Orderless and Eventually Durable File Systems

By

Vijay Chidambaram

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: August 20 2015

The dissertation is approved by the following members of the Final Oral Committee:

Andrea Arpaci-Dusseau, Professor, Computer Sciences
Remzi Arpaci-Dusseau, Professor, Computer Sciences
Mark Hill, Professor, Computer Sciences
Michael Swift, Associate Professor, Computer Sciences
Parmesh Ramanathan, Professor, ECE

Dedicated to my wonderful family without whom this would not have been possible.

Acknowledgments

This Ph.D would not have been possible without the support and encouragement of a number of people. In this section, I would like to express my heart-felt gratitude to these individuals.

First, I would like to thank Remzi and Andrea for showing me how research can be fun, and inspiring me to join academia. When I came to UW Madison, I was pursuing a Masters degree. Working with Remzi and Andrea was so interesting that I decided to stay for a PhD.

I have learned so much from Remzi and Andrea. I have learned how to look at the high-level picture, and figure out if the project as a whole made sense. I have learned to pay attention to the details, and to make sure I really understand what the system is doing. I have fond memories of Remzi and Andrea teaching me how to write the introduction to a paper, and explaining how a paper should flow.

Research is a hard process, filled with failure and rejection. But Remzi and Andrea somehow make it fun. They never take the research too seriously, and they always encourage students talking about the next deadline and the next project. They magnify success, and minimize failures. I could not have asked for better advisers.

I would also like to thank Michael Swift for his valuable advise throughout my PhD. His comments have been extremely useful in honing both my work and how I presented it. In particular, his extensive comments and feedback on different parts of my job application helped make it significantly better.

I would like to express my gratitude to the other members of my thesis committee: Mark Hill and Parmesh Ramanathan. Their feedback about my research has made it significantly stronger.

I am grateful to Bilge Multu for showing me how research in a different field works, for his invaluable comments on writing papers and giving talks. His feedback on my job talk was extremely useful. Watching him setup a new research lab in Wisconsin was inspiring, and I learned many things about managing a new lab from him.

I have been fortunate to do excellent internships throughout my PhD. I am grateful to James Mickens for his sound career advice, his jokes, and his insightful comments on my research. Our collaboration has been enjoyable and fruitful. I would like to thank Mahesh Balakrishnan for the discussions at Chaat House, the extended brainstorming sessions, and the feedback during my job hunt. I am indebted to Amar Phanishayee, Anirudh Badam, Marcos Aguilera, and Ted Wobber for the technical discussions during my internships.

I was fortunate to work with a set of highly motivated and smart students during my time at UW Madison: Thanumalayan (Thanu) Pillai, Ramnatthan Alagappan, Lanyue Lu, Yiying Zhang, Zev Weiss, Tyler Harter, Suli Yang, Jun He, Yupu Zhang, Thanh Do, Mohit Saxena, Asim Kadav, and Deepak Ramamurthi. In particular, I would like to thank Thanu, who I worked with on a number of papers. Interacting with him was a pleasure due to his clear thinking and analytical reasoning. His amazing hacking skills showed me how little time it takes to implement complex pieces of software if you know what you are doing. I am grateful to Lanyue Lu for all the late-night conversations while working on papers. I would also like to thank our group alumni, Swaminathan Sundaraman and Sriram Subramaniam. When I joined the group, they made me feel at home and patiently answered all my questions. I am grateful to Vishal Sivasubramaniam for helping run experiments for the probabilistic crash consistency work.

I was able to attempt the PhD only because I had a strong support group of friends. The PhD is a lonely journey; it was made considerable less lonely by the company of Sankaralingam Paneerselvam, Ramakrishnan Durairajan, Venkanathan Varadharajan, Sandeep Viswanathan, Ragothaman Avanasi, Surya Narayanan, and Sibin K Philip. Uthra and Srinath Sridharan are like family to me, and I will always remember the friday nights spent at their home playing Poker. I was able to recharge during summers thanks to the company of Sivaguru Chendamaraikannan, Pradep Kumar, Madhumitha Ravichandran, Karthik Narayan, Gokul Raj, Arjun Sharma, Sanjay Ravi, Aravindan Rajagopalan, Manoj Kumar Panguluru, Ilaval Manickam, Anand Balaji, Anitha Thangaraj, and other friends in Seattle and the Bay Area. My last years in the PhD were made memorable due to game nights with Anusha Prasad, Meenakshi Syamkumar, Vijay Kumar, Varun Joshi, Dinesh RT, and others. Without these friends to encourage me in times of self-doubt, it is questionable whether I would have finished the PhD.

Finally, I am immensely grateful to my family. Spending Thanksgiving with Venkat anna's family, Aruna, Nikhil, and Janani, allowed me to feel at home in the US. While other families actively discourage their children from doing a PhD, my family was unique in that they encouraged me to study further when all I could think about was getting a fat paycheck and buying expensive cars. My parents never complained that I wasn't helping support the family or helping buy real-estate like my peers. Instead, they encouraged me all throughout the PhD to follow my passion. I dedicate this dissertation to my family: appa, amma, and my brother Karthic.

Contents

Acknowledgments **iii**

Contents **vii**

List of Figures and Tables **xiii**

1 Introduction **1**

1.1 Crash-Consistency Analysis *3*

1.1.1 Probabilistic Crash Consistency *4*

1.1.2 Persistence Properties *5*

1.2 New Techniques for Crash Consistency *6*

1.2.1 Backpointer-Based Crash Consistency *7*

1.2.2 Optimistic Crash Consistency *8*

1.3 Contributions *10*

1.4 Overview *11*

2 Background **13**

2.1 Definitions *13*

2.2 The Environment *14*

2.3 The Path of a Write *15*

2.4 Failure Model *17*

2.5 File-System Data Structures *17*

2.6 Crash Consistency *20*

2.6.1 Application-Level Crash Consistency *20*

2.6.2 File-System Crash Consistency 21
2.7 Crash Consistency and Ordering 27
2.8 Pessimistic Journaling 29
2.9 Summary 32

3 Motivation 33
3.1 Flushing Performance Impact 33
3.2 Implications of High Flushing Cost 34
3.3 Probabilistic Crash Consistency 37
3.3.1 Quantifying Probabilistic Consistency 38
3.3.2 Factors affecting P_{inc} 39
3.3.3 Summary 44
3.4 Required Solutions 45
3.5 Summary 45

4 Studying Persistence Properties 47
4.1 Background 48
4.1.1 Application Update Protocols 49
4.1.2 Crash States 50
4.1.3 Protocol Goals 51
4.2 Motivation 51
4.3 Persistence Properties 54
4.3.1 Intuition 54
4.3.2 Definition 55
4.4 Block Order Breaker 57
4.4.1 Goals 58
4.4.2 Approach 58
4.4.3 Testing Persistence Properties with Bob 59
4.4.4 Limitations 62
4.5 Study of Persistence Properties 62
4.5.1 Persistence Properties Tested 64

4.5.2 Results *65*

4.5.3 Discussion *72*

4.6 Conclusion *73*

5 The No-Order File System 75

5.1 Goals and Assumptions *75*

5.2 Design *78*

5.2.1 Overview *78*

5.2.2 Backpointer-based consistency *79*

5.2.3 Non-persistent allocation structures *82*

5.3 Implementation *83*

5.3.1 Operating system environment *83*

5.3.2 Backpointers *84*

5.3.3 Non-persistent allocation structures *92*

5.4 Evaluation *95*

5.4.1 Reliability *95*

5.4.2 Performance *99*

5.5 Discussion *106*

5.5.1 Limitations *106*

5.5.2 Use Cases *109*

5.5.3 Implementation Challenge: Widespread Assumptions About Block Size *110*

5.6 Proof *111*

5.6.1 Notation *111*

5.6.2 Axioms *112*

5.6.3 Data Consistency *113*

5.6.4 Version Consistency *115*

5.7 Conclusion *116*

6 The Optimistic File System 117

6.1 Goals *118*

6.2 Optimistic Crash Consistency *118*
6.2.1 Asynchronous Durability Notification *119*
6.2.2 Optimistic Consistency Properties *121*
6.2.3 Optimistic Techniques *123*
6.2.4 Durability vs. Consistency *130*
6.3 Implementation of OptFS *131*
6.3.1 Asynchronous Durability Notifications *131*
6.3.2 Handling Data Blocks *132*
6.3.3 Optimistic Techniques *133*
6.4 Evaluation *135*
6.4.1 Reliability *135*
6.4.2 Performance *136*
6.4.3 Resource consumption *141*
6.4.4 Journal size *142*
6.5 Case Studies *143*
6.5.1 Atomic Update within Gedit *143*
6.5.2 Temporary Logging in SQLite *145*
6.6 Conclusion *146*

7 Discussion 147
7.1 Comparing NoFS and OptFS *147*
7.1.1 Hardware Requirements *147*
7.1.2 Usability *148*
7.1.3 Performance *149*
7.1.4 Summary *150*
7.2 Optimistic Techniques in Other Contexts *150*
7.2.1 Optimistic Techniques in Other Crash-Consistency Mechanisms *150*
7.2.2 Optimistic Techniques in Other Media *152*
7.2.3 Optimistic Techniques on NVM *154*
7.2.4 Optimistic Techniques in Distributed Systems *155*

