurrent interleavings of independent streams.

3.3 Evaluation Methodology

We evaluate the techniques for semantic inference and the various case studies utilizing the information, through prototype implementations. To prototype an SDS, we employ a software-based infrastructure. Our implementation inserts a pseudo-device driver into the kernel, which is able to interpose on traffic between the file system and the disk. Similar to a software RAID, our prototype appears to file systems above as a device upon which a file system can be mounted.

The primary advantage of our prototype is that it observes the same block-level information and traffic stream as an actual SDS, with no changes to the file system above; thus, conceptually, transferring the functionality from the pseudo driver to an actual hardware prototype is straightforward.

However, our current infrastructure differs in three important ways from a true SDS. First, and most importantly, our prototype does not have direct access to low-level drive internals (*e.g.*, current head position); using such information is thus made more difficult. Second, because the SDS runs on the same system as the host OS, there may be interference due to competition for resources; thus, the performance overheads incurred by our prototype could be pessimistic estimates of the actual overheads. Third, the performance characteristics of the microprocessor and memory system may be different than an actual SDS; however, high-end storage arrays already have significant processing power, and this processing capacity will likely trickle down into lower-end storage systems.

Platform

We have experimented with our prototype SDS in the Linux 2.2 and Linux 2.4 operating systems, underneath of the ext2, ext3, and VFAT file systems, respectively. We have also had limited experience underneath the Windows NTFS file system where we interpose underneath a virtual machine running Windows XP. Some of the initial case studies work only underneath the ext2 file system, while later case studies such as D-GRAID and FADED operate underneath other file systems. Most experiments in this paper are performed on a processor that is “slow” by modern standards, a 550 MHz Pentium III processor, with 5 10K-RPM IBM 9LZX disks. In some experiments, we employ a “fast” system, comprised of a 2.6 GHz Pentium IV, to gauge the effects of technology trends.

Chapter 4

Acquiring Semantic Knowledge

> "*To know that we know what we know, and that we do not know what we do not know, that is true knowledge.*" Confucius

In this chapter, we discuss how a semantically-smart disk system tracks file system-specific semantic information underneath a block-based interface. We identify two classes of semantic information that are pertinent to the storage system: *static* and *dynamic*, and present various techniques to track these two classes of information. We then discuss how file system asynchrony significantly limits the extent and accuracy of semantic information that can be tracked within an SDS. Finally, we evaluate the costs of our various techniques for semantic inference.

4.1 Static Information

The basic piece of information that an SDS requires is knowledge about the key on-disk data structures used by the file system. Such information is *static* because it does not change for a given version of a file system. The structure of an inode in the Linux ext2 file system and the specific fields in the superblock are examples of static information. It is important to note that static information does not include the entire realm of information about on-disk layout. Certain file systems could store file data within inodes for small files; in such cases, the format of the inode is not strictly static. However, the file system in this case has a way of determining whether a given inode stores data inline, perhaps based on some specific field in the inode. This field, together with the other static fields in the inode alone would constitute static information for such a file system.

Given the immutability of static knowledge for a given file system, imparting this information to the SDS is quite straight-forward; we directly embed this infor-

mation into the SDS. Instead of hardcoding this information into the SDS, we could also communicate it through a separate administrative channel in an offline fashion; most modern storage systems have such separate administrative channels [29]. Another alternative would be to encapsulate this information in an add-on hardware card that plugs into the storage system. In summary, all these techniques assume "white-box" knowledge of this static information.

Another approach to impart static information would be to automatically fingerprint the on-disk structures of the file system using *gray box* techniques [103]. Such techniques can sometimes enable automatic detection of minor changes to on-disk formats. However, for the remainder of this thesis, we will assume that the SDS has static information about on-disk format through one of the white-box techniques mentioned above.

Embedding static information about the file system into the SDS raises the concern of tying the storage system to the specific file system above. However, as discussed in Chapter 3, given the stability of on-disk formats and the relatively small number of file systems in popular use, creating such a dependency is reasonable; current storage systems already have instances of such dependency to file systems [29], or even to databases [15].

4.2 Dynamic Information

While static information about the on-disk structures of the file system is crucial in a semantic disk, static information alone is not sufficient to provide a large class of useful functionality. To enable useful functionality, the SDS requires dynamic information about the operation of the file system. Dynamic information involves properties of blocks that keep constantly changing at the file system, or higher level operations that the file system performs. For example, knowing from within the SDS whether a block is live or dead (*i.e.*, whether it contains valid data or it is free) is one example of dynamic information, because it constantly changes as blocks are allocated or deleted within the file system.

In this section, we describe various techniques to infer different kinds of dynamic information within an SDS. Many of the techniques involve exploiting the static knowledge about on-disk structures that the SDS already has, to carefully watch updates to those structures; for example, the SDS snoops on traffic to inode blocks and data bitmap blocks. In many cases, the SDS requires functionality to identify how a block has changed, in order to correlate those changes to higher level file system activity that could have led to the changes. For example, the SDS can infer that a data block has been allocated within the file system when it observes

the corresponding bit change from 0 to 1 in the data bitmap.

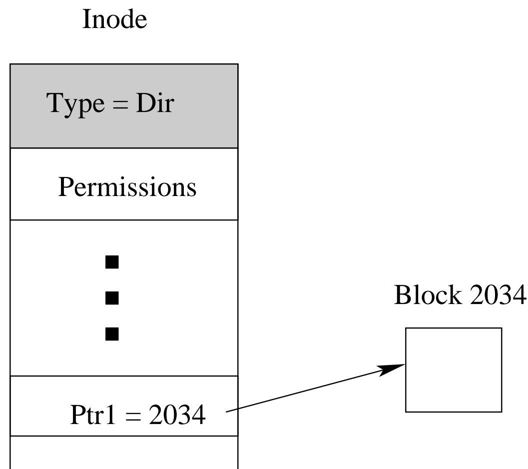
The SDS tracks changes to blocks via block differencing. Change detection is potentially one of the expensive operations within an SDS for two reasons. First, to compare the current block with the last version of the block, the SDS may need to fetch the old version of the block from disk; however, to avoid this overhead, a cache of blocks is employed. Second, the comparison itself may be expensive: to find the location of a difference, each byte in the new block must be compared with the corresponding byte in the old block. This cost can be reduced for certain meta-data blocks by skipping portions of the block that are uninteresting; for example, when differencing an inode block, the SDS might be able to skip scanning through inodes that are marked deleted. We quantify these costs in Section 4.4.

We describe below techniques for the two most common inferences made within an SDS: *classification* (*i.e.*, tracking the type of a block) and *association* (*e.g.*, which inode a block is associated with). We then discuss how the SDS can track higher level file system activity such as file creations and deletions through the process of *operation inferencing*. An SDS can use some or all of these techniques to implement its desired functionality.

For simplicity, we first discuss the techniques based on the assumption that the file system above is *synchronous* in its meta-data updates. The BSD FFS file system and Linux ext2 in synchronous mode fit into this assumption. In the next section, we discuss the implications of asynchrony and how it can complicate the techniques for tracking dynamic information.

4.2.1 Classification

The type of a block is one of the most useful pieces of information that an SDS can exploit. For example, if it identifies that a block is an important metadata block, it can replicate the block to a high degree for better reliability. Knowing block type also enables more sophisticated inferences to be built on top; for example, if a block is known to be an indirect block, the pointers within the block can be interpreted meaningfully. Block type can be determined through direct and indirect classification. With *direct classification*, blocks are easily identified by their location on disk. With *indirect classification*, blocks are identified only with additional information; for example, to identify directory data or indirect blocks, the corresponding inode must also be examined.



Inference: Block 2034 is directory

Figure 4.1: **Indirect Classification.**

Direct classification

Direct classification is the simplest and most efficient form of on-line block identification for an SDS, and the only information it requires is the static knowledge about the file system. The SDS determines the type of such *statically-typed* blocks by performing a simple bounds check to calculate into which set of block ranges the particular block falls. In an FFS-like file system, the superblock, bitmaps, and inode blocks are identified using this technique.

Indirect classification

Indirect classification is required when the type of a block can vary dynamically and thus simple direct classification cannot precisely determine the type of block. For example, in FFS-like file systems such as Linux ext2 or ext3, a given block in the data region of the file system can be either a data block, a directory block, or an indirect pointer block (*e.g.*, a single, double, or triple indirect block). Indirect classification is used in such cases to determine the precise type of the block. To illustrate these concepts we focus on how directory data is differentiated from file data; the steps for identifying indirect blocks versus pure data are similar, and we describe them briefly.

Identifying directory data:

The basic challenge in identifying whether a data block belongs to a file or a directory is to track down the inode that points to this data and check whether its type is a file or a directory. To perform this tracking, the SDS *snoops* on all inode traffic to and from the disk: when a directory inode is observed, the corresponding data block numbers are inserted into a `dir_blocks` hash table. The SDS removes data blocks from the hash table by observing when those blocks are freed (*e.g.*, by using block differencing on the bitmaps). When the SDS must later identify a block as a file or directory block, its presence in this table indicates that it is directory data.

One problem with the above approach is that the SDS may perform excess work if it obliviously inserts all data blocks into the hash table whenever a directory inode is read and written since this inode may have recently passed through the SDS, already causing the hash table to be updated. Therefore, to optimize performance, the SDS can infer whether or not a block has been added (or modified or deleted) since the last time this directory inode was observed, and thus ensure that only those blocks are added to (or deleted from) the hash table. This process of *operation inferencing* is described in detail in Section 4.2.3.

Identifying indirect blocks:

The process for identifying indirect blocks is almost identical to that for identifying directory data blocks. In this case, the SDS tracks new indirect block pointers in all inodes being read and written. By maintaining a hash table `indirect_blocks` of all single, double, and triple indirect block addresses, an SDS can determine if a block in the data region is an indirect block.

4.2.2 Association

Association refers to the ability of the SDS to re-establish the semantic relationship between various blocks. Various types of associations are possible within an SDS. For example, the SDS could associate an inode with the directory that contains the inode; thus, the SDS can infer the pathname corresponding to a certain inode, and consequently, the pathname of the file to which a given block belongs. This information can be useful if the SDS decides to perform optimizations specific to the type of a file as inferred from its pathname.

The most useful association is to connect data blocks with their inodes; for example, in Chapter 6, we show that the SDS can employ smarter layout of data in RAID systems for better availability, if it knows that a certain set of blocks belong to a single file. Association can be achieved with a simple but space-consuming approach. Similar to indirect classification, the SDS snoops on all inode traffic and inserts the data pointers into an `address_to_inode` hash table. One concern

about such a table is size; for accurate association, the table grows in proportion to the number of unique data blocks that have been read or written to the storage system since the system booted. However, if approximate information is tolerated by the SDS, the size of this table can be bounded.

4.2.3 Operation inferencing

Block classification and association provide the SDS with an efficient way for identifying special kinds of blocks; however, operation inferencing is necessary to understand the semantic meaning of the changes observed in those blocks. We now outline how an SDS can identify file system operations by observing certain key changes.

For illustrative purposes, in this section we examine how the SDS can infer file create and delete operations. The discussion below is specific to ext2, although similar techniques can be applied to other file systems.

File Creates and Deletes

There are two steps in identifying file creates and deletes. The first is the actual detection of a create or delete; the second is determining the inode that has been affected. We describe three different detection mechanisms and the corresponding logic for determining the associated inode.

The first detection mechanism involves the inode block itself. Whenever an inode block is written, the SDS examines it to determine if an inode has been created or deleted. A valid inode has a non-zero modification time and a zero deletion time. Therefore, whenever the modification time changes from zero to non-zero or the deletion time changes from non-zero to zero, it means the corresponding inode was newly made valid, *i.e.*, created. Similarly, a reverse change indicates a newly freed inode, *i.e.*, a deleted file. The inode number is calculated using the physical position of the inode on disk (on-disk inodes do not contain inode numbers).

The second detection mechanism involves the inode bitmap block. Whenever a new bit is set in the inode bitmap, it indicates that a new file has been created corresponding to the inode number represented by the bit position. Similarly, a newly reset bit indicates a deleted file.

The update of a directory block is a third indication of a newly created or deleted file. When a directory data block is written, the SDS examines the block for changes from the previous version. If a new directory entry (`dentry`) has been added, the name and inode number of the new file can be obtained from the

`dentry`; in the case of a removed `dentry`, the old contents of the `dentry` contain the name and inode number of the deleted file.

Given that any of these three changes indicate a newly created or deleted file, the choice of the appropriate mechanism (or combinations thereof) depends on the functionality being implemented in the SDS. For example, if the name of the newly created or deleted file must be known, the directory block-based solution would be the most appropriate, since it would readily convey the exact file name that has been added or removed.

Other File System Operations

The general technique of inferring logical operations by observing changes to blocks from their old versions can help detect other file system operations as well. We note that in some cases, for a conclusive inference on a specific logical operation, the SDS must observe correlated changes in multiple meta-data blocks. For example, the SDS can infer that a file has been renamed when it observes a change to a directory block entry such that the name changes but the inode number stays the same; note that the version number within the inode must stay the same as well. Similarly, to distinguish between the creation of a hard link and a normal file, both the directory entry and the file's inode must be examined.

4.2.4 Accuracy of inference

The above techniques assume that the file system above is synchronous, *i.e.*, reflects all metadata updates to disk immediately. Under such a file system, tracking dynamic information is accurate as long as the information pertains to on-disk state. However, sometimes an SDS may need to implicitly track in-memory state within the file system, *e.g.*, the contents of the file system's buffer cache [9]; such inference will be uncertain despite a synchronous file system.

In the next subsection, we show that under most modern file systems that do not exhibit synchronous metadata updates, even tracking dynamic information about on-disk state becomes uncertain and inaccurate.

4.3 Dealing with Asynchrony

The techniques in the previous section for classification, association, and operation inferencing ignored one basic trait of modern file systems: *asynchrony*. Most modern file systems exhibit asynchrony in their updates to data and metadata. File

systems typically delay updates to disk, and often reorder writes to disk for better scheduling. Such asynchrony significantly complicates techniques for tracking dynamic information.

4.3.1 Indirect classification

Indirect classification depends on the SDS observing the inode containing a block, before observing the actual block write. Specifically, when a data block is not present in the `indirect_blocks` hash table, the SDS infers that the data corresponds to a regular file block; however, in some cases, the concerned inode may not have yet been seen by the SDS and as a result is not yet in the hash table. Such a situation may occur when a new large file is created, or new blocks are allocated to existing files; if the file system does not guarantee that inode blocks are written before data blocks, the SDS may incorrectly classify newly written data blocks. This problem does not occur when classifying data blocks that are read. In the case of reads, the file system must read the corresponding inode block before the data block (to find the data block number); thus, the SDS will see the inode first and correctly identify subsequent data blocks.

To solve this problem, the SDS needs to buffer writes until the time when the classification can be made; this *deferred classification* occurs when the corresponding inode is written to disk or when the data block is freed, as can be inferred by monitoring data bitmap traffic.

We now move to a more subtle problem in indirect classification due to asynchrony. Let us consider the example of detecting an indirect pointer block. In order to detect a block as an indirect block, the SDS should observe that a certain inode's indirect pointer field contains a pointer to the address of the given block. More formally, to identify an indirect block B, the semantic disk must look for the inode that has block B in its indirect pointer field. Thus, when the relevant inode block I_B is written to disk, the disk infers that B is an indirect block and records this information. When it later observes block B written, it uses this information to classify and treat the block as an indirect block. However, due to the delayed write and reordering behavior of the file system, it is possible that in the time between the disk writes of I_B and B, block B was freed from the original inode and was reallocated to another inode with a different type, *i.e.*, as a normal data block. The disk does not know this since the operations took place in memory and were not reflected to disk. Thus, the inference made by the semantic disk on the block type could be wrong due to the inherent staleness of the information tracked.

4.3.2 Association

Similar uncertainty arises in tracking other forms of dynamic information such as association as well. For example, associating a block with a file is uncertain because between observing an inode owning a block and observing the actual block, the block could have been deleted from the old inode and reassigned to the new inode.

4.3.3 Operation inferencing

Techniques for operation inferencing are impacted by asynchrony as well. For example, if the SDS relies on the inode bitmap differencing technique to identify file creates and deletes (as described in Section 4.2.3), it would miss a create quickly followed by a subsequent delete of the same inode, because the SDS may not observe a change in the bitmap if the two operations are grouped due to a delayed write in the file system. One way by which the SDS can still track this information partially is by looking for a change in the version number of a valid inode, which indicates that a delete followed by a create occurred.

Asynchrony within the file system thus effectively results in suppression of information from the SDS, and imposes hard limits on the accuracy of what can be known within an SDS.

4.3.4 Impact of uncertainty

While asynchrony imposes fundamental limits to the accuracy of information that can be tracked in an SDS, it does not preclude utilizing such information to provide interesting functionality. Certain kinds of functionality are easily amenable to inaccurate information; they utilize semantic information in a way that does not impact correctness. For example, the SDS could utilize information on block type to selectively cache only metadata blocks in non-volatile RAM. In this case, a misclassification of a directory block as data block will at worst lead to sub-optimal utilization of NVRAM cache space, which is a performance issue, but there is no concern in terms of correctness. Another example is a functionality that utilizes semantic information to track the contents of the cache in the host system, so that the disk system can avoid caching the same blocks [9]. Such optimizations are thus inherently tolerant to inaccuracy.

However, a more challenging class of functionality is where semantic information is used in a way that can directly affect correctness, or the set of guarantees the system can provide. In Chapters 6 and 7, we describe two such case studies that have different levels of stringency in the degree to which they are correctness-sensitive. The techniques for circumventing inaccuracy are specific to the func-

	Indirect Classification		Block-Inode Association		Operation Inferencing	
	Sync	**Async**	**Sync**	**Async**	**Sync**	**Async**
$Create_0$	1.7	3.2	1.9	3.3	33.9	3.2
$Create_{32}$	60.6	3.8	324.4	16.4	279.7	3.8
$Delete_0$	4.3	3.6	6.7	3.9	50.9	3.6
$Delete_{32}$	37.8	6.9	80.1	28.8	91.0	6.9
Mkdir	56.3	8.6	63.6	11.1	231.9	8.6
Rmdir	49.9	106.2	57.8	108.5	289.4	106.2

Table 4.1: **SDS Time Overheads.** *The table breaks down the costs of indirect classification, block-inode association, and operation inferencing. Different microbenchmarks (one per row) stress various aspects of each action. The Create benchmark creates 1000 files, of size 0 or 32 KB, and the Delete benchmark similarly deletes 1000 such files. The Mkdir and Rmdir benchmarks create or remove 1000 directories, respectively. Each result presents the average overhead per operation in μs (i.e., how much extra time the SDS takes to perform classification, association, or inferencing). The experiments were run upon the "slow" system with the IBM 9LZX disk, with Linux ext2 mounted synchronously (***Sync***) or asynchronously (***Async***).*

tionality implemented in the SDS. We show through those case studies that it is indeed possible to build complex functionality that is correctness-sensitive, despite potentially inaccurate information.

4.4 Evaluation

In this section, we evaluate the costs of tracking dynamic information within an SDS. Specifically, we examine the time and space overheads associated with classification, association, and operation inferencing. As described in Chapter 3, because of our software-based prototype environment, the time overheads reported are pessimistic; a real hardware-based implementation would have a lower overhead due to the absence of CPU and memory contention.

4.4.1 Time overheads

Classification, association, and operation inferencing are potentially costly operations for an SDS. In this subsection, we employ a series of microbenchmarks to illustrate the various costs of these actions. The results of our experiments on an SDS underneath of Linux ext2 are presented in Table 4.1. For each action and mi-

crobenchmark we consider two cases. In the first case, the file system is mounted synchronously, ensuring that meta-data operations reach the SDS in order and thus allowing the SDS to guarantee correct classification with no additional effort; synchronous mounting in Linux ext2 is quite similar to traditional FFS in its handling of meta-data updates, in that metadata updates are written immediately to disk. In the second case, the file system is mounted asynchronously; in this case, correct classification and association cannot be guaranteed. The microbenchmarks perform basic file system operations, including file and directory creates and deletes, and we report the per-file or per-directory overhead of the action that is under test.

From our experiments, we make a number of observations. First, most operations tend to cost on the order of tens of microseconds per file or directory. Although some of the operations do require nearly 300 μs to complete, most of this cost is due to a per-block cost; for example, operation inferencing in synchronous mode with the Create_{32} workload takes roughly 280 μs, which corresponds to a 34 μs base cost (as seen in the Create_{0} workload) plus a cost of approximately 30 μs for each 4 KB block. Thus, although the costs rise as file size increases, the SDS incurs only a small per-block overhead compared to the actual disk writes, each of which may take some number of milliseconds to complete. Second, in most cases, the overheads when the ext2 file system is run in asynchronous mode are much lower than when run in synchronous mode. In asynchronous mode, numerous updates to meta-data blocks are batched and thus the costs of block differencing are amortized; in synchronous mode, each meta-data operation is reflected through to the disk system, incurring much higher overhead in the SDS. Third, we observe that in synchronous mode, classification is less expensive than association which is less expensive than inferencing. However, in asynchronous mode, the relative difference in time overheads between the various forms of dynamic inference is quite insignificant.

4.4.2 Space overheads

An SDS may require additional memory to perform classification, association, and operation inferencing; specifically, hash tables are required to track mappings between data blocks and inodes whereas caches are needed to implement efficient block differencing. We now quantify these memory overheads under a variety of workloads.

Table 4.2 presents the number of bytes used by each hash table to support classification, association, and operation inferencing. The sizes are the maximum reached during the run of a particular workload: NetNews [109], PostMark [55], and the modified Andrew benchmark [74]. For NetNews and PostMark, we vary

	Indirect Classification	Block-Inode Association	Operation Inferencing
$NetNews_{50}$	68.9 KB	1.19 MB	73.3 KB
$NetNews_{100}$	84.4 KB	1.59 MB	92.3 KB
$NetNews_{150}$	93.3 KB	1.91 MB	105.3 KB
$PostMark_{20}$	3.45 KB	452.6 KB	12.6 KB
$PostMark_{30}$	3.45 KB	660.7 KB	16.2 KB
$PostMark_{40}$	3.45 KB	936.4 KB	19.9 KB
Andrew	360 B	3.54 KB	1.34 KB

Table 4.2: **SDS Space Overheads.** *The table presents the space overheads of the structures used in performing classification, association, and operation inferencing, under three different workloads (NetNews, PostMark, and the modified Andrew benchmark). Two of the workloads (NetNews and PostMark) were run with different amounts of input, which correspond roughly to the number of "transactions" each generates (i.e., $NetNews_{50}$ implies 50,000 transactions were run). Each number in the table represents the maximum number of bytes stored in the requisite hash table during the benchmark run (each hash entry is 12 bytes in size). The experiment was run on the "slow" system with Linux ext2 in asynchronous mode on the IBM 9LZX disk.*

---

workload size, as described in the caption.

From the table, we see that the dominant memory overhead occurs in an SDS performing block-inode association. Whereas classification and operation inferencing require table sizes that are proportional to the number of unique meta-data blocks that pass through the SDS, association requires information on every unique data block that passes through. In the worst case, an entry is required for every data block on the disk, corresponding to 1 MB of memory for every 1 GB of disk space. Although the space costs of tracking association information are high, we believe they are not prohibitive. Further, if memory resources are scarce, the SDS can choose to either tolerate imperfect information (if possible), or swap portions of the table to disk.

In addition to the hash tables needed to perform classification, association, and operation inferencing, a cache of "old" data blocks is useful to perform block differencing effectively; recall that differencing is used to observe whether pointers have been allocated or freed from an inode or indirect block, to check whether time fields within an inode have changed, to detect bitwise changes in a bitmap, and to monitor directory data for file creations and deletions. The performance of the system is sensitive to the size of this cache; if the cache is too small, each difference calculation must first fetch the old version of the block from disk. To avoid the

extra I/O, the size of the cache must be roughly proportional to the active meta-data working set. For example, for the PostMark_{20} workload, we found that the SDS cache should contain approximately 650 4 KB blocks to hold the working set. When the cache is smaller, block differencing operations often go to disk to retrieve the older copy of the block, increasing run-time for the benchmark by roughly 20%.

4.5 Summary

In this chapter, we presented the basic techniques an SDS uses to track semantic information about the file system. Based on a minimal amount of static information about the on-disk layout of the file system, an SDS successively builds on this information to track more complex pieces of information.

We have also shown that there are fundamental limits to the extent and accuracy of semantic information that can be tracked in an SDS. Asynchrony in meta-data and data updates by the file system results in obfuscation of information at the semantic disk. In the next three chapters, we will explore how an SDS can utilize such potentially inaccurate information to implement new kinds of functionality that are precluded in today's storage systems.

Chapter 5

Exploiting Semantic Knowledge

> "*Tis not knowing much, but what is useful, that makes a wise man.*" Thomas Fuller

In this chapter, we describe a few simple case studies, each implementing new functionality in an SDS that would not be possible to implement within a drive or RAID without semantic knowledge. Some of the case studies presented in this chapter could be built into the file system proper; however, we present these case studies to illustrate the range of functionality that can be provided within an SDS. In the next two chapters, we describe more complex case studies, which represent entirely new pieces of functionality that cannot be implemented either within the storage system or within the file system today.

5.1 File System Model

Functionality implemented within a semantically-smart disk system is often based on certain assumptions about the dynamic behavior of the file system above. If the functionality is non-correctness-sensitive (*e.g.*, caching), and hence fundamentally tolerant to inaccuracy, it can afford to be lax in its assumptions about the dynamic file system behavior. However, for functionality that utilizes semantic information in a way that impacts correctness, the precise file system assumptions it is based on is crucial.

The first case study we consider in this chapter belongs to the non-correctness-critical category. The second case study is correctness-sensitive, and works underneath a very specific file system behavior. It assumes that the file system above is *synchronous*; in other words, the file system writes out metadata blocks syn-

chronously. The Linux ext2 file system in synchronous mount mode fits this assumption.

In the next two chapters, we will successively refine this simplistic file system model to consider more general and more detailed dynamic properties of the file system, to implement similar correctness-sensitive functionality. While D-GRAID (Chapter 6) considers a general asynchronous file system, FADED (Chapter 7) considers various refinements of asynchronous file systems that provide different kinds of guarantees on update behavior.

5.2 Semantic Caching

The first case study explores the use of semantic information in caching within an SDS. We examine how an SDS can use semantic knowledge to store important structures in non-volatile memory. Specifically, we exploit semantic knowledge to store in NVRAM the journal of Linux ext3, a journaling file system. Clearly, other variants of this basic idea are possible, such as caching only metadata blocks (*i.e.*, inodes, directories, etc.) in NVRAM, so this case study is only meant to be illustrative of the potential for semantically-aware NVRAM caching.

To implement the Journal Caching SDS (JC SDS), the SDS must recognize traffic to the journal and redirect it to the NVRAM. Doing so is straightforward, because the blocks allocated to the file system journal is part of static information about the file system; either a designated set of blocks are assigned for the journal, or a specific inode (number 2) points to the journal and the inode is pre-allocated to point to the set of journal blocks as part of file system initialization. Thus, by classifying and then caching data reads and writes to the journal file, the SDS can implement the desired functionality. For prototyping purposes, we treat a portion of volatile RAM as NVRAM.

Tables 5.1 shows the performance of the JC SDS. We can see that simple NVRAM caching of the journal in ext3 is quite effective at reducing run times, sometimes dramatically, by greatly reducing the time taken to write blocks to stable storage. An LRU-managed cache can also be effective in this case, but only when the cache is large enough to contain the working set. The worse performance of LRU_8 compared to default ext3 points to the overhead introduced by our pseudo-device driver layer. One of the main benefits of structural caching in NVRAM is that the size of the cached structures is known to the SDS and thus guarantees effective cache utilization. A hybrid may combine the best of both worlds, by storing important structures such as a journal or other meta-data in NVRAM, and managing the rest of available cache space in an LRU fashion.

	Create	Create+Sync
ext3	4.64	32.07
+LRU_8 SDS	5.91	11.96
+LRU_{100} SDS	2.39	3.35
+Journal Caching SDS	4.66	6.35

Table 5.1: **Journal Caching.** *The table shows the time in seconds to create 2000 32-KB files, under ext3 without an SDS, with an SDS that performs LRU NVRAM cache management using either 8 MB or 100 MB of cache, and with the Journal Caching SDS storing an 8 MB journal in NVRAM. The* **Create** *benchmark performs a single* `sync` *after all of the files have been created, whereas the* **Create+Sync** *benchmark performs a* `sync` *after each file creation, thus inducing a journaling-intensive workload. These experiments are run on the "slow" system running Linux 2.4 and utilizing IBM 9LZX disk.*

Semantic knowledge can also help efficient management of the main memory cache within the storage system. Simple LRU management of a disk cache is likely to duplicate the contents of the file system cache [119, 124], and thereby wastes memory in the storage system. This waste is particularly onerous in storage arrays, due to their large amounts of memory. In contrast, an SDS can use its understanding of the file system to cache blocks more intelligently. Specifically, the SDS can cache an exclusive set of blocks that are not cached within the file system, and thus avoid wasteful replication [9]. Also, caching within an SDS can be made more effective if the SDS identifies blocks that have been deleted, and removes them from the cache, thus freeing space for other live blocks.

Another related optimization that an SDS could do is smarter prefetching. Since the SDS has file awareness, it can make a better guess as to which block will next be read. For example, when the SDS observes a read to a directory block, it can prefetch the first few inode blocks pertaining to the inodes pointed to by the directory.

5.2.1 Tolerance to inaccuracy

The various optimizations discussed above for smarter caching within an SDS belong to a category of semantic disk functionality that is naturally tolerant to inaccurate information. Since none of the above optimizations utilize semantic information in a way that can impact correctness, they work underneath asynchronous file systems. Occasional inaccuracy will only make the optimizations marginally less effective, but with no correctness implications. Thus, these optimizations are

simpler to build within an SDS since they can ignore the possibility of inaccuracy in information. The next case study we explore has more stringent requirements on accuracy.

5.3 Journaling

Our next case study is more complex and demanding in its use of semantic information; we use semantic knowledge within the SDS to implement journaling underneath of an unsuspecting file system. As described in Chapter 2, journaling makes crash recovery both simple and efficient, by committing metadata operations as atomic transactions; journaling thus avoids expensive disk scans such as those found in FFS or FFS-like file systems [66]. The main difficulty addressed in this case study is how to track all of the necessary information in the disk itself, and to do so in an efficient manner.

We view the Journaling SDS as an extreme case which helps us to understand the amount of information we can obtain at the disk level. Unlike the smarter caching case study, the Journaling SDS requires a great deal of precise information about the file system.

In view of the extreme requirements on the accuracy of information required in a Journaling SDS, we implement the Journaling SDS underneath a synchronous file system (*i.e.*, Linux ext2 mounted in synchronous mode). By doing this simplification, we focus on the challenges of implementing complex functionality even assuming accurate information about the file system. In the next two chapters, we explore similarly complex case studies, but working under more general asynchronous file system behaviors.

5.3.1 Design and implementation

The fundamental difficulty in implementing journaling in an SDS arises from the fact that at the disk, transaction boundaries are blurred. For instance, when a file system does a file create, the file system knows that the inode block, the parent directory block, and the inode bitmap block are updated as part of the single logical create operation, and hence these block writes can be grouped into a single transaction in a straight-forward fashion. However, the SDS sees only a stream of meta-data writes, potentially containing interleaved logical file system operations. The challenge lies in identifying dependencies among those blocks and handling updates as atomic transactions.

The Journaling SDS maintains transactions at a coarser granularity than what a

Name	Type	Purpose
(a) CurrCache	Cache	Keeps needed blocks for current epoch
(b) NextCache	Cache	Keeps needed blocks for next epoch
(c) AddPtrs	Hash	Tracks added block ptrs in inodes
(d) AddBlks	Hash	Tracks added blocks in bitmaps
(e) AddInd	Hash	Tracks write of indirect blocks
(f) FreeBlks	Hash	Tracks freed inode pointers and bitmaps
(g) AddFile	Hash	Tracks files (inode, dentry, inode bitmap)
(h) DelFile	Hash	Tracks deleted files (inode, dentry, inode bitmap)
(i) Dirty	Hash	Tracks dirty blocks
(j) Fast	Hash	Performance optimization

Table 5.2: **Journaling SDS Structures.**

journaling file system would do. The basic approach is to buffer meta-data writes in memory within the SDS and write them to disk only when the in-memory state of the meta-data blocks together constitute a consistent meta-data state. This is logically equivalent to performing incremental in-memory fsck's on the current set of dirty meta-data blocks and then writing them to disk when the check succeeds. When the current set of dirty meta-data blocks form a consistent state, they can be treated as a single atomic transaction, thereby ensuring that the on-disk meta-data contents either remain at the previous (consistent) state or get fully updated with the current consistent state. One possible benefit of these coarse-grained transactions is that by batching commits, performance may be improved over more traditional journaling systems.

This coarse-grained approach to journaling brings out two important requirements. First, we need an efficient way of identifying and keeping track of dependencies among meta-data blocks and then checking if the current in-memory state of the blocks is consistent. Second, since the entire set of dirty meta-data blocks constitute a single transaction, we need to ensure that a continuous meta-data-intensive workload does not indefinitely prevent us from reaching a consistent state, thereby leading to an unbounded possible loss of data after a crash.

To meet these requirements, Journaling SDS maintains a cache of the old contents of meta-data blocks for efficiently identifying changes in them. Whenever it detects that an "interesting" change has occurred in one of the meta-data blocks, it records the change in one of multiple hash tables. Each hash table represents a single type of meta-data operation (like a newly added bit in the inode bitmap, a newly added directory entry, and so forth), and thus we have a separate hash table for every kind of meta-data operation that can occur. As we keep detecting changes to multiple meta-data blocks, we can *prune* these hash tables by canceling out changes that lead to a consistent state. For example, when a new file is created, the Journaling SDS sees a write of an inode-bitmap block, detects that a new inode bit is set and records the corresponding inode number in a suitable hash table. Subsequently, it gets a directory block write and records that a new directory entry pointing to a certain inode number has been added. When it finally gets the inode block write with the initialized inode, the state of those three blocks together would constitute a consistent state (assuming no other changes took place). To record the current state as consistent, the Journaling SDS removes the same inode number from the three hash tables where they were recorded. In a similar fashion, we manage hash tables for other meta-data operations. Given this technique for keeping track of changes, checking for consistency just involves checking that all the hash tables are empty, and thus is quite efficient.

To guarantee bounded loss of data after crash, the Journaling SDS has a mechanism of limiting the time that can elapse between two successive journal transaction commits. A journaling daemon wakes up periodically after a configurable interval and takes a copy-on-write snapshot of the current dirty blocks in the cache and the current hash table state. After this point, subsequent meta-data operations update a new copy of the cache and the hash tables, and therefore cannot introduce additional dependencies in the current *epoch*. From the commit decision-point until the actual commit, a meta-data operation is treated as part of the current epoch only if that operation contributes to resolving existing dependencies in the current epoch. This is slightly complicated by the coarse block-level granularity of the cache which can contain multiple meta-data operations, some of which need to be part of the current epoch and the rest, of the next epoch. Given that no additional dependencies are allowed into the current epoch, the maximum delay till the next commit is limited to the configured periodicity of the journaling daemon, together with the delayed write interval of the file system above the SDS.

A few idiosyncrasies of the ext2 file system made the implementation more complex. For instance, even in synchronous mode, ext2 does not write out the inode blocks when the size or reference count of an inode changes. This requires

the Journaling SDS to keep track of additional state and update them based on other operations seen. Similarly, when writing out inode blocks, ext2 does not preserve the temporal ordering of updates taking place on the inode. For instance, an inode block write that reflects a deleted inode may not reflect the creation of other inodes within the same block that happened in the past.

Table 5.2 displays the data structures used by the Journaling SDS. As one can see, there is a fair amount of complexity to tracking the needed information to detect consistency.

As mentioned above, the Journaling SDS implementation assumes that the file system has been mounted synchronously. To be robust, the SDS requires a way to verify that this assumption holds and to turn off journaling otherwise. Since the meta-data state written to disk by the Journaling SDS is consistent regardless of a synchronous or asynchronous mount, the only problem imposed by an asynchronous mount is that the SDS might miss some operations that were reversed (*e.g.*, a file create followed by a delete); this would lead to dependencies that are never resolved and indefinite delays in the journal transaction commit process. To avoid this problem, the Journaling SDS looks for a suspicious sequence of changes in meta-data blocks when only a single change is expected (*e.g.*, multiple inode bitmap bits change as part of a single write) and turns off journaling in such cases. As a fall-back, the Journaling SDS monitors elapsed time since the last commit; if dependencies prolong the commit by more than a certain time threshold, it suspects an asynchronous mount and aborts journaling. When it aborts journaling, the Journaling SDS relinquishes control over the "clean" flag used by the file system *fsck* program, thus forcing the file system to perform a consistency check on subsequent crashes. Thus, the Journaling SDS can never make consistency worse than traditional ext2, irrespective of whether it is mounted synchronously or asynchronously.

5.3.2 Evaluation

We evaluate both the correctness and performance of the Journaling SDS. To verify correctness of the implementation, we inserted crash points at specific places in our test workloads, and then ran our Journaling SDS recovery program that replays the log within the SDS to propagate committed transactions. After this, we ran fsck on the file system and verified that no inconsistencies were reported. We repeated this verification for numerous runs, and thus we are confident that our existing implementation provides at least as strong a consistency guarantee as ext2/fsck, but avoids the high cost of fsck-based recovery.

The performance of the Journaling SDS is summarized in Table 5.3. One interesting aspect to note in the performance of Journaling SDS is that despite requiring

	Create	Read	Delete
ext2 (2.2/sync)	63.9	0.32	20.8
ext2 (2.2/async)	0.28	0.32	0.03
ext3 (2.4)	0.47	0.13	0.26
ext2 (2.2/sync)+Journaling SDS	0.95	0.33	0.24

Table 5.3: **Journaling.** *The table shows the time to complete each phase of the LFS microbenchmark in seconds with 1000 32-KB files. Four different configurations are compared: ext2 on Linux 2.2 mounted synchronously, the same mounted asynchronously, the journaling ext3 under Linux 2.4, and the Journaling SDS under a synchronously mounted ext2 on Linux 2.2. This experiment took place on the "slow" system and the IBM 9LZX disk.*

SDS Infrastructure	
Initialization	395
Hash table and cache	2122
Direct classification	195
Indirect classification	75
Association	15
Operation inferencing	1105
Case Studies	
Journal Cache	305
Journaling SDS	2440

Table 5.4: **Code Complexity.** *The number of lines of code required to implement various aspects of an SDS are presented.*

the file system to be mounted synchronously, its performance is similar to the asynchronous versions of the file system. This effect is because of the fact that the Journaling SDS delays writing meta-data to disk due to its buffering until the operations reach a consistent state. In the read test the SDS has similar performance to the base file system (ext2 2.2), and in the delete test, it has similar performance to the journaling file system, ext3. It is only during file creation that the SDS pays a significant cost relative to ext3; the overhead of block differencing and hash table operations have a noticeable impact. Given that the purpose of this case study is to demonstrate that an SDS can implement complex functionality, we believe that this small overhead is quite acceptable.

5.4 Complexity Analysis

We briefly explore the complexity of implementing software for an SDS. Table 5.4 shows the number of lines of code for each of the components in our system and the case studies. From the table, one can see that most of the code complexity is found in the basic cache and hash tables, and the operation inferencing code. The smarter caching case study is trivial to implement on top of this base infrastructure; however, the Journaling SDS requires a few thousand lines of code due to its inherent complexity. Thus, we conclude that including this type of functionality within an SDS is quite pragmatic.

5.5 Summary

In this chapter, we have described two case studies of functionality that can be provided within an SDS. Both case studies provide file system-like functionality within the disk system - something that existing disk systems fundamentally cannot provide. While the first case study was naturally tolerant to inaccurate information, the second case study, *i.e.*, journaling, had more stringent requirements on accuracy. To accommodate those stringent requirements, the journaling case study required a synchronous file system on top. In the next two chapters, we will relax this assumption by building similar case studies that are complex and correctness-sensitive, but that work under more general asynchronous file system behaviors.