7.3 Using Osync *162*
7.4 Interactions with Industry *163*
7.5 Summary *164*

8 Related Work **167**
8.1 Crash Recovery *167*
8.2 Reliability Techniques *169*
8.2.1 Using Embedded Information *169*
8.2.2 Incremental FSCK in the Background *171*
8.2.3 Ordered Updates *171*
8.2.4 Delaying Durability *173*
8.3 Testing File Systems *174*
8.4 Interfaces for Consistency and Durability *175*

9 Future Work **179**
9.1 Software Async Durability Notifications *179*
9.2 Automatic Osync Substitution *180*
9.3 Tools for Testing File-System Crash Consistency *182*
9.4 Verifying Crash Consistency for Complex Stacks *183*
9.5 OptFS for SSDs *185*
9.6 Summary *186*

10 Lessons Learned and Conclusions **187**
10.1 Summary *188*
10.1.1 Crash-Consistency Analysis *188*
10.1.2 Crash-Consistency Solutions *189*
10.2 Lessons Learned *191*
10.2.1 The Importance of Interface Design *191*
10.2.2 The Importance of Asynchrony *192*
10.2.3 The Need for Better Tools *193*
10.3 Closing Words *195*

Bibliography **197**

A Appendix A **233**

A.1 Source Code *233*

List of Figures and Tables

Figure 2.1 **Ext2/3 Layout.** *The figure on top shows the layout of an ext2/3 file system. The disk address space is broken down into a series of block groups (akin to FFS cylinder groups), each of which is described by a group descriptor and has bitmaps to track allocations and regions for inodes and data blocks. The figure at the bottom shows the organization of an inode. An ext2/3 inode has twelve direct pointers to data blocks. If the file is large, indirect pointers are used.* 19

Table 2.2 **Different parts of a journaling transaction.** *The table lists the different kinds of blocks that make up a journaling transaction [20]. In the case of data journaling mode, M can also include data blocks being checkpointed to their final location in the file system.* 30

Figure 3.1 **The Cost of Flushing.** *The figure shows the performance of Filebench Varmail on different ext4 configurations. Performance increases 5X when flushes are disabled.* 34

Figure 3.2 **The Probability of Inconsistency (P_{inc}).** *An example of a window of vulnerability is shown. Blocks 1 through 6 were meant to be written in strict order to disk. However, block 5 (dark gray) is written early. Once 5 is committed, a window of vulnerability exists until blocks 3 and 4 (light gray) are committed; a crash during this time will lead to observable reordering. The probability of inconsistency is calculated by dividing the time spent in such a window (i.e., , $W = t_2 - t_1$) by the total runtime of the workload (i.e., , $t_{workload}$).* 38

Table 3.3 **Disk Model Parameters.** *The table lists the parameters for the Seagate Cheetah 15k.5 disk model used in our experiments.* . 40

Figure 3.4 **Workload Study of P_{inc}.** *The figure shows P_{inc} for six workloads. The first two workloads are sequential and random writes to a 1 GB file. Createfiles uses 64 threads to create 1M files. Fileserver, Webserver, and Varmail are part of the Filebench benchmark suite [171]. Fileserver performs a sequence of creates, deletes, appends, reads, and writes. Webserver emulates a multi-threaded web host server, performing sequences of open-read-close on multiple files plus a log file append. Varmail emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory. MySQL represents the OLTP benchmark from Sysbench [12]. Each bar is broken down into the percent contribution of the different types of misordering. Standard deviations are shown as well.* 41

Figure 3.5 **The Effect of Queue Size.** *The figure shows P_{inc} (left y-axis) and total I/O completion time (right y-axis) as the queue size of the simulated disk varies (x-axis). For this experiment, we use the Varmail workload.* 43

Figure 3.6 **The Effect of Distance.** *The figure shows* $\mathtt{P}_{\mathtt{inc}}$ *(left y-axis) and total I/O completion time (right y-axis) as the distance (in GB) between the data region and the journal of the simulated disk is increased (x-axis). For this experiment, we use the Varmail workload, with queue size set to 8.* 44

Figure 4.1 **Application Update Protocols.** *The figure shows pseudo-code for three different update protocols for SQLite (Version 3.7.17). Protocol (a) corresponds to using compilation option `SQLITE_ENABLE_ATOMIC_WRITE`. Protocol (b) corresponds to using option `PRAGMA database.synchronous = OFF`. Protocol (c) corresponds to using option `PRAGMA database.synchronous = FULL`. The first protocol is vulnerable to either a process or system crash. The second protocol is correct in the face of a process crash, but vulnerable in the event of a system crash. The last protocol is correct in the face of both process and system crashes.* 49

Table 4.2 **Possible Crash States for Example Protocol.** *The table shows different crash states for the update protocol in Listing 4.1. The second and third columns show the contents of the files `log` and `newfile`. The last column represents what the directory entry `existing_file` is pointing to.* 53

Figure 4.3 **Illustrating Atomicity and Ordering Properties.** *The figure shows the initial, final, and some of the intermediate crash states possible for the workload described in Section 4.3.2. X represents garbage data in the files. Intermediate states #A and #B represent different kinds of atomicity violations, while intermediate state #C represents an ordering violation.* 56

Figure 4.4 **BOB Workloads.** *The figure shows some of the code used in BOB Workloads. The first snippet shows that the directory is initialized with a number of files. The sample workloads does a number of metadata operations, both in synchronous (the True argument to CREAT) and asynchronous fashion. BOB would then generate different crash states and check if any of them contains re-ordered operations. For example, if the rename of file 02 is present but the unlink of 01 is not.* 61

Table 4.5 **File-System Configurations.** *The table shows the file-system configurations tested using the Block-Order Breaker. All file-system versions correspond to the versions released with Linux 3.2. The Git commit number of the latest patch applied to each file system is also provided to identify the file-system version. All mount options other than those explicitly mentioned are set to their default values.* 63

Table 4.6 **Persistence Properties.** *The table shows atomicity and ordering persistence properties that we empirically determined for different configurations of file systems. X → Y indicates that X is persisted before Y. [X,Y] → Z indicates that Y follows X in program order, and both become durable before Z. A × indicates that we have a reproducible test case where the property fails in that file system.* 66

Figure 5.1 **Backpointers.** *The figure shows a conceptual view of the backpointers present in NoFS. The file has a backpointer to the directory that it belongs to. The data block has a backpointer to the file it belong to. Files and directories have many backpointers while data blocks have a single backpointer.* 80

Table 5.2 **NoFS backpointer operations.** *The table lists the operations on backpointers caused by common file system operations. Note that all checks are done in memory.* 84

Figure 5.3 **Implementation of backpointers.** *The figure shows the different kinds of backpointers present in NoFS. foo is a child of the root inode /. This link is represented by a backlink from foo to /. Similarly, the data block is a part of foo, and hence has a backpointer to foo. Directory blocks also contain backpointers, in the form of dot entries to their owner's inode.* 86

Figure 5.4 **Failure Scenario: Rename.** *The figure presents three failure scenarios during the rename of a file. In each scenario, employing backpointers allows us to detect inconsistencies such as both the old and new parent directories claiming the renamed file.* 89

Figure 5.5 **Crash Scenario: File Create.** *The figure presents three failure scenarios during the creation of a file with 1 byte of data. In each scenario, employing backpointers allows us to detect inconsistencies such as the new file pointing to a data block that hasn't been updated.* 91

Table 5.6 **Reliability testing.** *The table shows how NoFS and ext2 react to various inconsistencies that occur due to updates not reaching the disk. NoFS detects all inconsistencies and reports an error, while ext2 lets most of the errors pass by undetected.* 97

Figure 5.7 **Micro-benchmark performance.** *This figure compares file-system performance on various micro-benchmarks. The sequential benchmarks involve reading and writing a 1 GB file. The random benchmarks involve 10K random reads and writes in units of 4088 bytes (4096 bytes - 8 byte backpointer) across a 1 GB file, with a fsync after 1000 writes. The creation and deletion benchmarks involve 100K files spread over 100 directories, with a fsync after every create or delete.* 100

Figure 5.8 **Macro-benchmark performance.** *The figure shows the throughput achieved on various application workloads. The sort benchmark is run on 500 MB of data. The varmail benchmark was run with parameters 1000 files, 100K mean dir width, 16K mean file size, 16 threads, 16K I/O size and 16K mean append size. The file and webserver benchmarks were run with the parameters 1000 files, 20 dir width, 1 MB I/O size and 16K mean append size. The mean file size was 128K for the fileserver benchmark and 16K for the webserver benchmark. Fileserver benchmark used 50 threads while webserver used 100 threads.* 101

Figure 5.9 **Interaction of Background Scan and File-System Activity.** *Figure (a) depicts the reduction in write bandwidth when sequential writes interleave with the background scan. Figure (b) shows that the running of the scan increases slowly with the amount of data in the file system.* 103

Figure 5.10 **Cost of the Background Scan.** *Figure (c) illustrates the rate at which data blocks are scanned. Figure (d) demonstrates the performance cost incurred when the stat system call is run on unverified inodes.* 104

Table 5.11 Notations on containers 112

Figure 6.1 **Optimistic Journaling.** *The figure shows four transactions in progress, involving writes to main memory, the on-disk journal, and to in-place checkpoints on disk. A rectangle block indicates that the file system has been notified that the write has been durably completed. Cloud-shaped blocks indicate that the write has been initiated, but the file system has not yet been notified of its completion and it may or may not be durable. Circles indicate dirty blocks in main memory that cannot be written until a previous write is durable; a dashed line indicates the write it is dependent on. Finally, the solid arrow indicates that the meta-data may refer to the data block.* 122

Figure 6.2 **Optimistic Journaling: Selective Data Journaling.** *The figure shows that selective data journaling may be used when transactions involve overwriting in-place data. Data blocks are now placed in the journal and checkpointed after the transaction is committed.* 129

Table 6.3 **Reliability Evaluation.** *The table shows the total number of simulated crashpoints, and the number of crashpoints resulting in a consistent state after remounting the file system and running recovery.* 135

Figure 6.4 **I/O Timeline.** *The figure illustrates how I/O are handled in ext4 with flushes and OptFS. Legend: A – application; B – buffer cache; C – disk cache; D – disk platter; W – `write()` system call; F – `fsync()` system call; O – `osync()` system call. The workload is the Filebech Varmail benchmark that emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory. In ext4, the application is blocked while blocks are flushed to the disk platter; in OptFS, the application is blocked only until I/O hits the disk cache.* 137

Figure 6.5 **Performance Comparison.** *Performance is shown normalized to ext4 ordered mode with flushes. The absolute performance of ordered mode with flushes is shown above each workload. Sequential writes are to 80 GB files. 200K random writes are performed over a 10 GB file, with an `fsync()` every 1K writes. The overwrite benchmark sequentially writes over a 32 GB file. Createfiles uses 64 threads to create 1M files. Fileserver emulates file-server activity, using 50 threads to perform a sequence of creates, deletes, appends, reads, and writes. Webserver emulates a multi-threaded web host server, performing sequences of open-read-close on multiple files plus a log file append, with 100 threads. Varmail emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory. Each workload was run for 660 seconds. MySQL OLTP benchmark performs 200K queries over a table with 1M rows.* 139

Table 6.6 **Dirty Block Background Writeout.** *The table shows how tuning dirty block writeout by `pdflush` threads affects file-system performance for the Filebench Createfiles workload. OptFS writes out dirty blocks lazily, collecting them into big batches. ext4 dirty data is written out eagerly (when 5 percent of memory is dirty) by background threads. When the background threads are tuned to behave more lazily, ext4 performance is similar to OptFS. Ext4 performs better when the background threads batch writes more than OptFS. Legend: DBR – Dirty Background Ratio. DR – Dirty Ratio. DEC – Dirty Expire Centiseconds (shown here in seconds).* 141

Figure 6.7 **Performance with Small Journals.** *The figure shows the variation in OptFS performance on the MySQL OLTP benchmark as the journal size is varied. When OptFS runs out of journal space, it issues flushes to safely checkpoint transactions. Note that even in this stress scenario, OptFS performance is 5x better than ext4 ordered mode with flushes.* 142

Table 6.8 **Resource Consumption.** *The table shows the average resource consumption by OptFS and ext4 ordered mode for a 660 second run of Filebench Varmail. OptFS incurs additional overhead due to optimistic techniques such as data transactional checksumming.* 144

Table 6.9 **Case Study: Gedit and SQLite.** *The table shows the number of simulated crashpoints that resulted in a consistent or inconsistent application state after remounting. It also shows the time required for an application operation.* 144

Table 7.1 **Access Latency of Different Storage Media.** *The table shows the read and write latency for small, random I/O on different storage media. These numbers are based on demonstrated prototypes [243, 298].* 152

Table 7.2 **ext4 and OptFS performance on SSDs.** *The table shows the performance of ext4 and OptFS in different modes for the Filebench Varmail workload running on top of an SSD.* 153

1 Introduction

Data is being generated at unprecedented rate today [162, 169, 306]. Apart from personal information [32, 74, 182], we trust digital storage services with financial data [28, 95, 125], medical information [69, 110, 127], and even government information [147, 189]. Widely used applications such as cloud services [1, 6, 13, 14, 74, 143, 315], mobile applications [36, 233, 241], key-value stores [66–68, 80, 90, 154, 300], and traditional database services [39, 43, 70, 104, 202, 227, 244, 263, 269, 271] all need to interact with and manage storage. Thus, storage services form a crucial part of everyday life for both individuals and businesses.

All storage services must solve a crucial problem: how to safely update data in the presence of interruptions? An update could be interrupted by a power loss or a system crash. Power-loss events have occurred at critical infrastructure buildings like hospitals [192], at major events such as the Super Bowl [53], and inside datacenters [148, 176, 183–186, 292, 307]. System crashes may result from bugs in the operating system [26, 129, 136, 152, 313] or in device drivers [131, 132, 178]. Carelessly updating data could result in data loss [120, 146] or corruption [61, 262].

Given that most applications access data via a file system such as Windows NTFS [191], Linux ext4 [167], or Apple HFS+ [18], the problem of file-system crash consistency (*i.e.*, preserving the integrity of file system in the face of crashes) becomes important. File systems update their state on storage carefully: the updates are sequenced so that the file system can recover correctly after a crash in the middle of the sequence. Ordering updates to the storage lies at the heart of most existing crash-consistency techniques such as journaling [220] or copy-on-write [114, 234].

Ordering updates to disk was a simple task when disks were first introduced [40, 107, 118, 267]. All writes were synchronous; the write command would not return until the data was written on the non-volatile surface (*i.e., persisted*). The order in which writes were submitted to the disk was the same order in which writes became persistent.

Unfortunately, the introduction of *write buffering* [260] in modern disks greatly complicates this apparently simple process. With write buffering enabled, a write submitted to the disk is first stored in on-board volatile RAM, and later written to the non-volatile surface. Thus, disk writes may complete out of order, as a smart disk scheduler may reorder requests for performance [124, 250, 308]; further, the notification received after a write issue implies only that the disk has received the request, not that the data has been written to the disk surface persistently.

Write ordering is achieved in modern drives via expensive *cache flush* operations [264]; such flushes cause all buffered dirty data in the drive to be persisted immediately. To ensure A is written before B, a client issues the write to A, and then a cache flush; when the flush returns, the client can safely assume that A reached the disk; the write to B can then be safely issued, knowing it will be persisted after A.

Unfortunately, cache flushing is expensive, sometimes prohibitively so. Flushes make I/O scheduling less efficient, as the disk has fewer requests to choose from. A flush also unnecessarily forces *all* previous writes to disk, whereas the requirements of the client may be less stringent. In addition, during a large cache flush, disk reads may exhibit extremely long latencies as they wait for pending writes to complete [255]. Finally, flushing conflates ordering and durability; if a client simply wishes to order one write before another, forcing the first write to disk is an expensive manner in which to achieve such an end. In short, the classic approach of flushing is *pessimistic*; it assumes a crash will occur and goes to great lengths to ensure that the disk is never in an inconsistent state via flush commands. The poor performance that results from pessimism has led some systems to disable flushing, apparently sacrificing correctness for performance; for example, for a significant amount of time, the Linux ext3 default configuration did not flush caches [58, 59].

The high cost of ordering writes has resulted in most file systems being run in one of two configurations: high performance with no crash consistency, or low performance with strong crash consistency. In this dissertation, we seek to answer the question: *is strong crash consistency possible with high performance?* In the first part of this dissertation, we analyze the behavior of applications and file systems. In the second part, we build on the insights gained from our analysis to develop new techniques that show it is possible to build systems that provide both strong crash consistency and high performance.

1.1 Crash-Consistency Analysis

Given the high cost of flushing, several systems disable flushing to increase performance [58, 59]. Some practitioners have reported that after disabling flushes, a crash does not always lead to file-system inconsistency [286]. It is important to understand the factors that affect whether a crash leads to file-system inconsistency. Our approach is to collect traces of the disk blocks written by different workloads, and examine the results of a crash at different points in the trace; this allows us to determine the probability of a crash leading to inconsistency for the given workload. We present the results of this analysis in this dissertation.

Apart from file-system crash consistency, users care about *application-level crash consistency*; for example, the contents of a user's browsing history in Google Chrome should *only* include web pages that they have visited. It is possible to have a consistent file system and an inconsistent application after a crash. Similar to file systems, applications need to order their writes to maintain crash consistency. Since ordering writes is expensive, applications avoid doing so, and instead depend on the file system to persist writes in the "correct" order. However, the correct order is application-specific, and different file systems persist writes in different ways, thus compromising application crash consistency. To better understand this problem, we define persistence properties – how file systems persist dirty writes – and examine how they vary among different file systems.

1.1.1 Probabilistic Crash Consistency

When disk flushes are disabled, there is the risk of file-system inconsistency after a crash; an inopportune crash could lead to corruption in user data or file-system internal structures [226]. We refer to such an approach as *probabilistic crash consistency* [46], in which a crash *might* lead to file system inconsistency, depending on many factors, including workload, system and disk parameters, and the exact timing of the crash or power loss.

Some practitioners observed that most crashes did not cause file-system inconsistency for some workloads [286]. Such observations further embolden other practitioners to disable flushes and risk corruption to gain performance. In this dissertation, one of our first contributions is the careful study of probabilistic crash consistency, wherein we show which exact factors affect the odds that a crash will leave the file system inconsistent.

Our approach is to collect the block traces (*i.e.*, the list of disk blocks written) of various workloads, and try to determine the *windows of vulnerability*: points in the workload's lifetime when a crash would have resulted in file-system inconsistency. For example, if data was added to the end of a file, there is a window of vulnerability in the time between the write of the file's book-keeping structure (*e.g.*, *inode*) that points to the new data block, and the write of the new data block. The re-ordering depends on a number of factors, such as the workload being used, the file-system layout on disk, and the size of the disk queue. We run experiments varying these factors and determining the resulting windows of vulnerabilities.