Chapter 6

Improving Availability with D-GRAID

> "*If a tree falls in the forest and no one hears it, does it make a sound?*"
> George Berkeley

In this chapter and the next, we demonstrate the feasibility of building complex functionality within an SDS despite asynchrony at the file system. This chapter presents the design, implementation, and evaluation of D-GRAID, a gracefully-degrading and quickly-recovering RAID storage array, that exploits semantic information to enable much better availability compared to existing storage systems [102]. D-GRAID ensures that most files within the file system remain available even when an unexpectedly high number of faults occur. D-GRAID achieves high availability through aggressive replication of semantically critical data, and fault-isolated placement of logically related data. D-GRAID also recovers from failures quickly, restoring only live file system data to a hot spare.

6.1 Introduction

Storage systems comprised of multiple disks are the backbone of modern computing centers, and when the storage system is down, the entire center can grind to a halt. Downtime is clearly expensive; for example, in the on-line business world, millions of dollars per hour are lost when systems are not available [56, 78].

Storage system *availability* is formally defined as the mean time between failure (MTBF) divided by the sum of the MTBF and the mean time to recovery (MTTR): $\frac{MTBF}{MTBF+MTTR}$ [39]. Hence, in order to improve availability, one can ei-

ther increase the MTBF or decrease the MTTR. Not surprisingly, researchers have studied both of these components of storage availability.

To increase the time between failures of a large storage array, data redundancy techniques can be applied [12, 16, 22, 40, 49, 73, 76, 77, 95, 117]. By keeping multiple copies of blocks, or through more sophisticated redundancy schemes such as parity-encoding, storage systems can tolerate a (small) fixed number of faults. To decrease the time to recovery, "hot spares" can be employed [48, 67, 76, 84]; when a failure occurs, a spare disk is activated and filled with reconstructed data, returning the system to normal operating mode relatively quickly.

6.1.1 The problem: Reduced availability due to semantic ignorance

Although various techniques have been proposed to improve storage availability, the narrow interface between file systems and storage [31] has curtailed opportunities for improving MTBF and MTTR. For example, RAID redundancy schemes typically export a simple failure model; if D or fewer disks fail, the RAID continues to operate correctly, but if more than D disks fail, the RAID is entirely unavailable until the problem is corrected, perhaps via a time-consuming restore from tape. In most RAID schemes, D is small (often 1); thus even when most disks are working, users observe a failed disk system. This "availability cliff" is a result of the storage system laying out blocks oblivious of their semantic importance or relationship; most files become corrupted or inaccessible after just one extra disk failure. Further, because the storage array has no information on which blocks are live in the file system, the recovery process must restore all blocks in the disk. This unnecessary work slows recovery and reduces availability.

An ideal storage array fails gracefully: if $\frac{1}{N}$th of the disks of the system are down, at most $\frac{1}{N}$th of the data is unavailable. An ideal array also recovers intelligently, restoring only live data. In effect, more "important" data is less likely to disappear under failure, and such data is restored earlier during recovery. This strategy for data availability stems from Berkeley's observation about falling trees: if a file isn't available, and no process tries to access it before it is recovered, is there truly a failure?

6.1.2 A solution: D-GRAID

To explore these concepts and provide a storage array with more graceful failure semantics, we present the design, implementation, and evaluation of D-GRAID, a RAID system that Degrades Gracefully (and recovers quickly). D-GRAID exploits semantic intelligence within the disk array to place file system structures

across the disks in a fault-contained manner, analogous to the fault containment techniques found in the Hive operating system [20] and in some distributed file systems [54, 94]. Thus, when an unexpected "double" failure occurs [39], D-GRAID continues operation, serving those files that can still be accessed. D-GRAID also utilizes semantic knowledge during recovery; specifically, only blocks that the file system considers live are restored onto a hot spare. Both aspects of D-GRAID combine to improve the effective availability of the storage array. Note that D-GRAID techniques are complementary to existing redundancy schemes; thus, if a storage administrator configures a D-GRAID array to utilize RAID Level 5, any single disk can fail without data loss, and additional failures lead to a proportional fraction of unavailable data.

We have built a prototype implementation of D-GRAID, which we refer to as *Alexander*. Alexander is an example of a semantically-smart disk system; it exploits semantic knowledge to implement graceful degradation and quick recovery. Alexander currently functions underneath unmodified Linux ext2 and VFAT file systems. By running under more general asynchronous file system behaviors, D-GRAID demonstrates that it is feasible to build complex functionality despite fundamentally imperfect information.

6.1.3 Key techniques

There are two key aspects to the Alexander implementation of graceful degradation. The first is *selective meta-data replication*, in which Alexander replicates naming and system meta-data structures of the file system to a high degree while using standard redundancy techniques for data. Thus, with a small amount of overhead, excess failures do not render the entire array unavailable. Instead, the entire directory hierarchy can still be traversed, and only some fraction of files will be missing, proportional to the number of missing disks. The second is a *fault-isolated data placement* strategy. To ensure that semantically meaningful data units are available under failure, Alexander places semantically-related blocks (*e.g.*, the blocks of a file) within the storage array's unit of fault-containment (*e.g.*, a disk). By observing the natural failure boundaries found within an array, failures make semantically-related groups of blocks unavailable, leaving the rest of the file system intact.

Unfortunately, fault-isolated data placement improves availability at a cost; related blocks are no longer striped across the drives, reducing the natural benefits of parallelism found within most RAID techniques [34]. To remedy this, Alexander also implements *access-driven diffusion* to improve throughput to frequently-accessed files, by spreading a copy of the blocks of "hot" files across the drives of the system. Alexander monitors access to data to determine which files to replicate

in this fashion, and finds space for those replicas either in a pre-configured *performance reserve* or opportunistically in the unused portions of the storage system.

We evaluate the availability improvements possible with D-GRAID through trace analysis and simulation, and find that D-GRAID does an excellent job of masking an arbitrary number of failures from most processes by enabling continued access to "important" data. We then evaluate our prototype, Alexander under microbenchmarks and trace-driven workloads. We find that the construction of D-GRAID is feasible; even with imperfect semantic knowledge, powerful functionality can be implemented within a block-based storage array. We also find that the run-time overheads of D-GRAID are small, but that the storage-level CPU costs compared to a standard array are high. We show that access-driven diffusion is crucial for performance, and that live-block recovery is effective when disks are under-utilized. The combination of replication, data placement, and recovery techniques results in a storage system that improves availability while maintaining a high level of performance.

The rest of this chapter is structured as follows. In Section 6.2, we present an extended motivation for graceful degradation. We present the design principles of D-GRAID in Section 6.3, and present trace analysis and simulations in Section 6.4. In Section 6.5, we outline the file system model on which our implementation of D-GRAID is based. In Section 6.6, we present our prototype implementation, and evaluate our prototype in Section 6.7. In Section 6.8, we present custom policies for different *levels* of D-GRAID. We discuss the resilience of D-GRAID to incorrect information in Section 6.9, and summarize in Section 6.10.

6.2 Extended Motivation

In this section, we motivate the need for graceful degradation during multiple failures. The motivation for graceful degradation arises from the fact that users and applications often do not require that the entire contents of a volume be present; rather, what matters to them is whether a particular set of files are available.

One question that arises is whether it is realistic to expect a catastrophic failure scenario within a RAID system. For example, in a RAID-5 system, given the high MTBF's reported by disk manufacturers, one might believe that a second disk failure is highly unlikely to occur before the first failed disk is repaired. However, multiple disk failures do occur, for two primary reasons. First, correlated faults are more common in systems than expected [41]. If the RAID has not been carefully designed in an orthogonal manner, a single controller fault or other component error can render a fair number of disks unavailable [22]; such redundant designs are

expensive, and therefore may only be found in higher end storage arrays. Second, Gray points out that system administration is the main source of failure in systems [39]. A large percentage of human failures occur during maintenance, where "the maintenance person typed the wrong command or unplugged the wrong module, thereby introducing a double failure" (page 6) [39].

Other evidence also suggests that multiple failures can occur. For example, IBM's ServeRAID array controller product includes directions on how to attempt data recovery when multiple disk failures occur within a RAID-5 storage array [53]. Within our own organization, data is stored on file servers under RAID-5. In one of the Computer Science department's servers, a single disk failed, but the indicator that should have informed administrators of the problem did not do so. The problem was only discovered when a second disk in the array failed; full restore from backup ran for days. In this scenario, graceful degradation would have enabled access to a large fraction of user data during the long restore.

One might think that the best approach to dealing with multiple failures would be to employ a higher level of redundancy [3, 16], thus enabling the storage array to tolerate a greater number of failures without loss of data. However, these techniques are often expensive (*e.g.*, three-way data mirroring) or bandwidth-intensive (*e.g.*, more than 6 I/Os per write in a P+Q redundant store). Graceful degradation is complementary to such techniques. Thus, storage administrators could choose the level of redundancy they believe necessary for common case faults; graceful degradation is enacted when a "worse than expected" fault occurs, mitigating its ill effect.

6.3 Design: D-GRAID Expectations

We now discuss the design of D-GRAID. We present background information on file systems, the data layout strategy required to enable graceful degradation, the important design issues that arise due to the new layout, and the process of fast recovery.

6.3.1 Graceful degradation

To ensure partial availability of data under multiple failures in a RAID array, D-GRAID employs two main techniques. The first is a *fault-isolated data placement* strategy, in which D-GRAID places each "semantically-related set of blocks" within a "unit of fault containment" found within the storage array. For simplicity of discussion, we assume that a file is a semantically-related set of blocks, and that a single disk is the unit of fault containment. We will generalize the former

below, and the latter is easily generalized if there are other failure boundaries that should be observed (*e.g.*, SCSI chains). We refer to the physical disk to which a file belongs as the *home site* for the file. When a particular disk fails, fault-isolated data placement ensures that only files that have that disk as their home site become unavailable, while other files remain accessible as whole files.

The second technique is *selective meta-data replication*, in which D-GRAID replicates naming and system meta-data structures of the file system to a high degree, *e.g.*, directory inodes and directory data in a UNIX file system. D-GRAID thus ensures that all live data is reachable and not orphaned due to multiple failures. The entire directory hierarchy remains traversable, and the fraction of missing user data is proportional to the number of failed disks.

Thus, D-GRAID lays out logical file system blocks in such a way that the availability of a single file depends on as few disks as possible. In a traditional RAID array, this dependence set is normally the entire set of disks in the group, thereby leading to entire file system unavailability under an unexpected failure. A UNIX-centric example of typical layout, fault-isolated data placement, and selective meta-data replication is depicted in Figure 6.1. Note that for the techniques in D-GRAID to work, a meaningful subset of the file system must be laid out within a single D-GRAID array. For example, if the file system is striped across multiple D-GRAID arrays, no single array will have a meaningful view of the file system. In such a scenario, D-GRAID can be run at the logical volume manager level, viewing each of the arrays as a single disk; the same techniques remain relevant.

Because D-GRAID treats each file system block type differently, the traditional RAID taxonomy is no longer adequate in describing how D-GRAID behaves. Instead, a more fine-grained notion of a RAID level is required, as D-GRAID may employ different redundancy techniques for different types of data. For example, D-GRAID commonly employs n-way mirroring for naming and system meta-data, whereas it uses standard redundancy techniques, such as mirroring or parity encoding (*e.g.*, RAID-5), for user data. Note that n, a value under administrative control, determines the number of failures under which D-GRAID will degrade gracefully. In Section 6.4, we will explore how data availability degrades under varying levels of namespace replication.

6.3.2 Design considerations

The layout and replication techniques required to enable graceful degradation introduce a number of design issues. We highlight the major challenges that arise.

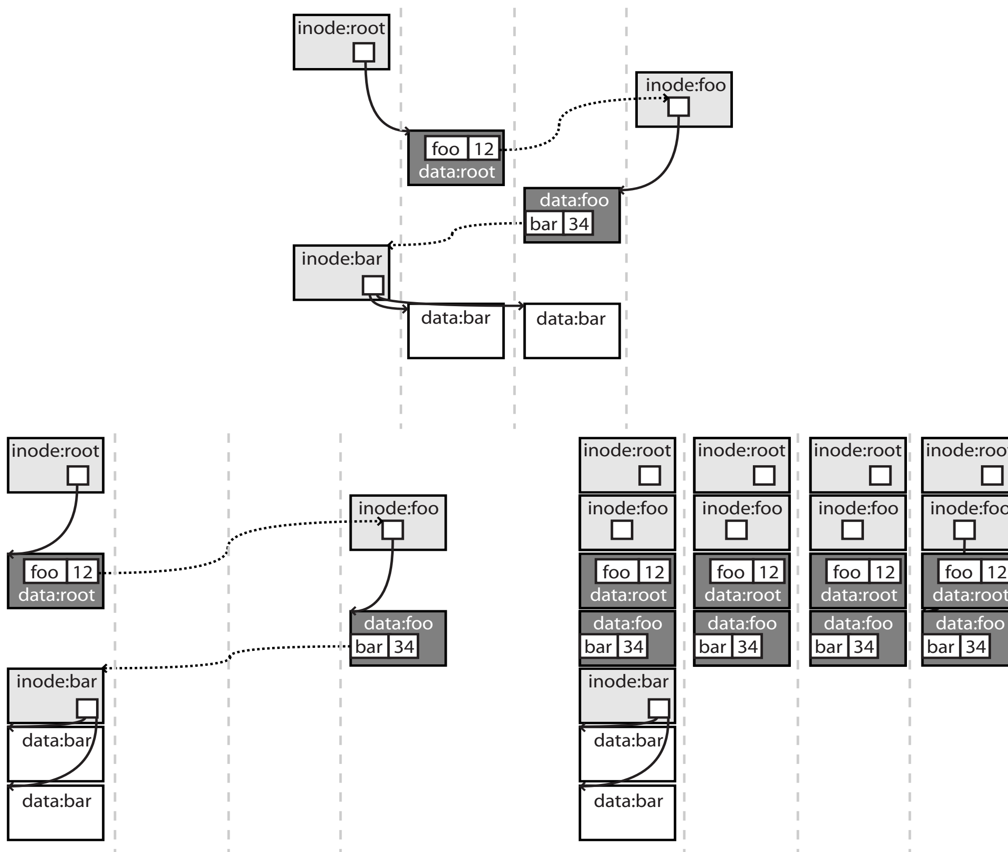


Figure 6.1: **A Comparison of Layout Schemes.** *These figures depict different layouts of a file "/foo/bar" in a* UNIX *file system starting at the root inode and following down the directory tree to the file data. Each vertical column represents a disk. For simplicity, the example assumes no data redundancy for user file data. On the top is a typical file system layout on a non-D-GRAID disk system; because blocks (and therefore pointers) are spread throughout the file system, any single fault will render the blocks of the file "bar" inaccessible. The left figure in the bottom is a fault-isolated data placement of files and directories. In this scenario, if one can access the inode of a file, one can access its data (indirect pointer blocks would also be constrained within the same disk). Finally, in the bottom right is an example of selective meta-data replication. By replicating directory inodes and directory blocks, D-GRAID can guarantee that users can get to all files that are available. Some of the requisite pointers have been removed from the rightmost figure for simplicity. Color codes are white for user data, light shaded for inodes, and dark shaded for directory data.*

Semantically-related blocks

With fault-isolated data placement, D-GRAID places a logical unit of file system data (*e.g.*, a file) within a fault-isolated container (*e.g.*, a disk). Which blocks D-GRAID considers "related" thus determines which data remains available under failure. The most basic approach is *file-based* grouping, in which a single file (including its data blocks, inode, and indirect pointers) is treated as the logical unit of data; however, with this technique a user may find that some files in a directory are unavailable while others are not, which may cause frustration and confusion. Other groupings preserve more meaningful portions of the file system volume under failure. With *directory-based* grouping, D-GRAID ensures that the files of a directory are all placed within the same unit of fault containment. Less automated options are also possible, allowing users to specify arbitrary semantic groupings which D-GRAID then treats as a unit.

Load balance

With fault-isolated placement, instead of placing blocks of a file across many disks, the blocks are isolated within a single home site. Isolated placement improves availability but introduces the problem of load balancing, which has both space and time components.

In terms of space, the total utilized space in each disk should be maintained at roughly the same level, so that when a fraction of disks fail, roughly the same fraction of data becomes unavailable. Such balancing can be addressed in the foreground (*i.e.*, when data is first allocated), the background (*i.e.*, with migration), or both. Files (or directories) larger than the amount of free space in a single disk can be handled either with a potentially expensive reorganization or by reserving large extents of free space on a subset of drives. Files that are larger than a single disk must be split across disks.

More pressing are the performance problems introduced by fault-isolated data placement. Previous work indicates that striping of data across disks is better for performance even compared to sophisticated file placement algorithms [34, 118]. Thus, D-GRAID makes additional copies of user data that are spread across the drives of the system, a process which we call *access-driven diffusion*. Whereas standard D-GRAID data placement is optimized for availability, access-driven diffusion increases performance for those files that are frequently accessed. Not surprisingly, access-driven diffusion introduces policy decisions into D-GRAID, including where to place replicas that are made for performance, which files to replicate, and when to create the replicas.

Meta-data replication level

The degree of meta-data replication within D-GRAID determines how resilient it is to excessive failures. Thus, a high degree of replication is desirable. Unfortunately, meta-data replication comes with costs, both in terms of space and time. For space overheads, the trade-offs are obvious: more replicas imply more resiliency. One difference between traditional RAID and D-GRAID is that the amount of space needed for replication of naming and system meta-data is dependent on usage, *i.e.*, a volume with more directories induces a greater amount of overhead. For time overheads, a higher degree of replication implies lowered write performance for naming and system meta-data operations. However, others have observed that there is a lack of update activity at higher levels in the directory tree [80], and lazy update propagation can be employed to reduce costs [95].

6.3.3 Fast recovery

Because the main design goal of D-GRAID is to ensure higher availability, fast recovery from failure is also critical. The most straightforward optimization available with D-GRAID is to recover only "live" file system data. Assume we are restoring data from a live mirror onto a hot spare; in the straightforward approach, D-GRAID simply scans the source disk for live blocks, examining appropriate file system structures to determine which blocks to restore. This process is readily generalized to more complex redundancy encodings. D-GRAID can potentially prioritize recovery in a number of ways, *e.g.*, by restoring certain "important" files first, where importance could be domain specific (*e.g.*, files in `/etc`) or indicated by users in a manner similar to the hoarding database in Coda [57].

6.4 Exploring Graceful Degradation

In this section, we use simulation and trace analysis to evaluate the potential effectiveness of graceful degradation and the impact of different semantic grouping techniques. We first quantify the space overheads of D-GRAID. Then we demonstrate the ability of D-GRAID to provide continued access to a proportional fraction of meaningful data after arbitrary number of failures. More importantly, we then demonstrate how D-GRAID can hide failures from users by replicating "important" data. The simulations use file system traces collected from HP Labs [89], and cover 10 days of activity; there are 250 GB of data spread across 18 logical volumes.

	Level of Replication		
	1-way	4-way	16-way
$ext2_{1KB}$	0.15%	0.60%	2.41%
$ext2_{4KB}$	0.43%	1.71%	6.84%
$VFAT_{1KB}$	0.52%	2.07%	8.29%
$VFAT_{4KB}$	0.50%	2.01%	8.03%

Table 6.1: **Space Overhead of Selective Meta-data Replication.** *The table shows the space overheads of selective meta-data replication as a percentage of total user data, as the level of naming and system meta-data replication increases. In the leftmost column, the percentage space overhead without any meta-data replication is shown. The next two columns depict the costs of modest (4-way) and paranoid (16-way) schemes. Each row shows the overhead for a particular file system, either ext2 or VFAT, with block size set to 1 KB or 4 KB.*

6.4.1 Space overheads

We first examine the space overheads due to selective meta-data replication that are typical with D-GRAID-style redundancy. We calculate the cost of selective meta-data replication as a percentage overhead, measured across all volumes of the HP trace data when laid out in either the ext2 or the VFAT file system. When running underneath ext2, selective meta-data replication is applied to the superblock, inode and data block bitmaps, and the inode and data blocks of directory files. The blocks replicated in the case of VFAT are those that comprise the FAT and the directory entries. We calculate the highest possible percentage of selective meta-data replication overhead by assuming no replication of user data; if user data is mirrored, the overheads are cut in half.

Table 6.1 shows that selective meta-data replication induces only a mild space overhead even under high levels of meta-data redundancy for both the Linux ext2 and VFAT file systems. Even with 16-way redundancy of meta-data, only a space overhead of 8% is incurred in the worst case (VFAT with 1 KB blocks). With increasing block size, while ext2 uses more space (due to internal fragmentation with larger directory blocks), the overheads actually decrease with VFAT. This phenomenon is due to the structure of VFAT; for a fixed-sized file system, as block size grows, the file allocation table itself shrinks, although the blocks that contain directory data grow.

6.4.2 Static availability

We next examine how D-GRAID availability degrades under failure with two different semantic grouping strategies. The first strategy is file-based grouping, which keeps the information associated with a single file within a failure boundary (*i.e.*, a disk); the second is directory-based grouping, which allocates files of a directory together. For this analysis, we place the entire 250 GB of files and directories from the HP trace onto a simulated 32-disk system, remove simulated disks, and measure the percentage of whole directories that are available. We assume no user data redundancy (*i.e.*, D-GRAID Level 0).

Figure 6.2 shows the percent of directories available, where a directory is available if all of its files are accessible (although subdirectories and their files may not be). From the figure, we observe that graceful degradation works quite well, with the amount of available data proportional to the number of working disks, in contrast to a traditional RAID where a few disk crashes would lead to complete data unavailability. In fact, availability sometimes degrades slightly less than expected from a strict linear fall-off; this is due to a slight imbalance in data placement across disks and within directories. Further, even a modest level of namespace replication (*e.g.*, 4-way) leads to very good data availability under failure. We also conclude that with file-based grouping, some files in a directory are likely to "disappear" under failure, leading to user dissatisfaction.

6.4.3 Dynamic availability

Finally, by simulating dynamic availability, we examine how often users or applications will be oblivious that D-GRAID is operating in degraded mode. Specifically, we run a portion of the HP trace through a simulator with some number of failed disks, and record what percent of processes observed no I/O failure during the run. Through this experiment, we find that namespace replication is not enough; certain files, that are needed by most processes, must be replicated as well.

In this experiment, we set the degree of namespace replication to 32 (full replication), and vary the level of replication of the contents of popular directories, *i.e.*, `/usr/bin`, `/bin`, `/lib` and a few others. Figure 6.3 shows that without replicating the contents of those directories, the percent of processes that run without ill-effect is lower than expected from our results in Figure 6.2. However, when those few directories are replicated, the percentage of processes that run to completion under disk failure is much better than expected. The reason for this is clear: a substantial number of processes (*e.g.*, `who`, `ps`, etc.) only require that their executable and a few other libraries are available to run correctly. With popular direc-

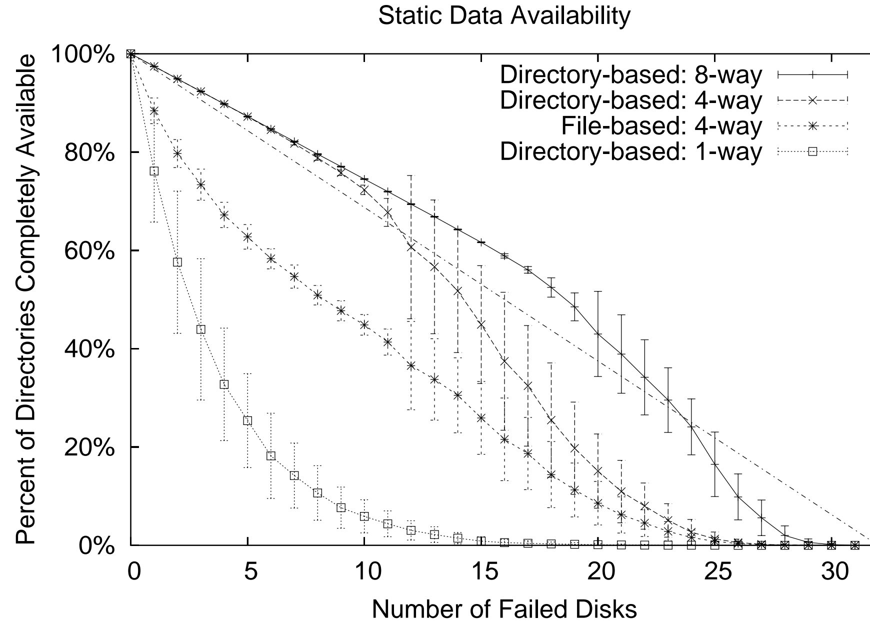


Figure 6.2: **Static Data Availability.** *The percent of entire directories available is shown under increasing disk failures. The simulated system consists of 32 disks, and is loaded with the 250 GB from the HP trace. Two different strategies for semantic grouping are shown: file-based and directory-based. Each line varies the level of replication of namespace metadata. Each point shows average and standard deviation across 30 trials, where each trial randomly varies which disks fail.*

tory replication, excellent availability under failure is possible. Fortunately, almost all of the popular files are in “read only” directories; thus, wide-scale replication will not raise write performance or consistency issues. Also, the space overhead due to popular directory replication is minimal for a reasonably sized file system; for this trace, such directories account for about 143 MB, less than 0.1% of the total file system size.

6.5 File System Model

The journaling case study in the previous chapter considered a synchronous file system, that committed metadata updates synchronously to disk. For this case study, we consider a more general file system model that is in tune with modern

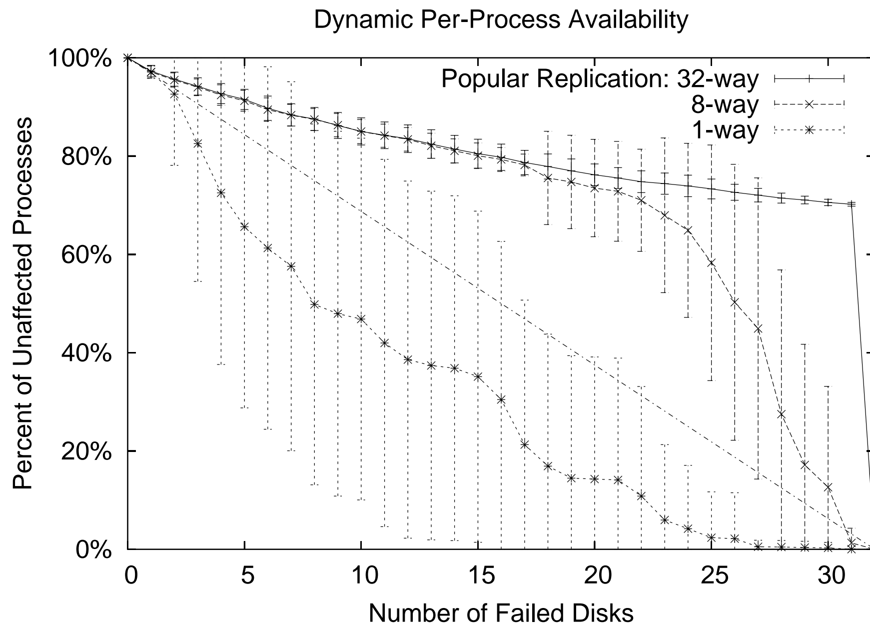


Figure 6.3: **Dynamic Data Availability.** *The figure plots the percent of processes that run unaffected under disk failure from one busy hour from the HP trace. The degree of namespace replication is set aggressively to 32. Each line varies the amount of replication for "popular" directories; 1-way implies that those directories are not replicated, whereas 8-way and 32-way show what happens with a modest and extreme amount of replication. Means and deviations of 30 trials are shown.*

file system behaviors. Specifically, our implementation of D-GRAID works under asynchronous file systems, such as ext2 in asynchronous mode.

The following generic behaviors hold for an asynchronous file system.

6.5.1 Arbitrary ordering

An arbitrary ordering file system orders updates to the file system arbitrarily; hence, no particular update order can be relied upon to garner extra information about the nature of disk traffic. For example, in FFS, meta-data updates are forced to disk synchronously, and thus will arrive at the disk before the corresponding data. Other file systems are very careful in how they order updates to disk [33], and therefore some ordering could be assumed; however, to remain as general as possible, we avoid any such assumptions.

In the next chapter, we will analyze this assumption in greater detail to explore how semantic inference techniques can be simplified if the file system provides certain simple ordering guarantees.

6.5.2 Delayed update

An asynchronous file system delays updates to disk, often for performance reasons. Delays are found in writing data to disk in many file systems, including LFS [91], which buffers data in memory before flushing it to disk; this improves performance both by batching small updates into a single large one, and also by avoiding the need to write data that is created but then deleted. A contrast to delayed updates is a file system that immediately reflects file system changes to disk; for example, ext2 can be mounted synchronously to behave in this manner.

6.5.3 Hidden operation

The hidden operation property, which refers to the file system not reflecting all its operations to disk, goes hand-in-hand with delayed updates. For example, a file system could delay the disk update associated with file creation; a subsequent delete obviates the need to reflect the create to disk.

6.6 Implementation: Making D-GRAID

We now discuss the prototype implementation of D-GRAID known as Alexander. Alexander uses fault-isolated data placement and selective meta-data replication to provide graceful degradation under failure, and employs access-driven diffusion to correct the performance problems introduced by availability-oriented layout. Currently, Alexander replicates namespace and system meta-data to an administrator-controlled value (*e.g.*, 4 or 8), and stores user data in either a RAID-0 or RAID-1 manner; we refer to those systems as D-GRAID Levels 0 and 1, respectively.

In this section, we present the implementation of graceful degradation and live-block recovery, with most of the complexity (and hence discussion) centered around graceful degradation. For simplicity of exposition, we focus on the construction of Alexander underneath the Linux ext2 file system. At the end of the section, we discuss differences in our implementation underneath VFAT.

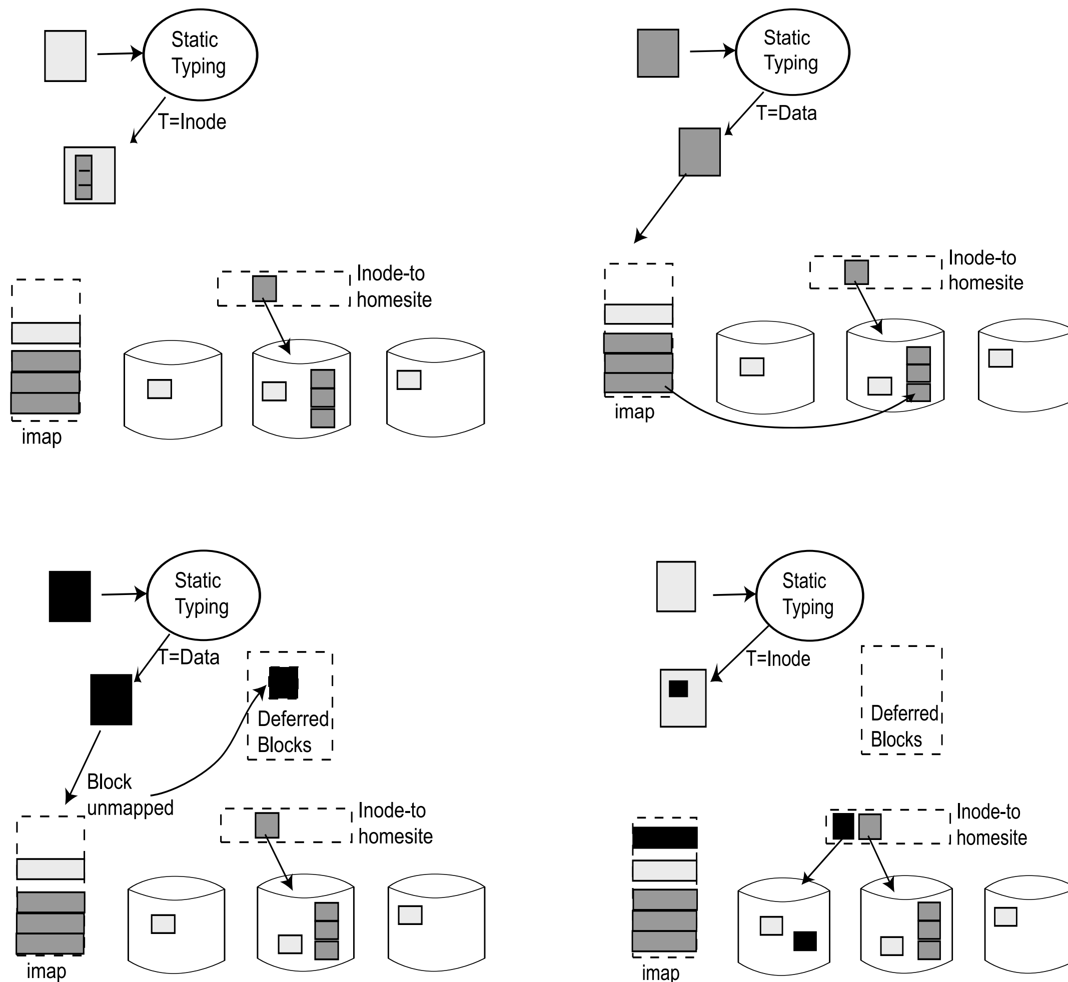


Figure 6.4: **Anatomy of a Write** *This figure depicts the control flow during a sequence of* `write` *operations in Alexander. In the first figure, an inode block is written; Alexander observes the contents of the inode block and identifies the newly added inode. It then selects a home site for the inode and creates physical mappings for the blocks of the inode, in that home site. Also, the inode block is aggressively replicated. In the next figure, Alexander observes a write to a data block from the same inode; since it is already mapped, the write goes directly to the physical block. In the third figure, Alexander gets a write to an unmapped data block; it therefore defers writing the block, and when Alexander finally observes the corresponding inode (in the fourth figure), it creates the relevant mappings, observes that one of its blocks is deferred, and therefore issues the deferred write to the relevant home site.*

6.6.1 Graceful degradation

We now present an overview of the basic operation of graceful degradation within Alexander. We first describe the simple cases before proceeding to the more intricate aspects of the implementation.

The Indirection Map

Similar to any other SCSI-based RAID system, Alexander presents host systems with a linear logical block address space. Internally, Alexander must place blocks so as to facilitate graceful degradation. Thus, to control placement, Alexander introduces a transparent level of indirection between the logical array used by the file system and physical placement onto the disks via the *indirection map (imap)*; similar structures have been used by others [30, 115, 117]. Unlike most of these other systems, this imap only maps every *live* logical file system block to its replica list, *i.e.*, all its physical locations. All *unmapped* blocks are considered free and are candidates for use by D-GRAID.

Reads

Handling block read requests at the D-GRAID level is straightforward. Given the logical address of the block, Alexander looks in the imap to find the replica list and issues the read request to one of its replicas. The choice of which replica to read from can be based on various criteria [117]; currently Alexander uses a randomized selection. However, in the presence of access-driven diffusion, the diffused copy is always given preference over the fault-isolated copy.

Writes

In contrast to reads, write requests are more complex to handle. Exactly how Alexander handles the write request depends on the *type* of the block that is written. Figure 6.4 depicts some common cases.

If the block is a static meta-data block (*e.g.*, an inode or a bitmap block) that is as of yet unmapped, Alexander allocates a physical block in each of the disks where a replica should reside, and writes to all of the copies. Note that Alexander can easily detect static block types such as inode and bitmap blocks underneath many UNIX file systems simply by observing the logical block address.

When an inode block is written, D-GRAID scans the block for newly added inodes; to understand which inodes are new, D-GRAID compares the newly written block with its old copy, a process referred to as block differencing. For every new

inode, D-GRAID selects a home site to lay out blocks belonging to the inode, and records it in the *inode-to-homesite* hashtable. This selection of home site is done to balance space allocation across physical disks. Currently, D-GRAID uses a greedy approach; it selects the home site with the most free space.

If the write is to an unmapped block in the data region (*i.e.*, a data block, an indirect block, or a directory block), the allocation cannot be done until D-GRAID knows which file the block belongs to, and thus, its actual home site. In such a case, D-GRAID places the block in a *deferred block list* and does not write it to disk until it learns which file the block is associated with. Since a crash before the inode write would make the block inaccessible by the file system anyway, the in-memory deferred block list is not a reliability concern.

D-GRAID also looks for newly added block pointers when an inode (or indirect) block is written. If the newly added block pointer refers to an unmapped block, D-GRAID adds a new entry in the imap, mapping the logical block to a physical block in the home site assigned to the corresponding inode. If any newly added pointer refers to a block in the deferred list, D-GRAID removes the block from the deferred list and issues the write to the appropriate physical block(s). Thus, writes are deferred only for blocks that are written *before* the corresponding owner inode blocks. If the inode is written first, subsequent data writes will be already mapped and sent to disk directly.

Another block type of interest that D-GRAID looks for is the data bitmap block. Whenever a data bitmap block is written, D-GRAID scans through it looking for newly freed data blocks. For every such freed block, D-GRAID removes the logical-to-physical mapping if one exists and frees the corresponding physical blocks. Further, if a block that is currently in the deferred list is freed, the block is removed from the deferred list and the write is suppressed; thus, data blocks that are written by the file system but deleted before their corresponding inode is written to disk do not generate extra disk traffic, similar to optimizations found in many file systems [91]. Removing such blocks from the deferred list is important because in the case of freed blocks, Alexander may never observe an owning inode. Thus, every deferred block stays in the deferred list for a bounded amount of time, until either an inode owning the block is written, or a bitmap block indicating deletion of the block is written. The exact duration depends on the delayed write interval of the file system.

Block Reuse

We now discuss a few of the more intricate issues involved with implementing graceful degradation. The first such issue is block reuse. As existing files are

deleted or truncated and new files are created, blocks that were once part of one file may be reallocated to some other file. Since D-GRAID needs to place blocks onto the correct home site, this reuse of blocks needs to be detected and acted upon. D-GRAID handles block reuse in the following manner: whenever an inode block or an indirect block is written, D-GRAID examines each valid block pointer to see if its physical block mapping matches the home site allocated for the corresponding inode. If not, D-GRAID changes the mapping for the block to the correct home site. However, it is possible that a write to this block (that was made in the context of the new file) went to the old home site, and hence needs to be copied from its old physical location to the new location. Blocks that must be copied are added to a *pending copies list*; a background thread copies the blocks to their new home site and frees the old physical locations when the copy completes.

Dealing with Imperfection

Another difficulty that arises in semantically-smart disks underneath typical file systems is that exact knowledge of the type of a dynamically-typed block is impossible to obtain, as discussed in Section 4.3. Thus, Alexander must handle incorrect type classification for data blocks (*i.e.*, file data, directory, and indirect blocks).

For example, D-GRAID must understand the contents of indirect blocks, because it uses the pointers therein to place a file's blocks onto its home site. However, due to lack of perfect knowledge, the fault-isolated placement of a file might be compromised (note that data loss or corruption is not an issue). Our goal in dealing with imperfection is thus to conservatively avoid it when possible, and eventually detect and handle it in all other cases.

Specifically, whenever a block construed to be an indirect block is written, we assume it is a valid indirect block. Thus, for every live pointer in the block, D-GRAID must take some action. There are two cases to consider. In the first case, a pointer could refer to an unmapped logical block. As mentioned before, D-GRAID then creates a new mapping in the home site corresponding to the inode to which the indirect block belongs. If this indirect block (and pointer) is valid, this mapping is the correct mapping. If this indirect block is misclassified (and consequently, the pointer invalid), D-GRAID detects that the block is free when it observes the data bitmap write, at which point the mapping is removed. If the block is allocated to a file before the bitmap is written, D-GRAID detects the reallocation during the inode write corresponding to the new file, creates a new mapping, and copies the data contents to the new home site (as discussed above).

In the second case, a potentially corrupt block pointer could point to an already mapped logical block. As discussed above, this type of block reuse results in a new

mapping and copy of the block contents to the new home site. If this indirect block (and hence, the pointer) is valid, this new mapping is the correct one for the block. If instead the indirect block is a misclassification, Alexander wrongly copies over the data to the new home site. Note that the data is still accessible; however, the original file to which the block belongs, now has one of its blocks in the incorrect home site. Fortunately, this situation is transient, because once the inode of the file is written, D-GRAID detects this as a reallocation and creates a new mapping back to the original home site, thereby restoring its correct mapping. Files which are never accessed again are properly laid out by an infrequent sweep of inodes that looks for rare cases of improper layout.