We find that for some workloads, such as large sequential writes or static web servers, the chances of file-system inconsistency are quite low. For other workloads, such as database update queries or email servers, there is a significant chance of inconsistency upon crash. Thus, although the probabilistic approach suffices for certain simple workloads, increasing performance while maintaining file-system consistency requires additional crash-consistency techniques.

1.1.2 Persistence Properties

Application-level crash consistency depends upon the order in which application writes are persisted on disk. Ideally, applications would carefully control the order of writes. Unfortunately, the only primitive available to applications to order writes is the `fsync()` system call [73, 153]. The `fsync()` call guarantees that when the call returns, all dirty data associated with the file (passed as an argument to `fsync()`) is persistent on the storage device. To ensure that the data is persistent, `fsync()` flushes the disk cache, thereby inheriting the performance penalties associated with the disk-cache flush.

Due to the high performance cost of `fsync()`, many applications do not invoke `fsync()` between two writes that need to be ordered, instead relying on the file system to persist writes in the correct order. Since the POSIX standard [278] does not define the order in which file systems should persist dirty writes (in the absence of `fsync()`), each file system persists dirty writes in a different fashion.

In this dissertation, we seek to define and understand the persistence behavior of file systems. We define *persistence properties* that capture the persistence behavior of the file system [216]. They break down into two global categories: the *atomicity* of operations (e.g., does the file system ensure that `rename()` is atomic in the event of a crash? [172]), and the *ordering* of operations (e.g., does the file system ensure that file creations are persisted in the same order they were issued?).

To analyze file-system persistence properties, we develop a simple tool, known as the *Block Order Breaker* (Bob). Bob collects the disk blocks written by the file system, and creates new crash states by persisting subsets of the collected disk blocks onto the initial disk state. With this simple approach, Bob can find which persistence properties do *not* hold for a given system. We use Bob to study six Linux file systems (ext2, ext3, ext4, reiserfs, btrfs, and xfs) in various configurations.

We find that persistence properties vary widely among the tested file systems. For example, appends to file A are persisted before a later rename of file B in the ordered journaling mode of ext3, but not in the same mode of ext4, unless a special option is enabled. Furthermore, persistence properties vary between different configurations of the same file system.

Applications should not assume that the underlying file system provides specific persistence properties. Instead, the application should use ordering primitives to ensure that the on-disk state is updated correctly.

1.2 New Techniques for Crash Consistency

We believe our analyses illustrate that current techniques that depend on flushing disk caches do not satisfy the needs of file systems and applications. File systems require new techniques that offer high performance while providing strong crash-consistency guarantees. Applications require new primitives that allow them to maintain crash consistency efficiently.

In the second part of the dissertation, we present our solutions to these problems. We present Backpointer-Based Consistency [47], a new crash-consistency technique that does not require any disk flushes in the file-system update protocol. We describe Optimistic Crash Consistency [46], a new crash-consistency technique that decouples ordering and durability. We introduce a new primitive, `osync()`, that allows applications to order their writes without making the writes immediately persistent.

1.2.1 Backpointer-Based Crash Consistency

As mentioned before, most file systems use ordered updates to maintain crash consistency. Thus, their update protocol (*i.e.*, the sequence of writes used to update on-disk state) has many *ordering points*. In the event of a crash, ordering points allow the file system to reason about which writes reached the disk and which did not, enabling the file system to take corrective measures, such as replaying the writes, to recover. Unfortunately, ordering points are not without their own set of problems. By their very nature, ordering points introduce waiting into the file-system code, thus potentially lowering performance. They constrain the scheduling of disk writes, both at the operating system level and at the disk driver level. They introduce complexity into the file-system code, which leads to bugs and decreased reliability [220, 221, 311, 312]. The use of ordering points also forces file systems to ignore the end-to-end argument [237], as the support of lower-level systems and disk firmware is required to implement imperatives such as the disk cache flush. When such imperatives are not properly implemented [247], file-system consistency is compromised [226]. In today's cloud computing environment [19], the operating system runs on top of a tall stack of virtual devices, and only one of them needs to neglect to enforce write ordering [294] for file-system consistency to fail.

We can thus summarize the current state of the art in file-system crash consistency as follows. At one extreme is a lazy, optimistic approach that writes blocks to disks in any order (e.g., ext2 [38]); this technique does not add overhead or induce extra delays at run-time, but requires an expensive (and often prohibitive) disk scan after a crash. At the other extreme are eager, pessimistic approaches that carefully order disk writes (e.g., ZFS or ext3); these techniques pay a perpetual performance penalty in return for consistency guarantees and quick recovery. We seek to obtain the best of both worlds: the simplicity and performance benefits of the lazy approach with the strong consistency and availability of eager file systems.

In this dissertation, we present the *No-Order File System* (NoFS) [47], a simple, optimistic, lightweight file system which maintains consistency without resorting to the use of ordering. NoFS employs a new approach to providing consistency called *backpointer-based consistency*, which is built upon references in each file-system object to the files or directories that own it. We extend a logical framework for file systems [256] to prove that the incorporation of backpointer-based consistency in an order-less file system guarantees a certain level of consistency. We simplify the update protocol through *non-persistent allocation structures*, reducing the number of blocks that need to reach disk to successfully complete an operation.

Through reliability experiments, we demonstrate that NoFS is able to detect and handle a wide range of inconsistencies. We compare the performance of NoFS with ext2, an order-less file system with no consistency guarantees, and ext3, a widely-used file system with strong consistency guarantees. We show that NoFS has excellent performance overall, matching or exceeding the performance of ext3 on various workloads.

1.2.2 Optimistic Crash Consistency

While backpointer-based consistency provides excellent performance, it does not support atomic primitives such as the `rename()` system call [101]. As the name suggests, the `rename()` call is used to atomically change the name of a given file. Since a large number of applications use `rename()` to atomically update files, the usability of NoFS is limited. Thus, we sought to develop new crash-consistency techniques and application-level primitives that meet our twin goals of providing high performance and enabling applications to build meaningful, efficient crash-consistency protocols.

We introduce *optimistic crash consistency* [46], a new approach to building a crash-consistent journaling file system. This optimistic approach takes advantage of the fact that in many cases, ordering can be achieved through other means and that crashes are rare events (similar to optimistic concurrency control [113, 142]). However, realizing consistency in an optimistic fashion is not without challenge; we thus develop a range of novel techniques, including a new extension of the transactional checksum [222] to detect data/metadata inconsistency, delayed reuse of blocks to avoid incorrect dangling pointers, and a selective data journaling technique to handle block overwrite correctly. The combination of these techniques leads to *both* high performance and *deterministic* consistency; in the rare event that a crash does occur, optimistic crash consistency either avoids inconsistency by design or ensures that enough information is present on the disk to detect and discard improper updates during recovery.

We demonstrate the power of optimistic crash consistency through the design, implementation, and analysis of the *optimistic file system (OptFS)*. OptFS builds upon the principles of optimistic crash consistency to implement *optimistic journaling*, which ensures that the file system is kept consistent despite crashes. Optimistic journaling is realized as a set of modifications to the Linux ext4 file system, but also requires a slight change in the disk interface to provide what we refer to as *asynchronous durability notification*, *i.e.*, , a notification when a write is persisted in addition to when the write has simply been received by the disk. We describe the implementation of OptFS and evaluate its performance. We show that for a range of workloads, OptFS significantly outperforms classic Linux ext4 with pessimistic journaling. Despite ensuring crash consistency, OptFS performs almost identically to Linux ext4 with probabilistic journaling.

Central to the performance benefits of OptFS is the separation of ordering and durability. By allowing applications to order writes without incurring a disk flush, and request durability when needed, OptFS enables application-level consistency at high performance. OptFS introduces two new file-system primitives: `osync()`, which ensures ordering between writes but only *eventual* durability, and `dsync()`, which ensures immediate durability as well as ordering.

We show how these primitives provide a useful base on which to build higher-level application consistency semantics. Specifically, we show how a document editing application can use `osync()` to implement the atomic update of a file (via a create and then atomic rename), and how the SQLite database management system can use file-system provided ordering to implement ordered transactions with eventual durability. We show that these primitives are sufficient for realizing useful application-level consistency at high performance.

Of course, the optimistic approach, while useful in many scenarios, is not a panacea. If an application requires *immediate, synchronous* durability (instead of eventual, asynchronous durability with consistent ordering), an expensive cache flush is still required. In this case, applications can use `dsync()` to request durability (as well as ordering). However, by decoupling the durability of writes from their ordering, OptFS provides a useful middle ground, thus realizing high performance and meaningful crash consistency for many applications.

1.3 Contributions

We describe the main contributions of this dissertation:

- We design a framework to investigate probabilistic crash consistency. We enumerate the ordering relationships that need to hold among different kinds of blocks in the journaling protocol to maintain crash consistency. We present the reasons why file-system inconsistency is rare upon crash for certain workloads.

- We define persistence properties, capturing the aspects of file-system behavior that affect application-level consistency. We study the persistence properties of sixteen file-system configurations and show that they vary widely.

- We design a new crash-consistency protocol, Backpointer-Based Crash Consistency, that provides strong guarantees without requiring the disk cache be flushed.

- We design Optimistic Crash Consistency, a new crash-consistency protocol that decouples ordering and durability, providing strong guarantees and excellent performance.

- We introduce `osync()`, a new ordering primitive that allows applications to build correct, efficient update protocols.

- We introduce Asynchronous Durability Notifications, a new disk interface that allows file systems and applications to discern the durability status of writes in the disk cache.

- We contribute open-source implementations of the Optimistic File System [277] and the No-Order File System [276], embodying the principles of optimistic crash consistency and backpointer-based consistency, respectively.

1.4 Overview

We briefly describe the contents of the different chapters in the dissertation.

- **Background**. In Chapter 2, we provide background on file-system data structures, and how applications interact with the file system. We define crash consistency for applications and file systems. We describe how crash consistency depends on ordering writes, and finally describe the journaling technique for file-system crash consistency.

- **Analysis**. In Chapter 3, we show the high performance cost of flushing disk caches. We analyze the effect of disabling cache flushes on crash consistency for different workloads, and describe the far-reaching effects of the expensive nature of flushes. In Chapter 4, we examine how application-level crash consistency depends upon specific behaviors of file systems. We define these behavior as *persistence properties*, and build a tool, the Block-Order Breaker [216], to analyze file-system persistence properties.

- **Solutions**. The next two chapters describe our new techniques for file-system crash consistency. In Chapter 5, we describe Backpointer-Based Consistency [47], a new technique that allows file systems to maintain crash consistency without ordering writes. In Chapter 6, we describe Optimistic Crash Consistency [46], a new form of crash consistency that decouples ordering from durability. In Chapter 7, we discuss how these new techniques compare with each other, and how they could be used in other contexts.

- **Related Work**. In Chapter 8, we describe prior work about crash consistency. We discuss work related to our specific techniques, and describe efforts similar to the Block-Order Breaker in analyzing crash behavior.

- **Future Work, Lessons Learned, and Conclusions**. In Chapter 9, we describe avenues in which our work could be extended. We describe how we can remove limitations of our work and apply our techniques in new contexts. In Chapter 10, we highlight the lessons learned and summarize our work. We believe our contributions will be valuable to a broad range of systems in the future.

2 Background

In this chapter, we provide background required for various aspects of this dissertation. First, we define terms that are used throughout the dissertation (§2.1). We then describe the environment in which applications interact with storage (§2.2) and how an application write propagates down the stack to the storage device (§2.3). We discuss the failure model (§2.4) and then describe file-system structures that are relevant to this dissertation (§2.5). We define crash consistency for applications and file systems (§2.6), and explain how crash consistency depends upon ordering writes to storage. We discuss how ordering is achieved at different levels in the storage stack (§2.7). Finally, we describe standard journaling as used in current file systems and explain why it is pessimistic (§2.8).

2.1 Definitions

Process Crash. A process crash is defined as the sudden termination of the process. There is no opportunity for clean-up activities. While any system calls in progress complete, all data in the process buffers are lost. For example, a process may be killed using the `kill -9` command.

System Crash. A system crash is defined as the sudden termination of the entire operating system. A system crash may occur due to bugs (in the kernel [49, 152, 156, 311, 312] or device drivers [132, 178]) or due to a power loss. If the whole system loses power, data in operating-system buffers and in volatile storage caches are lost. Otherwise, only data in the operating system buffers are lost. In the rest of this dissertation, we use "crash" to indicate a system crash with the whole system losing power.

Durability. Data is defined to be durable if it can be read back correctly from a storage device after a power loss. For example, data stored in volatile RAM is not durable, while data stored on magnetic media or solid state drives is usually considered durable.

Atomicity. An update is atomic with respect to failure if after a crash, the user obtains either the old version of the data in its entirety, or the new version of the data in its entirety. In the rest of this dissertation, we refer to such "failure-atomicity" as atomicity.

2.2 The Environment

Most applications persist their state on storage devices such as SATA drives [137]. There are many software layers between the application and the raw storage device. The raw storage and the software layers are collectively termed the storage stack. We briefly describe the major components of the environment that are of interest in this dissertation.

The File System. The file system presents the file abstraction to applications. Applications can read and write files without worrying exactly where the data is stored. Files can be logically contiguous even when physically stored at non-contiguous locations. Most applications use files to store their data. Some applications, such as Oracle's database products, can directly run on raw storage for increased performance [121]. This dissertation mainly deals with applications that use the file system.

The Page Cache. The Page Cache is a system-wide kernel-space cache for data read from or written to storage devices. The data is read or written in terms of pages or buffers (each of size 4096 bytes). Each page has metadata associated with it (*e.g.*, whether it is dirty). The page cache allows repeated access to pages to be significantly faster.

The I/O Subsystem. When pages are written from the page cache to the storage device, they go through the I/O subsystem [30]. The I/O subsystem comprises of several layers such as the block layer, the I/O scheduler, etc. These layers break up the pages into sector-sized requests and pass them to the device. For the purposes of this dissertation, we need to understand only two aspects of the I/O subsystem: that write requests (even a single page) will be broken up into smaller requests as they pass down the stack, and that requests will be re-ordered as they pass through the stack.

The Storage Device. The I/O subsystem submits read and write requests to the storage device. There are a number of storage devices available, ranging from the cheap SATA disk drives [137] to SCSI drives [81, 140, 304], SSDs [9] and RAID arrays [211]. PCM-based devices [144, 149], memresistors [270], and other new technology will be available soon. In this dissertation, we focus on the cheap, commodity drives (such as SATA) meant for server and desktop usage.

Storage Device Cache. Most storage devices available today include a small, on-board volatile RAM cache [226, 235, 260, 308]. These caches range in size from 8–128 MB. The caches are used both for buffering writes and caching reads; typically, the sizes of the read and write caches are not revealed by the manufacturer. If the device loses power, all dirty data in the cache is lost.

2.3 The Path of a Write

We now describe the path of a write from the application to storage. We describe the path in the context of the Linux Operating system [20, 30], but other operating systems such as Windows and Mac OSX behave in a similar manner.

Most applications use library functions (*e.g.*, `fwrite` from `libc`) to write to files. Before writing to a file, the application must open the file and get a handle to it (*e.g.*, a file descriptor). Arguments to the `open()` call determine how the write is performed: direct I/O, synchronous I/O, etc. In this section, I describe one particular case that is common in applications: asynchronous, buffered I/O.

After obtaining a handle to the file, the application calls a library function such as `fwrite` using the file handle as an argument. The library function in turns invokes a system call such as `write()`. The application's data is transferred from user-space buffers to system-wide buffers in the page cache. At this point, the `write()` system call completes.

Transferring the data from the buffer cache to the storage device happens at a later point of time. The background write-out is performed by the `pdflush` threads, and is triggered by multiple factors such as memory pressure, the amount of time the data has been dirty in memory, etc. The background threads submit the data to the storage device. The background threads do not submit the data in the order it was dirtied; the blocks are sorted and submitted such that performance is maximized [250].

Most storage devices today (such as SATA or SCSI drives) are equipped with small, on-board volatile caches. A write submitted to the device is first stored in the cache, and at a later point written to the non-volatile storage media. The storage device aims to write data to the non-volatile media in the most efficient manner possible; for example, a SATA drive tries to re-order writes so that disk seeks are minimized [308]. Thus, the order in which writes are submitted to the drive is not necessarily the order in which the writes are made durable.

2.4 Failure Model

In this dissertation, we are concerned mainly with fail-stop crashes that result in all volatile data being lost. Thus, the system has to recover based on data that was persisted on storage. Note that the data could be volatile at several layers in the storage stack – in the buffer cache, in I/O scheduler queues, in the storage device cache etc.

For the sake of simplicity, we assume that all data in volatile memory is lost in the case of crash. The techniques presented in this dissertation can be extended to handle cases where some subset of volatile data is lost, and some subset is later persisted to storage.

We do not handle a wide range of errors such as Latent Sector Errors [22] and silent data corruption [23]. We view such problems as orthogonal to crash consistency; existing solutions [221] can be adopted in addition to the techniques proposed in this dissertation.

2.5 File-System Data Structures

In this section, we introduce file-system data structures that we refer to throughout the dissertation. We define structures such as the inode that are generic and used by many existing file systems; we also define structures specific to the ext family of file systems, as most of the work in this dissertation involves ext2 [38], ext3 [288, 289], and ext4 [167].

Inode. An inode is a data structure that contain metadata about a user file, a directory, and or other special files (*e.g.*, symbolic links). The metadata includes file attributes (*e.g.*, size, access control lists) as well as pointers to data blocks on disk.

In ext2 and ext3 file systems, an inode has 12 direct pointers to its data blocks. If its data needs more blocks, the inode will use its indirect pointer that points to an *indirect block* which contains pointers to data blocks. If the indirect block is not enough, the inode will use a *double indirect block* which contains pointers to indirect blocks. At most, an inode can use a triple indirect block which contains pointers to double indirect blocks. In ext4, block pointers are replaced by extents representing a contiguous region of disk blocks.

Directory. A directory is a list of references to files and other directories. Most file systems are organized as a tree of directories, with empty directories or files forming the leaves of the tree. The root of file-system hierarchy is referred to as the "root" directory. A "directory entry" is a single reference to another file or directory.

In ext2 and ext3, directory entries are managed as linked lists of variable length entries. Each directory entry contains the inode number, the entry length, the file name and its length. In ext4, hash tables are used for storing directory entries.