Thus, without any optimizations, D-GRAID will eventually move data to the correct home site, thus preserving graceful degradation. However, to reduce the number of times such a misclassification occurs, Alexander makes an assumption about the contents of indirect blocks, specifically that they contain some number of valid unique pointers, or null pointers. Alexander can leverage this assumption to greatly reduce the number of misclassifications, by performing an integrity check on each supposed indirect block. The integrity check, which is reminiscent of work on conservative garbage collection [13], returns true if all the "pointers" (4-byte words in the block) point to valid data addresses within the volume and all non-null pointers are unique. Clearly, the set of blocks that pass this integrity check could still be corrupt if the data contents happened to exactly evade our conditions. However, a test run across the data blocks of our local file system indicates that only a small fraction of data blocks (less than 0.1%) would pass the test;only those blocks that pass the test *and* are reallocated from a file data block to an indirect block would be misclassified.[1]

Access-driven Diffusion

Another issue that D-GRAID must address is performance. Fault-isolated data placement improves availability but at the cost of performance. Data accesses to blocks of a large file, or, with directory-based grouping, to files within the same directory, are no longer parallelized. To improve performance, Alexander performs access-driven diffusion, monitoring block accesses to determine which block ranges are "hot", and then "diffusing" those blocks via replication across the disks of the system to enhance parallelism.

[1]By being sensitive to data contents, semantically-smart disks place a new requirement on file system traces to include user data blocks. However, the privacy concerns that such a campaign would encounter may be too difficult to overcome.

Access-driven diffusion can be achieved at both the logical and physical levels of a disk volume. In the logical approach, access to individual files is monitored, and those considered hot are diffused. However, per-file replication fails to capture sequentiality across multiple small files, for example, those in a single directory. Therefore we instead pursue a physical approach, in which Alexander replicates segments of the logical address space across the disks of the volume. Since file systems are good at allocating contiguous logical blocks for a single file, or to files in the same directory, replicating logical segments is likely to identify and exploit most common access patterns.

To implement access-driven diffusion, Alexander divides the logical address space into multiple segments, and during normal operation, gathers information on the utilization of each segment. A background thread selects logical segments that remain “hot” for a certain number of consecutive *epochs* and diffuses a copy across the drives of the system. Subsequent reads and writes first go to these replicas, with background updates sent to the original blocks. The imap entry for the block indicates which copy is up to date. Clearly, the policy for deciding which segments to diffuse is quite simplistic in our prototype implementation. A more detailed analysis of the policy space for access-driven diffusion is left for future work.

The amount of disk space to allocate to performance-oriented replicas presents an important policy decision. The initial policy that Alexander implements is to reserve a certain minimum amount of space (specified by the system administrator) for these replicas, and then opportunistically use the free space available in the array for additional replication. This approach is similar to that used by AutoRAID for mirrored data [117], except that AutoRAID cannot identify data that is considered “dead” by the file system once written; in contrast, D-GRAID can use semantic knowledge to identify which blocks are free.

6.6.2 Live-block recovery

To implement live-block recovery, D-GRAID must understand which blocks are live. This knowledge must be correct in that no block that is live is considered dead, as that would lead to data loss. Alexander tracks this information by observing bitmap and data block traffic. Bitmap blocks tell us the liveness state of the file system that has been reflected to disk. However, due to reordering and delayed updates, it is not uncommon to observe a write to a data block whose corresponding bit has not yet been set in the data bitmap. To account for this, D-GRAID maintains a duplicate copy of all bitmap blocks, and whenever it sees a write to a block, sets the corresponding bit in the local copy of the bitmap. The duplicate copy is synchronized with the file system copy when the data bitmap block is written by

the file system. This *conservative bitmap table* thus reflects a superset of all live blocks in the file system, and can be used to perform live-block recovery. Note that we assume the pre-allocation state of the bitmap will not be written to disk after a subsequent allocation; the locking in Linux and other modern systems already ensures this. Though this technique guarantees that a live block is never classified as dead, it is possible for the disk to consider a block live far longer than it actually is. This situation would arise, for example, if the file system writes deleted blocks to disk.

To implement live-block recovery, Alexander simply uses the conservative bitmap table to build a list of blocks which need to be restored. Alexander then proceeds through the list and copies all live data onto the hot spare.

6.6.3 Other aspects of Alexander

There are a number of other aspects of the implementation that are required for a successful prototype. In this subsection, we briefly describe some of the key aspects.

Physical block allocation

The logical array of blocks exported by SCSI has the property that block numbers that are contiguous in the logical address space are mapped to contiguous physical locations on disk. This property empowers file systems to place data contiguously on disk simply by allocating contiguous logical blocks to the data. In traditional RAID, this property is straightforward to preserve. Because physical blocks are assigned in a round-robin fashion across disks, the contiguity guarantees continue to hold; the physical block to assign for a given logical block is a simple arithmetic calculation on the logical block number.

However in D-GRAID, deciding on the physical block to allocate for a newly written logical block is not straightforward; the decision depends on the file to which the logical block belongs, and its logical offset within the file. Because of fault-isolated placement, a set of contiguous logical blocks (*e.g.*, those that belong to a single file) may all map to contiguous physical blocks on the same disk; thus, if a logical block L within that set is mapped to physical block P, block $L + k$ within the same set should be mapped to physical block $P + k$ in order to preserve contiguity expectations. However, at a larger granularity, since D-GRAID balances space utilization across files, the allocation policy should be different; for large values of k, block $L + k$ should map to physical block $P + (k/N)$ where N is the

number of disks in the array. The choice of which of these policies to use requires estimates of file size which are quite dynamic.

Our prototype addresses this issue with a simple technique of space reservations. Alexander utilizes its knowledge of inodes and indirect blocks to get *a priori* estimates of the exact size of the entire file (or a large segment of the file, as in the case of indirect block). When it observes a new inode written that indicates a file of size b blocks, it reserves b contiguous blocks in the home-site assigned for that file, so that when the actual logical blocks are written subsequently, the reserved space can be used. Note that since blocks are deferred until their inodes (or indirect blocks) are observed, a write to a new logical block will always have a prior reservation. Since inodes and indirect blocks are written only periodically (*e.g.*, once every 5 seconds), the size information obtained from those writes is quite stable.

Just-in-time commit

Space reservations depend on the size information extracted from inode and indirect blocks. However, given that indirect block detection is fundamentally inaccurate, a misclassified indirect block could result in spurious reservations that hold up physical space. To prevent this, Alexander employs lazy allocation, where actual physical blocks are committed only when the corresponding logical block is written. The reservation still happens *a priori*, but these reservations are viewed as *soft* and the space is reclaimed if required.

Interaction of deferred writes with `sync`

Alexander defers disk writes of logical blocks for which it has not observed an owning inode. Such arbitrary deferral could potentially conflict with application-level expectations after a `sync` operation is issued; when a `sync` returns, the application expects all data to be on disk. To preserve these semantics, D-GRAID handles inode and indirect block writes specially. D-GRAID does not return success on a write to an inode or indirect block until all deferred writes to blocks pointed to by that inode (or indirect) block have actually reached disk. Since the `sync` operation will not complete until the inode block write returns, all deferred writes are guaranteed to be complete before `sync` returns. The same argument extends for `fsync`, which will not return until all writes pertaining to the particular file complete. However, one weakness of this approach is that if the application performs an equivalent of `fdatasync` (*i.e.*, which flushes only the data blocks to disk, and not metadata), the above technique would not preserve the expected semantics.

Inconsistent fault behavior of Linux ext2

One interesting issue that required a change from our design was the behavior of Linux ext2 under partial disk failure. When a process tries to read a data block that is unavailable, ext2 issues the read and returns an I/O failure to the process. When the block becomes available again (*e.g.*, after recovery) and a process issues a read to it, ext2 will again issue the read, and everything works as expected. However, if a process tries to open a file whose inode is unavailable, ext2 marks the inode as "suspicious" and will never again issue an I/O request to the inode block, even if Alexander has recovered the block. To avoid a change to the file system and retain the ability to recover failed inodes, Alexander replicates inode blocks as it does namespace meta-data, instead of collocating them with the data blocks of a file.

Persistence of data structures

There are a number of structures that Alexander maintains, such as the imap, that must be reliably committed to disk and preferably, for good performance, buffered in a small amount of non-volatile RAM. Note that since the NVRAM only needs to serve as a cache of actively accessed entries in these data structures, its space requirements can be kept at an acceptable level. Though our current prototype simply stores these data structures in memory, a complete implementation would require them to be backed persistently.

Popular directory replication

The most important component that is missing from the Alexander prototype is the decision on which "popular" (read-only) directories such as `/usr/bin` to replicate widely, and when to do so. Although Alexander contains the proper mechanisms to perform such replication, the policy space remains unexplored. However, our initial experience indicates that a simple approach based on monitoring frequency of inode access time updates may likely be effective. An alternative approach allows administrators to specify directories that should be treated in this manner.

6.6.4 Alexander the FAT

Overall, we were surprised by the many similarities we found in implementing D-GRAID underneath ext2 and VFAT. For example, VFAT also overloads data blocks, using them as either user data blocks or directories; hence Alexander must defer classification of those blocks in a manner similar to the ext2 implementation.

As can be expected, the implementation of most of the basic mechanisms in D-GRAID such as physical block allocation, allocation of home sites to files, and tracking replicas of critical blocks is shared across both versions of D-GRAID.

However, there were a few instances where the VFAT implementation of D-GRAID differed in interesting ways from the ext2 version. For example, the fact that all pointers of a file are located in the file allocation table made a number of aspects of D-GRAID much simpler to implement; in VFAT, there are no indirect pointers to worry about. When a new copy of a FAT block is written, the new version can be directly compared with the previous contents of the block to get accurate information on the blocks newly allocated or deleted. We also ran across the occasional odd behavior in the Linux implementation of VFAT. For example, Linux would write to disk certain blocks that were allocated but then freed, avoiding an obvious and common file system optimization. Because of this behavior of VFAT, our estimate of the set of live blocks will be a strict superset of the blocks that are actually live. Although this was more indicative of the untuned nature of the Linux implementation, it served as yet another indicator of how semantic disks must be wary of any assumptions they make about file system behavior.

6.7 Evaluating Alexander

We now present a performance evaluation of Alexander. Similar to the SDS prototype described in Section 2, the Alexander prototype is constructed as a software RAID driver in the Linux 2.2 kernel. We focus primarily on the Linux ext2 variant, but also include some baseline measurements of the VFAT system. We wish to answer the following questions:

- Does Alexander work correctly?
- What time overheads are introduced?
- How effective is access-driven diffusion?
- How fast is live-block recovery?
- What overall benefits can we expect from D-GRAID?
- How complex is the implementation?

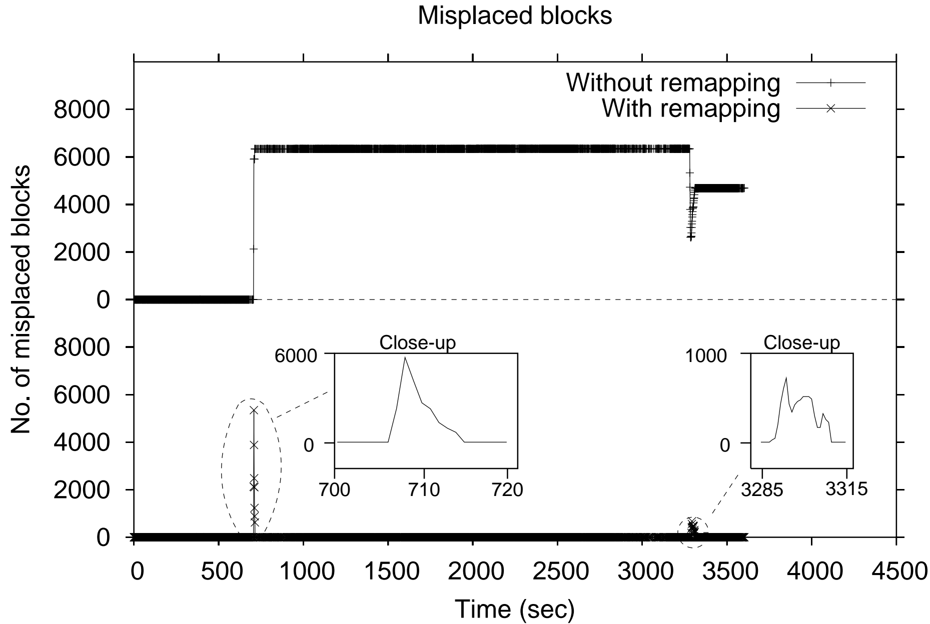


Figure 6.5: **Errors in Placement.** *The figure plots the number of blocks wrongly laid out by Alexander over time, while running a busy hour of the HP Trace. The experiment was run over 4 disks, and the total number of blocks accessed in the trace was 418000.*

6.7.1 Does Alexander work correctly?

Alexander is more complex than simple RAID systems. To gain confidence that Alexander operates correctly, we have put the system through numerous stress tests, moving large amounts of data in and out of the system without problems. We have also extensively tested the corner cases of the system, pushing it into situations that are difficult to handle and making sure that the system degrades gracefully and recovers as expected. For example, we repeatedly crafted microbenchmarks to stress the mechanisms for detecting block reuse and for handling imperfect information about dynamically-typed blocks. We have also constructed benchmarks that write user data blocks to disk that contain “worst case” data, *i.e.*, data that appears to be valid directory entries or indirect pointers. In all cases, Alexander was able to detect which blocks were indirect blocks and move files and directories into their proper fault-isolated locations.

To verify that Alexander places blocks on the appropriate disk, we instrumented the file system to log block allocations. In addition, Alexander logs events of inter-

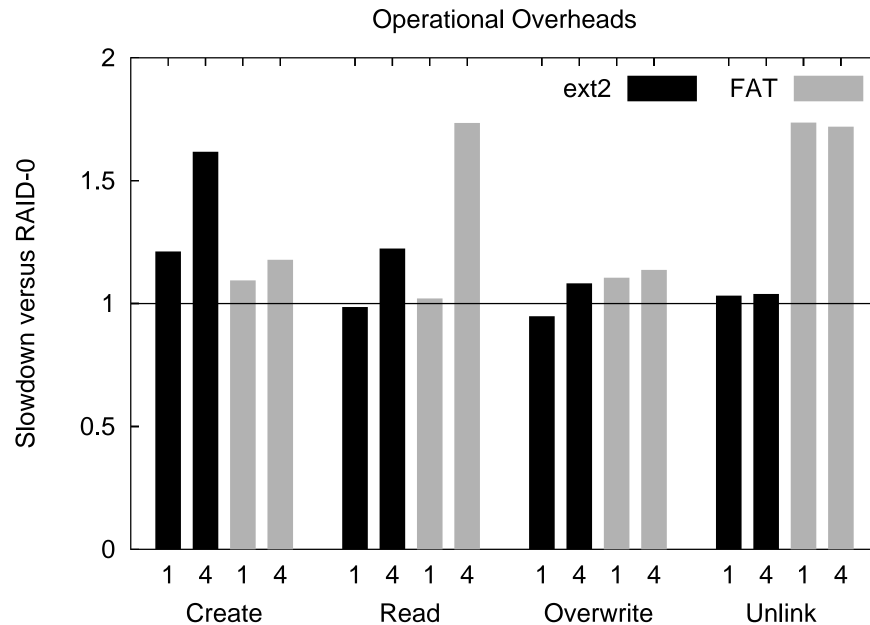


Figure 6.6: **Time Overheads.** *The figure plots the time overheads observed on D-GRAID Level 0 versus RAID Level 0 across a series of microbenchmarks. The tests are run on 1 and 4 disk systems. In each experiment, 3000 operations were enacted (e.g., 3000 file creations), with each operation on a 64 KB file.*

est such as assignment of a home site for an inode, creation of a new mapping for a logical block, re-mapping of blocks to a different homesite and receipt of logical writes from the file system. To evaluate the behavior of Alexander on a certain workload, we run the workload on Alexander, and obtain the time-ordered log of events that occurred at the file system and Alexander. We then process this log off-line and look for the number of blocks wrongly laid out at any given time.

We ran this test on a few hours of the HP Traces, and found that in many of the hours we examined, the number of blocks that were misplaced even temporarily was quite low, often less than 10 blocks. We report detailed results for one such hour of the trace where we observed the greatest number of misplaced blocks, among the hours we examined. Figure 6.5 shows the results.

The figure has two parts. The bottom part shows the normal operation of Alexander, with the capability to react to block reuse by remapping (and copying over) blocks to the correct homesite. As the figure shows, Alexander is able to quickly detect wrongly placed blocks and remap them appropriately. Further, the

	Run-time (seconds)	Blocks Written Total	Meta data	Unique
RAID-0	69.25	101297	–	–
D-GRAID$_1$	61.57	93981	5962	1599
D-GRAID$_2$	66.50	99416	9954	3198
D-GRAID$_3$	73.50	101559	16976	4797
D-GRAID$_4$	78.79	113222	23646	6396

Table 6.2: **Performance on postmark.** *The table compares the performance of D-GRAID Level 0 with RAID-0 on the Postmark benchmark. Each row marked D-GRAID indicates a specific level of metadata replication. The first column reports the benchmark run-time and the second column shows the number of disk writes incurred. The third column shows the number of disk writes that were to metadata blocks, and the fourth column indicates the number of unique metadata blocks that are written. The experiment was run over 4 disks.*

number of such blocks misplaced temporarily is only about 1% of the total number of blocks accessed in the trace. The top part of the figure shows the number of misplaced blocks for the same experiment, but assuming that the remapping did not occur. As can be expected, those delinquent blocks remain misplaced. The dip towards the end of the trace occurs because some of the misplaced blocks are later assigned to a file in that homesite itself (after a preceding delete), accidentally correcting the original misplacement.

6.7.2 What time overheads are introduced?

We now explore the time overheads that arise due to semantic inference. This primarily occurs when new blocks are written to the file system, such as during file creation. Figure 6.6 shows the performance of Alexander under a simple microbenchmark. As can be seen, allocating writes are slower due to the extra CPU cost involved in tracking fault-isolated placement. Reads and overwrites perform comparably with RAID-0. The high unlink times of D-GRAID on FAT is because FAT writes out data pertaining to deleted files, which have to be processed by D-GRAID as if it were newly allocated data. Given that the implementation is untuned and the infrastructure suffers from CPU and memory contention with the host, we believe that these are worst case estimates of the overheads.

Another cost of D-GRAID that we explore is the overhead of metadata replication. For this purpose, we choose Postmark [55], a metadata intensive file system

benchmark. We slightly modified Postmark to perform a `sync` before the deletion phase, so that all metadata writes are accounted for, making it a pessimistic evaluation of the costs. Table 6.2 shows the performance of Alexander under various degrees of metadata replication. As can be seen from the table, synchronous replication of metadata blocks has a significant effect on performance for metadata intensive workloads (the file sizes in Postmark range from 512 bytes to 10 KB). Note that Alexander performs better than default RAID-0 for lower degrees of replication because of better physical block allocation; since ext2 looks for a contiguous free chunk of 8 blocks to allocate a new file, its layout is sub-optimal for small files, since it does not pack them together.

The table also shows the number of disk writes incurred during the course of the benchmark. The percentage of extra disk writes roughly accounts for the difference in performance between different replication levels, and these extra writes are mostly to metadata blocks. However, when we count the number of unique physical writes to metadata blocks, the absolute difference between different replication levels is small. This suggests that lazy propagation of updates to metadata block replicas, perhaps during idle time or using freeblock scheduling, can greatly reduce the performance difference, at the cost of added complexity. For example, with lazy update propagation (*i.e.*, if the replicas were updated only once), D-GRAID$_4$ would incur only about 4% extra disk writes.

We also played back a portion of the HP traces for 20 minutes against a standard RAID-0 system and D-GRAID over four disks. The playback engine issues requests at the times specified in the trace, with an optional speedup factor; a speedup of 2$\times$ implies the idle time between requests was reduced by a factor of two. With speedup factors of 1$\times$ and 2$\times$, D-GRAID delivered the same per-second operation throughput as RAID-0, utilizing idle time in the trace to hide its extra CPU overhead. However, with a scaling factor of 3$\times$, the operation throughput lagged slightly behind, with D-GRAID showing a slowdown of up to 19.2% during the first one-third of the trace execution, after which it caught up due to idle time.

6.7.3 How effective is access-driven diffusion?

We now show the benefits of access-driven diffusion. In each trial of this experiment, we perform a set of sequential file reads, over files of increasing size. We compare standard RAID-0 striping to D-GRAID with and without access-driven diffusion. Figure 6.7 shows the results of the experiment.

As we can see from the figure, without access-driven diffusion, sequential access to larger files run at the rate of a single disk in the system, and thus do not benefit from the potential parallelism. With access-driven diffusion, performance

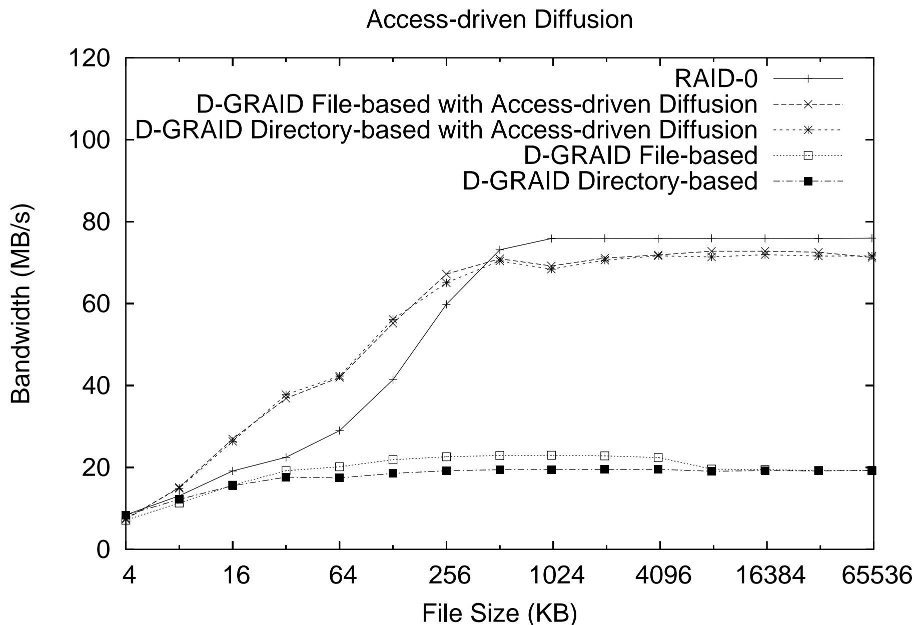


Figure 6.7: **Access-driven Diffusion.** *The figure presents the performance of D-GRAID Level 0 and standard RAID-0 under a sequential workload. In each experiment, a number of files of size x are read sequentially, with the total volume of data fixed at 64 MB. D-GRAID performs better for smaller files due to better physical block layout.*

is much improved, as reads are directed to the diffused copies across all of the disks in the system. Note that in the latter case, we arrange for the files to be already diffused before the start of the experiment, by reading them a certain threshold number of times. Investigating more sophisticated policies for when to initiate access-driven diffusion is left for future work.

6.7.4 How fast is live-block recovery?

We now explore the potential improvement seen with live-block recovery. Figure 6.8 presents the recovery time of D-GRAID while varying the amount of live file system data.

The figure plots two lines: worst case and best case live-block recovery. In the worst case, live data is spread throughout the disk, whereas in the best case it is compacted into a single portion of the volume. From the graph, we can see that live-block recovery is successful in reducing recovery time, particularly when

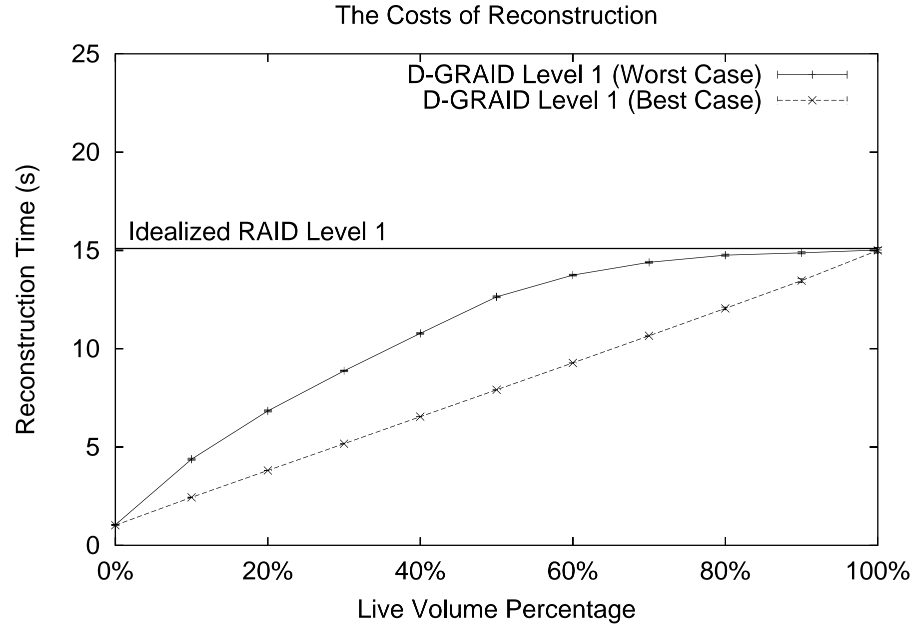


Figure 6.8: **Live-block Recovery.** *The figure shows the time to recover a failed disk onto a hot spare in a D-GRAID Level 1 (mirrored) system using live-block recovery. Two lines for D-GRAID are plotted: in the worst case, live data is spread across the entire 300 MB volume, whereas in the best case it is compacted into the smallest contiguous space possible. Also plotted is the recovery time of an idealized RAID Level 1.*

a disk is less than half full. Note also the difference between worst case and best case times; the difference suggests that periodic disk reorganization [93] could be used to speed recovery, by moving all live data to a localized portion.

6.7.5 What overall benefits can we expect from D-GRAID?

We next demonstrate the improved availability of Alexander under failures. Figure 6.9 shows the availability and performance observed by a process randomly accessing whole 32 KB files, running above D-GRAID and RAID-10. To ensure a fair comparison, both D-GRAID and RAID-10 limit their reconstruction rate to 10 MB/s.

As the figure shows, reconstruction of the 3 GB volume with 1.3 GB live data completes much faster (68 s) in D-GRAID compared to RAID-10 (160 s). Also, when the extra second failure occurs, the availability of RAID-10 drops to near

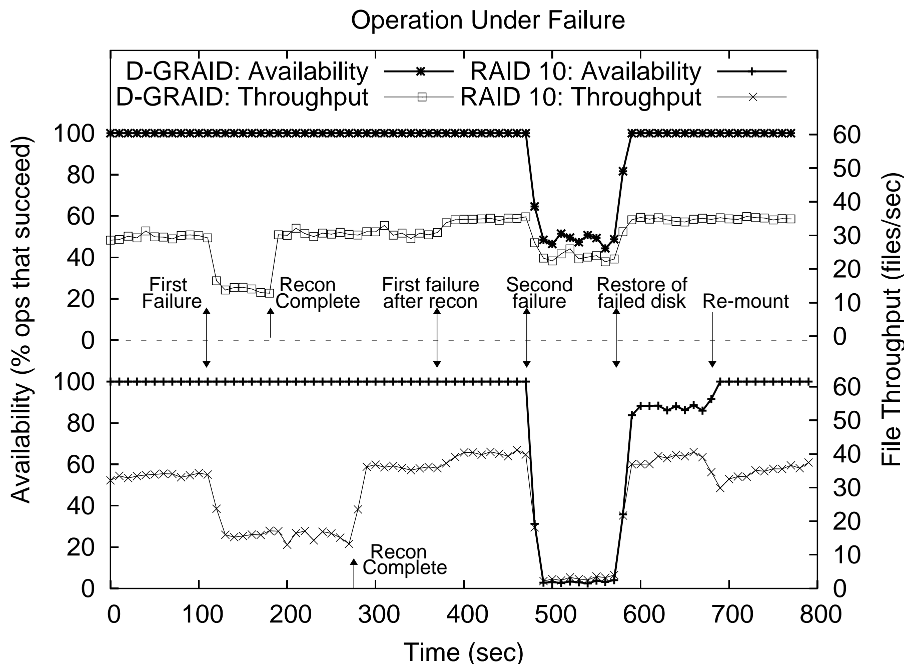


Figure 6.9: **Availability Profile.** *The figure shows the operation of D-GRAID Level 1 and RAID 10 under failures. The 3 GB array consists of 4 data disks and 1 hot spare. After the first failure, data is reconstructed onto the hot spare, D-GRAID recovering much faster than RAID 10. When two more failures occur, RAID 10 loses almost all files, while D-GRAID continues to serve 50% of its files. The workload consists of read-modify-writes of 32 KB files randomly picked from a 1.3 GB working set.*

zero, while D-GRAID continues with about 50 % availability. Surprisingly, after restore, RAID-10 still fails on certain files; this is because Linux does not retry inode blocks once they fail. A remount is required before RAID-10 returns to full availability.

6.7.6 How complex is the implementation?

We briefly quantify the implementation complexity of Alexander. Table 6.3 shows the number of C statements required to implement the different components of Alexander. From the table, we can see that the core file system inferencing module for ext2 requires only about 1200 lines of code (counted with number of semi-colons), and the core mechanisms of D-GRAID contribute to about 2000 lines of

	Semicolons	Total
D-GRAID Generic		
Setup + fault-isolated placement	1726	3557
Physical block allocation	322	678
Access driven diffusion	108	238
Mirroring + live block recovery	248	511
Internal memory management	182	406
Hashtable/Avl tree	724	1706
File System Specific		
SDS Inferencing: ext2	1252	2836
SDS Inferencing: VFAT	630	1132
Total	5192	11604

Table 6.3: **Code size for Alexander implementation.** *The number of lines of code needed to implement Alexander is shown. The first column shows the number of semicolons and the second column shows the total number of lines, including white-spaces and comments.*

code. The rest is spent on a hash table, AVL tree and wrappers for memory management. Compared to the tens of thousands of lines of code already comprising modern array firmware, we believe that the added complexity of D-GRAID is not that significant. Being an academic prototype, these complexity numbers could be a slight under-estimate of what would be required for a production quality implementation; thus, this analysis is only intended to be an approximate estimate.

6.8 D-GRAID Levels

Much of the discussion so far has focused on implementing D-GRAID over a storage system with no redundancy for user data (*i.e.*, RAID-0), or over a mirrored storage system (*i.e.*, RAID-10). However, as mentioned before, the layout *mechanisms* in D-GRAID are orthogonal to the underlying redundancy scheme. In this section, we formalize the different *levels* of D-GRAID, corresponding to the popular traditional RAID levels. We also present certain custom *policies* for each D-GRAID level that are tailored to the underlying redundancy mechanism. Note that in contrast to traditional RAID levels, the levels of D-GRAID differ only in the type of redundancy for normal user data; system meta-data is always maintained in RAID-1 with a certain configured replication degree.

6.8.1 D-GRAID-0: No redundancy

This is the simplest D-GRAID level where no redundancy mechanism is employed for normal user data. Thus, even a single disk failure results in data loss. In contrast to traditional RAID-0 where a single disk failure results in complete data loss, D-GRAID-0 ensures proportional data availability under failure. Figure 6.10(a) shows the D-GRAID-0 configuration.

Because of the absence of redundancy for normal data, the additional storage required for access-driven diffusion in D-GRAID-0 needs to come from a separate *performance reserve*, as described in Section 6.6. This reserve can be fixed to be a certain percentage (*e.g.*, 10% of the storage volume size) or can be tunable by the administrator. Tuning this parameter provides the administrator control over the trade-off between performance and storage efficiency. One issue with changing the size of the performance reserve dynamically is that file systems may not be equipped to deal with a variable volume size. This limitation can be addressed by a simple technique: the administrator creates a file in the file system with a certain reserved name (*e.g.*, */.diffuse*). The size of this file implicitly conveys to D-GRAID the size of its performance reserve. Since the file system will not use the blocks assigned to this reserved file to any other file, D-GRAID is free to use this storage space. When the file system runs short of storage, the administrator can prune the size of this special file, thus dynamically reducing the size of the performance reserve.

6.8.2 D-GRAID-10: Mirroring

A mirrored D-GRAID system stripes data across multiple mirrored pairs, similar to RAID-10. Note that D-GRAID is not meaningful in a storage system comprised of a single mirrored pair (*i.e.*, RAID-1) because such a system fundamentally has no partial failure mode. The access-driven diffusion policy in D-GRAID-10 is quite similar to D-GRAID-0 where a dynamic performance reserve is used to hold diffused copies; Figure 6.10(b) depicts this configuration. Note that the diffused copies are not mirrored; thus D-GRAID-10 requires only half the percentage of space that D-GRAID-0 requires, in order to achieve the same level of diffusion.

A slight variant of D-GRAID-10 can make access-driven diffusion much more effective, though at the cost of a slight degradation in reliability. Instead of the disks in a mirrored pair being physical mirrors as discussed above, we could employ *logical* mirroring, where we just impose that each logical disk block has two copies in two different disks. With such a relaxed definition, D-GRAID could store one copy of a file in the traditional striped fashion, while the other copy of the file is

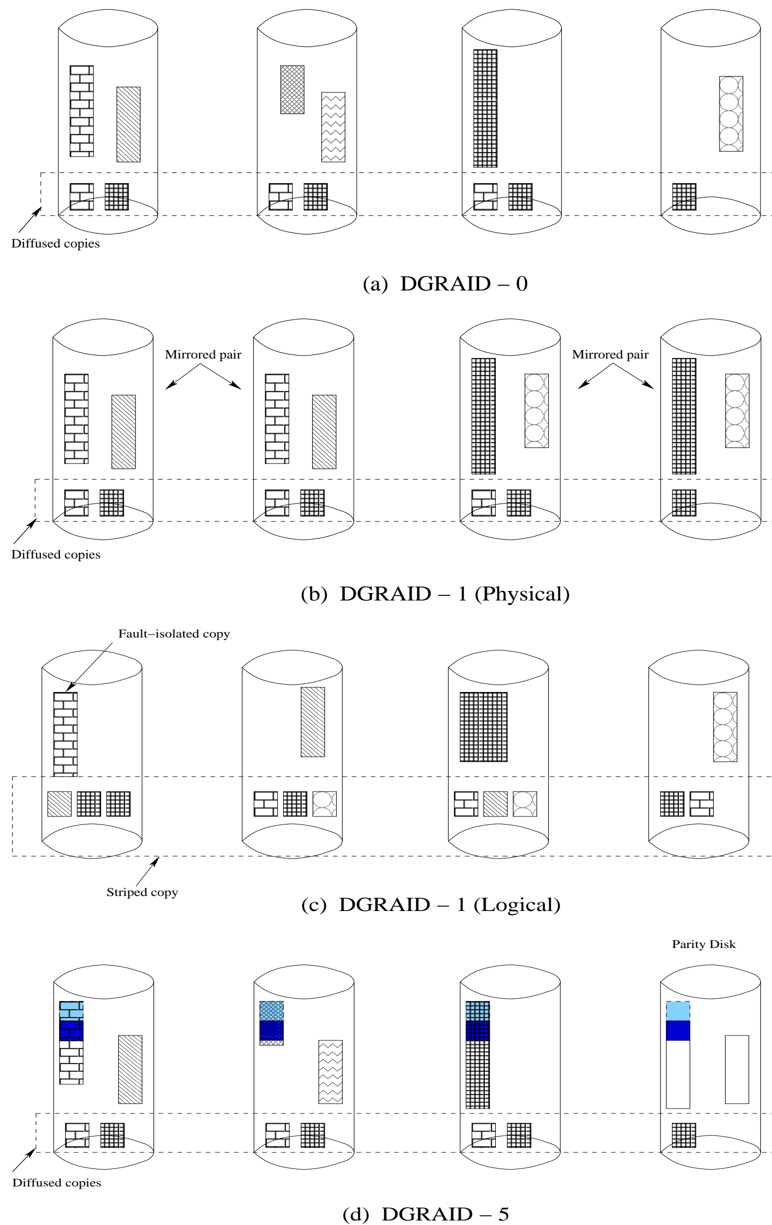


Figure 6.10: **D-GRAID Levels.** *The figures depict the data layout of D-GRAID under various redundancy schemes. Each style of shading represents a different file. In the D-GRAID-5 figure, the color of shading indicates a physical RAID-5 stripe. The diffusion segments and the striped region in D-GRAID-1(Logical) are indicated as separate regions of the disk for simplicity; in practice, they will be interleaved with the fault-isolated copies.*

stored in fault-isolated fashion. Figure 6.10(c) depicts this configuration. Each file has a fault-isolated copy laid out in a single disk, and another copy striped across all the other disks, so that a single disk failure will not result in any data loss. Such logical mirroring of data achieves the benefits of fault isolated placement with almost no impact on performance, because parallelism is still available out of the striped copies. Note that in such a scenario no extra space is required for access-driven diffusion.

Although the above variant of D-GRAID-10 improves performance by more efficient access-driven diffusion, it reduces reliability compared to a traditional D-GRAID-10. In a traditional D-GRAID-10 (*i.e.*, physical mirroring), after a single disk failure, only the failure of its mirror disk will lead to loss of data. However, in logical mirroring, the second failure always results in loss of data, though proportionally, irrespective of which disk incurred the failure.

6.8.3 D-GRAID-5: Parity

D-GRAID-5 is the counterpart of traditional RAID-5; redundancy for user data is maintained in the form of parity encoding on a small number of disks (usually 1), resulting in better space efficiency. While it may appear that the fine grained block-level striping that is fundamental to RAID-5 would be in conflict with the fault isolated placement in D-GRAID, these techniques are quite orthogonal. The fine-grained striping required for RAID-5 occurs at the *physical* level, across actual physical disk blocks, while fault isolated placement is just a logical assignment of files onto those physical blocks. Thus, D-GRAID-5 would still maintain the invariant that the kth parity block is the XOR of the kth block in every disk; the only difference is that the kth block in each disk would contain data pertaining to a different file in D-GRAID, while in RAID, they would usually be part of the same file. This configuration is shown in Figure 6.10(d), where blocks belonging to the same physical RAID-5 stripe are shaded with the same color.

However, fault isolated placement with RAID-5 like redundancy leads to a performance issue. Since blocks within a RAID-5 stripe are no longer part of a single file (and thus not logically related), full stripe writes become uncommon. Thus with the block allocation policies described so far, most writes will be to partial stripes; such *small writes* have the well known performance problem of requiring four disk operations for every block written [77].

To address the small write problem in D-GRAID-5, we need a customized block allocation policy. While the allocation policies described in Section 6.6 are targeted at preserving the logical contiguity perceived by the file system, D-GRAID-5 requires a policy that minimizes the impact of small writes. One example

of such a policy is log-structured allocation [91, 117], where blocks are not written in place, but allocated from empty *segments*, invalidating the old locations.

With such log structured allocation, D-GRAID-5 would simply divide each disk into multiple segments; at any given time, D-GRAID-5 would operate on a *segment stripe*, which comprises of the kth segment in each disk. When a write arrives, the fault isolation module of D-GRAID-5 would decide which disk the block needs to be laid out in, and then would allocate the tail physical block of the corresponding segment to that logical block. Considering that in a typical workload, writes are spread across multiple files, and given that D-GRAID balances space utilization across disks, it is most likely that writes to such multiple files are spread across different segments within the current segment stripe, thus resulting in full stripe writes. Note however that for this technique to be effective, the *log cleaner* should co-ordinate cleaning across the entire set of disks, so that the set of freed segments comprise full segment stripes.

6.8.4 Summary

In summary, we find that the basic layout techniques in D-GRAID are orthogonal to the underlying redundancy mechanism. By building on top of any physical redundancy scheme, D-GRAID strictly improves the availability of the storage array. However, custom policies (*e.g.*, for access driven diffusion, physical block allocation, etc.) often make D-GRAID more effective for a given redundancy mechanism.

6.9 Discussion: The Impact of Being Wrong

As described in Section 6.6, there is a fair amount of complexity in identifying the logical file to which a block belongs, in order to place it in the correct home site for graceful degradation. An interesting question that arises in the light of such complexity is: what happens if D-GRAID makes a wrong inference? For example, what happens if D-GRAID permanently associates a block with the wrong file, and thus places it in the wrong home site? Such incorrect inferences affect different parts of the D-GRAID design differently.