Data Block. A data block can contain user's data or directory entries. If an inode represents a user file, its data blocks contain user's data. If an inode represents a directory, its data blocks contain directory entries.

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

File-System Layout. Figure 2.1 (from Haryadi Gunawi's disseration [103]) depicts the ext2/3 on-disk layout. In this organization (which is loosely based on FFS [172]), the disk is split into a number of *block groups*; within each block group are bitmaps, an inode table, and data blocks. Each block group also contains a redundant copy of crucial file-system control information such as the superblock and the group descriptors.

The ext2 and ext3 file systems maintain bitmaps that store the allocation information of inodes and data blocks. There is a single bit representing

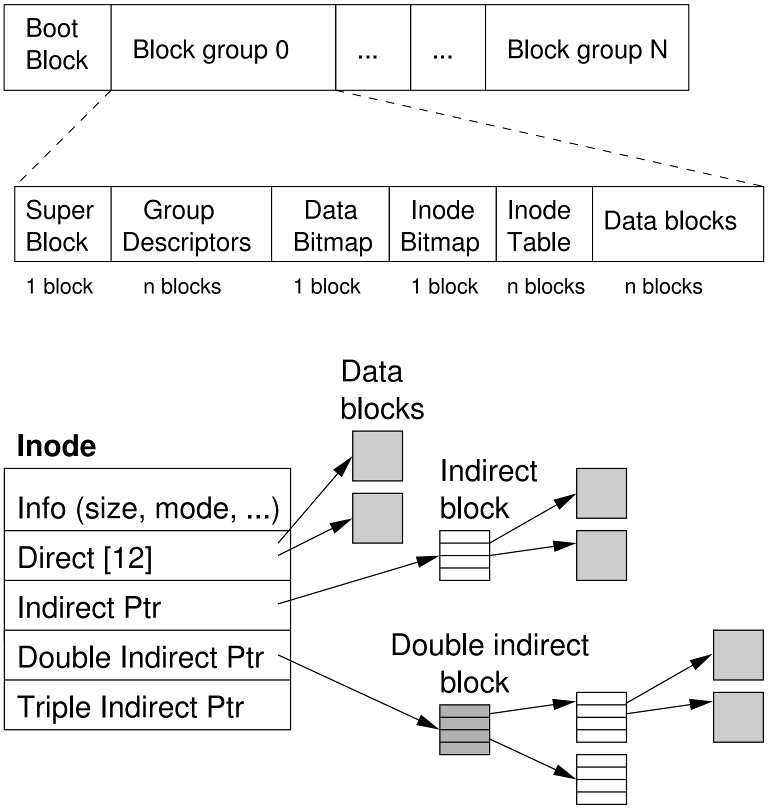


Figure 2.1: **Ext2/3 Layout.** *The figure on top shows the layout of an ext2/3 file system. The disk address space is broken down into a series of block groups (akin to FFS cylinder groups), each of which is described by a group descriptor and has bitmaps to track allocations and regions for inodes and data blocks. The figure at the bottom shows the organization of an inode. An ext2/3 inode has twelve direct pointers to data blocks. If the file is large, indirect pointers are used.*

each inode or data block present on disk. A group descriptor describes a block group. It contains information such as the location of the inode table, block bitmap, and inode bitmap for the corresponding group. In addition, it also keeps track of allocation information such as the number of free blocks, free inodes, and used directories in the group. The inode table consists of an array of inodes (as defined above), and it can span multiple blocks.

2.6 Crash Consistency

We first define crash consistency, and describe application-level and file-system crash consistency, and their relation to each other. We discuss existing solutions for application-level and file-system crash consistency.

Crash Consistency. A system is consistent if certain system-specific invariants are maintained. A system is *crash-consistent* if the required invariants are maintained after a crash. The crash may be either a system crash or a process crash. Note that crash consistency applies to file systems and any application that requires invariants to be maintained for correct operation.

2.6.1 Application-Level Crash Consistency

Modern applications employ a large number of data structures to achieve good user experience. These data structures are persisted in a large number of files. We ran `strace -e trace=file` when starting up the Google Chrome browser (64-bit version 43.0.2357) [94]. We observed that Chrome accesses over 960 files under the home directory alone while starting up.

Applications must maintain invariants over their persistent data structures. For example, the contents of a user's browsing history in Google Chrome should *only* include web pages that they have visited. Another invariant is that the history should contain *all* the web pages previously visited. When all the invariants of an application hold, it is deemed *consistent*.

Many applications are required to be consistent (after recovery) even if the process or the entire system crashes or loses power. After a system crash, the application must recover using only the on-disk state. Hence, the on-disk state must be kept consistent at all times.

Applications achieve crash consistency by carefully designing *application update procotols* that update the on-disk state one piece at a time [216]. The order in which on-disk state is updated is fixed; based on this, appropriate recovery code is written. For example, assume triplet (A, B, C) needs to be updated to (X, Y, Z), and that only one item in the triplet can be updated at a time. If the order of updates is fixed as X, then Y, and finally Z, the recovery code needs to handle intermediate states such as (X, B, C) and (X, Y, C). Thus, the order in which on-disk application state is updated is crucial; re-ordering them could lead to inconsistent application state after recovery [216, 316].

Existing Solutions. Application-level crash consistency is implemented in an ad-hoc manner, with several applications re-using the same techniques (such as atomically updating the whole file using `rename()` or logging) [216]. Applications could use existing databases such as SQLite [207] to consistently store and update their state; however, this usually comes at a heavy performance cost [126].

2.6.2 File-System Crash Consistency

We first describe invariants that file systems seek to maintain across crashes. We then discuss different levels of file-system crash consistency. We discuss how file-system crash consistency is related to application-level crash consistency, and finally describe existing solutions for crash consistency in file systems.

File-System Crash Invariants

A file system's metadata structures (*e.g.*, allocation structures) need to be kept consistent in the event of a crash. Actions such as creating a new file require several separate on-disk metadata structures (*e.g.*, the file inode, the directory, etc.) to be updated in an atomic fashion. Thus, a file system's crash invariants involve relationships between different on-disk metadata structures. For example, there are three invariants mentioned in the Soft Updates paper [86, 87, 251]:

1. A pointer always points to an initialized structure (*e.g.*, an inode must be initialized before a directory entry references it).
2. A resource must be used only after nullifying all previous pointers to it (*e.g.*, a pointer to a data block must be nullified before that disk block may be reallocated for a new inode).
3. The last pointer to a resource must be reset only after a new pointer has set to the same resource (*e.g.*, when renaming a file, do not remove the old name for an inode until after the new name has been written)

Crash-Consistency Levels

There are three different levels of crash consistency in file systems, depending upon the invariants that are maintained [47, 256]. We describe each in turn, starting with the weakest.

Metadata consistency: The metadata structures of the file system are entirely consistent with each other. There are no dangling files and no duplicate pointers. The counters and bitmaps of the file system, which keep track of resource usage, match with the actual usage of resources on the disk. Therefore a resource is in use if and only if the bitmaps say that it is in use. Metadata consistency does not provide any guarantees about data.

Data consistency: Data consistency is a stronger form of metadata consistency. Along with the guarantee about metadata, there is the additional guarantee that all data that is read by a file belongs to that file. In other words, a read of file A may not return garbage data, or data belonging to some file B. It is possible that the read may return an older version of the data of file A.

Version consistency: Version consistency is a stronger form of data consistency with the additional guarantee that the version of the metadata matches the version of the referred data. For example, consider a file with a single data block. The data block is overwritten, and a new block is added, thereby changing the file version: the old version had one block, and the new version has two blocks. Version consistency guarantees that a read of the file does not return old data from the first block and new data from the second block (since the read would return the old version of the data block and the new version of the file metadata).

Relation with Application-Level Crash Consistency

Note that application-level crash consistency is quite different from file-system crash consistency. File-system crash consistency is primarily concerned with file-system metadata; as long as the various invariants about file-system metadata structures are satisfied, the file system is consistent. For example, even an empty file system is consistent. In contrast, application-level consistency is concerned with application-level invariants, and do not care about the file-system structures. It is possible to have an application be consistent while running on top of an inconsistent file system. For example, after a crash, ext2 is inconsistent until `fsck` is run [175]. If the application does its own logging or journaling (and depending on the inconsistencies introduced by the crash), it is quite possible the application is consistent while the file system is not.

Existing Solutions

There have been a number of techniques developed over the years to maintain file-system consistency after a crash. We briefly describe each of these techniques.

Ordered Writes. The most basic technique to maintain crash consistency is to order file-system updates such that the sequence does not violate crash invariants at any point. Early file systems such as the Unix 4.2 BSD File System [204], the Fast File System [172], the DOS file systems [77], and VMS file system [170] sequenced metadata updates with synchronous writes. Synchronous writes result in significant performance degradation [177, 203, 249], prompting file systems to not maintain some or all crash invariants, choosing instead to fix the file system on reboot using a utility such as fsck [175, 213, 232]. Instead of synchronously writing in the desired order, another approach is to pass the required order down to the disk scheduler, and allow the scheduler to make write durable in the correct order [37, 87]; the drawback is that this greatly increases the complexity of the driver, since detailed dependency information must be passed down to the scheduler to increase performance.

Using a file-system checker has two main disadvantages. First, the checker must be run on the entire disk, causing file-system activity to be halted for a significant amount of time (especially for large RAID arrays [211]). Although several optimizations were developed to reduce the running time of the file-system check [111, 173, 213], it is still too expensive for large volumes, prompting the file-system community to turn to other solutions. Second, file-system checkers are often complex, containing thousands of lines of code; unsurprisingly, they often contain bugs leading to corruption and data loss [102, 311, 312].

File systems that depend upon on the file-system check alone for consistency cannot provide data consistency, since there is no way for the file system to differentiate between valid data and garbage in a data block. Therefore, file reads may return garbage after a crash. At the end of the scan, the checker holds complete information about every object in the file system: it knows all the files that are reachable through the directory hierarchy, and all the data blocks that are reachable through the files. Hence, duplicate resource allocation and orphan resources can be handled, ensuring metadata consistency.

Journaling. Journaling uses the idea of write-ahead logging [51, 106] to solve the consistency problem: metadata (and sometimes data) is first logged to a separate location on disk, and when all writes have safely reached the disk, the information is written into its original place in the file system. Over the years, this technique has been incorporated into a number of file systems such as NTFS [191], JFS [27], XFS [272], ReiserFS [229], and ext3 [288, 289].

Journaling file systems offer data or metadata consistency based on whether data is journaled or not [220]. Both journaling modes use at least one ordering point in their update protocols, where they wait for the journal writes to be persisted on disk before writing the commit block. Journaling file systems often perform worse than their order-less peers, since information needs to be first written to the log and then later to the correct location on disk. Recovery of the journal is needed after a crash, but it is usually much faster than the file-system check.

Copy-on-Write. Copy-on-write file systems use the shadow paging technique [42, 44, 234, 269]. Shadow paging directs a write to a metadata or data block to a new copy of the block, never overwriting the block in place. Once the write is persisted on disk, the new information is added to the file-system tree. The ordering point is in-between these two steps, where the file system atomically changes between the old view of the metadata to one which includes the new information. Copy-on-write has been used in a number of file systems [114, 234], with the most recent being ZFS [29] and btrfs [166].

Copy-on-write file systems provide metadata, data, and version consistency due to the use of logging and transactions. Modern copy-on-write file systems like ZFS achieve good performance, though at the cost of high complexity. The large size of these file systems (tens of thousands of lines of code [242]) is partly due to the copy-on-write technique, and partly due to advanced features such as storage pools and snapshots.

Soft Updates. Soft updates involves tracking dependencies among in-memory copies of metadata blocks, and carefully ordering the writes to disk such that the disk always sees content that is consistent with the other disk metadata. In order to do this, it may sometimes be necessary to roll back updates to a block at the time of write, and roll-forward the update later. Soft updates was implemented for FFS, and enabled FFS to achieve performance close to that of a memory-based file system [87] . However, it was extremely tricky to implement the ordering rules correctly, leading to numerous bugs. Although the Featherstitch project [84] reduces the complexity of soft updates, the idea has not spread beyond the BSD distributions.

Soft updates provide metadata and data consistency at low cost. FFS with soft updates cannot tell the difference between different versions of data, and hence does not provide version consistency. Soft updates also provide high availability since a blocking file-system check is not required; instead, upon reboot after a crash, a snapshot of the file-system state is taken, and the file-system check is run on the snapshot in the background [173].

Battery-backed RAM. Another solution is to simply use a battery-backed RAM cache [57]. The file system can then simply use the synchronous writes approach, but achieve significantly higher performance. Battery-backed RAM has been used in systems such as eNVy [309], Rio File Cache [45], Lucent DNCP [33], Conquest [299] and RAMCloud [201]. The disadvantages of this approach include the higher cost of the battery-backed RAM and managing the data transfer between the RAM cache and non-volatile storage.

Every technique we have described in this section is built upon carefully ordering writes to a cache or to storage. Just as file-system crash consistency is built upon ordering writes to storage, application-level crash consistency is built upon ordering operations to files. In the next section, we describe how file systems and applications order their requests.

2.7 Crash Consistency and Ordering

We now describe how applications and file systems order their updates.

Application-Level Ordering. Applications update their on-disk state via file metadata operations (*e.g.*, creating a file, linking a file, renaming a file, etc.) and file data operations (*e.g.*, writing to a file, truncating a file, etc.). To maintain crash consistency, applications require that these operations be performed on disk in a specific order. For example, many applications atomically update multiple blocks of a file using the following sequence:

1. Create temporary file `tmp`
2. Write new contents into `tmp`
3. Rename `tmp` over original file

In this sequence, it is vital that step 2 occur before step 3. Otherwise, a crash could lead to an empty original file; this has been observed in practice [59]. Applications use the `fsync()` system call to order updates [216, 217, 257, 275]. The `fsync()` system call ensures that when it returns successfully, dirty data and metadata associated with the file has been flushed to storage [278]. The above sequence then becomes:

1. Create temporary file `tmp`
2. Write new contents into `tmp`
3. Persist `tmp` using `fsync()`
4. Rename `tmp` over original file

File-System Level Ordering. File-system operations such as appending to a file involve updating multiple data structures on disk. For example, a file append involves changes to the file inode, the appended data block, and an allocation structure for data blocks. Most file systems carefully order updates to these data structures to maintain crash consistency. File systems submit writes to the I/O subsystem, which in turn submits writes to the storage device. Thus, the writes could be re-ordered by either the I/O subsystem or the storage device itself.

The I/O subsystem, or more specifically, the I/O scheduler, may re-order writes to achieve better performance [21]. To preserve ordering, barriers are provided: no I/O before the barrier may complete after the barrier, and vice versa. Barrier flags can be attached to I/O requests, and barrier semantics are respected by I/O scheduler as they re-order requests.

Barriers are a construct of the I/O subsystem; storage devices do not understand barriers, and hence they need to be translated into lower-level commands by the I/O subsystem. On devices such as IDE, SATA, and SCSI, a barrier command translates into a FLUSH command [137, 264]. As the name indicates, the FLUSH command ensures that all dirty data in the device cache is written to non-volatile storage. The FLUSH command is coarse-grained: the caller cannot pick what must be made durable; the whole cache is flushed, often at high performance cost [226]. Using finer-grained interfaces such as Tagged Queuing [174, 266] has met with limited success [60].

In 2010, the Linux kernel stopped using barriers to order requests (as of version 2.6.36) [60]. Pushing down ordering information down into the I/O subsystem via barriers was found to reduce performance significantly. Since file systems were the layer that had the most context for ordering decisions, it was decided to retain ordering information at that level. File systems now issue writes in the correct order and wait for them to complete. Ordering was achieved by tagging requests with the Forced Unit Access (FUA) flag [266]. The FUA flag forces the request to by-pass the cache entirely. When a write request tagged with FUA completes, it indicates that the write has been made durable on stable storage. The FUA flag is used in conjunction with the FLUSH flag to effectively order requests. For example, to write A before B, but to also ensure that both A and B are durable, a client might write A, then write B with both cache flushing and FUA enabled; this ensures that when B reaches the drive, A (and other dirty data) will be forced to disk; subsequently, B will be forced to disk due to the FUA. On devices that do not support FUA, an extra FLUSH request is used to achieve the same effect.

When applications issue a `fsync()` system call, it results in the file system issuing a FLUSH request in turn. File systems submit all the dirty data and metadata associated with the file to the storage device, and then issue a FLUSH request to ensure that the file becomes persistent. The application-level ordering primitive is connected to the file-system ordering primitive in this manner.

Thus, ordering is an important primitive on which application-level and file-system crash consistency depends. The correctness, usability, and performance of ordering primitives significantly affects how easy it is to build correct, crash-consistent applications and file systems.

2.8 Pessimistic Journaling

Given the ordering mechanisms described in Section 2.7, we now describe how a journaling file system safely commits data to disk in order to maintain consistency in the event of a system crash. We base our discussion on ordered-mode Linux ext3 and ext4 [287, 288], though much of what we say is applicable to other journaling file systems such as SGI XFS [272], Windows NTFS [258], and IBM JFS [27]. In ordered mode, file-system metadata is journaled to maintain its consistency; data is not journaled, as writing each data block twice reduces performance substantially.

When an application updates file-system state, either metadata, user data, or (often) both need to be updated in a persistent manner. For

Symbol	Blocktype	Must be persisted before
D	Data block	–
J_M	Metadata block that has been journalled	–
J_C	Transaction commit block	D, J_M
M	Metadata block written to file-system	D, J_M, J_C

Table 2.2: **Different parts of a journaling transaction.** *The table lists the different kinds of blocks that make up a journaling transaction [20]. In the case of data journaling mode, M can also include data blocks being checkpointed to their final location in the file system.*

example, when a user appends a block to a file, a new *data* block (D) must be written to disk (at some point); in addition, various pieces of *metadata* (M) must be updated as well, including the file's inode and a bitmap marking the block as allocated.

We refer to the atomic update of metadata to the journal as a *transaction*. Before committing a transaction Tx to the journal, the file system first writes any data blocks (D) associated with the transaction to their final destinations; writing data before transaction commit ensures that committed metadata does not point to garbage. After these data writes complete, the file system uses the journal to log metadata updates; we refer to these journal writes as J_M. After these writes are persisted, the file system issues a write to a commit block (J_C); when the disk persists J_C, the transaction Tx is said to be *committed*. Finally, after the commit, the file system is free to update the metadata blocks in place (M); if a crash occurs during this *checkpointing* process, the file system can recover simply by scanning the journal and replaying committed transactions. Details can be found elsewhere [220, 288].