The graceful degradation component of D-GRAID is quite robust to incorrect inferences; an incorrect association of a block to the wrong file would only affect fault isolation, and not impact correctness. Even if D-GRAID miscalculates a large fraction of its associations, the reliability of the resulting storage layout will still be strictly better than the corresponding traditional RAID level. This is because

D-GRAID builds on top of existing RAID redundancy. An incorrect association may lead to a layout that is not completely fault isolated, but such a layout will still exhibit better fault isolation compared to traditional RAID. Thus even in the face of incorrect inference, the storage system correctness is not affected, thus making D-GRAID an ideal candidate to make aggressive use of such semantic information.

In contrast, the live block recovery component of D-GRAID does depend on semantic information for correctness. Although it requires only a conservative estimate of the set of live blocks in the volume, D-GRAID requires this estimate to be *strictly* conservative; a live block should never be inferred to be dead, since that could lead to loss of data. However, as described in Section 6.6, tracking such block liveness information conservatively is quite simple, and thus is straightforward to realize. If liveness tracking is not accurate under a different file system, D-GRAID could still reconstruct the blocks it thinks are live first, and to be conservative, recover the remaining blocks as well in the background.

Thus, D-GRAID requires accuracy only for a very simple piece of semantic information for implementing fast recovery. Much of the design and complexity of D-GRAID is on fault isolation for graceful degradation; this component is much more robust to incorrect inference, and cannot be "wrong" in any bad way.

6.10 Summary

D-GRAID turns the simple binary failure model found in most storage systems into a continuum, increasing the availability of storage by continuing operation under partial failure and quickly restoring live data after a failure does occur. In this chapter, we have shown the potential benefits of D-GRAID, explored the limits of semantic knowledge, and have shown how a successful D-GRAID implementation can be achieved despite these limits. Through simulation and the evaluation of a prototype implementation, we have found that D-GRAID can be built in a semantically-smart disk system without any file system modification, and that it delivers graceful degradation and live-block recovery, and, through access-driven diffusion, good performance.

Chapter 7

Exploiting Liveness Knowledge in FADED

> "*Life is pleasant. Death is peaceful. It's the transition that's troublesome.*"
> Isaac Asimov

D-GRAID, presented in the previous chapter, was naturally amenable to approximate semantic inference, because wrong information cannot lead to correctness issues. In this chapter, we present a more aggressive piece of SDS functionality that has stringent requirements on correctness. The specific functionality involves performing *secure deletion* within the disk system by exploiting knowledge on liveness of data [101]. We discuss techniques for tracking various forms of liveness in general, and then apply those techniques in the context of secure deletion.

7.1 Introduction

Liveness of blocks is a key piece of semantic information that is useful in storage systems. Previous work has demonstrated the utility of such knowledge: dead blocks can be used to store rotationally optimal replicas of data [122] or to provide zero-cost writes [115]. In this chapter, we describe how a semantically-smart disk system can acquire information on liveness, and demonstrate its application with a case study.

Before presenting the specific techniques for tracking liveness, we first formalize the notion of liveness within storage. Specifically, we identify three useful classes of liveness (content, block, and generation liveness), and present techniques for explicit and implicit tracking of each type. Because techniques for tracking liveness are dependent on the characteristics of the file system, we study a range of file

systems, including ext2, ext3 and VFAT; in doing so, we identify key file system properties that impact the feasibility and complexity of such techniques.

We then demonstrate the utility and applicability of the techniques by describing the design, implementation, and evaluation of a prototype *secure deleting disk* that infers logical deletes occurring at the file system, and shreds the deleted blocks, making deleted data irrecoverable [44]. Unlike the D-GRAID case study explored in the previous chapter, secure delete poses new challenges due to its extreme requirements on the type and accuracy of liveness information. Through the case study, we show that even underneath modern asynchronous file systems, one can implement SDS functionality that has extreme correctness requirements.

Finally, we compare the SDS approach of tracking liveness with an alternative approach where the interface to the disk system is changed to add an explicit "free" command. This comparison helps bring out the complexity and performance costs of the semantic inference approach in comparison to the explicit approach.

This chapter is organized as follows. We first present an extended motivation (§7.2), followed by a taxonomy of liveness (§7.3), and a list of file system properties that impact techniques for tracking liveness information (§7.4). We then discuss the specific techniques for tracking liveness (§7.5), and present the secure delete case study (§7.6). We then describe our initial experience with liveness tracking under NTFS, a closed-source file system (§7.7). Finally, we describe the explicit notification approach (§7.8), compare the merits of the two approaches (§7.9), and conclude (§7.10).

7.2 Extended Motivation

Liveness information enables a variety of functionality and performance enhancements within the storage system. Most of these enhancements cannot be implemented at higher layers because they require low-level control available only within the storage system.

Eager writing: Workloads that are write-intensive can run faster if the storage system is capable of *eager writing*, *i.e.*, writing to "some free block closest to the disk arm" instead of the traditional in-place write [30, 115]. However, in order to select the closest block, the storage system needs information on which blocks are live. Existing proposals function well as long as there exist blocks that were never written to; once the file system writes to a block, the storage system cannot identify subsequent death of the block as a result of a delete. A disk empowered with liveness information can be more effective at eager writing.

Adaptive RAID: Information on block liveness within the storage system can also

facilitate dynamic, adaptive RAID schemes such as those in the HP AutoRAID system [117]; AutoRAID utilizes free space to store data in RAID-1 layout, and migrates data to RAID-5 when it runs short of free space. Knowledge of block death can make such schemes more effective.

Optimized layout: Techniques to optimize on-disk layout transparently within the storage system have been well explored. Adaptive reorganization of blocks within the disk [93] and replication of blocks in rotationally optimal locations [122] are two examples. Knowing which blocks are free can greatly facilitate such techniques; live blocks can be collocated together to minimize seeks, or the "free" space corresponding to dead blocks can be used to hold rotational replicas.

Smarter NVRAM caching: Buffering writes in NVRAM is a common optimization in storage systems. For synchronous write workloads that do not benefit much from in-memory delayed writes within the file system, NVRAM buffering improves performance by absorbing multiple overwrites to a block. However, in delete-intensive workloads, unnecessary disk writes can still occur; in the absence of liveness information, deleted blocks occupy space in NVRAM and need to be written to disk when the NVRAM fills up. From real file system traces [90], we found that up to 25% of writes are deleted *after* the typical delayed write interval of 30 seconds, and thus will be unnecessarily written to disk. Knowledge about block death within storage removes this overhead.

Intelligent prefetching: Modern disks perform aggressive prefetching; when a block is read, the entire track in which the block resides is often prefetched [97], and cached in the internal disk cache. In an aged (and thus, fragmented) file system, only a subset of blocks within a track may be live, and thus, caching the whole track may result in suboptimal cache space utilization. Although reading in the whole track is still efficient for disk I/O, knowledge about liveness can enable the disk to selectively cache only those blocks that are live.

Faster recovery: Liveness information enables faster recovery in storage arrays. A storage system can reduce reconstruction time during disk failure by only reconstructing blocks that are live within the file system, as described in the previous chapter.

Self-securing storage: Liveness information in storage can help build intelligent security functionality in storage systems. For example, a storage level intrusion detection system (IDS) provides another perimeter of security by monitoring traffic, looking for suspicious access patterns such as deletes or truncates of log files [79]; detecting these patterns requires liveness information.

Secure delete: The ability to delete data in a manner that makes recovery impossible is an important component of data security [10, 44, 52]. Government

Liveness type	Description	Currently possible?	Example utility
Content	Data within block	Yes	Versioning
Block	Whether a block holds valid data currently	No	Eager write, fast recovery
Generation	Block's lifetime in the context of a file	No	Secure delete, storage IDS

Table 7.1: **Forms of liveness.**

regulations require strong guarantees on sensitive data being "forgotten", and such requirements are expected to become more widespread in both government and industry in the near future [2]. Secure deletion requires low-level control on block placement that is available only within the storage system; implementing storage level secure delete requires liveness information within the storage system. We explore secure deletion further in Section 7.6.

7.3 Liveness in Storage: A Taxonomy

Having discussed the utility of liveness information within a storage system, we now present a taxonomy of the forms of liveness information that are relevant to storage. Such liveness information can be classified along three dimensions: *granularity*, *accuracy*, and *timeliness*.

7.3.1 Granularity of liveness

Depending on the specific storage-level enhancement that utilizes liveness information, the logical unit of liveness to be tracked can vary. We identify three granularities at which liveness information is meaningful and useful: content, block and generation. A summary is presented in Table 7.1.

Content liveness

Content liveness is the simplest form of liveness. The unit of liveness is the actual data in the context of a given block; thus, "death" at this granularity occurs on every overwrite of a block. When a block is overwritten with new data, the storage system can infer that the old contents are dead. An approximate form of content liveness is readily available in existing storage systems, and has been explored in

previous work; for example, Wang *et al.*'s virtual log disk frees the past location of a block when the block is overwritten with new contents [115]. Tracking liveness at this granularity is also useful in on-disk versioning, as seen in self-securing storage systems [108]. However, to be completely accurate, the storage system also needs to know when a block is freed within the file system, since the contents stored in that block are dead even without it being overwritten.

Block liveness

Block liveness tracks whether a given disk block currently contains valid data, *i.e.*, data that is accessible through the file system. The unit of interest in this case is the "container" instead of the "contents". Block liveness is the granularity required for many applications such as intelligent caching, prefetching, and eager writing. For example, in deciding whether to propagate a block from NVRAM to disk, the storage system just needs to know whether the block is live at this granularity. This form of liveness information cannot be tracked in traditional storage systems because the storage system is unaware of which blocks the file system thinks are live. However, a weak form of this liveness can be tracked; a block that was never written to can be inferred to be dead.

Generation liveness

The generation of a disk block is the lifetime of the block in the context of a certain file. Thus, by death of a generation, we mean that a block that was written to disk (at least once) in the context of a certain file becomes either free or is reallocated to a different file. Tracking generation liveness ensures that the disk can detect every logical file system delete of a block whose contents had reached disk in the context of the deleted file. An example of a storage level functionality that requires generation liveness is secure delete, since it needs to track not just whether a block is live, but also whether it contained data that belonged to a file generation that is no longer alive. Another application that requires generation liveness information is storage-based intrusion detection. Generation liveness cannot be tracked in existing storage systems.

7.3.2 Accuracy of liveness information

The second dimension of liveness is accuracy, by which we refer to the degree of trust the disk can place in the liveness information available to it. Inaccuracy in liveness information can lead the disk into either overestimating or underestimating the

set of live entities (blocks or generations). The degree of accuracy required varies with the specific storage application. For example, in delete-squashing NVRAM, it is acceptable for the storage system to slightly overestimate the set of live blocks, since it is only a performance issue and not a correctness issue; on the other hand, underestimating the set of live blocks is catastrophic since the disk would lose valid data. Similarly, in generation liveness detection for secure delete, it is acceptable to miss certain intermediate generation deaths of a block as long as the latest generation death of the block is known.

7.3.3 Timeliness of information

The third and final axis of liveness is timeliness, which defines the time between a death occurring within the file system and the disk learning of the death. The periodicity with which the file system writes metadata blocks imposes a bound on the timeliness of the liveness information inferred. In many applications, such as eager writing and delete-aware caching, this delayed knowledge of liveness is acceptable, as long as the information has not changed in the meantime. However, in certain applications such as secure delete, timely detection may provide stronger guarantees.

7.4 File System Model

While D-GRAID in the previous chapter considered an asynchronous file system, the file system model it was based on assumed the worst-case effects of asynchrony, namely, arbitrary reordering and delayed writes. However, some modern file systems provide certain kinds of guarantees in terms of how they update data to disk. In this case study, we study the range of such dynamic update properties that hold in modern file systems, and study how such properties affect our techniques for semantic inference. Towards this goal, we experimented underneath three different file systems: ext2, ext3, and VFAT. We have also experimented with NTFS, but only on a limited scale due to lack of source code access; our NTFS experience is described in Section 7.7. Given that ext2 has two modes of operation (synchronous and asynchronous modes) and ext3 has three modes (writeback, ordered, and data journaling modes), all with different update behaviors, we believe these form a rich set of file systems.

Based on our experience with the above file systems, we identify some key high level behavioral properties of a file system that are relevant in the context of tracking liveness information. Table 7.2 summarizes these properties. In the next

Property	Ext2	Ext2+sync	VFAT	Ext3-wb	Ext3-ord	Ext3-data
Reuse ordering		×		×	×	×
Block exclusivity	×	×	×			
Generation marking	×	×		×	×	×
Delete suppression	×	×	×	×	×	×
Consistent metadata				×	×	×
Data-metadata coupling						×

Table 7.2: **File system properties.** *The table summarizes the various properties exhibited by each of the file systems we study.*

two sections, we will discuss how these properties influence techniques for storage-level liveness tracking.

Reuse ordering

If the file system guarantees that it will not reuse disk blocks until the freed status of the block (*e.g.*, bitmaps or other metadata that pointed to the block) reaches disk, the file system exhibits *reuse ordering*. This property is necessary (but not sufficient) to ensure data integrity; in the absence of this property, a file could end up with partial contents from some other deleted file after a crash, even in a journaling file system. While VFAT and the asynchronous mode of ext2 do not have reuse ordering, all three modes of ext3, and ext2 in synchronous mode, exhibit reuse ordering.

Block exclusivity

Block exclusivity requires that for every disk block, there is at most one dirty copy of the block in the file system cache. It also requires that the file system employ adequate locking to prevent any update to the in-memory copy while the dirty copy is being written to disk. This property holds for certain file systems such as ext2 and VFAT. However, ext3 does not conform to this property. Because of its snapshot-based journaling, there can be two dirty copies of the same metadata block, one for the “previous” transaction being committed and the other for the current transaction.

Generation marking

The *generation marking* property requires that the file system track reuse of file pointer objects (*e.g.*, inodes) with version numbers. Both the ext2 and ext3 file systems conform to this property; when an inode is deleted and reused for a different file, the version number of the inode is incremented. VFAT does not exhibit this property.

Delete suppression

A basic optimization found in most file systems is to suppress writes of deleted blocks. All file systems we discuss obey this property for data blocks. VFAT does not obey this property for directory blocks.

Consistent metadata

This property indicates whether the file system conveys a consistent metadata state to the storage system. All journaling file systems exhibit the consistent metadata property; transaction boundaries in their on-disk log implicitly convey this information. Ext2 and VFAT do not exhibit this property.

Data-metadata coupling

Data-metadata coupling builds on the consistent metadata property, and it requires the notion of consistency to be extended also to data blocks. In other words, a file system conforming to this property conveys a consistent metadata state together with the set of data blocks that were dirtied in the context of that transaction. Among the file systems we consider, only ext3 in data journaling mode conforms to this property.

7.5 Techniques for Liveness Detection

In this section, we analyze various issues in inferring liveness information from within the storage system. Because semantic inference is file system dependent, we discuss the feasibility and generality of implicit liveness detection by considering three different file systems: ext2, ext3, and VFAT. In Section 7.7, we discuss our initial experience with detecting liveness underneath the Windows NTFS file system.

Among the different forms of liveness we address, we only consider the granularity and accuracy axes mentioned in Section 7.3. Along the accuracy axis, we

consider *accurate* and *approximate* inferences; the *approximate* instance refers to a strict *over-estimate* of the set of live entities. On the timeliness axis, we address the more common (and complex) case of lack of timely information; under most modern file systems that are asynchronous and hence delay metadata updates, timeliness is not guaranteed.

7.5.1 Content liveness

As discussed in Section 7.3, when the disk observes a write of new contents to a live data block, it can infer that the previous contents stored in that block has suffered a content death. However, to be completely accurate, content liveness inference requires information on block liveness.

7.5.2 Block liveness

Block liveness information enables a storage system to know whether a given block contains valid data at any given time. To track block liveness, the storage system monitors updates to structures tracking allocation. In ext2 and ext3, there are specific data bitmap blocks which convey this information; in VFAT this information is embedded in the FAT itself, as each entry in the FAT indicates whether or not the corresponding block is free. Thus, when the file system writes an allocation structure, the storage system examines each entry and concludes that the relevant block is either dead or live.

Because allocation bitmaps are buffered in the file system and written out periodically, the liveness information that the storage system has is often stale, and does not account for new allocations (or deletes) that occurred during the interval. Table 7.3 depicts a time line of operations which leads to an incorrect inference by the storage system. The bitmap block M_B tracking the liveness of B is written in the first step indicating B is dead. Subsequently, B is allocated to a new file I_1 and written to disk while M_B (now indicating B as live) is still buffered in memory. At this point, the disk wrongly believes that B is dead while the on-disk contents of B are actually valid.

To address this inaccuracy, the disk tracks a *shadow copy* of the bitmaps internally, as described in Chapter 6; whenever the file system writes a bitmap block, the disk updates its shadow copy with the copy written. In addition, whenever a data block is written to disk, the disk pro-actively sets the corresponding bit in its shadow bitmap copy to indicate that the block is live. In the above example, the write of B leads the disk to believe that B is live, thus preventing the incorrect conclusion from being drawn.

Operation	In-memory	On-disk
Initial	$M_B \Rightarrow B$ free	
M_B write to disk		B free
I_1 alloc	$I_1 \rightarrow B$	
	$M_B \Rightarrow B$ alloc	
B write to disk		B written
Liveness belief	B live	B free

Table 7.3: **Naive block liveness detection.** *The table depicts a time line of events that leads to an incorrect liveness inference. This problem is solved by the shadow bitmap technique.*

File system properties for block liveness

The shadow bitmap technique tracks block liveness accurately only underneath file systems that obey either the block exclusivity or data-metadata coupling property.

Block exclusivity guarantees that when a bitmap block is written, it reflects the current liveness state of the relevant blocks. If the file system tracks multiple snapshots of the bitmap block (*e.g.*, ext3), it could write an old version of a bitmap block M_B (indicating B is dead) after a subsequent allocation and write of B. The disk would thus wrongly infer that B is dead while in fact the on-disk contents of B are valid, since it belongs to a newer snapshot; such uncertainty complicates block liveness inference.

If the file system does not exhibit block exclusivity, block liveness tracking requires the file system to exhibit data-metadata coupling, *i.e.*, to group metadata blocks (*e.g.*, bitmaps) with the actual data block contents in a single consistent group; file systems typically enforce such consistent groups through transactions. By observing transaction boundaries, the disk can then reacquire the temporal information that was lost due to lack of block exclusivity. For example, in ext3 data journaling mode, a transaction would contain the newly allocated data blocks together with the bitmap blocks indicating the allocation as part of one consistent group. Thus, at the commit point, the disk conclusively infers liveness state from the state of the bitmap blocks in that transaction. Since data writes to the actual in-place locations occur only after the corresponding transaction commits, the disk is guaranteed that until the next transaction commit, all blocks marked dead in the previous transaction will remain dead. In the absence of data-metadata coupling, a newly allocated data block could reach its in-place location before the corresponding transaction commits, and thus will become live in the disk before the disk detects it.

Operation	In-memory	On-disk
Initial	$M_B \Rightarrow B$ alloc	B live
	$I_1 \rightarrow B$	$I_1 \rightarrow B$
B write to disk		B written
I_1 delete	$M_B \Rightarrow B$ free	
I_2 alloc	$I_2 \rightarrow B$	
	$M_B \Rightarrow B$ alloc	
M_B write to disk		B live
Liveness belief		(Missed gen. death)

Table 7.4: **Missed generation death under block liveness.** *The table shows a scenario to illustrate that simply tracking block liveness is insufficient to track generation deaths.*

For accuracy, block liveness also requires the file system to conform to the delete suppression property; if delete suppression does not hold, a write of a block does not imply that the file system views the block as live, and thus the shadow bitmap technique will overestimate the set of live blocks until the next bitmap write. From Table 7.2, ext2, VFAT, and ext3 in data journaling mode thus readily facilitate block liveness detection.

7.5.3 Generation liveness

Generation liveness is a stronger form of liveness than block liveness, and hence builds upon the same shadow bitmap technique. With generation liveness, the goal is to find, for each on-disk block, whether a particular "generation" of data (*e.g.*, that corresponding to a particular file) stored in that block is dead. Thus, block liveness is a special case of generation liveness; a block is dead if the latest generation that was stored in it is dead. Conversely, block liveness information is not sufficient to detect generation liveness because a block currently live could have stored a dead generation in the past. Table 7.4 depicts this case. Block B initially stores a generation of inode I_1, and the disk thinks that block B is live. I_1 is then deleted, freeing up B, and B is immediately reallocated to a different file I_2. When M_B is written the next time, B continues to be marked live. Thus, the disk missed the generation death of B that occurred between these two bitmap writes.

Generation liveness under reuse ordering

Although tracking generation liveness is in general more challenging, a file system that follows the reuse ordering property makes it simple to track. With reuse or-

dering, before a block is reused in a different file, the deleted status of the block reaches disk. In the above example, before B is reused in I_2, the bitmap block M_B will be written, and thus the disk can detect that B is dead. In the presence of reuse ordering, tracking block liveness accurately implies accurate tracking of generation liveness. File systems such as ext3 that conform to reuse ordering, thus facilitate *accurate* tracking of generation liveness.

Generation liveness without reuse ordering

Underneath file systems such as ext2 or VFAT that do not exhibit the reuse ordering property, tracking generation liveness requires the disk to look for more detailed information. Specifically, the disk needs to monitor writes to metadata objects that link blocks together into a single logical file (such as the inode and indirect blocks in ext2, the directory and FAT entries in VFAT). The disk needs to explicitly track the "generation" a block belongs to. For example, when an inode is written, the disk records that the block pointers belong to the specific inode.

With this extra knowledge about the file to which each block belongs, the disk can identify generation deaths by looking for *changes in ownership*. For example, in Table 7.4, if the disk tracked that B belongs to I_1, then eventually when I_2 is written, the disk will observe a change of ownership, because I_2 owns a block that I_1 owned in the past; the disk can thus conclude that a generation death must have occurred in between.

A further complication arises when instead of being reused in I_2, B is reused again in I_1, now representing a new file. Again, since B now belongs to a new generation of I_1, this scenario has to be detected as a generation death, but the ownership change monitor would miss it. To detect this case, we require the file system to track reuse of inodes (*i.e.*, the generation marking property). Ext2 already maintains such a version number, and thus enables detection of these cases of generation deaths. With version numbers, the disk now tracks for each block the "generation" it belonged to (the generation number is a combination of the inode number and the version number). When the disk then observes an inode written with an incremented version number, it concludes that all blocks that belonged to the previous version of the inode should have incurred a generation death. We call this technique *generation change monitoring*.

Finally, it is pertinent to note that the generation liveness detection through generation change monitoring is only *approximate*. Let us assume that the disk observes that block B belongs to generation G_1, and at a later time observes that B belongs to a different generation G_2. Through generation change monitoring, the disk can conclude that there was a generation death of B that occurred in between.

Liveness type	Properties
Block_{Approx}	Block exclusivity *or* Data-metadata coupling
$\text{Block}_{Accurate}$	[Block_{Approx}] + Delete suppression
$\text{Generation}_{Approx}$	[Block_{Approx}] + Generation marking
$\text{Generation}_{Accurate}$	[$\text{Block}_{Accurate}$] + Reuse ordering

Table 7.5: **FS properties for liveness inference.** *$Approx$ indicates the set of live entities is over-estimated.*

However, the disk cannot know exactly *how many* generation deaths occurred in the relevant period. For example, after being freed from G_1, B could have been allocated to G_3, freed from G_3 and then reallocated to G_2, but the disk never saw G_3 owning B due to delayed write of G_3. However, as we show in our case study, this weaker form of generation liveness is still quite useful.

A summary of the file system properties required for various forms of liveness inference is presented in Table 7.5.

7.6 Case Study: Secure Delete

To demonstrate our techniques for imparting liveness to storage, we present the design, implementation, and evaluation of a *secure deleting disk*. There are two primary reasons why we chose secure deletion as our case study. First, secure delete requires tracking of generation liveness, which is the most challenging to track. Second, secure delete uses the liveness information in a context where correctness is paramount. A false positive in detecting a delete would lead to irrevocable deletion of valid data, while a false negative would result in the long-term recoverability of deleted data (a violation of secure deletion guarantees).

Our secure deletion prototype is called FADED (A File-Aware Data-Erasing Disk); FADED works underneath three different file systems: ext2, VFAT, and ext3. Because of its complete lack of ordering guarantees, ext2 presented the most challenges. Specifically, since ext2 does not have the reuse ordering property, detecting generation liveness requires tracking generation information within the disk, as described in Section 7.5.3. We therefore mainly focus on the implementation of FADED underneath ext2, and finally discuss some key differences in our implementation for other file systems.

7.6.1 Goals of FADED

The desired behavior of FADED is as follows: for every block that reaches the disk in the context of a certain file F, the delete of file F should trigger a secure overwrite (*i.e.*, *shred*) of the block. This behavior corresponds to the notion of *generation liveness* defined in Section 7.3. A *shred* involves multiple overwrites to the block with specific patterns so as to erase remnant magnetic effects of past layers (that could otherwise be recovered through techniques such as magnetic scanning tunneling microscopy [44]). Recent work suggests that two such overwrites are sufficient to ensure non-recoverability in modern disks [52].

Traditionally, secure deletion is implemented within the file system [10, 105, 106]; however, such implementations are unreliable given modern storage systems. First, for high security, overwrites need to be *off-track* writes (*i.e.*, writes straggling physical track boundaries), which external erase programs (*e.g.*, the file system) cannot perform [51]. Further, if the storage system buffers writes in NVRAM [117], multiple overwrites done by the file system may be collapsed into a single write to the physical disk, making the overwrites ineffective. Finally, in the presence of block migration [29, 117] within the storage system, an overwrite by the file system will only overwrite the current block location; stray copies of deleted data could remain. Thus, the storage system is the proper locale to implement secure deletion.

Note that FADED operates at the granularity of an entire volume; there is no control over which individual files are shredded. However, this limitation can be dealt with by storing "sensitive" files in a separate volume on which the secure delete functionality is enabled.

7.6.2 Basic operation

As discussed in Section 7.5.3, FADED monitors writes to inode and indirect blocks and tracks the inode generation to which each block belongs. It augments this information with the block liveness information it collects through the shadow bitmap technique. Note that since ext2 obeys the block exclusivity and delete suppression properties, block liveness detection is reliable. Thus, when a block death is detected, FADED can safely shred that block.

On the other hand, if FADED detects a generation death through the ownership change or generation change monitors (*i.e.*, the block is live according to the block liveness module), FADED cannot simply shred the block, because FADED does not know if the current contents of the block belong to the generation that was deleted, or to a new generation that was subsequently allocated the same block due to block

reuse. If the current contents of the block are valid, a shredding of the block would be catastrophic.

We deal with such uncertainty through a conservative approach to generation-death inference. By being conservative, we convert an apparent correctness problem into a performance problem, *i.e.*, we may end up performing more overwrites than required. Fundamental to this approach is the notion of a *conservative overwrite*.

Conservative overwrites

A conservative overwrite of block B erases past layers of data on the block, but leaves the current contents of B intact. Thus, even if FADED does not know whether a subsequent valid write occurred after a predicted generation death, a conservative overwrite on block B will be safe; it can never shred valid data. To perform a conservative overwrite of block B, FADED reads the block B into non-volatile RAM, then performs a normal secure overwrite of the block with the specific pattern, and ultimately restores the original data back into block B.

The problem with a conservative overwrite is that if the block contents that are restored after the conservative overwrite are in fact the old data (which had to be shredded), the conservative overwrite was ineffective. In this case, FADED can be guaranteed to observe one of two things. First, if the block had been reused by the file system for another file, the new, valid data will be written eventually (*i.e.*, within the delayed write interval of the file system). When FADED receives this new write, it buffers the write, and before writing the new data to disk, FADED performs a shred of the concerned block once again; this time, FADED knows that it need not restore the old data, because it has the more recent contents of the block. To identify which writes to treat in this special manner, FADED tracks the list of blocks that were subjected to a conservative overwrite in a *suspicious blocks* list, and a write to a block in this list will be committed only after a secure overwrite of the block; after the second overwrite, the block is removed from the suspicious list. Note that the suspicious list needs to be stored persistently, perhaps in NVRAM, in order to survive crashes.

Second, if the block is not reused by the file system immediately, then FADED is guaranteed to observe a bitmap reset for the corresponding block, which will be flagged as a block death by the block liveness detector. Since block liveness tracking is reliable, FADED can now shred the block again, destroying the old data. Thus, in both cases of wrongful restore of old data, FADED is guaranteed to get another opportunity to make up for the error.

Cost of conservatism

Conservative overwrites come with a performance cost; every conservative overwrite results in the concerned block being treated as "suspicious", regardless of whether the data restored after the conservative overwrite was the old or new data, because FADED has no information to find it at that stage. Because of this uncertainty, even if the data restored were the new data (and hence need not be overwritten again), a subsequent write of the block in the context of the same file would lead to a redundant shredding of the block. Here we see one example of the performance cost FADED pays to circumvent the lack of perfect information.

7.6.3 Coverage of deletes

In the previous subsection, we showed that for all generation deaths detected, FADED ensures that the appropriate block version is overwritten, without compromising valid data. However, for FADED to achieve its goals, these detection techniques must be *sufficient* to identify *all* cases of deletes at the file system level that need to be shredded. In this section, we show that FADED can indeed detect all deletes, but requires two minor modifications to ext2.

Undetectable deletes

Because of the weak properties of ext2, certain deletes can be missed by FADED. We present the two specific situations where identification of deletes is impossible, and then propose minor changes to ext2 to fix those scenarios.

File truncates: The generation change monitor assumes that the version number of the inode is incremented when the inode is reused. However, the version number in ext2 is only incremented on a complete delete and reuse; partial truncates do not affect the version number. Thus if a block is freed due to a partial truncate and is reassigned to the same file, FADED misses the generation death. Although such a reuse after a partial truncate could be argued as a logical overwrite of the file (and thus, not a *delete*), we adopt the more complex (and conservative) interpretation of treating it as a delete.

To handle such deletes, we propose a small change to ext2; instead of incrementing the version number on a reallocation of the inode, we increment it on every truncate. Alternatively, we could introduce a separate field to the inode that tracks this version information. This is a non-intrusive change, but is effective at providing the disk with the requisite information. This technique could result in extra overwrites in the rare case of partial truncates, but correctness is guaranteed

Operation	In-memory	On-disk
Initial	$I_1 \rightarrow B^{Ind}$	$I_1 \rightarrow B^{Ind}$
I_1 delete	B free	
I_2 alloc	$I_2 \rightarrow B$	
B write to disk		$I_1 \rightarrow B^{Ind}$ (wrong type)

Table 7.6: **Misclassified indirect block.** *The table shows a scenario where a normal data block is misclassified as an indirect block.* B^{Ind} *indicates that* B *is treated as an indirect block. Reuse ordering for indirect blocks prevents this problem.*

because the "spurious" overwrites would be conservative and would leave data intact.

Reuse of indirect blocks: A more subtle problem arises due to the presence of indirect pointer blocks. Indirect blocks share the data region of the file system with other user data blocks; thus the file system can reuse a normal user data block as an indirect block and vice versa. In the presence of such *dynamic typing*, the disk cannot reliably identify an indirect block, as shown in Chapter 4.

The only way FADED can identify a block B as an indirect block is when it observes an inode I_1 that contains B in its indirect pointer field. FADED then records the fact that B is an indirect block. However, when it later observes a write to B, FADED cannot be certain that the contents indeed are those of the indirect block, because in the meanwhile I_1 could have been deleted, and B could have been reused as a user data block in a different inode I_2. This scenario is illustrated in Table 7.6.

Thus, FADED cannot trust the block pointers in a suspected indirect block; this uncertainty can lead to missed deletes in certain cases. To prevent this occurrence, a data block should never be misclassified as an indirect block. To ensure this, before the file system allocates, and immediately after the file system frees an indirect block B^{Ind}, the concerned data bitmap block $M_{B^{Ind}}$ should be flushed to disk, so that the disk will know that the block was freed. Note that this is a weak form of reuse ordering only for indirect blocks. As we show later, this change has very little impact on performance, since indirect blocks tend to be a very small fraction of the set of data blocks.

Practicality of the changes: The two changes discussed above are minimal and non-intrusive; the changes together required modification of 12 lines of code in ext2. Moreover, they are required only because of the weak ordering guarantees of ext2. In file systems such as ext3 which exhibit reuse ordering, these changes are

Operation	In-memory	On-disk
Initial	B free	B free
I_1 alloc	$I_1 \rightarrow B$	
B write to disk		B written
I_1 delete	B free	
I_2 alloc	$I_2 \rightarrow B$	
I_2 write to disk		$I_2 \rightarrow B$
		(Missed delete of B)

Table 7.7: **Missed delete due to an orphan write.** *The table illustrates how a delete can be missed if an orphan block is not treated carefully. Block B, initially free, is allocated to I_1 in memory. Before I_1 is written to disk, B is written. I_1 is then deleted and B reallocated to I_2. When I_2 is written, FADED would associate B with I_2 and would miss the overwrite of B.*

not required. Our study of ext2 is aimed as a limit study of the minimal set of file system properties required to reliably implement secure deletion at the disk.

Orphan allocations

Orphan allocations refer to a case where the file system considers a block dead while the disk considers it live. Assume that a block is newly allocated to a file and is written to disk in the context of that file. If a crash occurs at this point (but before the metadata indicating the allocation is written to disk), the disk would assume that the block is live, but on restart, the file system views the block as dead. Since the on-disk contents of the block belong to a file that is no longer extant in the file system, the block has suffered a generation death, but the disk does not know of this.

Implicit block liveness tracking in FADED already addresses this in the case of ext2; when ext2 recovers after a crash, the fsck utility writes out a copy of all bitmap blocks; the block liveness monitor in FADED will thus detect death of those orphan allocations.

Orphan writes

Due to arbitrary ordering in ext2, FADED can observe a write to a newly allocated data block before it observes the corresponding owning inode. Such *orphan writes* need to be treated carefully because if the owning inode is deleted before being written to disk, FADED will never know that the block once belonged to that

inode. If the block is reused in another inode, FADED would miss overwriting the concerned block which was written in the context of the old inode. Table 7.7 depicts such a scenario.

One way to address this problem is to *defer* orphan block writes until FADED observes an owning inode [102], a potentially memory-intensive solution. Instead, we use the suspicious block list used in conservative overwrites to also track orphan blocks. When FADED observes a write to an orphan block B, it marks B suspicious; when a subsequent write arrives to B, the old contents are shredded. Thus, if the inode owning the block is deleted before reaching disk, the next write of the block in the context of the new file will trigger the shred. If the block is not reused, the bitmap reset will indicate the delete.

This technique results in a redundant secure overwrite anytime an orphaned block is overwritten by the file system in the context of the same file, again a cost we pay for conservatism. Note that this overhead is incurred only the first time an orphan block is overwritten.

Delayed overwrites

Multiple overwrites of the same block cause additional disk I/Os that can hurt performance if incurred on the critical path. For better performance, FADED delays overwrites until idle time in the workload [38] (or optionally, until up to n minutes of detection). Thus, whenever FADED decides to shred a block, it just queues it; a low priority thread services this queue if FADED had not observed useful foreground traffic for more than a certain duration. Delayed overwrites help FADED to present writes to the disk in a better, sequential ordering, besides reducing the impact on foreground performance. Delaying also reduces the number of overwrites if the same block is deleted multiple times. The notion of conservative overwrites is crucial to delaying overwrites arbitrarily, even after the block that had to be overwritten is written in the context of a new file. Note that if immediate shredding is required, the user needs to perform a `sync`.

A summary of the key data structures and components of FADED is presented in Figure 7.1.

Guaranteed detection of deletes

We now demonstrate how the basic techniques outlined so far together ensure that FADED captures all relevant cases of deletes. We prove that for every block B that is deleted by the file system after it has reached disk, FADED always overwrites the deleted contents of B.

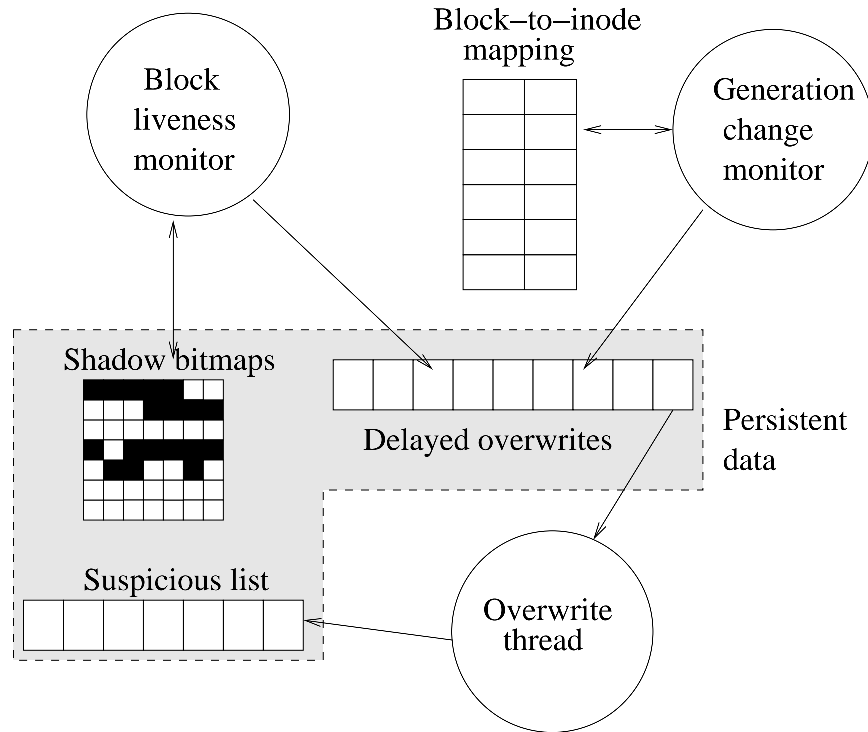


Figure 7.1: **Key components of FADED.**

When a delete of an inode I_1 occurs within ext2, a set of blocks are freed from a file; this results in an increment of the version number of I_1, and the reset of relevant bits in the data bitmap block pertaining to the freed blocks. Let us consider one such block B that is freed. Let us assume that B had already been written to disk in the context of I_1. If B had not been written to disk, the disk does not need to perform any overwrite, so we do not consider that case. Let the bitmap block containing the status of B be M_B, and let B_I be the block containing the inode I_1. Now, there are two possibilities: either B is reused by the file system *before* M_B is written to disk, or B is not reused until the write of M_B.

Case 1: Block B not reused

If B is not reused immediately to a different file, the bitmap block M_B, which is dirtied, will be eventually written to disk, and the disk will immediately know of the delete through the bitmap reset indicator, and thus overwrite B.

Case 2: Block B is reused

Let us now consider the case where B is reused in inode I_2. There are three

possibilities in this case: *at the point of receiving the write of* B, the disk either thinks B belongs to I_1, or it thinks B is free, or that B belongs to some other inode I_x.

Case 2a: Disk thinks $I_1 \rightarrow B$

If the disk knew that $I_1 \rightarrow B$, the disk would have tracked the previous version number of I_1. Thus, when it eventually observes a write of B_I, (which it will, since B_I is dirtied because of the version number increment), the disk will note that the version number of I_1 has increased, and thus would overwrite all blocks that it thought belonged to I_1, which in this case includes B. Thus B would be overwritten, perhaps restoring a newer value. As discussed in Section 7.6.2, even if this was a conservative overwrite, the old contents are guaranteed to be shredded.

Case 2b: Disk thinks B is free

If the disk thinks B is free, it would treat B as an orphan block when it is written, and mark it suspicious. Consequently, when B is written again in the context of the new inode I_2, the old contents of B will be shredded.

Case 2c: Disk thinks $I_x \rightarrow B$

To believe that $I_x \rightarrow B$, the disk should have observed I_x pointing to B at some point before the current write to B.[1] The disk could have observed $I_x \rightarrow B$ either before or after B was allocated to I_1 by the file system.

Case 2c-i: $I_x \rightarrow B$ before $I_1 \rightarrow B$

If the disk observed $I_x \rightarrow B$ *before* it was allocated to I_1, and still thinks $I_x \rightarrow B$ when B is written in the context of I_1, it means the disk never saw $I_1 \rightarrow B$. However, in this case, block B was clearly deleted from I_x at some time in the past in order to be allocated to I_1. This would have led to the version number of I_x incrementing, and thus when the disk observes I_x written again, it would perform an overwrite of B since it thinks B used to belong to I_x.

Case 2c-ii: $I_x \rightarrow B$ after $I_1 \rightarrow B$

If this occurs, it means that I_x was written to disk owning B after B got deleted from I_1 but before B is written. In this case, B will only be written in the context of I_x which is still not deleted, so it does not have to be overwritten. As discussed in Section 9.5, this is true because of the block exclusivity property of ext2.

Note that the case of a block being deleted from a file and then quickly reallocated to the same file is just a special case of *Case 2c* , with $I_1 = I_x$.

Thus, in all cases where a block was written to disk in the context of a certain file, the delete of the block from the file will lead to a shred of the deleted contents.

[1] If indirect block detection was uncertain, the disk can wrongly think $I_x \rightarrow B$ because of a corrupt "pointer" in a false indirect block. However, with our file system modification for reuse ordering in indirect blocks, that case does not occur.

□

7.6.4 FADED for other file systems

We have also implemented FADED underneath other file systems, and in each case, validated our implementation with the same testing methodology as will be described in Section 7.6.5. However, for the sake of brevity, we only point to the key differences we observed relative to ext2.

FADED for VFAT

Like ext2, VFAT also does not conform to reuse ordering, so FADED needs to track generation information for each block in order to detect deletes. One key difference in VFAT compared to ext2 is that there are no pre-allocated, uniquely addressable "inodes", and consequently, no "version" information as well. Dynamically allocated directory blocks contain a pointer to the start block of a file; the FAT chains the start block to the other blocks of the file. Thus, detecting deletes reliably underneath unmodified VFAT is impossible. We therefore introduced an additional field to a VFAT directory entry that tracks a globally unique *generation number*. The generation number gets incremented on every create and delete in the file system, and a newly created file is assigned the current value of generation number. With this small change (29 lines of code) to VFAT, the generation change monitor accurately detects all deletes of interest.

FADED for ext3

Since ext3 exhibits reuse ordering, tracking generation liveness in ext3 is the same as tracking block liveness. However, since ext3 does not obey the block exclusivity property, tracking block liveness accurately is impossible except in the data journaling mode which has the useful property of data-metadata coupling. For the ordered and writeback modes, we had to make a small change: when a metadata transaction is logged, we also made ext3 log a list of data blocks that were allocated in the transaction. This change (95 lines of code), coupled with the reuse ordering property, enables accurate tracking of deletes.

7.6.5 Evaluation

In this section, we evaluate our prototype implementation of secure delete. The enhanced disk is implemented as a pseudo-device driver in the Linux 2.4 kernel, as described in Chapter 3.

Config	Delete	Overwrite	Excess	Miss
No changes	76948	68700	11393	854
Indirect	76948	68289	10414	28
Version	76948	69560	11820	0
Both	76948	67826	9610	0

Table 7.8: **Correctness and accuracy.** *The table shows the number of overwrites performed by the FADED under various configurations of ext2. The columns (in order) indicate the number of blocks deleted within the file system, the total number of logical overwrites performed by FADED, the number of unnecessary overwrites, and the number of overwrites missed by FADED. Note that deletes that occurred before the corresponding data write do not require an overwrite.*

Correctness and accuracy

To test whether our FADED implementation detected all deletes of interest, we instrument the file system to log every delete, and correlate it with the log of writes and overwrites by FADED, to capture cases of unnecessary or missed overwrites. We tested our system on various workloads with this technique, including a few busy hours from the HP file system traces [89]. Table 7.8 presents the results of this study on the trace hour 09 00 of $11/30/00$.

In this experiment, we ran FADED under four versions of Linux ext2. In the first, marked "No changes", a default ext2 file system was used. In "Indirect", we used ext2 modified to obey reuse ordering for indirect blocks. In "Version", we used ext2 modified to increment the inode version number on every truncate, and the "Both" configuration represents both changes (the correct file system implementation required for FADED). The third column gives a measure of the extra work FADED does in order to cope with inaccurate information. The last column indicates the number of missed overwrites; in a correct system, the fourth column should be zero.

We can see that the cost of inaccuracy is quite reasonable; FADED performs roughly 14% more overwrites than the minimal amount. Also note that without the version number modification to ext2, FADED indeed misses a few deletes. The reason no missed overwrites are reported for the "Version" configuration is the rarity of the case involving a misclassified indirect block.

Config	Reads	Writes	Run-time(s)
No changes	394971	234664	195.0
Version	394931	234648	195.5
Both	394899	235031	200.0

Table 7.9: **Impact of FS changes on performance.** *The performance of the various file system configurations under a busy hour of the HP Trace is shown. For each configuration, we show the number of blocks read and written, and the trace run-time.*

	Run-time (s)	
	PostMark	**HP Trace**
Default	166.8	200.0
SecureDelete_2	177.7	209.6
SecureDelete_4	178.4	209.0
SecureDelete_6	179.0	209.3

Table 7.10: **Foreground impact: Postmark and HP trace.** *The run-times for Postmark and the HP trace are shown for FADED, with 2, 4 and 6 overwrite passes. Postmark was configured with 40K files and 40K transactions.*

Performance impact of FS changes

We next evaluate the performance impact of the two changes we made to ext2, by running the same HP trace on different versions of ext2. Table 7.9 shows the results. As can be seen, even with both changes, the performance reduction is only about 2% and the number of blocks written is marginally higher due to synchronous bitmap writes for indirect block reuse ordering. We thus conclude that the changes are quite practical.

Performance of secure delete

We now explore the foreground performance of FADED, and the cost of overwrites.
Foreground performance impact: Tracking block and generation liveness requires FADED to perform extra processing. This cost of reverse engineering directly impacts application performance because it is incurred on the critical path of every disk operation. We quantify the impact of this extra processing required at FADED on foreground performance. Since our software prototype competes for CPU and memory resources with the host, these are worst case estimates of the overheads.

We run the Postmark file system benchmark [55] and the HP trace on a file system running on top of FADED. Postmark is a metadata intensive small-file benchmark, and thus heavily exercises the inferencing mechanisms of FADED. To arrive at a pessimistic estimate, we perform a `sync` at the end of each phase of Postmark, causing all disk writes to complete and account that time in our results. Note that we do not wait for completion of delayed overwrites. Thus, the numbers indicate the performance perceived by the foreground task.

Table 7.10 compares the performance of FADED with a default disk. From the table, we can see that even for 4 or 6 overwrite passes, foreground performance is not affected much. Extra CPU processing within FADED causes only about 4 to 7% lower performance compared to the modified file system running on a normal disk.

Idle time required: We now quantify the cost of performing overwrites for shredding. We first run a microbenchmark that repeatedly creates a 64 MB file, flushes it to disk, deletes it, and then waits for a certain delay period before repeating these steps, for a total of 10 iterations. Varying the amount of delay between each phase, we measure the time for the whole benchmark (including delayed overwrites) to complete.

First, we run this experiment in a way that no block reuse occurs at the file system; each of the 10 iterations creates files in a different directory (ext2 places directories on different block groups). The overwrite process thus has to overwrite about 64 MB for each iteration, no matter how long the overwrites are delayed. The left graph in Figure 7.2 shows the results. As can be expected, for each extra overwrite pass, the extra time required to finish those overwrites is roughly equal to the default benchmark run-time with zero delay, since the amount of data written in a single overwrite pass is the same as that written due to the create. Since FADED delays overwrites and presents them to disk in sequential order, the overwrites achieve close to sequential disk performance.

We next run the same experiment, but allowing block reuse by the file system. This might be more representative of typical workloads where deletes and new creates occur roughly in the same set of directories. In this experiment, all phases create the file in the same directory. Thus, roughly the same set of blocks are repeatedly assigned to the files, and therefore delaying overwrites can yield significant benefit. While overwriting blocks synchronously would have incurred a 64MB overwrite for every phase, delayed overwrites will ideally only incur the overhead once. The right graph in Figure 7.2 shows the benchmark run time on FADED with varying number of overwrite passes. Note that the time taken for overwrites is not fixed as in the case of the previous experiment; this is because with more idle time,

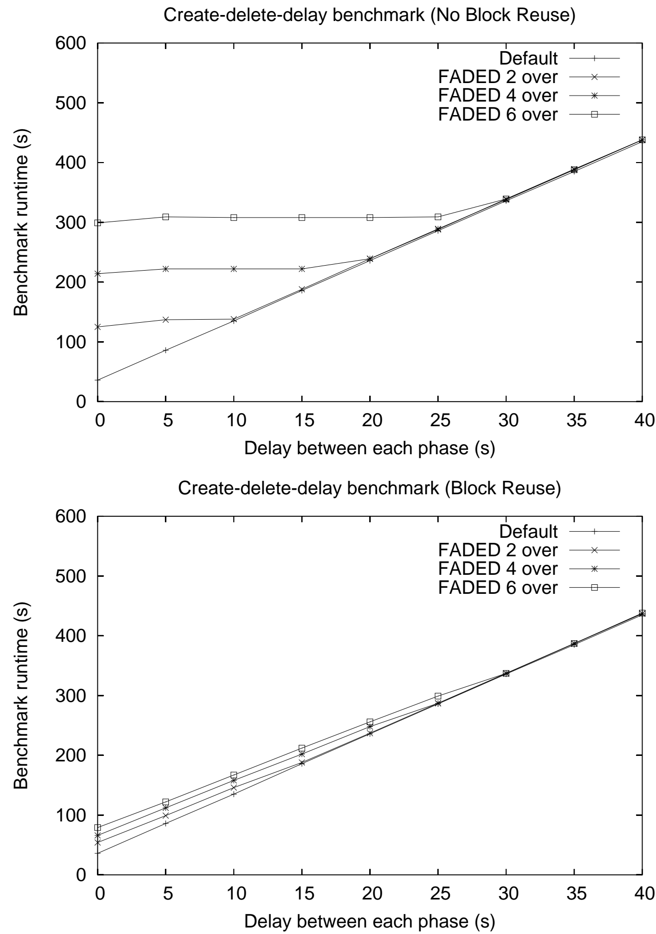


Figure 7.2: **Idle time requirement for microbenchmark.** *The figure plots the time taken by a create-delete-delay microbenchmark. The left graph shows no block reuse while the right graph includes block reuse. The baseline execution time for the benchmark is indicated by the "Default" line.*

FADED performs more unnecessary overwrites in the intermediate phases. Thus, in the presence of block reuse, very little idle time is required.

We next explore the time required for overwrites. First, we use the same Postmark configuration as above, but measure the time for the benchmark to complete including delayed overwrites. Since Postmark deletes all files at the end of its run, we face a worst case scenario where the entire working set of the benchmark has to

	Run-time with overwrites (s)	
	PostMark	**HP Trace**
Default	166.8	200.0
SecureDelete_2	466.6	302.8
SecureDelete_4	626.4	345.6
SecureDelete_6	789.3	394.3

Table 7.11: **Idle time requirement.** *The table shows the total run-time of two benchmarks, Postmark and the HP trace. The time reported includes completion of all delayed overwrites.*

be overwritten, accounting for the large overwrite times reported in Table 7.11. In the HP-trace, the overwrite times are more reasonable. Since most blocks deleted in the HP trace are then reused in subsequent writes, most of the overwrites performed here are conservative. This accounts for the steep increase from 0 to 2 overwrite passes, in the implicit case.

7.7 Implicit Detection Under NTFS

In this section, we present our experience building support for implicit liveness detection underneath the Windows NTFS file system. The main challenge we faced underneath NTFS was the absence of source code for the file system. While the basic on-disk format of NTFS is known [107], details of its update semantics and journaling behavior are not publicly available. As a result, our implementation currently tracks only block liveness which requires only knowledge of the on-disk layout; generation liveness tracking could be implemented if the details of NTFS journaling mechanism were known.

The fundamental piece of metadata in NTFS is the Master File Table (MFT); each record in the MFT contains information about a unique file. Every piece of metadata in NTFS is treated as a regular file; file 0 is the MFT itself, file 2 is the recovery log, and so on. The allocation status of all blocks in the volume is maintained in a file called the cluster bitmap, which is similar to the block bitmap tracked by ext2. On block allocations and deletions, NTFS regularly writes out modified bitmap blocks.

Our prototype implementation runs as a device driver in Linux, similar to the setup described earlier for other file systems. The virtual disk on which we interpose is exported as a logical disk to a virtual machine instance of Windows XP running over VMware Workstation [113]. To track block liveness, our implemen-

tation uses the same shadow bitmap technique mentioned in Section 7.5.2. By detailed empirical observation under long-running workloads, we found that NTFS did not exhibit any violation of the block exclusivity and delete suppression properties mentioned in Section 9.5; however, due to the absence of source code, we cannot assert that NTFS *always* conforms to these properties. This limitation points to the general difficulty of using implicit techniques underneath closed-source file systems; one can never be certain that the file system conforms to certain properties unless those are guaranteed by the file system vendor. In the absence of such guarantees, the utility of implicit techniques is limited to optimizations that can afford to be occasionally "wrong" in their implicit inference.

Our experience with NTFS also points to the utility of characterizing the precise set of file system properties required for various forms of liveness inference. This set of properties now constitutes a minimal "interface" for communication between file system and storage vendors. For example, if NTFS confirmed its conformance to the block exclusivity and delete suppression properties, the storage system could safely implement aggressive optimizations that rely on its implicit inference.

7.8 Explicit Liveness Notification

We have so far considered how to infer liveness within an SDS, and the costs of doing so. A pertinent issue to consider is how the liveness inference techniques compare to an approach where one had the luxury of changing the interface to support a new command that explicitly indicates deletes. In this section, we discuss such an *explicit notification* approach, where we assume that special `allocate` and `free` commands are added to SCSI. As an optimization, we obviate the need for an explicit `allocate` command by treating a `write` to a previously freed block as an implicit `allocate`.

We first describe the issues in incorporating such an explicit command into existing file systems. Although modifying file systems to use this interface may seem trivial, we find that supporting the `free` command has ramifications in the consistency management of the file system under crashes. We then implement secure deletion in such an explicit notification scenario and compare the costs of doing secure delete with the semantic inference approach.

We have modified the Linux ext2 and ext3 file systems to use this `free` command to communicate liveness information; we discuss the issues therein. The `free` command is implemented as an *ioctl* to a pseudo-device driver, which serves as our enhanced disk prototype.

7.8.1 Granularity of `free` notification

One issue that arises with explicit notification is the exact semantics of the `free` command, given the various granularities of liveness outlined in Section 7.3. For example, if only block liveness or content liveness needs to be tracked, the file system can be lazy about initiating `free` commands (thus suppressing `free` to blocks that are subsequently reused). For generation liveness, the file system needs to notify the disk of every delete of a block whose contents reached disk in the context of the deleted file. However, given multiple intermediate layers of buffering, the file system may not know exactly whether the contents of a block reached disk in the context of a certain file.

To simplify file system implementation, the file system should not be concerned about what form of liveness a particular disk functionality requires. In our approach, the file system invokes the `free` command for every logical delete. On receiving a `free` command for a block, the disk marks the block dead in its internal allocation structure (*e.g.*, a bitmap), and on a `write`, it marks the corresponding block live. The responsibility for mapping these `free` commands to the appropriate form of liveness information lies with the disk. For example, if the disk needs to track generation deaths, it will only be interested in a `free` command to a block that it thinks is live (as indicated by its internal bitmaps); a redundant `free` to a block that is already free within the disk (which happens if the block is deleted before being written to disk) will not be viewed as a generation death. For correct operation, the file system should guarantee that it will not write a block to disk without a prior allocation; if the `write` itself is treated as an implicit `allocate`, this guarantee is the same as the *delete suppression* property. A `write` to a freed block without an allocation will result in incorrect conclusion of generation liveness within the disk. Note that after a `free` is issued for a block, the disk can safely use that block, possibly erasing its contents.

7.8.2 Timeliness of `free` notification

Another important issue that arises in explicit notification of a `free` is *when* the file system issues the notification. One option is *immediate notification*, where the file system issues a “free” immediately when a block gets deleted in memory. Unfortunately, this solution can result in loss of data integrity in certain crash scenarios. For example, if a crash occurs immediately after the `free` notification for a block B but before the metadata indicating the corresponding delete reaches disk, the disk considers block B as dead, while upon recovery the file system views block B as live since the delete never reached disk. Since a live file now contains a freed block,

this scenario is a violation of data integrity. While such violations are acceptable in file systems such as ext2 which already have weak data integrity guarantees, file systems that preserve data integrity (such as ext3) need to *delay* notification until the effect of the delete reaches disk.

Delayed notification requires the file system to conform to the *reuse ordering* property; otherwise, if the block is reused (and becomes live within the file system) before the effect of the previous delete reaches disk, the delayed `free` command would need to be suppressed, which means the disk would miss a generation death.

7.8.3 Orphan allocations

Finally, explicit notification needs to handle the case of *orphan allocations*, where the file system considers a block dead while the disk considers it live. Assume that a block is newly allocated to a file and is written to disk in the context of that file. If a crash occurs at this point (but before the metadata indicating the allocation is written to disk), the disk would assume that the block is live, but on restart, the file system views the block as dead. Since the on-disk contents of the block belong to a file that is no longer extant in the file system, the block has suffered a generation death, but the disk does not know of this. The `free` notification mechanism should enable accurate tracking of liveness despite orphan allocations. Handling orphan allocations is file system specific, as we describe below.

7.8.4 Explicit notification in ext2

As mentioned above, because ext2 does not provide data integrity guarantees on a crash, the notification of deletes can be immediate; thus ext2 invokes the `free` command synchronously whenever a block is freed in memory. Dealing with orphan allocations in ext2 requires a relatively simple but expensive operation; upon recovery, the fsck utility conservatively issues `free` notifications to every block that is currently dead within the file system.

7.8.5 Explicit notification in ext3

Because ext3 guarantees data integrity in its ordered and data journaling modes, `free` notification in ext3 has to be delayed until the effect of the corresponding delete reaches disk. In other words, the notification has to be delayed until the transaction that performed the delete commits. Therefore, we record an in-memory list of blocks that were deleted as part of a transaction, and issue `free` notifications

for all those blocks when the transaction commits. Since ext3 already conforms to the reuse ordering property, such delayed notification is feasible.

However, a crash could occur during the invocation of the `free` commands (*i.e.*, immediately after the commit of the transaction); therefore, these `free` operations should be redo-able on recovery. For this purpose, we also log special `free` records in the journal which are then replayed on recovery, as part of the delete transaction.

During recovery, since there can be multiple committed transactions which will need to be propagated to their on-disk locations, a block deleted in a transaction could have been reallocated in a subsequent committed transaction. Thus, we cannot replay all logged `free` commands. Given our guarantee of completing all `free` commands for a transaction before committing the next transaction, we should only replay `free` commands for the last successfully committed transaction in the log (and not for any earlier committed transactions that are replayed).

To deal with orphan allocations, we log block numbers of data blocks that are about to be written, before they are actually written to disk. On recovery, ext3 can issue `free` commands to the set of orphan data blocks that were part of the uncommitted transaction.

7.8.6 Explicit secure delete

We have also built secure deletion under the explicit notification framework. We modified the ext2 and ext3 file systems to notify the disk of every logical delete (as described in §7.8). The file system modifications accounted for 14 and 260 lines of code respectively. Upon receiving the notification, the disk decides to shred the block. However, similar to FADED, the disk delays overwrites until idle time to minimize impact on foreground performance.

Performance of "explicit" secure delete

To evaluate our explicit secure delete implementation, we run the same hour of the HP trace; the results are presented in Table 7.12. Since the file system does not require the changes that FADED required, the default performance corresponds to the "No changes" row in Table 7.9.

In terms of the foreground performance, the explicit implementation performs better because it does not incur the overhead of inference. Further, it does not require the file system modifications reported in Table 7.9 (this corresponds to the "No changes" row in Table 7.9). Note that we do not model the cost of sending a `free` command across the SCSI bus; thus the overheads in the explicit case

System	Run-time (s)	
	Foreground	**With overwrites**
Default	195.0	–
Ex.SD_2	195.5	280.0
Ex.SD_4	196.8	316.2
Ex.SD_6	196.4	346.1

Table 7.12: **HP trace performance under explicit secure delete.** *The table shows the foreground run time and the total completion time of overwrites of the HP Trace on explicit secure delete.*

are optimistic. Also, since the explicit mode has perfect information, it avoids unnecessary overwrites, thus resulting in 8-13% lower overwrite times compared to FADED. Thus, we can conclude that the extra costs of semantic inference in comparison to the explicit approach are quite reasonable.

7.9 Discussion

In this section, we reflect on the lessons learned from our case study to refine our comparison on the strengths and weaknesses of the explicit and implicit approaches.

The ideal scenario for the implicit approach is where changes are required only in the storage system and not in the file system or the interface. However, in practice, accurate liveness detection requires certain file system properties, which means the file system needs to be modified if it does not conform to those requisite properties. In the face of such changes to both the storage system and the file system, it might appear that the implicit approach is not much more pragmatic than the explicit approach of changing the interface also. There are two main reasons why we believe the implicit approach is still useful.

First, file system changes are not required if the file system already conforms to the requisite properties. For example, many file systems (e.g. ext2, VFAT, ext3-data journaling, and perhaps NTFS) are already amenable to block liveness detection without any change to the file system. The ext3 file system in data journaling mode already conforms to the properties required for generation liveness detection. Clearly, in such cases, the implicit approach enables non-intrusive deployment of functionality.

Second, we believe that modifying the file system to conform to a set of well-defined properties is more general than modifying the file system (and the inter-

face) to convey a specific piece of information. Although we have discussed the file system properties from the viewpoint of implicit liveness detection, some of the properties enable richer information to be inferred; for example, the association between a block and its owning inode (required for certain applications such as file-aware layout [102]) can be tracked accurately if the file system obeys the reuse ordering or the consistent metadata properties. Our ultimate goal is to arrive at a set of properties that enable a wide variety of information to be tracked implicitly, thus outlining how file systems may need to be designed to enable such transparent extension within the storage system. In contrast, the approach of changing the interface requires introducing a new interface every time a different piece of information is required.

7.10 Summary

In this chapter, we explored various techniques to infer liveness information in an SDS and then utilized those techniques to implement reliable secure delete. By this implementation, we have demonstrated that even functionality that relies on semantic information for correctness can indeed be embedded at the storage system, despite fundamental uncertainty due to asynchrony. By being conservative in dealing with imperfect information, the storage system can convert apparent correctness problems into a minor performance degradation. We have also quantified the performance and complexity costs of implementing such functionality in an SDS by comparing it to an alternative implementation that explicitly changes the interface to achieve the same functionality.

The inference techniques and case studies thus far have assumed that the storage system is being used by a file system. Another popular usage scenario for a storage system is in the context of a DBMS. In the next chapter, we will explore applying similar semantic inference techniques underneath a database system.

Chapter 8

Semantic Disks for Database Systems

"*"Today we have this sort of simple-minded model that a disk is one arm on one platter and [it holds the whole database]. And in fact [what's holding the database] is RAID arrays, it's storage area networks, it's all kinds of different architectures underneath that hood, and it's all masked over by a logical volume manager written by operating system people who may or may not know anything about databases. Some of that transparency is really good because it makes us more productive and they just take care of the details. ... But on the other hand, optimizing the entire stack would be even better. So, we [in the two fields] need to talk, but on the other hand we want to accept some of the things that they're willing to do for us." [98].*"
-Pat Selinger

We have so far considered the range of functionality that an SDS can provide if it understood semantic information about the file system. In this chapter, we extend this philosophy by having the storage system understand even higher layers of the system. Specifically, we consider techniques by which an SDS can infer information about a database management system, and how it can utilize those techniques to provide improved functionality.

8.1 Introduction

In order to explore the applicability of the semantic disk technology in the realm of databases, we describe two of our earlier case studies as applied to database management systems. Each of the case studies is based on our work that successfully applied the corresponding idea underneath a file system. Specifically, we study

how to improve storage system availability by building a DBMS-specific version of D-GRAID, a RAID system that degrades gracefully under failure. We also implement a DBMS-specialized version of FADED, a storage system that guarantees that data is unrecoverable once the user has deleted it. We evaluate the case studies by prototype implementations underneath the Predator DBMS [99] built upon the SHORE storage manager [63].

In D-GRAID, we find that the DBMS-specific version did not work as well as its file-system-specific counterpart. Upon further investigation, we found that there were fundamental underlying reasons for this unexpected discrepancy. First, data structures and inter-relationships among data items are more complex in databases than in file systems; hence, improving reliability proved more difficult than anticipated. Second, databases tend to keep less general-purpose metadata than file systems, and certainly do not reflect such information to stable storage.

We conclude by discussing a set of evolutionary changes that database management systems could incorporate to become more amenable to semantically-smart disk technology. Specifically, we believe that a DBMS should export knowledge of data relationships to the storage system, and that it should keep general purpose usage statistics and periodically reflect them to disk. With such incremental changes in place, we believe that database systems will be well poised to take advantage of more intelligent storage systems.

The rest of this chapter is organized as follows. In Section 8.2, we discuss how a semantic disk extracts information about the DBMS. In Sections 8.3 and 8.4, we present our two DBMS-specific case studies. We present a general discussion of additional DBMS support required for semantic disks in Section 8.5, and conclude in Section 8.6.

8.2 Extracting Semantic Information

To implement database-specific functionality within a storage system, the storage system needs to understand higher level semantic information about the DBMS that uses the storage system. Similar to the file system case, database-specific semantic information can be broadly categorized into two types: static and dynamic. In this section, we describe the types of information a semantic disk requires underneath a DBMS, and discuss how such information can be acquired.

Since our experience has primarily been with the Predator DBMS [99] built upon the SHORE storage manager [63], we illustrate our techniques with specific examples from Predator; however, we believe the techniques are general across other database systems.

8.2.1 Static information

Static information is comprised of facts about the database that do not change while the database is running. The structure of the log record of the database and the format of a database page are examples of static information. Such information can be embedded into the storage system firmware, or can be communicated to the storage system through an out-of-band channel once during system installation. As in the file systems case, such on-disk formats change very slowly; thus, embedding this information within the storage system firmware is practical.

To gain some insight as to how often a storage vendor would have to deliver "firmware" updates in order to keep pace with DBMS-level changes, we studied the development of Postgres [81] looking for times in its revision history when a dump/restore was required to migrate to the new version of the database. We found that a dump/restore was needed every 9 months on average, a number higher than we expected. However, in commercial databases that store terabytes of data, requiring a dump/restore to migrate is less tolerable to users; indeed, more recent versions of Oracle go to great lengths to avoid on-disk format changes.

Thus, we assume that the storage system has static information about a few relevant data structures of the DBMS; the main challenge lies in building upon this minimal information to automatically infer higher level *operations* performed by the database system.

8.2.2 Dynamic information

Dynamic information pertains to information about the DBMS that continually changes as the database system operates. Tracking the set of disk blocks allocated to a certain table is an instance of dynamic information, as it keeps changing as records are inserted into or deleted from the table. Similarly, tracking whether a certain disk block belongs to a table or an index is another example.

Unlike static information, dynamic information needs to be continually tracked by the disk. To track dynamic information, a semantic disk utilizes static information about data structure formats to monitor changes to key data structures; these changes are then correlated to specific higher level DBMS operations that could have caused the change. In the case of file systems, we showed that this process of dynamic inference is significantly complicated by buffering and reordering of writes within the file system.

Fortunately, tracking dynamic information accurately and reliably underneath a DBMS is quite straightforward, because of the write-ahead log(WAL) that databases use. The log records every operation that leads to any change to on-disk contents,

and because of the WAL property, the log of an operation reaches disk *before* the effect of the operation reaches disk. This strong ordering guarantee makes inferences underneath a DBMS accurate and straightforward. We now describe how a semantic disk infers different kinds of dynamic information about the DBMS. Note that not every case study requires all types of information; we quantify and discuss this further in Section 8.5.

Log snooping

The basic technique that a semantic disk uses to track dynamic information about the DBMS is to *snoop* on the log records written out by the DBMS. Each log record contains a Log Sequence Number(LSN) [69], which is usually the byte offset of the start of that record in the log volume. The LSN gives the semantic disk accurate information on the exact ordering of events that occurred in the database. To track these events, the disk maintains an *expected LSN* pointer that tracks the LSN of the next log record expected to be seen by the disk; thus, when the semantic disk receives a write request to a log block, it knows exactly where in the block to look for the next log record. It processes that log record, and then advances the expected LSN pointer to point to the next record. When the DBMS does group commits, log blocks could arrive out of order, but the semantic disk utilizes the LSN ordering to process them in order; log blocks arriving out of order are *deferred* until the expected LSN reaches that block.

Tracking block ownership

An important piece of information that is useful within a semantic disk is the logical grouping of blocks into tables and indices. This involves associating a block with the corresponding table or index *store* that logically *owns* the block. Since the allocation of a block is an update to on-disk data that must be recoverable, the DBMS logs it before performing the allocation. For example, SHORE writes out a `create_ext` log record with the block number and the ID of the store it is allocated to. When the semantic disk observes this log entry, it records this information in an internal `block_to_store` hash table.

Once the semantic disk knows the numerical store ID to which a page belongs, it needs to map the store ID to the actual table (or index) name. To get this mapping, the disk utilizes static knowledge of the system catalog table that tracks this mapping at the DBMS. In the case of Predator, this mapping is maintained in a B-Tree called the *RootIndex*. Thus, the disk monitors `btree_add` records in the log; such records contain the store-id of the B-Tree and the entry added. Since the logi-

cal store ID of the RootIndex is known (as part of static information), the semantic disk identifies newly created mappings and tracks them in a `store_to_name` hash table.

Tracking block type

Another useful piece of information that a semantic disk may require is the *type* of a store (or a block), *e.g.*, whether the block is a data page or an index page. To track this information, the semantic disk watches updates to the system catalog tables that track this information. The semantic disk detects inserts to this table by looking for `page_insert` records in the log, which contain the tuple added, together with the page number. Since the disk already knows the store ID to which the page belongs (block ownership), it is straightforward to monitor inserts to the store IDs corresponding to the catalogs. Note that the table names of these catalog tables are part of the static information known to the disk. In Predator, we are mainly interested in the _SINDXS table which contains the list of indexes in the database. Each tuple in the _SINDXS table contains the name of the index, the name of the table and the attribute on which it is built.

Relationship among stores

The most useful type of relationship is that between a table and the set of indices built on the table. Tracking this information again involves monitoring updates to the _SINDXS catalog table, which contains the association. Note that this relationship is maintained in the catalogs only in terms of table and index *names* and not the store IDs; hence the need to track the store-to-name mapping as described earlier.

Fine-grained information

While the aforementioned dynamic information can be extracted by log snooping, tracking certain pieces of information requires the semantic disk to also probe into the contents of a page. For example, to find whether a B-Tree page is an internal page or a leaf page, the disk needs to look at the relevant field within the page. Similarly, to find byte ranges within a page that got deleted, the semantic disk needs to scan the page looking for “holes”. Such fine-grained information thus requires more static knowledge within the storage system, since it has to understand the format of B-Tree and data pages.

8.3 Partial Availability with D-GRAID

In this section, we describe our first case study D-GRAID, a prototype semantically-smart storage system that utilizes DBMS-specific information to lay out blocks in a way that ensures graceful degradation of database availability under unexpected multiple failures. Thus, D-GRAID enables continued operation of the database instead of complete unavailability under multiple failures. We motivated the need for partial availability and showed that this approach significantly improves the availability of file systems in Chapter 6.

8.3.1 Design

The basic goal of D-GRAID is to make *semantically meaningful* fragments of data available under failures, so that queries that access only those parts of the data can still run to completion, oblivious of data loss in other parts of the database. The key idea is to lay out all blocks of a *semantic fragment* within a single disk, so that even after arbitrary disk failures, semantic fragments will be available or unavailable in their entirety. Thus, data present in the live disks will be meaningful in isolation. The choice of which blocks constitute a semantic fragment is dependent on the database workload.

D-GRAID exports a linear logical address space to the DBMS running above (like any SCSI disk). Internally, it needs to place these logical blocks in the appropriate physical disk to ensure partial availability. Like in the file system case, the key structure D-GRAID uses to enable such flexibility in block placement is the *indirection map*, which maps every logical block to the corresponding physical block; similar structures are used in modern arrays [117]. If the logical block is replicated, a bit in the indirection map entry tracks the most up to date copy.

We now present various layout strategies for partial DBMS availability. First, we discuss what structures need to be aggressively replicated for partial availability. Then, we explore two classes of techniques for fragmentation, targeting widely different database usage patterns: coarse-grained fragmentation and fine-grained fragmentation. Finally, we discuss how partial availability interacts with the transaction and recovery mechanism of the DBMS, and the issue of availability of the database log.

System metadata replication

Before delving into semantic fragmentation, there are some data structures within a DBMS that must be available for *any* query in the system to be able to run. For

example, system catalogs (that contain information about each table and index) are frequently consulted; if such structures are unavailable under partial failure, the fact that most data remains accessible is of no practical use. Therefore, D-GRAID aggressively replicates the system catalogs and the *extent map* in the database that tracks allocation of blocks to stores. In our experiments, we employ 8-way replication of important meta-data; we believe that 8-way replication is quite feasible given the "read-mostly" nature of such meta-data and the minimal space overhead (less than 1%) this entails.

Coarse-grained fragmentation

When the database has a large number of moderately sized tables, a semantic fragment can be defined in terms of entire tables. This subsection presents layout strategies for improved availability in such a scenario.

A. Scans:

Many queries, such as selection queries that filter on a non-indexed attribute or aggregate queries on a single table, just involve a sequential scan of one entire table. Since a scan requires the entire table to be available in order to succeed, a simple choice of a semantic fragment is the set of all blocks belonging to a table; thus, an entire table is placed within a single disk, so that when failures occur, a subset of tables are still available in their entirety, and therefore scans just involving those tables will continue to operate oblivious of failure.

B. Index lookups:

Index lookups form another common class of queries. When a selection condition is applied based on an indexed attribute, the DBMS looks up the corresponding index to find the appropriate tuple RIDs, and then reads the relevant data pages to retrieve the tuples. Since traversing the index requires access to multiple pages in the index, collocation of a whole index improves availability. However, if the index and table are viewed independently for placement, an index query fails if either the index or the table is unavailable, decreasing availability. A better strategy to improve availability is to collocate a table with its indices. We call the latter strategy as *dependent index placement*.

C. Joins:

Many queries involve joins of multiple tables. Such queries typically require all the joined tables to be available, in order to succeed. To improve availability of join queries, D-GRAID collocates tables that are likely to be joined together into a single semantic fragment, which is then laid out on a single disk. Note that identification of such "join groups" requires extra access statistics to be tracked by the DBMS. Specifically, we modified the Predator DBMS to record the set of

stores (tables and indexes) accessed for each query, and construct an approximate correlation matrix that indicates the access correlation between each pair of stores. This information is written to disk periodically (once every 5 seconds). D-GRAID then uses this information to collocate tables that are likely to be accessed together.

Fine-grained fragmentation

While collocation of entire tables and indices within a single disk provides enhanced availability, certain scenarios may preclude such layout. For example, a single table or index may be too large to fit within a single disk, or a table can be "popular" (*i.e.*, accessed by most queries); placing such a hot table within a single disk will lead to significant loss of availability upon failure of that particular disk. In such scenarios, we require a fine-grained approach to semantic fragmentation. In this approach, D-GRAID stripes tables and indexes across multiple disks (similar to a traditional RAID array), but adopts new techniques to enable graceful degradation, as detailed below.

A. Scans:

Scans fundamentally require the entire table to be available, and thus any striping strategy will impact availability of scan queries. However, with disk capacities roughly doubling every year [42], tables that are too large to fit into a single disk may become increasingly rare. In such cases, a hierarchical approach is possible: a large table can be split across the minimal number of disks that can hold it, and the disk group can be treated as a logical fault-boundary; D-GRAID can be applied over such logical fault-boundaries. Finally, if the database supports approximate queries [47], it can provide partial availability for scan queries even with missing data.

B. Index lookups:

With large tables, index-based queries are likely to be more common. For example, an OLTP workload normally involves index lookups on a small number of large tables. These queries do not require the entire index or table to be available. D-GRAID uses two simple techniques to improve availability for such queries. First, the internal pages of the B-tree index are aggressively replicated, so that a failure does not take away, for instance, the root of the B-tree. Second, an index page is collocated with the data pages corresponding to the tuples pointed to by the index page. For this collocation, D-GRAID uses a probabilistic strategy; when a leaf index page is written, D-GRAID examines the set of RIDs contained in the page, and for each RID, determines which disk the corresponding tuple is placed in. It then places the index page on the disk which has the greatest number of matching tuples. Note that we assume the table is clustered on the index attribute; page-level

collocation may not be effective in the case of non-clustered indices.

C. Joins:

Similar to indices, page-level collocation can also be applied across tables of a join group. For such collocation to be feasible, all tables in the join group should be clustered on their join attribute. Alternatively, if some tables in the join group are "small", they can be replicated across disks where the larger tables are striped.

Diffusion for performance

With coarse-grained fragmentation, an entire table is placed within a single disk. If the table is large or is accessed frequently, this can have a performance impact since the parallelism that can be obtained across the disks is wasted. To remedy this, D-GRAID monitors accesses to the logical address space and tracks logical segments that are likely to benefit from parallelism. D-GRAID then creates an extra copy of those blocks and spreads them across the disks in the array, like a normal RAID would do. Thus, for blocks that are "hot", D-GRAID regains the lost parallelism due to collocated layout, while still providing partial availability guarantees. Reads and writes are first sent to the diffused copy, with background updates being sent to the actual copy.

8.3.2 Transactions and Recovery

A pertinent issue to consider is how partial availability interacts with the transaction and recovery mechanisms within a DBMS. Databases declare a transaction committed once it is recorded in the log, *i.e.*, potentially before the actual data blocks are written; since ACID guarantees impose that this commit be durable in the face of database crashes, the recovery process needs to read through the log and *redo* operations that were not committed to disk. However, when the storage system underneath supports partial availability, some of the blocks could be "missing"; if one of the blocks referred to in a REDO record is missing, the recovery process cannot execute the redo.

However, this problem has been considered and solved in ARIES [69], in the context of handling offline objects during *deferred restart*. Specifically, the solution is to just ignore redo of those blocks until the blocks actually become available again (when it is restored from tape, for instance). The *PageLSN* in the unavailable page would have remained the same while it was unavailable, and when it does become available, the REDO record will be more recent than the PageLSN of the page, and thus the redo applied. Accesses to the page once it becomes available, should be prevented until the recovery takes place on that page. Similarly undo

operations can be handled, either at the logical or physical level. The key idea is to turn off access to those pages by normal transactions (this can be done by holding a long-standing lock on those pages until they are recovered) [69].

Another side-effect of the partial availability semantics occurs even during normal transaction execution. A transaction could be committed after recording changes in the log, but when the actual data pages are being updated, the corresponding disk blocks can be unavailable, thereby forcing the database to violate the durability guarantees. To prevent this, D-GRAID guarantees that writes will never fail as long as there is some space in atleast one of the disks that is still alive. When a write request arrives to a dead disk, it automatically remaps it to a new block in a disk that is still alive, and updates the *imap* accordingly. If there was no space, still D-GRAID does not do worse than a normal storage system in which such a multiple failure occured in the middle of query execution; the database would abort, and REDO during log replay after the disk is recovered, will restore sanity to the contents of the block.

Availability of the database log

The database log plays a salient role in the recoverability of the database, and its ability to make use of partial availability. It is therefore important for the log to be available under multiple failures. We believe that providing high availability for the log is indeed possible. Given that the size of the “active portion” of the log is determined by the length of the longest transaction factored by the concurrency in the workload, the portion of the log that needs to be kept highly available is quite reasonable. Modern storage arrays have large amounts of persistent RAM, which are obvious locales to place the log for high availability, perhaps replicating it across multiple NVRAM stores. This, in addition to normal on-disk storage of the log, can ensure that the log remains accessible in the face of multiple disk failures.

8.3.3 Evaluation

We evaluate the availability improvements and performance of D-GRAID through a prototype implementation; our D-GRAID prototype functions as a software RAID driver in the Linux 2.4 kernel, and operates underneath the Predator/Shore DBMS.

Availability improvements

To evaluate availability improvements with D-GRAID, we use a D-GRAID array of 16 disks, and study the fraction of queries that the database serves successfully under an increasing number of disk failures. Since layout techniques in D-GRAID are

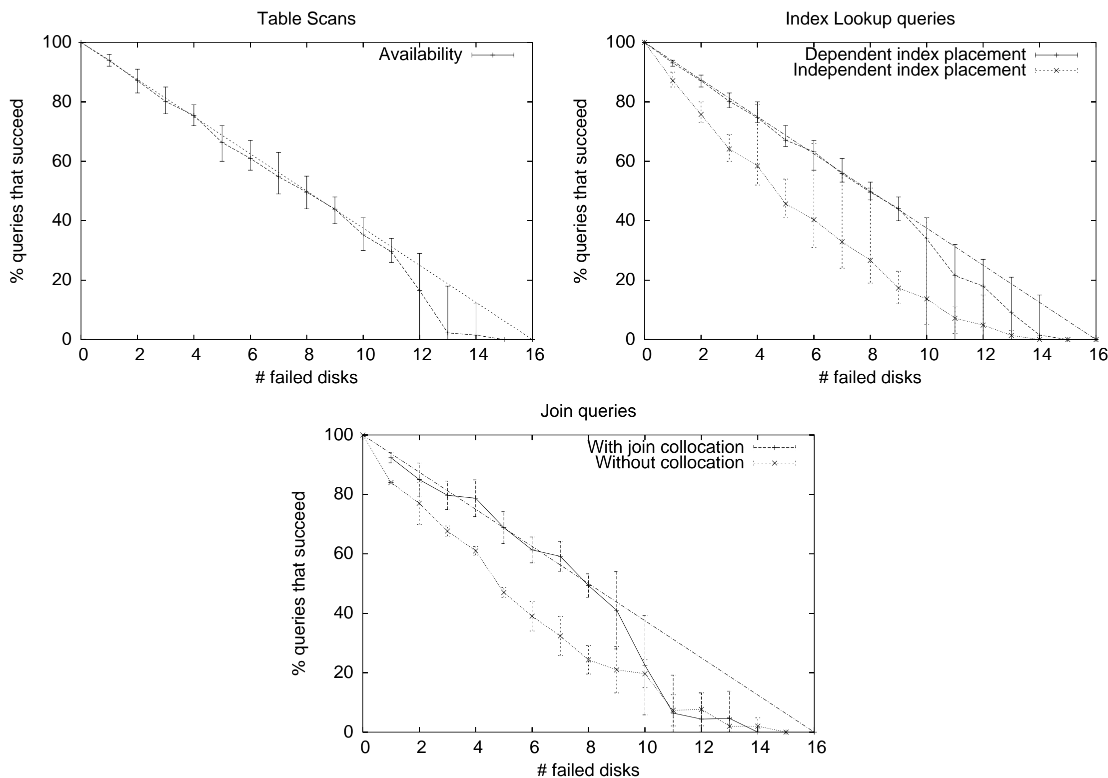


Figure 8.1: **Coarse-grained fragmentation.** *The graphs show the availability degradation for scans, index lookups and joins under varying number of disk failures. A 16-disk D-GRAID array was used. The steeper fall in availability for higher number of failures is due to the limited (8-way) replication of metadata. The straight diagonal line depicts "ideal" linear degradation.*

complementary to existing RAID schemes such as parity or mirroring, we show D-GRAID Level 0 (*i.e.*, no redundancy for data) in our measurements, for simplicity. We mainly use microbenchmarks to analyze the availability provided by various layout techniques in D-GRAID.

A. Coarse-grained fragmentation

We first evaluate the availability improvements due to the coarse-grained fragmentation techniques in D-GRAID. Figure 8.1 presents the availability of various types of queries under synthetic workloads.

The first graph in Figure 8.1 shows the degradation in availability for scan queries, under multiple disk failures. The database had 200 tables, each with 10,000

tuples. The workload is as follows: each query chooses a table at random and computes an average over a non-indexed attribute, thus requiring a scan of the entire table. The percentage of such queries that complete successfully is shown. As the graph shows, collocation of whole tables enables the database to be partially available, serving a proportional fraction of queries. In comparison, just one failure in a traditional RAID-0 system results in complete unavailability. Note that if redundancy is maintained (*i.e.*, parity or mirroring), both D-GRAID and traditional RAID will tolerate up to one failure without any availability loss.

The middle graph in Figure 8.1 shows the availability for index lookup queries under a similar workload. We consider two different layouts; in both layouts, an entire "store" (*i.e.*, an index or a table) is collocated within one disk. In *independent index placement*, D-GRAID treats the index and table as independent stores and hence possibly allocates different disks for them, while with *dependent index placement*, D-GRAID carefully allocates the index on the same disk as the corresponding table. As can be seen, dependent placement leads to much better availability under failure.

Finally, to evaluate the benefits of join-group collocation, we use the following microbenchmark: the database contains 100 pairs of tables, with joins always involving tables in the same pair. We then have join queries randomly selecting a pair and joining the corresponding two tables. The bottom-most graph in Figure 8.1 shows that by collocating joined tables, D-GRAID achieves higher availability. Note that for this experiment, the DBMS was modified to report additional statistics on access correlation between tables.

B. Fine-grained fragmentation

We now evaluate the effectiveness of fine-grained fragmentation in D-GRAID. We mainly focus on the availability of index lookup queries since they are the most interesting in this category. The workload we use for this study consists of index lookup queries on randomly chosen values of a primary key attribute in a single large table. We plot the fraction of queries that succeed under varying number of disk failures. The top graph in Figure 8.2 shows the results.

There are three layouts examined in this graph. The lowermost line shows availability under simple striping with just replication of system catalogs. We can see that the availability falls drastically under multiple failures due to loss of internal B-tree nodes. The middle line depicts the case where internal B-tree nodes are replicated aggressively; as can be expected, this achieves better availability. Finally, the third line shows the availability when data and index pages are collocated, in addition to internal B-tree replication. Together, these two techniques ensure near linear degradation of availability.

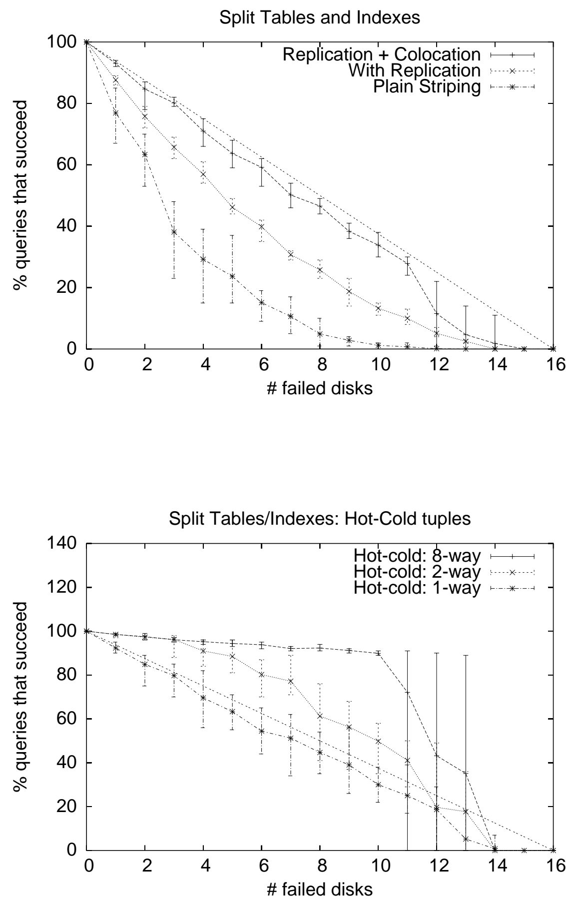


Figure 8.2: **Index Lookups under fine-grained fragmentation.**

The right graph in Figure 8.2 considers a similar workload, but a small subset of tuples are much "hotter" compared to the others. Specifically, 5% of the tuples are accessed in 90% of the queries. Even under such a workload, simple replication and collocation provide near linear degradation in availability since hot pages are spread nearly uniformly across the disks. However, under such a hot-cold workload, D-GRAID can improve availability further by replicating data and index pages containing such hot tuples. The other two lines depict availability when such hot pages are replicated by factors of 2 and 8. Thus, when a small fraction of (read mostly) data is hot, D-GRAID utilizes that information to enhance availability through selective replication.

	D-GRAID	RAID-0	Slowdown
Table Scan	7.97 s	6.74 s	18.1%
Index Lookup	51 ms	49.7 ms	2.7%
Bulk Load	186.61 s	176.14 s	5.9%
Table Insert	11.4 ms	11 ms	3.6%

Table 8.1: **Time Overheads of D-GRAID.** *The table compares the performance of D-GRAID with default RAID-0 under various microbenchmarks. An array of 4 disks is used.*

Performance overheads

We now evaluate the performance implications of fault-isolated layout in D-GRAID. For all experiments in this section, we use a 550 MHz P-III system with a 4-disk D-GRAID array comprised of 9.1 GB IBM UltraStar 9LZX disks with peak throughput of 20 MB/s. The database used has a single table of 500,000 records, each sized 110 bytes, with an index on the primary key.

A. Time and space overheads

We first explore the time and space overheads incurred by our D-GRAID prototype for tracking information about the database and laying out blocks to facilitate graceful degradation. Table 8.1 compares the performance of D-GRAID with fine-grained fragmentation to Linux software RAID 0 under various basic query workloads. The workloads examined are a scan of the entire table, an index lookup of a random key in the table, bulk load of the entire indexed table, and inserts into the indexed table. D-GRAID performs within 5% of RAID-0 for all workloads except scans. The poor performance in scans is due to a Predator anomaly, where the scan workload completely saturated the CPU (6.74 s for a 50 MB table across 4 disks). Thus, the extra CPU cycles required by D-GRAID impacts the scan performance by about 18%. This interference is because our prototype competes for resources with the host; in a hardware RAID system, such interference would not exist. Overall, we find that the overheads of D-GRAID are quite reasonable.

We also evaluated the space overheads due to aggressive metadata replication and found them to be minimal; the overhead scales with the number of tables, and even in a database with 10,000 tables, the overhead is only about 0.9% for 8-way replication of important data.

B. Parallelism for Collocated Tables

We now evaluate the benefits of diffusing an extra copy of popular tables. Table 8.2 shows the time taken for a scan of the table described above, under coarse-

grained fragmentation in D-GRAID. As can be seen, simple collocation leads to poor scan performance due to the lost parallelism. With the extra diffusion aimed at performance, D-GRAID performs much closer to default RAID-0.

8.3.4 Discussion

In comparing our implementation of D-GRAID underneath a DBMS with our file system work discussed in Chapter 6, we uncovered some fundamental challenges that are unique to a DBMS. First, the notion of semantically-related groups is much more complex in a DBMS because of the various inter-relationships that exist across tables and indexes. In the file system case, whole files or whole directories were reasonable approximations of semantic groupings. In a DBMS, since the goal of D-GRAID is to enable serving as many higher level queries as possible, the notion of semantic grouping is *dynamic*, *i.e.*, it depends on the query workload. Second, identifying "popular" data that needs to be aggressively replicated, is relatively easier in file systems; standard system binaries and libraries were obvious targets, independent of the specific file system running above. However, in a DBMS, the set of popular tables varies with the DBMS and is often dependent on the query workload.

Thus, effective implementation of D-GRAID underneath a DBMS requires modification of the DBMS to record additional information. Given that D-GRAID enables the database system to serve a large fraction of its queries even under multiple failures that, in traditional storage arrays would lead to complete unavailability of the database, we believe that such simple modifications to the DBMS to enable D-GRAID are reasonable and feasible. We outline the exact pieces of information required in Section 8.5.

8.4 Secure Delete with FADED

In this section, we describe FADED, a prototype semantically-smart disk that detects deletes of records and tables at the DBMS level and securely overwrites (*shreds*) the relevant data to make it irrecoverable. The file system instance of FADED was discussed in Chapter 7.

8.4.1 Table-level deletes

The simplest granularity of secure delete is a whole table. When a `drop table` command is issued, FADED needs to shred all blocks that belonged to the table. FADED uses log snooping to identify log records that indicate freeing of extents

	Scan Time (s)
RAID-0	6.74
D-GRAID	15.69
D-GRAID + Diffusion	7.35

Table 8.2: **Diffusing Collocated Tables.** *The table shows the scan performance observed over a 4-disk array under various configurations.*

from stores. In SHORE, a `free_ext_list` log record is written for every extent freed. Once FADED knows the list of freed blocks, it can issue secure overwrites to those pages. However, since the contents of the freed pages may be required by the database if the transaction aborts (thus undoing the deletes), FADED needs to delay the overwrites until the transaction that performed the delete commits. Thus, FADED keeps track of a list of "pending" transactions, and snoops for `commit` records in the log to identify if those transactions have committed, so that the relevant secure overwrites can be initiated.

8.4.2 Record-level deletes

Handling record level deletes in FADED is more challenging. When specific tuples are deleted (via the SQL `delete from` statement), specific byte ranges in the pages that contained the tuples need to be shredded. On a delete, a DBMS typically marks the relevant page "slot" free, and increments the freespace count in the page. Since such freeing of slots is logged, FADED can learn of such record deletes by log snooping.

However, FADED cannot shred the whole page on detecting the delete because other records in the page could still be valid. The shredding is therefore deferred until FADED receives a write to the page that reflects the relevant delete. On receiving such a write, FADED shreds the entire page in the disk, and then writes the new data received. Thus, past layers of data in the disk pertaining to the deleted records disappear.

There are two main problems in implementing the above technique. The first problem relates to identifying the correct version of the page that contained the deleted record. Assume that FADED observed a record delete d in page P, and waits for a subsequent write of P. When P is written, FADED needs to detect if the version written reflects d. The version could be stale if the DBMS wrote the page out sometime before the delete, but the block got reordered by the disk scheduler,

	Run time (s)	
	Workload I	Workload II
Default	52.0	66.0
FADED_2	78.3	128.5
FADED_4	91.0	160.0
FADED_6	104.5	190.2

Table 8.3: **Overheads of secure deletion.** *This table shows the performance of FADED with 2,4 and 6 overwrites, under two workloads. Workload I deletes contiguous records, while Workload II deletes records randomly across the table.*

and arrives late at the disk. To identify whether the current contents of P reflects the delete, FADED uses the *PageLSN* field in the page [69]. The *PageLSN* of a page tracks the sequence number of the latest log record describing a change in the page. Thus, FADED simply needs to compare the *PageLSN* to the LSN of the delete d it is tracking.

The second issue is that the DBMS typically does not zero out bytes that belonged to deleted records; as a result, old data still remains in the page. Thus, when FADED observes the page write, it has to scan the page looking for free space, and zero out deleted byte ranges. Since the page could still reside in the DBMS cache, all subsequent writes to the page also need to be scanned and zeroed out in their freespace.

8.4.3 Performance

In this subsection, we briefly evaluate the cost of secure deletion in FADED through a prototype implementation. Similar to the D-GRAID prototype, FADED is implemented as a device driver in the Linux 2.4 kernel, and works underneath Predator [99]. We consider two workloads operating on a table with 500,000 110-byte records. In the first workload, we perform a `delete from` in such a way that all rows in the second half of the table are deleted (*i.e.*, the deleted pages are contiguous). The second workload is similar, but the tuples to be deleted are selected in random.

Table 8.3 compares FADED under various number of overwrite passes, with the default case, a plain disk. As expected, secure deletion comes at a performance cost due to the extra disk I/O for the multiple passes of overwrites. However, since such overhead is incurred only on deletes, and only sensitive data needs to be deleted in

this manner, we believe the costs are reasonable in situations where the additional security is required.

8.4.4 Discussion

The fine record-level granularity of deletes in a DBMS makes secure deletion more complex in the case of DBMS, compared to its file system counterpart. Although our implementation does not require any changes to the DBMS, it requires detailed information about the on-disk page layout of the DBMS. Thus, we make an assumption that on-disk page formats are relatively stable across DBMS releases; alternatively, when the format changes, the storage system would require a corresponding firmware upgrade. However, secure delete is beginning to be recognized as a crucial piece of functionality especially in the context of recent legislations on data retention. Given that a semantic disk is an ideal locale to implement it, incurring this extra dependency seems worth the cost.

8.5 Towards a Semantic Disk-Friendly DBMS

In this section, we first discuss what exact types of DBMS-specific information are required by the storage system for our case studies. We then suggest a few evolutionary changes that would make a DBMS more amenable to semantically-smart disks.

8.5.1 Information required

In Section 8.2, we presented various techniques that a storage system can employ to learn more about a DBMS, but as these techniques were general, not all case studies require each of the techniques. By listing the exact types of information required for the case studies, we see how intimate the relationship between storage and the DBMS above must be for the semantic-disk technology to be successful.

The exact information required for each of our case studies is listed in Table 8.4. The table presents each case study (with variants listed as appropriate), and in each column shows which type of information is needed from the database. The columns are broken down into three major categories: static, dynamic, and extra information.

As discussed before, static information is the type of information that does not change as the DBMS is running (*e.g.*, the format of a log record); rather such information might change when a new version of the database is installed. Dynamic information continually changes during operation (*e.g.*, which blocks on disk a

	Static *embedded*				Dynamic *automatically tracked*				Extra *provided by DBMS*
	Catalog tables	Log record format	B-tree page format	Data page format	Block ownership	Block type	Relation among stores	Transaction status	Access statistics
D-GRAID									
basic	×	×			×	×	×		
+fine-grained frags	×	×	×		×	×	×		
+join-collocation	×	×			×	×	×		×
FADED									
basic	×	×			×	×		×	
+record-level delete	×	×		×	×	×		×	

Table 8.4: **DBMS Information required for case studies** *The table lists different pieces of information about the DBMS that our case studies require. The first column specifies static information that must be embedded into the semantic disk, the second column lists dynamic state automatically tracked by the disk, and the third column lists additional information tracked and communicated to the disk by the DBMS.*

table is allocated upon). Finally, extra information is what we needed to add to the DBMS in order for the storage system to implement certain classes of functionality.

Probably the biggest concern from the perspective of database vendors is what types of static information are required; if a storage system assumes it understands the contents of a catalog table, for example, the database vendor may be loathe to change it or feel constrained by this dependency. As we can see from the table, the amount of static information needed by each case study varies quite a bit. All the case studies need to know the structure of catalog tables and log records. In addition, implementing features such as fine-grained fragmentation in D-GRAID or record-level delete in FADED required an understanding of more detailed aspects of the DBMS, including the B-tree page format and the data page format, respectively. Given this level of detail, one must weigh whether the additional functionality provided merits the increased level of interdependence.

8.5.2 How DBMSes can help semantic disks

Despite the simplicity of tracking dynamic information underneath a DBMS, we encountered some fundamental challenges in implementing the various case studies underneath a DBMS, that do not arise in their file system counterparts. Being general-purpose, file systems track richer information, sometimes implicitly, about the way files are accessed. For example, the fact that a certain set of files lies within a single directory implicitly conveys information to the storage system that those files are likely to be accessed together. Similarly, most file systems track the last time each file was accessed. Such information is required for effective implementation of one of our case studies.

However, database systems do not typically track detailed information on how tables and indexes are accessed. In this subsection, we draw on our experiences from the case studies to identify simple ways in which a DBMS can assist semantically-smart disks, by tracking more information. We provide three key suggestions that could be easily incorporated into DBMS systems, as demonstrated in our case studies above.

First, we require the DBMS to track statistics on relationships between logical entities such as tables and indexes, and write them out periodically to disk. For example, the DBMS could record for each query, the set of tables and indexes accessed, and aggregate this information across queries. These statistics capture the semantic correlation between the accesses to different tables, and thus inform the storage system on general semantic groupings that need to be collocated for improved reliability. These statistics can be written as additional catalog tables. Because these updates are primarily meant to be performance hints, they need not be transactional, and thus can avoid the logging overhead.

Second, the DBMS needs to track statistics on the popularity of various pieces of data, such as the number of queries that accessed a given table over a certain duration. This piece of information conveys to the storage system the importance of various tables and indexes. The storage system can use this information to to aggressively replicate "hot" data to improve availability.

Interestingly, some modern database systems already track a modest amount of similar access information, though at a coarser granularity, for purposes of performance diagnosis; for example, the Automatic Workload Repository in Oracle 10g maintains detailed access statistics and commits them to disk periodically [72].

Finally, a key point to note is the stability of on-disk data layout in a DBMS, such as the format of a data page. As described in Chapter 3, in the case of file systems, on-disk formats rarely change. In the case of a DBMS, format changes are more of a concern. To facilitate semantic disks, DBMSes need to be judicious

in format changes. We believe that this trend already holds to an extent in modern DBMSes given that format changes require a dump and restore of existing data, an arduous task for large databases. For example, modern versions of Oracle take great care in preserving on-disk format.

8.6 Summary

We have demonstrated in this chapter that semantic knowledge enables the construction of powerful functionality within a storage system, tailored to a system as complex as a modern DBMS, and yet avoiding complex changes to the database in order to harness said functionality. We have also shown that simple evolutionary changes to the DBMS can make database-specific storage systems more effective. Instead of viewing storage systems and database systems as black boxes relative to each other, the SDS approach enables new classes of functionality by making the storage system DBMS-aware.

Chapter 9

A Logic of File Systems and Semantic Disks

"*In theory, there's no difference between theory and practice, but in practice, there is.*" Anonymous

In the previous chapters, we presented detailed techniques to extract semantic information underneath modern file systems and database systems, and various case studies to utilize that information to provide enhanced functionality within storage. As could be clearly seen from the description of some of our case studies, extracting and utilizing semantic information under modern file systems is quite complex. When building functionality within the SDS that utilizes information in a way that can impact correctness (such as FADED), one needs a clear understanding of the kinds of information that can be tracked accurately in a semantic disk given a set of file system behaviors.

In order to simplify and systematize the process of reasoning about the kinds of information that can be tracked accurately in a semantic disk, we formulate in this chapter a logic framework for formally reasoning about file systems and semantic disks. Although the intended initial goal of this logic was to model semantic disks, it was very soon clear that reasoning about information available to a semantic disk has a strong parallel to reasoning about file system consistency management, since in both cases, the information purely pertains to what can be "known" from on-disk state. Thus, we present this logic as a way to model file systems and reason about their correctness properties, and then show how we can use it to reason about semantic disks.

9.1 Introduction

Reliable data storage is the cornerstone of modern computer systems. File systems are responsible for managing persistent data, and it is therefore essential to ensure that they function correctly.

Unfortunately, modern file systems have evolved into extremely complex pieces of software, incorporating sophisticated performance optimizations and features. Because disk I/O is the key bottleneck in file system performance, most optimizations aim at minimizing disk access, often at the cost of complicating the interaction of the file system with the storage system; while early file systems adopted simple update policies that were easy to reason about [65], modern file systems have significantly more complex interaction with the disk, mainly stemming from asynchrony in updates to metadata [11, 32, 46, 68, 86, 111, 112].

Reasoning about the interaction of a file system with disk is paramount to ensuring that the file system never corrupts or loses data. However, with complex update policies, the precise set of guarantees that the file system provides is obscured, and reasoning about its behavior often translates into a manual intuitive exploration of various scenarios by the developers; such *ad hoc* exploration is arduous [112], and possibly error-prone. For example, recent work [121] has found major correctness errors in widely used file systems such as ext3, ReiserFS and JFS.

In this chapter, we present a formal logic for modeling the interaction of a file system with the disk. With formal modeling, we show that reasoning about file system correctness is simple and foolproof. The need for such a formal model is illustrated by the existence of similar frameworks in many other areas where correctness is paramount; existing models for authentication protocols [18], database reliability [45], and database recovery [58] are a few examples. While general theories for modeling concurrent systems exist [8, 59], such frameworks are too general to model file systems effectively; a domain-specific logic greatly simplifies modeling [18].

A logic of file systems serves three important purposes. First, it enables us to prove properties about existing file system designs, resulting in better understanding of the set of guarantees and enabling aggressive performance optimizations that preserve those guarantees. Second, it significantly lowers the barrier to providing new mechanisms or functionality in the file system by enabling rigorous reasoning about their correctness; in the absence of such a framework, designers tend to stick with "time-tested" alternatives. Finally, the logic helps design functionality in semantically-smart disk systems by facilitating precise characterization and proof of their properties.

A key goal of the logic framework is *simplicity*; in order to be useful to general

file system designers, the barrier to entry in terms of applying the logic should be low. Our logic achieves this by enabling *incremental modeling*. One need not have a complete model of a file system before starting to use the logic; instead, one can simply model a particular piece of functionality or mechanism in isolation and prove properties about it.

Through case studies, we demonstrate the utility and efficacy of our logic in reasoning about file system correctness properties. First, we represent and prove the soundness of important guarantees provided by existing techniques for file system consistency, such as soft updates and journaling. We then use the logic to prove that the Linux ext3 file system is needlessly conservative in its transaction commits, resulting in sub-optimal performance; this case study demonstrates the utility of the logic in enabling aggressive performance optimizations.

To illustrate the utility of the logic in developing new file system functionality, we propose a new file system mechanism called *generation pointers* to enable *consistent undelete* of files. We prove the correctness of our design by incremental modeling of this mechanism in our logic, demonstrating the simplicity of the process. We then implement the mechanism in the Linux ext3 file system, and verify its correctness. As the logic indicates, we empirically show that inconsistency does indeed occur in undeletes in the absence of our mechanism.

Finally, we demonstrate the value of the logic in reasoning about semantic disks. We consider one specific type of semantic information, namely block type, and logically prove that under a certain set of file system behaviors, inference of that information is guaranteed to be accurate.

The rest of this chapter is organized as follows. We first present an extended motivation (§9.2), and a quick context on file systems (§9.3). We present our logic (§9.4), and represent some common file system properties using the logic (§9.5). We then use the logic to prove consistency properties of existing systems (§9.6), prove the correctness of an unexploited performance optimization in ext3 (§9.7), and reason about a new technique for consistent undeletes (§9.8). We then apply our logic to semantic disks (§9.9), and then summarize (§9.10).

9.2 Extended Motivation

A systematic framework for reasoning about the interaction of a file system with the disk has multifarious benefits. We describe three key applications of the framework.

9.2.1 Reasoning about existing file systems

An important usage scenario for the logic is to model existing file systems. There are three key benefits to such modeling. First, it enables a clear understanding of the precise guarantees that a given mechanism provides, and the assumptions under which those guarantees hold. Such an understanding enables correct implementation of functionality at *other* system layers such as the disk system by ensuring that they do not adversely interact with the file system assumptions. For example, write-back caching in disks often results in reordering of writes to the media; this can negate the assumptions journaling is based on.

The second benefit is that it enables more aggressive performance optimizations. When reasoning about complex interactions becomes hard, file system developers tend to be conservative (*e.g.*, perform unnecessarily more synchronous writes). Our logic helps remove this barrier; it enables developers to be more aggressive in their performance optimizations while still being confident of their correctness. In Section 9.7, we analyze a real example of such an opportunity for aggressive performance optimization in the Linux ext3 file system, and show that the logic framework can help prove its correctness.

The final benefit of the logic framework is its potential use in implementation-level model checkers [121]; having a clear model of expected behavior against which to validate an existing file system would perhaps enable more comprehensive and efficient model checking, instead of the current technique of relying on the *fsck* mechanism which is quite expensive; the cost of an fsck on every explored state limits the scalability of such model checking.

9.2.2 Building new file system functionality

Recovery and consistency are traditionally viewed as "tricky" issues to reason about and get right. A classic illustration of this view arises in the context of database recovery; the widely used ARIES [69] algorithm pointed to correctness issues with many earlier proposals. The success of ARIES has also stalled innovation in database recovery, because of the notion that it is too hard to be confident about the correctness of new techniques.

Given that a significant range of innovation within the file system deals with its interaction with the disk and can have correctness implications, this inertia against changing "time-tested" alternatives stifles the incorporation of new functionality in file systems. A systematic framework to reason about a new piece of functionality can greatly reduce this barrier to entry. In Section 9.8, we propose a new file system functionality and use our logic to prove its correctness. To further illustrate the effi-

cacy of the logic in reasoning about new functionality, we examine in Section 9.6.2 a very common file system feature, *i.e.*, journaling, and show that starting from a simple logical model of journaling, we can systematically arrive at the various corner cases that need to be handled, some of which involve admittedly complex interactions [112].

A key attraction of the logic framework is that it enables *incremental modeling*; one need not have a complete model of a file system in order to start proving properties, but instead can model the specific subset of the file system that is impacted by the new functionality.

9.2.3 Designing semantically-smart disks

The logic framework also significantly simplifies reasoning about semantically-smart disk systems. As we showed in our various case studies, inferring information accurately underneath modern file systems is quite complex, especially because it is dependent on the dynamic properties of the file system. In Section 9.9, we show that the logic can simplify reasoning about a semantic disk; this can in turn enable more aggressive classes of functionality in semantic disks.

9.3 Background

In this section, we provide some basic context information about file systems. While Chapter 2 provided detailed background information about file systems, this section provides a high-level background for a general model of a file system.

A file system organizes disk blocks into logical files and directories. In order to map blocks to logical entities such as files, the file system tracks various forms of *metadata*; correct maintenance of this metadata is crucial to enabling proper retrieval of data by the file system. In this section, we first describe the forms of metadata that file systems track, and then discuss the issue of file system consistency. Finally, we describe the asynchrony of file systems, that is a major source of complexity in its interaction with disk.

9.3.1 File system metadata

File system metadata can be classified into three types:

Directories: Directories map a logical file name to per-file metadata. Since the file mapped for a name can be a directory itself, directories enable a hierarchy of files. When a user opens a file specifying its *path name*, the file system locates the per-file metadata for the file, reading each directory in the path if required.

File metadata: File metadata contains information about a specific file. Examples of such information are the set of disk blocks that comprise the file, the size of the file, and so on. In certain file systems such as FAT, file metadata is embedded in the directory entries, while in most other file systems, file metadata is stored separately (*e.g.*, inodes) and is pointed to by the directory entries. The pointers from file metadata to the actual disk blocks can sometimes be indirected through *indirect pointer* blocks which in turn contain pointers to actual data blocks.

Allocation structures: File systems manage various resources on disk such as the set of free disk blocks that can be allocated to new files, and the set of free file metadata instances. To track such resources, file systems maintain structures (*e.g.*, bitmaps, free lists) that indicate for each instance of the resource, whether it is free or busy.

In addition, file systems track other metadata (*e.g.*, super block for bootstrapping, etc.), but we mainly focus on the three major types discussed above.

9.3.2 File system consistency

For proper operation, the internal metadata of the file system and its data blocks should be in a *consistent* state. By consistency, we mean that the state of the various metadata structures obeys a set of invariants that the file system relies on. For example, a directory entry should only point to a valid file metadata structure; if a directory points to file metadata that is uninitialized (*i.e.*, marked free), the file system is said to be *inconsistent*.

Most file systems provide metadata consistency, since that is crucial to correct operation. A stronger form of consistency is *data consistency*, where the file system guarantees that data block contents always correspond to the file metadata structures that point to them. We discuss this issue in Section 9.6.1. Many modern file systems such as Linux ext3 and ReiserFS provide data consistency.

9.3.3 File system asynchrony

An important characteristic of most modern file systems is the *asynchrony* they exhibit during updates to data and metadata. Updates are simply buffered in memory and are written to disk only after a certain delay interval, with possible reordering among those writes. While such asynchrony is crucial for performance, it complicates consistency management. Due to asynchrony, a system crash leads to a state where an arbitrary subset of updates has been applied on disk, potentially leading to an inconsistent on-disk state. Asynchrony of updates is the principal reason for complexity in the interaction of a file system with the disk.

9.4 The Formalism

In this section, we define a formal model of a file system. We first define the basic entities in the model and the various relationships among them. We then present basic operations and logical postulates.

9.4.1 Basic entities

The basic entities in our model are *containers*, *pointers*, and *generations*. A file system is simply a collection of containers. Containers are linked to each other through pointers. Each file system differs in the exact types of containers it defines and the relationship it allows between those container types; this abstraction based on containers and pointers is thus general to describe any file system.

Containers in a file system can be *freed* and *reused*; A container is considered to be free when it is not pointed to by any other container; it is *live* otherwise. The instance of a container between a reuse and the next free is called a *generation*; thus, a generation is a specific incarnation of a container. Generations are never reused. When a container is reused, the previous generation of that container is freed and a new generation of the container comes to life. A generation is thus fully defined by its container plus a logical *generation number* that tracks how many times the container was reused. Note that generation does *not* refer to the *contents* of a container, but is an abstraction for its current incarnation; contents can change without affecting the generation.

We illustrate the notion of containers and generations with a simple example from a typical UNIX-based file system. If the file system contains a fixed set of designated *inodes*, each inode slot is a *container*. At any given point, an inode slot in use is associated with an inode *generation* that corresponds to a specific file. When the file is deleted, the corresponding inode generation is deleted (forever), but the inode container is simply marked free. A different file created later can reuse the same inode container for a logically different inode generation. Similarly, a directory container is the block in which the directory entries are stored; the block can have a different generation at a later time. Pointers from the directory to inodes are pointers to the respective inode *containers*, because directory entries typically just contain the inode number.

Note that a single container (*e.g.*, an inode) can point to multiple containers (*e.g.*, data blocks). A single container can also be sometimes pointed to by multiple containers (*e.g.*, hard links in UNIX file systems). Data block containers typically do not point to any other container.

Symbol	Description
$\&A$	set of entities that point to container A
$*A$	set of entities pointed to by container A
$\|A\|$	container that tracks if container A is live
$\&a$	set of entities that point to generation a
$*a$	set of entities pointed to by generation a
$A \rightarrow B$	denotes that container A has a pointer to B
$\&A = \emptyset$	denotes that no entity points to A
A^k	the kth epoch of container A
$t(A^k)$	type of kth epoch of container A
$g(A^k)$	generation of the kth epoch of container A
$C(a)$	container associated with generation a
A_k	generation k of container A

Table 9.1: **Notations on containers and generations.**

Notations

The notations used to depict the basic entities and the relationships across them are listed in Table 9.1. Containers are denoted by upper case letters, while generations are denoted by lower case letters. An "entity" in the description represents a container or a generation. Note that some notations in the table are defined only later in the section.

Memory and disk versions of containers

A file system needs to manage its structures across two domains: volatile memory and disk. Before accessing the contents of a container, the file system needs to *read* the on-disk version of the container into memory. Subsequently, the file system can make modifications to the in-memory copy of the container, and such modified contents are periodically written to disk after a certain duration of residence in memory. Thus, until the file system writes a modified container to disk, the contents of a modified container in memory will be different from that on disk.

9.4.2 Beliefs and actions

We formulate the logic in terms of *beliefs* and *actions*. A belief represents a certain state in memory or disk, while an action is some operation performed by the file system, resulting in a certain set of beliefs.

A belief represents the creation or existence of a certain *state*. Any statement enclosed within $\{\}$ represents a belief. Beliefs can be either *in memory* beliefs or *on disk* beliefs, and are denoted as either $\{\}_M$ or $\{\}_D$ respectively. For example $\{A \rightarrow B\}_M$ indicates that $A \rightarrow B$ is a belief in the file system memory, *i.e.*, container A currently points to B in memory, while $\{A \rightarrow B\}_D$ means it is a disk belief. While memory beliefs just represent the state the file system tracks in memory, on-disk beliefs are defined as follows: a belief holds on disk at a given time, if on a crash, the file system can conclude with the same belief purely based on a scan of on-disk state at that time. On-disk beliefs are thus solely dependent on on-disk data.

Since the file system manages free and reuse of containers, its beliefs can be in terms of *generations*; for example $\{A_k \rightarrow B_j\}_M$ is valid (note that A_k refers to generation k of container A). However, on-disk beliefs can only deal with containers, since generation information is lost at the disk. In Sections 9.8 and 9.9, we propose techniques to expose generation information to the disk, and show that it enables better guarantees.

The other component of our logic is *actions*, which result in changes to the set of beliefs that hold at a given time. There are two actions defined in our logic, both performed by the file system on the disk:

- $read(A)$ – This operation is used by the file system to read the contents of an on-disk container (and thus, its current generation) into memory. The file system needs to have the container in memory before it can modify it. After a $read$, the contents of A in memory and on-disk are the same, *i.e.*, $\{A\}_M = \{A\}_D$.
- $write(A)$ – This operation results in flushing the current contents of a container to disk. After this operation, the contents of A in memory and on-disk are the same, *i.e.*, $\{A\}_D = \{A\}_M$.

9.4.3 Ordering of beliefs and actions

A fundamental aspect of the interaction of a file system with disk is the *ordering* among its actions. The ordering of actions also determines the order in which beliefs are established. The operation of a file system can be viewed as a partial ordering on the beliefs and actions it goes through. To order actions and the resulting beliefs, we use the *before* ($\ll$) and *after* ($\gg$) operators. Thus, $\alpha \ll \beta$ means that α occurred before β in time. Note that by *ordering* beliefs, we are using the $\{\}$ notation as both a way of indicating the *event* of creation of the belief, and the *state* of existence of a belief. For example, the belief $\{B \rightarrow A\}_M$ represents the event where the file system assigns A as one of the pointers from B.

We also use a special ordering operator called *precedes* ($\prec$). Only a belief can appear to the left of a $\prec$ operator. The $\prec$ operator is defined as follows: $\alpha \prec \beta$ means that belief α occurs before β (*i.e.*, $\alpha \prec \beta \Rightarrow \alpha \ll \beta$); further, it means that belief α holds at least until β occurs. This implies there is no intermediate action or event between α and β that invalidates belief α. The operator $\prec$ is *not* transitive; $\alpha \prec \beta \prec \gamma$ does not imply $\alpha \prec \gamma$, because belief α needs to hold only until β and not necessarily until γ (note that $\alpha \prec \beta \prec \gamma$ is simply a shortcut for $(\alpha \prec \beta) \wedge (\beta \prec \gamma) \wedge (\alpha \ll \gamma)$).

9.4.4 Proof system

Given our primitives for sequencing beliefs and actions, we can define *rules* in our logic in terms of an *implication* of one event sequence given another sequence. We use the traditional operators: $\Rightarrow$ (implication) and $\Leftrightarrow$ (double implication, *i.e.*, if and only if). We also use the logical AND operator ($\wedge$) and logical OR ($\vee$) to combine sequences.

An example of a logical rule is: $\alpha \ll \beta \Rightarrow \gamma$. This notation means that *every time* an event or action β occurs after α, event γ occurs *at the point of occurrence of* β. Another example of a rule is $\alpha \ll \beta \Rightarrow \alpha \ll \gamma \ll \beta$; this rule denotes that every time β occurs after α, γ should have occurred sometime between α and β. Note that in such a rule where the same event occurs in both sides, the event constitutes a temporal reference point by referring to the same time instant in both LHS and RHS. This temporal interpretation of identical events is crucial to the above rule serving the intended implication; otherwise the RHS could refer to some other instant where $\alpha \ll \beta$.

Rules such as the above can be used in logical proofs by *event sequence substitution*; for example, with the rule $\alpha \ll \beta \Rightarrow \gamma$, whenever the subsequence $\alpha \ll \beta$ occurs in a sequence of events, it logically implies the event γ. We could then apply the above rule to any event sequence by replacing any subsequence that matches the left half of the rule, with the right half; thus, with the above rule, we have the following postulate: $\alpha \ll \beta \ll \delta \Rightarrow \gamma \ll \delta$.

9.4.5 Attributes of containers

To make the logic expressive for modern file systems, we extend the vocabulary of the logic with attributes on a container; a generation has the same attributes as its container. We define three attributes: epoch, type, and sharing.

Epoch

The *epoch* of a container is defined as follows: every time the *contents* of a container change *in memory*, its epoch is incremented. Since the file system can batch multiple changes to the contents due to buffering, the set of epochs visible at the disk is a subset of the total set of epochs a container goes through. We denote an epoch by the superscript notation; A^k denotes the kth epoch of A. Note that our definition of epoch is only used for expressivity of our logic; it does not imply that we expect the file system to track such an epoch. Also note the distinction between an *epoch* and a *generation*; a generation change occurs only on a reuse of the container, while an epoch changes on every change in contents *or* when the container is reused.

Type

Containers can have a certain *type* associated with them. The type of a container can either be *static*, *i.e.*, it does not change during the lifetime of the file system, or can be *dynamic*, where the same container can belong to different types at different points in time. For example, in typical FFS-based file systems, *inode* containers are statically typed, while block containers may change their type between data, directory and indirect pointers. We denote the type of a container A by the notation $t(A)$.

Shared vs. unshared

A container that is pointed to by more than one container is called a *shared container*; a container that has exactly one pointer leading into it is unshared. By default, we assume that containers are shared. We denote unshared containers with the $\oplus$ operator. $\oplus A$ indicates that A is unshared. Note that being unshared is a property of the container *type* that the file system always ensures; a container belonging to a type that is unshared, will always have only one pointer pointing into it. For example, most file systems designate data block containers to be unshared.

9.4.6 Logical postulates

In this subsection, we present the fundamental rules that govern the behavior of event sequences and beliefs. We first define the composition of our operators, and then present the rules of our logic using those operators.

Operator composition

We define the following rules on how the operators in our logic compose with each other:

- The operator AND ($\wedge$) is distributive with the precedes($\prec$) operator and the other ordering operators ($\ll$ and $\gg$).

$$\alpha \wedge (\beta \prec \gamma) \quad \Rightarrow \quad (\alpha \wedge \beta) \prec (\alpha \wedge \gamma) \tag{9.1}$$

- The $\prec$ operator distributes over itself according to the following rule (note the parentheses used to group the beliefs, which are otherwise not transitive):

$$(\{st1\} \prec \{st2\}) \prec \{st3\} \quad \Rightarrow \quad \begin{aligned} &\{st1\} \prec \{st2\} \wedge \\ &\{st1\} \prec \{st3\} \wedge \\ &\{st2\} \prec \{st3\} \end{aligned} \tag{9.2}$$

Basic rules

We now present the basic rules that govern the transition of beliefs across memory and disk, and the actions leading to them.

- If a container B points to A in memory, its current generation also points to A in memory.

$$\{B^x \rightarrow A\}_M \quad \Leftrightarrow \quad \{g(B^x) \rightarrow A\}_M \tag{9.3}$$

- If B points to A in memory, a $write$ of B will lead to the disk belief that B points to A.

$$\{B \rightarrow A\}_M \prec write(B) \quad \Rightarrow \quad \{B \rightarrow A\}_D \tag{9.4}$$

  The converse states that the disk belief implies that the same belief first occurred in memory.

$$\{B \rightarrow A\}_D \quad \Rightarrow \quad \{B \rightarrow A\}_M \ll \{B \rightarrow A\}_D \tag{9.5}$$

- Similarly, if B points to A on disk, a $read$ of B will result in the file system inheriting the same belief.

$$\{B \rightarrow A\}_D \prec\ read(B) \quad \Rightarrow \quad \{B \rightarrow A\}_M \tag{9.6}$$

- If the on-disk contents of container A pertain to epoch y, some generation c should have pointed to generation $g(A^y)$ in memory, followed by $write(A)$. The converse also holds:

$$\{A^y\}_D \Rightarrow \quad \{c \rightarrow g(A^y)\}_M \prec\ write(A) \ll \{A^y\}_D \tag{9.7}$$

$$\{c \rightarrow A_k\}_M \prec\ write(A) \Rightarrow \{A^y\}_D \wedge\ (g(A^y) = k) \tag{9.8}$$

- If $\{b \rightarrow A_k\}$ and $\{c \rightarrow A_j\}$ hold in memory at two different points in time, container A should have been freed between those instants.

$$\begin{aligned} &\{b \rightarrow A_k\}_M \ll\ \{c \rightarrow A_j\}_M \wedge\ (k \neq j) \\ &\Rightarrow \{b \rightarrow A_k\}_M \ll\ \{\&A = \emptyset\}_M \prec\ \{c \rightarrow A_j\}_M \end{aligned} \tag{9.9}$$

Note that the rule includes the scenario where an intermediate generation A_l occurs between A_k and A_j.

- If container B pointed to A on disk, and subsequently the file system removes the pointer from B to A in memory, a write of B will lead to the disk belief that B does not point to A.

$$\begin{aligned} &\{B \rightarrow A\}_D \prec\ \{A \notin\ *B\}_M \prec\ write(B) \\ &\Rightarrow \{A \notin\ *B\}_D \end{aligned} \tag{9.10}$$

Further, if A is an unshared container, the write of B will lead to the disk belief that no container points to A, *i.e.*, A is *free*.

$$\begin{aligned} &\oplus A \wedge\ (\{B \rightarrow A\}_D \prec\ \{\&A = \emptyset\}_M \prec\ write(B)) \\ &\Rightarrow \{\&A = \emptyset\}_D \end{aligned} \tag{9.11}$$

- If A is a dynamically typed container, and its type at two instants are different, A should have been freed in between.

$$\begin{aligned} &(\{t(A) = x\}_M \ll\ \{t(A) = y\}_M) \wedge\ (x \neq y) \\ &\Rightarrow \{t(A) = x\}_M \ll\ \{\&A = \emptyset\}_M \prec\ \{t(A) = y\}_M \end{aligned} \tag{9.12}$$

9.5 File System Properties

Various file systems provide different guarantees on their update behavior. Each guarantee translates into new rules to the logical model of the file system, and can be used to complement our basic rules when reasoning about that file system. In this section, we discuss three such properties.

9.5.1 Container exclusivity

A file system exhibits *container exclusivity* if it guarantees that for every on-disk container, there is at most one dirty copy of the container's contents in the file system cache. It also requires the file system to ensure that the in-memory contents of a container do not change while the container is being written to disk. Many file systems such as BSD FFS, Linux ext2 and VFAT exhibit container exclusivity; some journaling file systems like ext3 do not exhibit this property. In our equations, when we refer to containers in memory, we refer to the latest epoch of the container in memory, in the case of file systems that do not obey container exclusivity. For example, in eq. 9.11, $\{\&A = \emptyset\}_M$ means that at that time, there is no container whose latest epoch in memory points to A; similarly, $write(B)$ means that the latest epoch of B at that time is being written. When referring to a specific version, we use the epoch notation. Of course, if container exclusivity holds, only one epoch of any container exists in memory.

Under container exclusivity, we have a stronger converse for eq. 9.4:

$$\{B \to A\}_D \quad \Rightarrow \quad \{B \to A\}_M \prec \{B \to A\}_D \tag{9.13}$$

If we assume that A is unshared, we have a stronger equation following from equation 9.13, because the only way the disk belief $\{B \to A\}_D$ can hold is if B was written by the file system. Note that many containers in typical file systems (such as data blocks) are unshared.

$$\begin{aligned} \{B \to A\}_D \quad \Rightarrow \quad & \{B \to A\}_M \prec \\ & (write(B) \ll \{B \to A\}_D) \end{aligned} \tag{9.14}$$

9.5.2 Reuse ordering

A file system exhibits *reuse ordering* if it ensures that before reusing a container, it commits the freed state of the container to disk. For example, if A is pointed to by generation b in memory, later freed (*i.e.*, $\&A = \emptyset$), and then another generation c

is made to point to A, the freed state of A (*i.e.*, the container of generation b, with its pointer removed) is written to disk before the reuse occurs.

$$\{b \rightarrow A\}_M \prec \{\&A = \emptyset\}_M \prec \{c \rightarrow A\}_M$$
$$\Rightarrow \{\&A = \emptyset\}_M \prec write(C(b)) \ll \{c \rightarrow A\}_M$$

Since every reuse results in such a commit of the freed state, we could extend the above rule as follows:

$$\{b \rightarrow A\}_M \ll \{\&A = \emptyset\}_M \prec \{c \rightarrow A\}_M$$
$$\Rightarrow \{\&A = \emptyset\}_M \prec write(C(b)) \ll \{c \rightarrow A\}_M \tag{9.15}$$

FFS with soft updates [32], and Linux ext3 are two examples of file systems that exhibit reuse ordering.

9.5.3 Pointer ordering

A file system exhibits *pointer ordering* if it ensures that before writing a container B to disk, the file system writes all containers that are pointed to by B.

$$\{B \rightarrow A\}_M \prec write(B)$$
$$\Rightarrow \{B \rightarrow A\}_M \prec (write(A) \ll write(B)) \tag{9.16}$$

FFS with soft updates is an example of a file system that exhibits pointer ordering.

9.6 Modeling Existing Systems

Having defined the basic formalism of our logic, we proceed to using the logic to model and reason about file system behaviors. In this section, we present proofs for two properties important for file system consistency. First, we discuss the *data consistency* problem in a file system. We then model a journaling file system and reason about the *non-rollback* property in a journaling file system.

9.6.1 Data consistency

We first consider the problem of *data consistency* of the file system after a crash. By data consistency, we mean that the contents of data block containers have to be consistent with the metadata that references the data blocks. In other words, a file

should not end up with data from a different file when the file system recovers after a crash. Let us assume that B is a file metadata container (i.e. contains pointers to the data blocks of the respective file), and A is a data block container. Then, if the disk belief that B^x points to A holds, and the on-disk contents of A were written when k was the generation of A, then epoch B^x should have pointed (at some time in the past) exactly to the kth generation of A in memory, and not a different generation. The following rule summarizes this:

$$\begin{aligned} \{B^x \rightarrow A\}_D \wedge \{A^y\}_D \quad \Rightarrow \quad & (\{B^x \rightarrow A_k\}_M \ll \{B^x \rightarrow A\}_D) \\ & \wedge (k = g(A^y)) \end{aligned}$$

For simplicity, let us make a further assumption that the data containers in our file system are nonshared ($\oplus A$), i.e. different files do not share data block pointers. Let us also assume that the file system obeys the container exclusivity property. Many modern file systems such as ext2 and VFAT have these properties. Since under block exclusivity $\{B^x \rightarrow A\}_D \Rightarrow \{B^x \rightarrow A\}_M \prec \{B^x \rightarrow A\}_D$ (by eq. 9.13), we can rewrite the above rule as follows:

$$\begin{aligned} & (\{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D) \wedge \{A^y\}_D \\ & \Rightarrow (k = g(A^y)) \end{aligned} \tag{9.17}$$

If this rule does not hold, it means that the file represented by the generation $g(B^x)$ points to a generation k of A, but the contents of A were written when its generation was $g(A^y)$, clearly a case of data corruption.

To show that this rule does not always hold, we assume the negation and prove that it is reachable as a sequence of valid file system actions ($\alpha \Rightarrow \beta \equiv \neg(\alpha \wedge \neg\beta)$).

From eq. 9.7, we have $\{A^y\}_D \Rightarrow \{c \rightarrow g(A^y)\}_M \prec write(A)$. Thus, we have two event sequences implied by the LHS of eq. 9.17:

$$\begin{aligned} i. \quad & \{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \\ ii. \quad & \{c \rightarrow g(A^y)\}_M \prec write(A) \end{aligned}$$

Thus, in order to prove eq. 9.17, we need to prove that every possible interleaving of the above two sequences, together with the clause $(k \neq g(A^y))$ is invalid. To disprove eq. 9.17, we need to prove that at least one of the interleavings is valid.

Since $(k \neq g(A^y))$, and since $\{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D$, the event $\{c \rightarrow g(A^y)\}_M$ cannot occur in between those two events, due to container exclusivity and because A is unshared. Similarly $\{B^x \rightarrow A_k\}_M$ cannot occur between $\{c \rightarrow g(A^y)\}_M \prec write(A)$. Thus, we have only two interleavings:

1. $\{B^x \to A_k\}_M \prec \{B^x \to A\}_D \ll \{c \to g(A^y)\}_M \prec write(A)$
2. $\{c \to g(A^y)\}_M \prec write(A) \ll \{B^x \to A_k\}_M \prec \{B^x \to A\}_D$

Case 1:

Applying eq. 9.3,

$$\begin{aligned} \Rightarrow \quad & \{g(B^x) \to A_k\}_M \prec \{B^x \to A\}_D \\ & \ll \{c \to g(A^y)\}_M \prec write(A) \wedge (k \neq g(A^y)) \end{aligned}$$

Applying eq. 9.9,

$$\begin{aligned} \Rightarrow \quad & \{g(B^x) \to A_k\}_M \prec \{B^x \to A\}_D \\ & \ll \{\&A = \emptyset\}_M \prec \{c \to g(A^y)\}_M \prec write(A) \end{aligned} \tag{9.18}$$

Since step 9.18 is a valid sequence in file system execution, where generation A^k could be freed due to delete of file represented by generation $g(B^x)$ and then a subsequent generation of the block is reallocated to the file represented by generation c in memory, we have shown that this violation could occur.

Let us now assume that our file system obeys reuse ordering, i.e. equation 9.15. Under this additional constraint, equation 9.18 would imply the following:

$$\begin{aligned} \Rightarrow \quad & \{g(B^x) \to A_k\}_M \prec \{B^x \to A\}_D \prec \\ & \{\&A = \emptyset\}_M \prec write(B) \ll \\ & \{c \to g(A^y)\}_M \prec write(A) \end{aligned}$$

By eq. 9.11,

$$\begin{aligned} \Rightarrow \quad & \{g(B^x) \to A_k\}_M \prec \{B^x \to A\}_D \prec \\ & \{\&A = \emptyset\}_D \ll \{c \to g(A^y)\}_M \prec \\ & write(A) \\ \Rightarrow \quad & \{\&A = \emptyset\}_D \wedge \{A_c\}_D \end{aligned} \tag{9.19}$$

This is however, a contradiction under the initial assumption we started off with, i.e. $\{\&A = B\}_D$. Hence, under reuse ordering, we have shown that this particular scenario does not arise at all.

Case 2: $\{c \to g(A^y)\}_M \prec write(A) \ll \{B^x \to A_k\}_M \prec \{B^x \to A\}_D \wedge (k \neq g(A^y))$

Again, applying eq. 9.3,

$$\begin{aligned} \Rightarrow \quad & (k \neq g(A^y)) \wedge \{c \to g(A^y)\}_M \prec write(A) \ll \\ & \{g(B^x) \to A_k\}_M \prec \{B^x \to A\}_D \end{aligned}$$

By eqn 9.9,

$$\Rightarrow \quad \{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{\&A = \emptyset\}_M \\ \prec \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \tag{9.20}$$

Again, this is a valid file system sequence where file generation c pointed to data block generation $g(A^y)$, the generation $g(A^y)$ gets deleted, and a new generation k of container A gets assigned to file generation $g(B^x)$. Thus, consistency violation can also occur in this scenario.

Interestingly, when we apply eq. 9.15 here, we get

$$\Rightarrow \quad \{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{\&A = \emptyset\}_M \\ \prec write(C(c)) \ll \{g(B^x) \rightarrow A_k\}_M \\ \prec \{B^x \rightarrow A\}_D$$

However, we cannot apply eq. 9.11 in this case because the belief $\{C \rightarrow A\}_D$ need not hold. Even if we did have a rule that led to the belief $\{\&A = \emptyset\}_D$ immediately after $write(C(c))$, that belief will be overwritten by $\{B^x \rightarrow A\}_D$ later in the sequence. Thus, eq. 9.15 does not invalidate this sequence; reuse ordering thus does not guarantee data consistency in this case.

Let us now make another assumption, that the file system also obeys pointer ordering (eq. 9.16).

Since we assume that A is unshared, and that container exclusivity holds, we can apply eq. 9.14 to equation 9.20.

$$\Rightarrow \quad \{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{\&A = \emptyset\}_M \prec \\ \{g(B^x) \rightarrow A_k\}_M \prec write(B) \ll \{B^x \rightarrow A\}_D \tag{9.21}$$

Now applying the pointer ordering rule (eqn 9.16,),

$$\Rightarrow \quad \{c \rightarrow g(A^y)\}_M \prec write(A) \ll \{\&A = \emptyset\}_M \prec \\ \{g(B^x) \rightarrow A_k\}_M \prec write(A) \ll write(B) \\ \ll \{B^x \rightarrow A\}_D$$

By eq. 9.8,

$$\Rightarrow \quad \{c \rightarrow A\}_M \prec write(A) \ll \{\&A = \emptyset\}_M \prec \\ \{A^y\}_D \ll write(B) \ll \{B^x \rightarrow A\}_D \wedge (k = g(A^y)) \\ \Rightarrow \quad \{A^y\}_D \wedge \{B^x \rightarrow A\}_D \wedge (k = g(A^y)) \tag{9.22}$$

This is again a contradiction, since this implies that the contents of A on disk belong to the same generation A_k, while we started out with the assumption that

$g(A^y) \neq k$.

Thus, under reuse ordering and pointer ordering, the file system never suffers a data consistency violation. If the file system does not obey any such ordering (such as ext2), data consistency could be compromised on crashes. Note that this inconsistency is fundamental, and cannot be fixed by scan-based consistency tools such as *fsck*.

9.6.2 Modeling file system journaling

We now extend our logic with rules that define the behavior of a journaling file system. We then use the model to reason about a key property in a journaling file system.

Journaling is a technique commonly used by file systems to ensure metadata consistency. When a single file system operation spans multiple changes to metadata structures, the file system groups those changes into a *transaction* and guarantees that the transaction commits atomically; either all changes reach disk or none of them reach disk, thus preserving consistency. To provide atomicity, the file system first writes the changes to a *write-ahead log (WAL)*, and propagates the changes to the actual on-disk location only after the transaction is *committed* to the log. A transaction is committed when all changes are logged, and a special "commit" record is written to log indicating completion of the transaction. When the file system recovers after a crash, a checkpointing process *replays* all changes that belong to committed transactions.

To model journaling, we consider a logical "transaction" object that determines the set of *log record* containers that belong to that transaction, and thus logically contains pointers to the log copies of all containers modified in that transaction. We denote the log copy of a journaled container by the ˆ symbol on top of the container name; $\hat{A}^x$ is thus a container in the *log*, *i.e.*, *journal* of the file system (note that we assume *physical logging*. such as the block-level logging in ext3). The physical realization of the transaction object is the "commit" record, since it logically points to all containers that changed in that transaction. For WAL property to hold, the commit container should be written only after the log copy of all modified containers that the transaction points to, are written.

If T is the commit container, the WAL property leads to the following two rules:

$$\{T \rightarrow \hat{A}^x\}_M \prec write(T) \Rightarrow \{T \rightarrow \hat{A}^x\}_M \prec (write(\hat{A}^x) \ll write(T)) \tag{9.23}$$

$$\{T \rightarrow \hat{A}^x\}_M \prec write(A^x) \Rightarrow \{T \rightarrow \hat{A}^x\}_M \prec (write(T) \ll write(A^x)) \tag{9.24}$$

The first rule states that the transaction is not committed (*i.e.*, commit record not written) until all containers belonging to the transaction are written to disk. The second rule states that the on-disk copy of a container is written only after the transaction in which the container was modified, is committed to disk. Note that unlike the normal pointers considered so far that point to containers or generations, the pointers from container T in the above two rules point to *epochs*. These *epoch pointers* are used because a commit record is associated with a specific epoch (*e.g.*, snapshot) of the container.

The replay or checkpointing process can be depicted by the following two rules.

$$\{T \rightarrow \hat{A^x}\}_D \wedge \{T\}_D \;\; \Rightarrow \;\; write(A^x) \ll \{A^x\}_D \tag{9.25}$$

$$\{T_1 \rightarrow \hat{A^x}\}_D \wedge \{T_2 \rightarrow \hat{A^y}\}_D \wedge (\{T_1\}_D \ll \{T_2\}_D) \Rightarrow write(A^y) \ll \{A^y\}_D \tag{9.26}$$

The first rule says that if a container is part of a transaction and the transaction is committed on disk, the on-disk copy of the container is updated with the logged copy pertaining to that transaction. The second rule says that if the same container is part of multiple committed transactions, the on-disk copy of the container is updated with the copy pertaining to the last of those transactions.

The following belief transitions hold:

$$(\{T \rightarrow \hat{B^x}\}_M \wedge \{B^x \rightarrow A\}_M) \prec write(T) \Rightarrow \{B^x \rightarrow A\}_D \tag{9.27}$$

$$\{T \rightarrow \hat{A^x}\}_M \prec write(T) \Rightarrow \{A^x\}_D \tag{9.28}$$

Rule 9.27 states that if B^x points to A and $\hat{B^x}$ belongs to transaction T, the commit of T leads to the disk belief $\{B^x \rightarrow A\}_D$. Rule 9.28 says that the disk belief $\{A^x\}_D$ holds immediately on commit of the transaction which $\hat{A^x}$ is part of; creation of the belief does not require the checkpoint write to happen. As described in §9.4.2, a disk belief pertains to the belief the file system would reach, if it were to start from the current disk state.

In certain journaling file systems, it is possible that only containers of certain types are journaled; updates to other containers directly go to disk, without going through the transaction machinery. In our proofs, we will consider the cases of both complete journaling (where all containers are journaled) and selective journaling (only containers of a certain type). In the selective case, we also address the possibility of a container changing its type from a journaled type to a non-journaled type and vice versa. For a container B that belongs to a journaling type, we have the following converse of equation 9.27:

$$\{B^x \rightarrow A\}_D \;\Rightarrow\; (\{T \rightarrow \hat{B^x}\}_M \wedge \{B^x \rightarrow A\}_M) \prec write(T) \ll \{B^x \rightarrow A\}_D \quad (9.29)$$

We can show that in complete journaling, data inconsistency never occurs; we omit this due to space constraints.

The non-rollback property

We now introduce a new property called *non-rollback* that is pertinent to file system consistency. We first formally define the property and then reason about the conditions required for it to hold in a journaling file system.

The non-rollback property states that the contents of a container on disk are never overwritten by *older* contents from a previous epoch. This property can be expressed as:

$$\{A^x\}_D \ll \{A^y\}_D \;\Rightarrow\; \{A^x\}_M \ll \{A^y\}_M \quad (9.30)$$

The above rule states that if the on-disk contents of A move from epoch x to y, it should logically imply that epoch x occurred before epoch y in memory as well. The non-rollback property is crucial in journaling file systems; absence of the property could lead to data corruption.

If the disk believes in the xth epoch of A, there are only two possibilities. If the type of A^x was a journaled type, A^x should have belonged to a transaction and the disk must have observed the commit record for the transaction; as indicated in eq 9.28, the belief of $\{A^x\}_D$ occurs immediately after the commit. However, at a later point the actual contents of A^x will be written by the file system as part of its checkpoint propagation to the actual on-disk location, thus re-establishing belief $\{A^x\}_D$. If J is the set of all journaled types,

$$\{A^x\}_D \wedge \{t(A^x) \in J\}_M \Rightarrow (\{A^x\}_M \wedge \{T \rightarrow \hat{A^x}\}_M) \prec write(T) \ll \{A^x\}_D \ll write(A^x) \ll \{A^x\}_D \quad (9.31)$$

The second possibility is that A^x is of a type that is not journaled. In this case, the only way the disk could have learnt of it is by a prior commit of A^x.

$$\{A^x\}_D \wedge \{t(A^x) \notin J\}_M \;\Rightarrow\; \{A^x\}_M \prec write(A^x) \ll \{A^x\}_D \quad (9.32)$$

A^x and A^y are journaled:

Let us first assume that both A^x and A^y belong to a journaled type. To prove the non-rollback property, we consider the LHS of eq 9.30: $\{A^x\}_D \ll \{A^y\}_D$; since both A^x and A^y are journaled, we have the following two sequence of events that led to the two beliefs (by eq. 9.31):

$$(\{A^x\}_M \wedge \{T_1 \rightarrow \hat{A^x}\}_M) \prec write(T_1) \ll \{A^x\}_D$$
$$\ll write(A^x) \ll \{A^x\}_D$$

$$(\{A^y\}_M \wedge \{T_2 \rightarrow \hat{A^y}\}_M) \prec write(T_2) \ll \{A^y\}_D$$
$$\ll write(A^y) \ll \{A^y\}_D$$

Omitting the $write$ actions in the above sequences for simplicity, we have the following sequences of events:

$$i. \quad \{A^x\}_M \ll \{A^x\}_D \ll \{A^x\}_D$$
$$ii. \quad \{A^y\}_M \ll \{A^y\}_D \ll \{A^y\}_D$$

Note that in each sequence, there are two instances of the *same* disk belief being created: the first instance is created when the corresponding transaction is committed, and the second instance, when the checkpoint propagation happens at a later time. In snapshot-based coarse-grained journaling systems (such as ext3), transactions are always committed in order. Thus, if epoch A^x occurred before A^y, T_1 will be committed before T_2 (*i.e.*, the first instance of $\{A^x\}_D$ will occur before the first instance of $\{A^y\}_D$. Another property true of such journaling is that the checkpointing is in-order as well; if there are two committed transactions with different copies of the same data, only the version pertaining to the later transaction is propagated during checkpoint.

Thus, the above two sequences of events lead to only two interleavings, depending on whether epoch x occurs before epoch y or vice versa. Once the ordering between epoch x and y is fixed, the rest of the events are constrained to a single sequence:

Interleaving 1:

$$(\{A^x\}_M \ll \{A^y\}_M) \wedge (\{A^x\}_D \ll \{A^y\}_D \ll \{A^y\}_D)$$
$$\Rightarrow \quad \{A^x\}_M \ll \{A^y\}_M$$

Interleaving 2:

$$\Rightarrow \quad (\{A^y\}_M \ll \{A^x\}_M) \wedge (\{A^y\}_D \ll \{A^x\}_D \ll \{A^x\}_D)$$

$$\Rightarrow \quad \{A^y\}_D \ll \{A^x\}_D$$

Thus, the second interleaving results in a contradiction from our initial statement we started with (*i.e.*, $\{A^x\}_D \ll \{A^y\}_D$). Therefore the first interleaving is the only legal way the two sequences of events could be combined. Since the first interleaving implies that $\{A^x\}_M \ll \{A^y\}_M$, we have proved that if the two epochs are journaled, the non-rollback property holds.

A^y is journaled, but A^x is not:

We now consider the case where the type of A changes between epochs x and y, such that A^y belongs to a journaled type and A^x does not.

We again start with the statement $\{A^x\}_D \ll \{A^y\}_D$. From equations 9.31 and 9.32, we have the following two sequences of events:

$$\begin{array}{ll} i. & (\{A^y\}_M \wedge \{T \rightarrow \hat{A}^y\}_M) \prec write(T) \\ & \ll \{A^y\}_D \ll write(A^y) \ll \{A^y\}_D \\ ii. & \{A^x\}_M \prec write(A^x) \ll \{A^x\}_D \end{array}$$

Omitting the $write$ actions for the sake of readability, the sequences become:

$$\begin{array}{ll} i. & \{A^y\}_M \ll \{A^y\}_D \ll \{A^y\}_D \\ ii. & \{A^x\}_M \ll \{A^x\}_D \end{array}$$

To prove the non-rollback property, we need to show that every possible interleaving of the above two sequences where $\{A^y\}_M \ll \{A^x\}_M$ results in a contradiction, *i.e.*, cannot co-exist with $\{A^x\}_D \ll \{A^y\}_D$.

The interleavings where $\{A^y\}_M \ll \{A^x\}_M$ are:

1. $\{A^y\}_M \ll \{A^x\}_M \ll \{A^x\}_D \ll \{A^y\}_D \ll \{A^y\}_D$
2. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^x\}_M \ll \{A^x\}_D \ll \{A^y\}_D$
3. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^y\}_D \ll \{A^x\}_M \ll \{A^x\}_D$
4. $\{A^y\}_M \ll \{A^x\}_M \ll \{A^y\}_D \ll \{A^x\}_D \ll \{A^y\}_D$
5. $\{A^y\}_M \ll \{A^x\}_M \ll \{A^y\}_D \ll \{A^y\}_D \ll \{A^x\}_D$
6. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^x\}_M \ll \{A^y\}_D \ll \{A^x\}_D$

Scenarios 3,5,and 6 imply $\{A^y\}_D \ll \{A^x\}_D$ and are therefore invalid interleavings.

Scenarios 1,2, and 4 are valid interleavings that do not contradict our initial assumption of disk beliefs, but at the same time, imply $\{A^y\}_M \ll \{A^x\}_M$; these scenarios thus violate the non-rollback property. Therefore, under dynamic typing, the above journaling mechanism does not guarantee non-rollback. Due to this

violation, file contents can be corrupted by stale metadata generations.

Scenario 2 and 4 occur because the checkpoint propagation of earlier epoch A^y which was journaled, occurs *after* A was overwritten as a non-journaled epoch. To prevent this, we need to impose that the checkpoint propagation of a container in the context of transaction T does not happen if the on-disk contents of that container were updated *after* the commit of T. The *journal revoke records* in ext3 precisely guarantee this; if a revoke record is encountered during log replay, the corresponding block is not propagated to the actual disk location.

Scenario 1 happens because a later epoch of A is committed to disk before the transaction which modified an earlier epoch is committed. To prevent this, we need a form of *reuse ordering*, which imposes that before a container changes type (i.e. is reused in memory), the transaction that freed the previous generation be committed. Since transactions commit in order, and the freeing transaction should occur *after* transaction T which used A^y in the above example, we have the following guarantee:

$$\{t(A^y) \in J\}_M \wedge \{t(A^x) \notin J\}_M \wedge (\{A^y\}_M \ll \{A^x\}_M) \\ \Rightarrow \{A^y\}_M \prec write(T) \ll \{A^x\}_M$$

With this additional rule, scenario 1 becomes the same as scenarios 2 and 4 and is handled by the revoke record solution. Thus, under these two properties, the non-rollback property is guaranteed.

9.7 Redundant Synchrony in Ext3

We examine a performance problem with the ext3 file system where the transaction commit procedure artificially limits parallelism due to a redundant synchrony in its disk writes [82]. The *ordered mode* of ext3 guarantees that a newly created file will never point to stale data blocks after a crash. Ext3 ensures this guarantee by the following ordering in its commit procedure: when a transaction is committed, ext3 first writes to disk the data blocks allocated in that transaction, waits for those writes to complete, then writes the journal blocks to disk, waits for those to complete, and then writes the commit block, If I is an inode container, F is a file data block container, and T is the transaction commit container, the commit procedure of ext3 can be expressed by the following equation:

$$(\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I^x}\}_M) \prec write(T) \\ \Rightarrow (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I^x}\}_M) \\ \prec write(F) \ll write(\hat{I^x}) \ll write(T) \quad (9.33)$$

To examine if this is a necessary condition to ensure the no-stale-data guarantee, we first formally depict the guarantee that the ext3 ordered mode seeks to provide, in the following equation:

$$\{I^x \rightarrow F_k\}_M \ll \{I^x \rightarrow F\}_D \quad \Rightarrow \quad \{F_y\}_D \ll \{I^x \rightarrow F\}_D \wedge (g(F^y) = k) \tag{9.34}$$

The above equation states that if the disk acquires the belief that $\{I^x \rightarrow F\}$, then the contents of the data container F on disk should *already* pertain to the generation of F that I^x pointed to in memory. Note that because ext3 obeys reuse ordering, the ordered mode guarantee only needs to cater to the case of a *free* data block container being allocated to a new file.

We now prove equation 9.34, examining the conditions that need to hold for this equation to be true. We consider the LHS of the equation:

$$\{I^x \rightarrow F_k\}_M \ll \{I^x \rightarrow F\}_D$$

Applying equation 9.29 to the above, we get

$$\Rightarrow \quad (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I^x}\}_M) \prec write(T) \ll \{I^x \rightarrow F\}_D$$

Applying equation 9.33, we get

$$\Rightarrow \quad (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I^x}\}_M) \prec write(F) \ll write(\hat{I^x}) \ll write(T) \ll \{I^x \rightarrow F\}_D \tag{9.35}$$

By equation 9.8,

$$\Rightarrow \quad (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I^x}\}_M) \prec \{F^y\}_D \ll write(\hat{I^x}) \ll write(T) \ll \{I^x \rightarrow F\}_D \wedge (g(F^y) = k)$$
$$\Rightarrow \quad \{F^y\}_D \ll \{I^x \rightarrow F\}_D \wedge (g(F^y) = k)$$

Thus, the current ext3 commit procedure (equation 9.33) guarantees the no-stale-data property. However, to see if all the waits in the above procedure are required, let us reorder the two actions $write(F)$ and $write(\hat{I^x})$ in eq. 9.35:

$$\Rightarrow \quad (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I^x}\}_M) \prec write(\hat{I^x}) \ll write(F) \ll$$

$$write(T) \ll \{I^x \rightarrow F\}_D$$

Once again, applying equation 9.8,

$$\Rightarrow \quad \{F^y\}_D \ll \{I^x \rightarrow F\}_D \wedge (g(F^y) = k)$$

Thus, we can see that the ordering between the actions $write(F)$ and $write(\hat{I^x})$ is inconsequential to the guarantee that ext3 ordered mode attempts to provide. We can hence conclude that the wait that ext3 employs after the write to data blocks is redundant, and unnecessarily limits parallelism between data and journal writes. This can have especially severe performance implications in settings where the log is stored on a separate disk, because ext3 is unnecessarily limited to single disk bandwidth.

We believe that this specific example from ext3 points to a general problem with file system design. When developers do not have rigorous frameworks to reason about correctness, they tend to be *conservative*. Such conservatism often translates into unexploited opportunities for performance optimization. A systematic framework such as our logic enables more aggressive performance optimizations while not compromising correctness.

9.8 Support for Consistent Undelete

In this section, we demonstrate that our logic enables one to quickly formulate and prove properties about new file system features and mechanisms. We explore a functionality that is traditionally not considered a part of core file system design: the ability to *undelete* deleted files with certain consistency guarantees. The ability to recover deleted files is useful, as demonstrated by the large number of tools available for the purpose [83, 87]. Such tools try to rebuild deleted files by scavenging through on-disk metadata; this is possible to an extent because file systems do not normally zero out freed metadata containers (they are simply marked free). For example, in a UNIX file system, the block pointers in a deleted inode would indicate the blocks that used to belong to that deleted file.

However, none of the existing tools for undelete can guarantee *consistency* (*i.e.*, assert that the recovered contents are valid). While undelete is fundamentally only best-effort (files cannot be recovered if the blocks were subsequently reused in another file), the user needs to know how trustworthy the recovered contents are. We demonstrate using our logic that with existing file systems, such *consistent* undelete is impossible. We then provide a simple solution, and prove that the solution guarantees consistent undelete. Finally, we present an implementation of the solution

in ext3.

9.8.1 Undelete in existing systems

In order to model undelete, the logic needs to be capable of expressing pointers from containers holding a *dead* generation. We introduce the $\rightsquigarrow$ notation to indicate such a pointer, which we call a *dead pointer*. We also define a new operator $\tilde{\&}$ on a container that denotes the set of all dead *and* live entities pointing to the container. Let $undel(B)$ be an action to initiate undelete of container B.

The undelete process can be summarized by the following equation:

$$\begin{aligned}&undel(B) \wedge \{B^x \rightsquigarrow A\}_D \wedge \{\tilde{\&}A = \{B\}\}_D \\ &\Leftrightarrow \{B^x \rightsquigarrow A\}_D \prec \{B^y \rightarrow A\}_D \wedge (g(B^x) = g(B^y))\end{aligned} \tag{9.36}$$

In other words, if the dead (free) container B^x points to A on disk, and is the only container (alive or dead) pointing to A, the undelete makes the generation $g(B^x)$ live again, and makes it point to A.

The guarantee we want to hold for consistency is that if a dead pointer from B^x to A is brought alive, the on-disk contents of A at the time the pointer is brought alive, must be corresponding to the same generation that epoch B^x originally pointed to in memory (similar to the data consistency formulation in §9.6.1):

$$\begin{aligned}&\{B^x \rightarrow A_k\}_M \ll \{B^x \rightsquigarrow A\}_D \prec \{B^y \rightarrow A\}_D \\ &\wedge (g(B^x) = g(B^y)) \\ &\Rightarrow \{B^x \rightsquigarrow A\}_D \wedge \{A^z\}_D \wedge (g(A^z) = k)\end{aligned}$$

Note that the clause $g(B^x) = g(B^y)$ is required in the LHS to cover only the case where the *same* generation is brought to life, which would be true only for undelete.

To show that the above guarantee does not hold necessarily, we consider the negation of the RHS, *i.e.*, $\{A^z\}_D \wedge (g(A^z) \neq k)$, and show that this condition can co-exist with the conditions required for undelete as described in equation 9.36. In other words, we show that $undel(B) \wedge \{B^x \rightsquigarrow A\}_D \wedge \{\tilde{\&}A = \{B\}\}_D \wedge \{A^z\}_D \wedge (g(A^z) \neq k)$ can arise from valid file system execution.

We utilize the following implications for the proof:

$$\begin{aligned}\{B^x \rightsquigarrow A\}_D \quad &\Leftrightarrow \quad \{B^x \rightarrow A_k\}_M \prec \{\&A = \emptyset\}_M \prec write(B) \\ \{A^z\}_D \quad &\Rightarrow \quad \{c \rightarrow g(A^z)\}_M \prec write(A) \text{ (eq. 9.7)}\end{aligned}$$

Let us consider one possible interleaving of the above two event sequences:
$\{c \rightarrow g(A^z)\}_M \prec\ write(A) \ll\ \{B^x \rightarrow A_k\}_M \prec\ \{\&A = \emptyset\}_M \prec\ write(B)$

This is a valid file system sequence where a file represented by generation c points to $g(A^z)$, A^z is written to disk, then block A is freed from c thus killing the generation $g(A^z)$, and a new generation A_k of A is then allocated to the generation $g(B^x)$. Now, when $g(B^x)$ is deleted, and B is written to disk, the disk has both beliefs $\{B^x \rightsquigarrow A\}_D$ and $\{A^z\}_D$. Further, if the initial state of the disk was $\{\tilde{\&}A = \emptyset\}_D$, the above sequence would also simultaneously lead to the disk belief $\{\tilde{\&}A = \{B\}\}_D$. Thus we have shown that the conditions $\{B^x \rightsquigarrow A\}_D \wedge\ \{\tilde{\&}A = \{B\}\}_D \wedge\ \{A^z\}_D \wedge$ $(k \neq g(A^z))$ can hold simultaneously. An undelete of B at this point would lead to violation of the consistency guarantee, because it would associate a stale generation of A with the undeleted file $g(B^x)$. It can be shown that neither reuse ordering nor pointer ordering would guarantee consistency in this case.

9.8.2 Undelete with generation pointers

We now propose the notion of *generation pointers* and show that with such pointers, consistent undelete is guaranteed. So far, we have assumed that *pointers* on disk point to *containers* (as discussed in Section 9.4). If instead, each pointer pointed to a specific *generation*, it leads to a different set of file system properties. To implement such "generation pointers", each on-disk container contains a generation number that gets incremented every time the container is reused. In addition, every on-disk pointer will embed this generation number in addition to the container name. With generation pointers, the on-disk contents of a container will implicitly indicate its generation. Thus, $\{B_k\}_D$ is a valid belief; it means that the disk knows the contents of B belong to generation k.

Under generation pointers, the criterion for doing undelete (eq. 9.36) becomes:

$$\begin{aligned} &undel(B) \wedge\ \{B^x \rightsquigarrow A_k\}_D \wedge\ \{A_k\}_D \\ &\Leftrightarrow \{B^x \rightsquigarrow\ A_k\}_D \prec\ \{B^y \rightarrow A_k\}_D \end{aligned} \tag{9.37}$$

Let us introduce an additional constraint $\{A^z\}_D \wedge\ (k \neq g(A^z))$ into the left hand side of the above equation (as we did in the previous subsection):

$$\{B^x \rightsquigarrow A_k\}_D \wedge\ \{A_k\}_D \wedge\ \{A^z\}_D \wedge\ (k \neq g(A^z)) \tag{9.38}$$

Since $k \neq g(A^z)$, let us denote $g(A^z)$ as h. Since every on-disk container holds the generation number too, we have $\{A_h\}_D$. Thus, the above equation becomes:

$$\{B^x \rightsquigarrow A_k\}_D \wedge\ \{A_k\}_D \wedge\ \{A_h\}_D \wedge\ (k \neq h)$$

This is clearly a contradiction, since it means the on-disk container A has the two different generations k and h simultaneously. Thus, it follows that an undelete would not occur in this scenario (or alternatively, this would be flagged as inconsistent). Thus, all undeletes occurring under generation pointers are consistent.

9.8.3 Implementation of undelete in ext3

Following on the proof for consistent undelete, we implemented the generation pointer mechanism in Linux ext3. Each block has a generation number that gets incremented every time the block is reused. Although the generation numbers are maintained in a separate set of blocks, ensuring atomic commit of the generation number and the block data is straightforward in the data journaling mode of ext3, where we simply add the generation update to the create transaction. The block pointers in the inode are also extended with the generation number of the block. We implemented a tool for undelete that scans over the on-disk structures, restoring all files that can be undeleted *consistently*. Specifically, a file is restored if the generation information in all its metadata block pointers match with the corresponding block generation of the data blocks.

We ran a simple microbenchmark creating and deleting various directories from the linux kernel source tree, and observed that out of roughly 12,200 deleted files, 2970 files (roughly 25%) were detected to be inconsistent and not undeletable, while the remaining files were successfully undeleted. This illustrates that the scenario proved in Section 9.8.1 actually occurs in practice; an undelete tool without generation information would wrongly restore these files with corrupt or misleading data.

9.9 Application to Semantic Disks

We now move to the application of our logic framework for reasoning about semantically-smart disk systems. As seen from our case studies, this reasoning process is quite hard for complex functionality. Our formalism of memory and disk beliefs fits the SDS model, since the extra file system state tracked by an SDS is essentially a disk belief. In this section, we first use our logic to explore the feasibility of tracking block type within a semantic disk. We then show that the usage of generation pointers by the file system simplifies information tracking within an SDS.

9.9.1 Block typing

An important piece of information required within a semantic disk is the *type* of a disk container [102]. While identifying the type of statically-typed containers is straightforward, dynamically typed containers are hard to deal with. The type of a dynamically typed container is determined by the contents of a *parent* container; for example, an indirect pointer block can be identified only by observing a parent inode that has this block in its indirect pointer field. Thus, tracking dynamically typed containers requires correlating type information from a type-determining parent, and then using the information to interpret the contents of the dynamic container.

For accurate type detection in an SDS, we want the following guarantee to hold:

$$\{t(A^x) = k\}_D \quad \Rightarrow \quad \{t(A^x) = k\}_M \tag{9.39}$$

In other words, if the disk interprets the contents of an epoch A^x to be belonging to type k, those contents should have belonged to type k in memory as well. This guarantees, for example, that the disk would not wrongly interpret the contents of a normal data block container as an indirect block container. Note however that the equation does not impose any guarantee on *when* the disk identifies the type of a container; it only states that whenever it does, the association of type with the contents is correct.

To prove this, we first state an algorithm of how the disk arrives at a belief about a certain type [102]. An SDS snoops on metadata traffic, looking for type-determining containers such as inodes. When such a container is written, it observes the pointers within the container and concludes on the type of each of the pointers. Let us assume that one such pointer of type k points to container A. The disk then examines if container A was written since the last time it was freed. If yes, it interprets the current contents of A as belonging to type k. If not, when A is written at a later time, the contents are associated with type k. We have the following equation:

$$\begin{aligned} \{t(A^x) = k\}_D \quad \Rightarrow \quad & \{B^y \rightarrow A\}_D \wedge (f(B^y, A) = k) \\ & \wedge \{A^x\}_D \end{aligned} \tag{9.40}$$

In other words, to interpret A^x as belonging to type k, the disk must believe that some container B points to A, and the current on-disk epoch of B (*i.e.*, B^y) must indicate that A is of type k; the function $f(B^y, A)$ abstracts this indication. Further, the disk must contain the contents of epoch A^x in order to associate the contents with type k.

Let us explore the logical events that should have led to each of the components

on the right side of equation 9.40. Applying eq. 9.13,

$$\begin{aligned} &\{B^y \rightarrow A\}_D \wedge\ (f(B^y, A) = k) \\ &\Rightarrow (\{B^y \rightarrow A\}_M \wedge\ (f(B^y, A) = k)) \prec\ \{B^y \rightarrow A\}_D \\ &\Rightarrow (\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D \end{aligned} \tag{9.41}$$

Similarly for the other component $\{A^x\}_D$,

$$\{A^x\}_D \quad \Rightarrow \quad write(A^x) \ll \{A^x\}_D \tag{9.42}$$

To verify the guarantee in equation 9.39, we assume that it does not hold, and then observe if it leads to a valid scenario. Thus, we can add the clause $\{t(A^x) = j\}_M \wedge (j \neq k)$ to equation 9.40, and our equation to prove is:

$$\{B^y \rightarrow A\}_D \wedge\ (f(B^y, A) = k) \wedge\ \{A^x\}_D \wedge\ \{t(A^x) = j\}_M$$

We thus have two event sequences (from eq. 9.41 and 9.42):

1. $(\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D$
2. $\{t(A^x) = j\}_M \wedge\ write(A^x)$

Since the type of an epoch is unique, and a $write$ of a container implies that it already has a type,
$\{t(A^x) = j\}_M \wedge\ write(A^x) \Rightarrow\ \{t(A^x) = j\}_M \prec\ write(A^x)$.

These sequences can only be interleaved in two ways. The epoch A^x occurs either before or after the epoch in which $\{t(A) = k\}_M$.

Interleaving 1:

$$\begin{aligned} &(\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D \\ &\ll\ \{t(A^x) = j\}_M \prec\ write(A^x) \end{aligned}$$

By eq. 9.12,

$$\begin{aligned} \Rightarrow \quad &(\{B^y \rightarrow A\}_M \wedge\ \{t(A) = k\}_M) \prec\ \{B^y \rightarrow A\}_D \\ &\ll\ \{\&A = \emptyset\}_M \prec\ \{t(A^x) = j\}_M \prec\ write(A^x) \end{aligned}$$

This is a valid sequence where the container A is freed after the disk acquired the belief $\{B \rightarrow A\}$ and a later version of A is then written when its actual type has changed to j in memory, thus leading to incorrect interpretation of A^x as belonging to type k.

However, in order to prevent this scenario, we simply need the reuse ordering rule (eq. 9.15). With that rule, the above sequence would imply the following:

$$\Rightarrow \quad (\{B^y \rightarrow A\}_M \wedge \{t(A)=k\}_M) \prec \{B^y \rightarrow A\}_D$$
$$\ll \{\&A=\emptyset\}_M \prec write(B) \ll \{t(A^x)=j\}_M \prec write(A^x)$$
$$\Rightarrow \quad (\{B^y \rightarrow A\}_M \wedge \{t(A)=k\}_M) \prec \{B^y \rightarrow A\}_D$$
$$\ll \{\&A=\emptyset\}_D \prec \{t(A^x)=j\}_M \prec write(A^x)$$

Thus, when A^x is written, the disk will be treating A as free, and hence will not wrongly associate A with type k.

Interleaving 2:

Proceeding similarly with the second interleaving where epoch A^x occurs before A is assigned type k, we arrive at the following sequence:

$$\Rightarrow \quad \{t(A^x)=j\}_M \prec write(A^x) \ll \{\&A=\emptyset\}_M$$
$$\prec (\{B^y \rightarrow A\}_M \wedge \{t(A)=k\}_M) \prec \{B^y \rightarrow A\}_D$$

We can see that simply applying the reuse ordering rule does not prevent this sequence. We need a stronger form of reuse ordering where the “freed state” of A includes not only the containers that pointed to A, but also the allocation structure $|A|$ tracking liveness of A. With this rule, the above sequence becomes:

$$\Rightarrow \quad \{t(A^x)=j\}_M \prec write(A^x) \ll \{\&A=\emptyset\}_M$$
$$\prec write(|A|) \ll (\{B^y \rightarrow A\}_M \wedge \{t(A)=k\}_M)$$
$$\prec \{B^y \rightarrow A\}_D \tag{9.43}$$

We also need to add a new behavior to the SDS which states that when the SDS observes an allocation structure indicating that A is free, it inherits the belief that A is free.

$$\{\&A=\emptyset\}_M \prec write(|A|) \quad \Rightarrow \quad \{\&A=\emptyset\}_D$$

Applying the above SDS operation to eqn 9.43, we get

$$\Rightarrow \quad \{t(A^x)=j\}_M \prec write(A^x) \ll \{\&A=\emptyset\}_D$$
$$\ll (\{B^y \rightarrow A\}_M \wedge \{t(A)=k\}_M) \prec \{B^y \rightarrow A\}_D$$

In this sequence, because the SDS does not observe a write of A since it was treated as “free”, it will not associate type k to A until A is subsequently written.

Thus, we have shown that an SDS cannot accurately track dynamic type underneath a file system without any ordering guarantees. We have also shown that

if the file system exhibits a strong form of reuse ordering, dynamic type detection can indeed be made reliable within an SDS.

9.9.2 Utility of generation pointers

In this subsection, we explore the utility of file system-level "generation pointers" (§ 9.8.2) in the context of an SDS. To illustrate their utility, we show that tracking dynamic type in an SDS is straightforward if the file system tracks generation pointers.

With generation pointers, equation 9.40 becomes:

$$\{t(A_g) = k\}_D \quad \Rightarrow \quad \{B^y \rightarrow A_g\}_D \wedge \ (f(B^y, A_g) = k) \\ \wedge \{A_g\}_D$$

The two causal event sequences (as explored in the previous subsection) become:

$$(\{B^y \rightarrow A_g\}_M \wedge \ \{t(A_g) = k\}_M) \prec \ \{B^y \rightarrow A_g\}_D$$

$$\{t(A_g) = j\}_M \wedge \ write(A_g)$$

Since the above sequences imply that the same generation g had two different types, it violates rule 9.12. Thus, we straightaway arrive at a contradiction that proves that violation of rule 9.39 can never occur.

9.10 Summary

As the need for dependability of computer systems becomes more important than ever, it is essential to have systematic formal frameworks to verify and reason about their correctness. Despite file systems being a critical component of system dependability, formal verification of their correctness has been largely ignored. Besides making file systems vulnerable to hidden errors, the absence of a formal framework also stifles innovation, because of the skepticism towards the correctness of new proposals, and the proclivity to stick to "time-tested" alternatives. In this chapter, we have taken a step towards bridging this gap in file system design by showing that a logical framework can substantially simplify and systematize the process of verifying file system correctness. We have also shown how such a logic systematizes the process of reasoning about semantic disks.

Chapter 10

Related Work

The related work on semantically-smart disk systems can be grouped into two classes: work on smarter storage in general, and work on implicit systems. We discuss each in turn. Finally, we also discuss related work pertaining to some of our case studies and the logic framework.

10.1 Smarter Storage

The idea of embedding intelligence into storage systems is a well-explored one, dating back to the "logic per track" devices [104] suggested by Slotnick in 1970, and database machines proposed in the early 1980s [14]. The range of previous work on smart disks fall into four categories. The first category assumes that the interface between file and storage systems is fixed and cannot be changed, the category under which an SDS belongs. Research in the second group proposes changes to the storage interface, requiring that file systems be modified to leverage this new interface. The third group proposes changes not only to the interface, but also to the programming model for applications. Finally, the fourth group advocates placing all storage-like smartness within file systems, reverting back to the old model of "dumb" storage.

10.1.1 Fixed interfaces

The focus of this thesis is on the integration of smart disks into a traditional file system environment. In this environment, the file system has a narrow, SCSI-like interface to storage, and uses the disk as a persistent store for its data structures. An early example of a smart disk controller is Loge [30], which harnessed its process-

ing capabilities to improve performance by writing blocks near the current disk-head position. Wang *et al.*'s log-based programmable disk [115] extended this approach in a number of ways, namely quick crash-recovery and free-space compaction. Neither of these systems assume or require any knowledge of file system structures, and thus are limited in the range of optimizations they can provide.

When storage system interfaces are more developed than that provided in the local setting, there are more opportunities for new functionality. The use of a network packet filter within the Slice virtual file service [5] allows Slice to interpose on NFS traffic in clients, and thus implement a range of optimizations (*e.g.*, preferential treatment of small files). Interposing on an NFS traffic stream is simpler than doing so on a SCSI-disk block stream because the contents of NFS packets are well-defined.

High-end RAID products are the perfect place for semantic smartness, because a typical enterprise storage system has substantial processing capabilities and memory capacity. For example, an EMC Symmetrix server contains about 100 processors and can be configured with up to 256 GB of memory [29]. Some high-end RAID systems currently leverage their resources to perform a bare minimum of semantically-smart behavior; for example, storage systems from EMC can recognize an Oracle data block and provide an extra checksum to assure that a block write (comprised of multiple sector writes) reaches disk atomically [15]. This thesis explores the acquisition and exploitation of more detailed knowledge of file system behavior.

10.1.2 More expressive interfaces

Given that one of the primary factors that limits the addition of new functionality in a smart disk is the narrow interface between file systems and storage, it is not surprising that there has been research that investigates changing this interface. Mime investigates an enhanced interface in the context of an intelligent RAID controller [19]; specifically, Mime adds primitives to allow clients to control both when updates to storage become visible to other traffic streams and the commit order of operations. Logical disks expand the interface by allowing the file system to express grouping preferences with lists [23]; thus, file systems are simplified since they do not need to maintain this information. E×RAID exposes per-disk information to an informed file system (namely, I·LFS), providing performance optimizations, more control over redundancy, and improved manageability of storage [24]. Finally, Ganger suggests that a reevaluation of this interface is needed [31], and outlines two relevant case studies: track-aligned extents [97] and freeblock scheduling [62].

Distributed storage has been another domain where researchers have considered changing the interface to storage, in order to provide better functionality. For example, the Petal storage system [60] proposes a sparse virtual disk abstraction to its file system counterpart, Frangipani [110]. More recently, the Boxwood storage system provides higher level abstractions such as B-Trees to software layers above [64]; such a high level interface to storage naturally conveys rich semantic information to the storage system.

Also, recent work in the storage community suggests that the next evolution in storage will place disks on a more general-purpose network and not a standard SCSI bus [36]. Some have suggested that these network disks export a higher-level, object-like interface [37], thus moving the responsibilities of low-level storage management from the file system into the drives themselves. Although the specific challenges would likely be different in this context, the fixed object-based interface between file systems and storage will likely provide an interesting avenue for further research into the utility of semantic awareness.

10.1.3 New programming environments

In contrast to integration underneath a traditional file system, other work has focused on incorporating active storage into entirely new parallel programming environments. Recent work on "active disks" includes that by Acharya *et al.* [1], Riedel *et al.* [88], Amiri *et al.* [4], and Sivathanu *et al.* [100]. Much of this research involves shipping computation to the storage system, and focuses on how to partition applications across host and disk CPUs to minimize data transferred across system busses or to reduce latency.

10.1.4 Smarter file systems

A different approach that is considered by certain custom file systems to circumvent the narrow interface limitation is to revert back to the old model of storage, *i.e.*, treat storage as a collection of dumb disks. The file system then manages storage-like functionality such as RAID layout, placement and migration of data, etc. The Google file system is one example in this category [35]. GFS implements all the smarts within the file system and is simply built on top of a collection of IDE disks. Although an interesting alternative, this approach is fraught with limitations. Storage-level placement and optimizations often depend very much on the low-level details of the storage system such as power supply and actual busses connecting the disks, etc. GFS was successful because it was so custom-made that it had specific information and control over these various low-level details. In more

general purpose environments, customizing the file system to such low-level details is quite infeasible, so it is no surprise that large enterprise storage systems today constitute a multi-billion dollar industry; most general-purpose customers such as banks and e-commerce site tend to deploy sophisticated enterprise storage systems that take care of the low-level complexity.

10.2 Implicit Systems

Semantically-smart disk systems are based on the general philosophy of implicit inference of information around existing interfaces. This general approach has been formalized under the term "Gray-box systems" by Arpaci-Dusseau *et al.* [6], where the authors propose techniques to infer the state of the operating system based on high-level "gray-box" assumptions about the expected behavior of the operating system. This approach has subsequently been applied to infer and control various aspects of the operating system, such as cache replacement [17], file placement [71], and CPU scheduling [85].

Implicit inference of information has been explored also in other boundaries besides the application to operating system boundary. For example, in programming language research, Hsieh *et al.* investigate the automatic extraction of bit-level instruction encodings by feeding permutations of instructions into an assembler and analyzing the resulting machine code [50]. Schindler *et al.*, have proposed implicit techniques for tracking various internal parameters of disk drives by utilizing timing information for various carefully chosen microbenchmarks [96, 120]. More recent work has looked at deconstructing internal information about RAID systems with a similar approach of careful microbenchmarking [25]. As described in Chapter 3, although such microbenchmarking techniques are useful in tracking information about the storage system, the implicit channel they rely on is a fragile timing channel, and thus such techniques cannot track the wealth of dynamic information existent in modern storage systems.

Other examples of systems where usage of implicit information has been explored are in inferring TCP behavior [75], inferring the policies used in a commodity storage cluster [43], and co-ordinated process scheduling in distributed systems [7].

10.3 Partial Availability

Our case study D-GRAID for partial availability draws on related work from a number of different areas, including distributed file systems and traditional RAID

systems.

10.3.1 Distributed file systems

Designers of distributed file systems have long ago realized the problems that arise when spreading a directory tree across different machines in a system. For example, Walker *et al.* discuss the importance of directory namespace replication within the Locus distributed system [80]. The Coda mobile file system also takes explicit care with regard to the directory tree [57]. Specifically, if a file is cached, Coda makes sure to cache every directory up to the root of the directory tree. By doing so, Coda can guarantee that a file remains accessible should a disconnection occur. Perhaps an interesting extension to our work would be to reconsider host-based in-memory caching with availability in mind. Also, Slice [5] tries to route namespace operations for all files in a directory to the same server.

More recently, work in wide-area file systems has also re-emphasized the importance of the directory tree. For example, the Pangaea file system aggressively replicates the entire tree up to the root on a node when a file is accessed [94]. The Island-based file system also points out the need for "fault isolation" but in the context of wide-area storage systems; their "one island principle" is quite similar to fault-isolated placement in D-GRAID [54].

Finally, P2P systems such as PAST that place an entire file on a single machine have similar load balancing issues [92]. However, the problem is more difficult in the p2p space due to the constraints of file placement; block migration is much simpler in a centralized storage array.

10.3.2 Traditional RAID systems

We also draw on the long history of research in classic RAID systems. From AutoRAID [117] we learned both that complex functionality could be embedded within a modern storage array, and that background activity could be utilized successfully in such an environment. From AFRAID [95], we learned that there could be a flexible trade-off between performance and reliability, and the value of delaying updates.

Much of RAID research has focused on different redundancy schemes. While early work stressed the ability to tolerate single-disk failures [12, 76, 77], later research introduced the notion of tolerating multiple-disk failures within an array [3, 16]. We stress that our work is complementary to this line of research; traditional techniques can be used to ensure full file system availability up to a certain number of failures, and D-GRAID techniques ensure graceful degradation

under additional failures. A related approach is parity striping [40] which stripes only the parity and not data; while parity striping would achieve a primitive form of fault isolation, the layout is still oblivious of the semantics of the data; blocks will have the same level of redundancy irrespective of their importance (*i.e.*, meta-data vs data), so multiple failures could still make the entire file system inaccessible. Also, file systems typically spread out large files across the logical address space, hence parity striping cannot ensure collocation of the blocks of a file. A number of earlier works also emphasize the importance of hot sparing to speed recovery time in RAID arrays [48, 67, 76]. Our work on semantic recovery is also complementary to those approaches.

Finally, note that term "graceful degradation" is sometimes used to refer to the performance characteristics of redundant disk systems under failure [49, 84]. This type of graceful degradation is different from what we discuss in this thesis; indeed, none of those systems continues operation when an unexpected number of failures occurs.

10.4 Logical Modeling of Systems

In this section, we examine prior work related to our logic framework for modeling file systems and semantic disks.

Previous work has recognized the need for modeling complex systems with formal frameworks, in order to facilitate proving correctness properties about them. The logical framework for reasoning about authentication protocols, proposed by Burrows *et al.* [18], is the most related to our work in spirit; in that paper, the authors formulate a domain-specific logic and proof system for authentication, showing that protocols can be verified through simple logical derivations. Other domain-specific formal models exist in the areas of database recovery [58] and database reliability [45].

A different body of related work involves generic frameworks for modeling computer systems. The well-known TLA framework [59] is an example. The I/O automaton [8] is another such framework. While these frameworks are general enough to model most complex systems, their generality is also a curse; modeling various aspects of a file system to the extent we have in this paper, is quite tedious with a generic framework. Tailoring the framework by using domain-specific knowledge makes it simpler to reason about properties using the framework, thus significantly lowering the barrier to entry in terms of adopting the framework [18]. Specifications and proofs in our logic take 10 to 20 lines in contrast to the thousands of lines that TLA specifications take [123]. However, automated theorem-proving

through model checkers is one of the benefits of using a generic framework such as TLA.

Previous work has also explored verification of the correctness of system *implementations*. The recent body of work on using model checking to verify implementations is one example [70, 121]. We believe that this body of work is complementary to our logic framework; our logic framework can be used to build the model and the invariants that should hold in the model, which the implementation can be verified against.

Chapter 11

Conclusions and Future Work

"*It's not even the beginning of the end, but it's, perhaps, the end of the beginning.*" Winston Churchill, 1942

The philosophy of building systems as a hierarchy of layers is one of the oldest and most influential in system design [26]. Each layer in such a system communicates with the others through well-defined interfaces. While layering has clear advantages such as enabling independent innovation and evolution of system components, it also has a cost: interfaces between layers are formulated based on implicit assumptions about the layers, but as layers evolve over time, they sometimes invalidate those assumptions, thus making the interface obsolete, sub-optimal, or overly limiting. To keep up with evolving system layers, interfaces ideally need to evolve as well, but practical concerns often preclude or retard interface evolution. The problem of narrow interface to storage that we have addressed in this thesis is one instance of this general problem with system evolution.

In this thesis, we have presented a new approach to interface evolution by implicit inference of information around fixed existing interfaces. The implicit interface evolution approach addresses the fundamental bootstrapping problem with explicit interface change; by facilitating demonstration of benefits due to a potential new interface without actual interface change, implicit inference techniques can move industry more rapidly towards the new interface, thus catalyzing an explicit interface change. Also, given the long time-scales at which explicit interface changes occur, implicit techniques provide a way to innovate in the short term; for example, with the SDS approach, a storage vendor can provide smart functionality today and ship the systems without waiting for industry to embrace a new storage interface. Of course, implicit interface evolution comes with costs in terms of added system complexity and a small amount of performance overhead; whether

these costs justify the benefits of a potential new functionality is a decision that can now be taken on a case-by-case basis.

In the remainder of this chapter, we first outline some key lessons we learned through the work presented in this thesis, and then explore various avenues for future research based on various aspects of this thesis.

11.1 Lessons Learned

In this section, we reflect on some of the key lessons we learned during our experience designing and implementing semantically-smart disk systems. We believe that these lessons have a broader relevance beyond just our specific techniques and case studies.

Limited knowledge within the disk does not imply limited functionality.

One of the main contributions of this thesis is a demonstration of both the limits of semantic knowledge that can be tracked underneath modern file systems, as well as the "proof" via implementation that despite such limitations, interesting functionality can be built inside of a semantically-smart disk system. We believe that any semantic disk system must be careful in its assumptions about file system behavior, and hope that our work can guide others who pursue a similar course.

Semantically-smart disks would be easier to build with some help from above.

Because of the way file systems reorder, delay, and hide operations from disks, reverse engineering exactly what they are doing at the SCSI level is difficult. We have found that small modifications to file systems could substantially lessen this difficulty. For example, if the file system could inform the disk whenever it believes the file system structures are in a consistent on-disk state, many of the challenges in the disk would be lessened. Similarly, if the file system is careful about when it reuses blocks, it would avoid a significant source of uncertainty within the semantic disk. In Chapter 7, we summarize various such useful dynamic properties that hold in certain file systems. In file systems that do not conform to these properties, such small alterations could ease the burden of semantic disk development.

Dependence on dynamic properties of the file system is a double-edged blade.

While semantic disks can be significantly simplified if they make assumptions on dynamic ordering guarantees provided by the file system, such a dependence on dynamic properties might be more of a concern than just dependence on the static on-disk layout of the file system. D-GRAID was very general in its assumptions (arbitrary reordering and delay of writes) and thus will be robust to changes in dynamic properties of the file system. However, an aggressive functionality such as secure deletion need to make more assumptions about dynamic properties in order to guarantee correctness. Given the added level of dependence, one needs to be careful in deciding if the extra functionality it enables is worth the cost.

Semantically-smart disks stress file systems in unexpected ways.

File systems were not built to operate on top of semantic disks that behave as D-GRAID does, for example; specifically, they may not behave particularly well when part of a volume address space becomes unavailable. Perhaps because of its heritage as an OS for inexpensive hardware, Linux file systems handle unexpected conditions fairly well. However, the exact model for dealing with failure is inconsistent: data blocks could be missing and then reappear, but the same is not true for inodes. As semantically-smart disks push new functionality into storage, file systems may potentially need to evolve to accommodate them.

Conservative techniques are crucial to working around fundamental uncertainty.

At various points in our work, we stumbled upon what appeared to be a fundamental limitation in terms of semantic inference, but soon resolved it based on some conservative technique or abstraction. The most pronounced example of this was in secure delete which had strong requirements on correctness, but the semantic information it had to be based on was fundamentally imprecise. With the mechanism of conservative overwrites, we were able to solve this apparent limitation. One of the crucial aspects to developing complex, especially correctness-sensitive functionality within semantic disks is tolerance to uncertainty through such conservative mechanisms and abstractions.

Detailed traces of workload behavior are invaluable.

Because of the excellent level of detail available in the HP traces [89], we were able to simulate and analyze the potential of D-GRAID under realistic settings. Many other traces do not contain per-process information, or anonymize file references to the extent that pathnames are not included in the trace, and thus we could not utilize them in our study. One remaining challenge for tracing is to include user data blocks, as semantically-smart disks may be sensitive to the contents. However, the privacy concerns that such a campaign would encounter may be too difficult to overcome.

Domain specific formal models at the right granularity greatly simplify reasoning.

Our logic framework for modeling file systems and semantic disks pointed to us the utility of formalizing complex systems. By making the framework domain specific and at a coarse granularity, equations and proofs in the logic are extremely intuitive to understand and conceivably accessible to real file system designers, in comparison to generic, low-level frameworks such as TLA. We found that the framework significantly helped in uncovering implicit assumptions that we sometimes made, and helped systematize the process of showing that a given set of mechanisms precisely provides the guarantees it claims to. We believe that similar frameworks that use domain knowledge can significantly aid construction of other complex systems.

11.2 Future Work

In this section, we describe various extensions to the work described in this thesis that would be interesting to consider. These relate to general applications of the overall implicit inference approach beyond semantic disks, new directions with our logic framework, and direct extensions to semantic disk technology.

11.2.1 Implicit inference in other domains

While this thesis has explored implicit inference of semantic inference underneath modern file systems, the general philosophy and techniques apply to other layer boundaries in the system stack that are subject to the problem of being stuck with obsolete or limiting interfaces. For instance, a similar problem occurs in the area of virtual machine monitors [28, 114], where the VMM has a very narrow interface to the guest operating systems. From the I/O perspective, the VMM exports

a virtual disk to each guest operating system, and then observes block-level reads and writes into this virtual disk, very much like a storage system underneath SCSI. Because of this narrow interface, the VMM has no information to prioritize I/O requests from multiple guest operating systems; if the VMM can distinguish between foreground writes (on which some application within the guest OS is waiting on) and background delayed writes, scheduling can be much more effective. Similarly, semantic knowledge can help in better disk space allocation within the VMM by compacting free space due to deleted files within guest operating systems. One interesting aspect of semantic inference within a VMM is that unlike a storage system underneath SCSI which could only passively observe requests, the VMM is a more active entity; for example, the VMM can change memory contents that the guest operating system observes, and can thus actively *control* the operating system.

Another application of semantic information within a VMM is in robust intrusion detection and tracing. Since virtual machines are an additional protection boundary isolated from the operating system, they provide an attractive vantage point to perform tracing and analysis to detect intrusions even in cases where the operating system is compromised. Today, such intrusion analysis within the VMM is fundamentally limited due to lack of semantic knowledge. With semantic information, VMM-level tools could monitor changes to key files within the file system or changes to registry contents, without being circumvented by the intruder.

11.2.2 Integrating logic into implementation checkers

Our logic framework for modeling file systems presents another interesting avenue for future research. It would be interesting to explore how such a framework can augment existing techniques for verifying file system implementations. For example, the logic could be used to construct a set of precise constraints that the update traffic from a file system should conform to, and then a layer of software underneath the file system can verify online if those constraints are violated.

A more ambitious goal would be to explore if one can start with a logic specification of a file system's interaction with the disk, and automatically arrive at an implementation that preserves the ordering guarantees mentioned in the specification. Although the general problem of going from a logic to an implementation is quite hard, it could be tractable if the logic is sufficiently domain specific. Recent research has demonstrated serious correctness bugs in widely used file systems; automating the implementation of file system consistency management can help alleviate this problem.

11.2.3 More semantic disk functionality

While we have investigated a wide range of case studies in this thesis for demonstrating the utility of semantic knowledge within a storage system, there are plenty of opportunities that we have left unexplored. For example, in D-GRAID, we built most of the mechanisms required, but the policy space remains largely uninvestigated. There are various design points within D-GRAID where interesting policy decisions arise. For example, deciding when to perform access-driven diffusion, and exactly which blocks to diffuse is a difficult problem, depending on various aspects such as update frequency, popularity of blocks, and so on. Deciding on the degree of metadata replication is another example. The problem of deciding the degree of replication can perhaps be viewed as an optimization problem where the utility of replicating a block depends on its importance (for example, how high in the directory tree the block is); one could then arrive at the optimal level of replication for each type of block given a fixed space budget or a performance tolerance.

Semantic knowledge could also be applied for other kinds of functionality enhancements within storage systems. For example, power consumption is a critical issue in many enterprise systems [21]. If the storage system has semantic knowledge, it could collocate active data in a smaller set of disks (similar to D-GRAID), and then spin down the remaining disks, thus conserving power. Semantic information can also help the storage system to predict which blocks are likely to be active in the near future, and pro-actively power up those disks in anticipation. For example, if a directory block is read, the disk system can anticipate that blocks belonging to inodes pointed to by the directory, or even those files in entirety could be accessed, and thus perform the needful. Another functionality that is possible in semantic disks is consistent block-level snapshotting. Incremental snapshotting is a popular feature in file server appliances today [116], but block-level storage systems cannot provide this functionality because they have no information on when the on-disk contents represents a consistent file system state; with semantic inference, storage systems can acquire this information. Our journaling case study in Chapter 5 detects consistent state underneath synchronous file systems; extending this underneath general purpose asynchronous file systems will be interesting.

11.2.4 Making semantic disks more semantic

Much of the work in this thesis pertained to inferring higher level knowledge about the file system. A natural extension to this would be to use the same approach to infer even higher-level information. For example, in the case of D-GRAID, the collocation policy explored was quite straight-forward in that it considered whole

files and directories. If the storage system was more aggressive in its semantic inference, it could have looked at file names and file content to perform more interesting collocation. For example, if it identifies a file with a `html` extension, it can look at the contents and place all embedded links within the same fault boundary. Similarly, if it looked at a file called Makefile, it could perhaps infer that the entire source code base is related and collocate those files. The database-specific storage work described in Chapter 8 is one instance of this general approach. However, this direction of work has to be considered judiciously; extending too far up into the stack could result in dependencies on fragile assumptions. Similar to our work which drew a line on static format information that is unlikely to change, the challenge will be on identifying stable dependencies that are worth creating.

11.3 Summary

Semantically-smart disk systems enable new classes of functionality and improvements in storage by exploiting information about higher layers of the system. Pragmatism has been one of the driving goals behind the SDS approach; deployment of innovative functionality is more likely to be successful if it is as less intrusive on existing infrastructure as possible. By requiring no changes to the storage interface and in most cases, to the layers above, semantic disks achieve just that. We believe that our general approach of implicit evolution of ossified interfaces has value beyond the realm of file systems and storage, and can lead to a new way of thinking about system evolution.

Bibliography

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, California, October 1998.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB 28)*, Hong Kong, China, August 2002.

[3] G. A. Alvarez, W. A. Burkhard, and F. Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, Denver, Colorado, May 1997.

[4] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, San Diego, California, June 2000.

[5] D. Anderson, J. Chase, and A. Vahdat. Interposed Request Routing for Scalable Network Storage. *ACM Transactions on Computer Systems*, 20(1), 2002.

[6] A. Arpaci-Dusseau and R. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[7] A. C. Arpaci-Dusseau, D. E. Culler, and A. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the 1998 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '98)*, Madison, Wisconsin, June 1998.

[8] P. C. Attie and N. A. Lynch. Dynamic Input/Output Automata, a Formal Model for Dynamic Systems. In *ACM Symposium on Principles of Distributed Computing*, pages 314–316, 2001.

[9] L. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, Munich, Germany, June 2004.

[10] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

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

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

[13] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[14] H. Boral and D. J. DeWitt. Database Machines: An Idea Whose Time has Passed? In *3rd International Workshop on Database Machines*, 1983.

[15] J. Brown and S. Yamaguchi. Oracle's Hardware Assisted Resilient Data (H.A.R.D.). *Oracle Technical Bulletin (Note 158367.1)*, 2002.

[16] W. Burkhard and J. Menon. Disk Array Storage System Reliability. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 432–441, Toulouse, France, June 1993.

[17] N. C. Burnett, J. Bent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[18] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, Litchfield Park, Arizona, December 1989.

[19] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: A high performance parallel storage device. Technical Report HPL-CSP-92-9, HP Labs, 1992.

[20] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

[21] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[22] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[23] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville, North Carolina, December 1993.

[24] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[25] T. E. Denehy, J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deconstructing Storage Arrays. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, pages 59–71, Boston, Massachusetts, October 2004.

[26] E. W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[27] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[28] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[29] EMC Corporation. Symmetrix Enterprise Information Storage Systems. http://www.emc.com, 2002.

[30] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, San Francisco, California, January 1992.

[31] G. R. Ganger. Blurring the Line Between Oses and Storage Devices. TR SCS CMU-CS-01-166, Dec. 2001.

[32] G. R. Ganger, M. K. McKusick, C. A. Soules, and Y. N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems*, 18(2), May 2000.

[33] G. R. Ganger and Y. N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, California, November 1994.

[34] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt. Disk Subsystem Load Balancing: Disk Striping vs. Conventional Data Placement. In *HICSS '93*, 1993.

[35] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing (Lake George), New York, October 2003.

[36] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, California, October 1998.

[37] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, Carnegie Mellon University, 1997.

[38] R. A. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, pages 201–212, 1995.

[39] J. Gray. Why Do Computers Stop and What Can We Do About It? In *6th International Conference on Reliability and Distributed Databases*, June 1987.

[40] J. Gray, B. Horst, and M. Walker. Parity Striping of Disc Arrays: Low-cost Reliable Storage with Acceptable Throughput. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB 16)*, pages 148–159, Brisbane, Australia, August 1990.

[41] S. D. Gribble. Robustness in Complex Systems. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloss Elmau, Germany, May 2001.

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

[43] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Schindler. Deconstructing Commodity Storage Clusters. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, Madison, Wisconsin, June 2005.

[44] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

[45] V. Hadzilacos. A Theory of Reliability in Database Systems. *Journal of the ACM*, 35(1):121–145, 1988.

[46] R. 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.

[47] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*, Tucson, Arizona, May 1997.

[48] M. Holland, G. Gibson, and D. Siewiorek. Fast, on-line failure recovery in redundant disk arrays. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, June 1993.

[49] H.-I. Hsiao and D. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *6th International Data Engineering Conference*, 1990.

[50] W. C. Hsieh, D. Engler, and G. Back. Reverse-Engineering Instruction Encodings. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

[51] G. Hughes. Personal communication, 2004.

[52] G. Hughes and T. Coughlin. Secure Erase of Disk Drive Data. IDEMA Insight Magazine, 2002.

[53] IBM. ServeRAID - Recovering from multiple disk failures. http://www.pc.ibm.com/qtechinfo/MIGR-39144.html, 2001.

[54] M. Ji, E. Felten, R. Wang, and J. P. Singh. Archipelago: An Island-Based File System For Highly Available And Scalable Internet Services. In *4th USENIX Windows Symposium*, August 2000.

[55] J. Katcher. PostMark: A New File System Benchmark. NetApp TR-3022, October 1997.

[56] K. Keeton and J. Wilkes. Automating data dependability. In *Proceedings of the 10th ACM-SIGOPS European Workshop*, pages 93–100, Saint-Emilion, France, September 2002.

[57] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[58] D. Kuo. Model and Verification of a Data Manager Based on ARIES. *ACM Transactions on Database Systems*, 21(4):427–479, 1996.

[59] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Language and Systems*, 16(3):872–923, 1994.

[60] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[61] Z. Li, Z. Chen, S. M. Srivivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Symposium on*

File and Storage Technologies (FAST '04), pages 173–186, San Francisco, California, April 2004.

[62] C. Lumb, J. Schindler, and G. Ganger. Freeblock Scheduling Outside of Disk Firmware. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[63] M. Carey et. al. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*, Minneapolis, Minnesota, May 1994.

[64] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 105–120, San Francisco, California, December 2004.

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

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

[67] J. Menon and D. Mattson. Comparison of Sparing Alternatives for Disk Arrays. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, Gold Coast, Australia, May 1992.

[68] J. C. Mogul. A Better Update Policy. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '94)*, Boston, Massachusetts, June 1994.

[69] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[70] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. 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.

[71] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 311–324, San Antonio, Texas, June 2003.

[72] Oracle. The Self-managing Database: Automatic Performance Diagnosis. https://www.oracleworld2003.com/pub-lished/40092/40092.doc, 2003.

[73] C. U. Orji and J. A. Solworth. Doubly Distorted Mirrors. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, Washington, DC, May 1993.

[74] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '90)*, June 1990.

[75] J. Padhye and S. Floyd. On Inferring TCP Behavior. In *Proceedings of SIGCOMM '01*, San Diego, California, August 2001.

[76] A. Park and K. Balasubramanian. Providing fault tolerance in parallel secondary storage systems. Technical Report CS-TR-057-86, Princeton, November 1986.

[77] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, Chicago, Illinois, June 1988.

[78] D. A. Patterson. Availability and Maintainability $>>$ Performance: New Focus for a New Century. Key Note at FAST '02, January 2002.

[79] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based Intrusion Detection: Watching Storage Activity For Suspicious Behavior. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[80] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, Pacific Grove, California, December 1981.

[81] Postgres. The PostgreSQL Database. http://www.postgresql.com.

[82] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, Anaheim, California, April 2005.

[83] R-Undelete. R-Undelete File Recovery Software. http://www.r-undelete.com/.

[84] A. L. N. Reddy and P. Banerjee. Gracefully Degradable Disk Arrays. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 401–408, Montreal, Canada, June 1991.

[85] J. Regehr. Inferring Scheduling Behavior with Hourglass. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

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

[87] Restorer2000. Restorer 2000 Data Recovery Software. http://www.bitmart.net/.

[88] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB 24)*, New York, New York, August 1998.

[89] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[90] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, San Diego, California, June 2000.

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

[92] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[93] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-91-156, Hewlett Packard Laboratories, 1991.

[94] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

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

[96] J. Schindler and G. R. Ganger. Automated Disk Drive Characterization. TR CMU-CS-99-176, 1999.

[97] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[98] P. Selinger and M. Winslett. Pat Selinger Speaks Out. *SIGMOD Record*, 32(4):93–103, December 2003.

[99] P. Seshadri and M. Paskin. PREDATOR: An OR-DBMS with Enhanced Data Types. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*, Tucson, Arizona, May 1997.

[100] M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Evolving RPC for Active Storage. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 264–276, San Jose, California, October 2002.

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

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

[103] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[104] D. Slotnick. *Logic Per Track Devices*, volume 10, pages 291–296. Academic Press, 1970.

[105] SourceForge. SRM: Secure File Deletion for POSIX Systems. http://srm.sourceforge.net, 2003.

[106] SourceForge. Wipe: Secure File Deletion. http://wipe.sourceforge.net, 2003.

[107] SourceForge. The Linux NTFS Project. http://linux-ntfs.sf.net/, 2004.

[108] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.

[109] K. Swartz. The Brave Little Toaster Meets Usenet. In *LISA '96*, pages 161–170, Chicago, Illinois, October 1996.

[110] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, Saint-Malo, France, October 1997.

[111] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[112] S. C. Tweedie. EXT3, Journaling File System. http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[113] VMWare. VMWare Workstation 4.5. http://www.vmware.com/products/, 2004.

[114] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[115] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[116] N. Wilhelm-Olsen, J. Desai, G. Melvin, and M. Federwisch. Data protection strategies for network appliance storage systems. NetApp Technical Report TR3066, 2003.

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

[118] J. L. Wolf. The Placement Optimization Problem: A Practical Solution to the Disk File Assignment Problem. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '89)*, pages 1–10, Berkeley, California, May 1989.

[119] T. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[120] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, Ottawa, Canada, May 1995.

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

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

[123] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. *Lecture Notes in Computer Science*, (1703):54–66, 1999.

[124] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.