We thus have the following set of ordered writes that must take place: D before J_M before J_C before M, or more simply: $D \rightarrow J_M \rightarrow J_C \rightarrow M$. Note that D, J_M, and M can represent more than a single block (in larger transactions), whereas J_C is always a single sector (for the sake of write atomicity). Table 2.2 summarizes the different blocks in journaling and their ordering requirements. To achieve this ordering, the file system issues a cache flush wherever order is required (*i.e.*, , where there is a $\rightarrow$ symbol).

Optimizations to this protocol have been suggested in the literature, some of which have been realized in Linux ext4. For example, some have noted that the ordering between data and journaled metadata ($\mathrm{D} \rightarrow \mathrm{J_M}$) is superfluous; removing that ordering can sometimes improve performance ($\mathrm{D|J_M} \rightarrow \mathrm{J_C} \rightarrow \mathrm{M}$) [220].

Others have suggested a "transactional checksum" [222] which can be used to remove the ordering between the journal metadata and journal commit ($\mathrm{J_M}$ and $\mathrm{J_C}$). In the normal case, the file system cannot issue $\mathrm{J_M}$ and $\mathrm{J_C}$ together, because the drive might reorder them; in that case, $\mathrm{J_C}$ might hit the disk first, at which point a system crash (or power loss) would leave that transaction in a seemingly committed state but with garbage contents. By computing a checksum over the entire transaction and placing its value in $\mathrm{J_C}$, the writes to $\mathrm{J_M}$ and $\mathrm{J_C}$ can be issued together, improving performance; with the checksum present, crash recovery can avoid replay of improperly committed transactions. With this optimization, the ordering is $\mathrm{D} \rightarrow \overline{\mathrm{J_M|J_C}} \rightarrow \mathrm{M}$ (where the bar over the journal updates indicates their protection via checksum).

Interestingly, these two optimizations do not combine, *i.e.*, , $\mathrm{D|}\overline{\mathrm{J_M|J_C}} \rightarrow \mathrm{M}$ is not correct; if the file system issues D, $\mathrm{J_M}$, and $\mathrm{J_C}$ together, it is possible that $\mathrm{J_M}$ and $\mathrm{J_C}$ reach the disk first. In this case, the metadata commits before the data; if a crash occurs before the data is written, an inode (or indirect block) in the committed transaction could end up pointing to garbage data. Oddly, ext4 allows this situation with the "right" set of mount options.[1]

We should note that one other important ordering exists among up-

[1] The options are `journal_checksum` and `journal_async_commit`.

dates, specifically the order *between* transactions; journaling file systems assume transactions are committed to disk in order (*i.e.*, , $\mathsf{Tx}_i \to \mathsf{Tx}_{i+1}$) [288]. Not following this ordering could lead to odd results during crash recovery. For example, a block B could have been freed in Tx_i, and then reused in Tx_{i+1}; in this case, a crash after Tx_{i+1} committed but before Tx_i did would lead to a state where B is allocated to two files.

Finally, and most importantly, we draw attention to the *pessimistic* nature of this approach. Whenever ordering is required, an expensive cache flush is issued, thus forcing *all* pending writes to disk, when perhaps only a subset of them needed to be flushed. In addition, the flushes are issued even though the writes may have gone to disk in the correct order anyhow, depending on scheduling, workload, and other details; flushes induce extra work that may not be necessary. Finally, and perhaps most harmful, is the fact that the burden of flushes is added despite the fact that crashes are rare, thus exacting a heavy cost in anticipation of an extremely occasional event.

2.9 Summary

In this chapter, we presented background material essential for this dissertation. We described how a write propagates down the storage stack from the application to the storage media. We introduced the relevant file-system data structures. We defined crash consistency, and explained how crash consistency is maintained in applications and file systems. We described how crash consistency is dependent on ordering writes; we discussed mechanisms available to applications and file systems to order writes. Finally, we provided an overview of the standard journaling protocol used in file systems such as ext3.

3 Motivation

We use the background material presented in Chapter 2 to motivate this dissertation. We saw that ordering primitives are crucial to crash consistency on both file systems and applications. On current systems, the most common ordering primitive that is used is the FLUSH command. Thus, the performance of both applications and file systems is tied to the performance of the FLUSH command.

In this chapter, we first demonstrate the significant performance degradation due to flushing disk caches (§3.1). We discuss the numerous and complex implications of the high cost of flushing (§3.2). One effect of the high performance cost is that some practitioners turn off flushing; we analyze the result of disabling flushes on different workloads (§3.3). Finally, we describe what is required to solve this problem (§3.4).

3.1 Flushing Performance Impact

To better understand the performance impact of cache flushing during pessimistic journaling, we performed a simple experiment. Specifically, we ran the Varmail benchmark (from the Filebench suite [82]) atop Linux ext4 [167] running on a SATA disk drive, both with and without cache flushing; with cache flushing enabled, we also enabled transactional checksums [221] to see their performance impact. Varmail is a good choice here as it simulates an email server and includes many small synchronous updates to disk, thus stressing the journaling machinery described above.

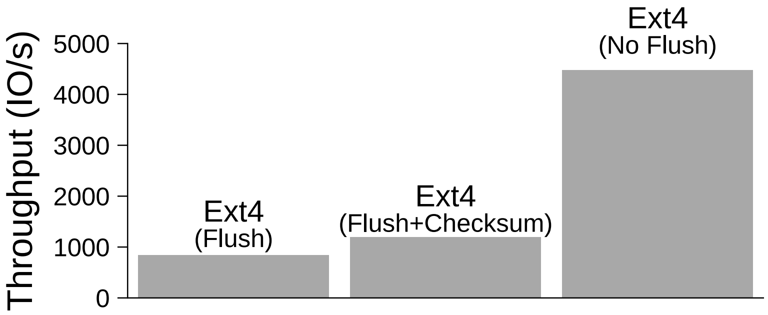


Figure 3.1: **The Cost of Flushing.** *The figure shows the performance of Filebench Varmail on different ext4 configurations. Performance increases 5X when flushes are disabled.*

The experimental setup for this benchmark and configuration is described in more detail in Section 6.4.2.

From Figure 3.1, we observe the following. First, transactional checksums increase performance slightly, showing how removing a single ordering point (via a checksum) can help. Second, and most importantly, there is a vast performance improvement when cache flushing is turned off, in this case nearly a factor of five. Given this large performance difference, in some installations, cache flushing is disabled, which leads to the following question: what kind of crash consistency is provided when flushing is disabled? Surprisingly, the answer is not "none", as we now describe.

3.2 Implications of High Flushing Cost

The FLUSH command is the primary ordering primitive that is used by applications and file systems today. The `fsync()` system call results in one or more flushes, depending upon the file system being used. The high performance cost of the flush has resulted in several implications.

Systems Turn Off Flushing. First, as we saw in Section 3.3, several systems simply turn off flushes, choosing to risk corruption and data loss, rather than have their systems run so slowly. For a number of years, the default distribution of Linux ext3 had barriers turned off [58]. This was not an accident; the Linux kernel maintainers choose this dangerous default intentionally:

> *Last time this came up lots of workloads slowed down by 30 percent so I dropped the patches in horror. I just don't think we can quietly go and slow everyone's machines down by this much... There are no happy solutions here, and I'm inclined to let this dog remain asleep and continue to leave it up to distributors to decide what their default should be.*

As another example, the `fsync()` system call on Mac OS X does not flush the disk cache [153]. The system call ensures that the data is pushed to the storage device, but does not flush the device cache. Quoting from the man page:

> *Note that while fsync() will flush all data from the host to the drive (i.e. the "permanent storage device"), the drive itself may not physically write the data to the platters for quite some time and it may be written in an out-of-order sequence. Specifically, if the drive loses power or the OS crashes, the application may find that only some or none of their data was written. The disk drive may also re-order the data so that later writes may be present, while earlier writes are not. This is not a theoretical edge case. This scenario is easily reproduced with real world workloads and drive power failures.*

For developers that seek to ensure durability of the data, Apple provides the `F_FULLFSYNC` flag to be used with the `fcntl` system call.

Drives Lie About Flushing. Since the performance of applications depends directly upon the performance of the FLUSH command, drives can seem faster on application benchmarks by having the FLUSH command return before the data was made durable. Users have reported seeing this behavior [247, 262]. Recent work on power-fail testing of drives has also confirmed this behavior [317].

Application Developers Avoid Using Fsync. If implemented correctly, `fsync()` is extremely expensive; therefore, application developers avoid using it. However, without `fsync()`, application updates could get reordered on different file systems. File systems such as ext4 support the most common update patterns (*e.g.*, atomically updating a file using `rename()`), erroneously leading developers to believe their application is correct on all file systems [216, 275]. When the file-system behavior changes, it leads to applications breaking. For example, when ext4 introduced delayed allocation, it resulted in widespread data loss because applications were not using `fsync()` [146, 280].

Virtualized Stacks Disable Flushing. With the advent of virtualization and cloud computing [19], applications run inside virtual machines on tall storage stacks. For example, the Windows I/O stack contains 18 layers between the application and the storage [283]. For applications to be crash consistent on virtualized I/O stacks, every layer in the stack has to obey ordering primitives. Unfortunately, some layers do not honor flush requests. For example, VirtualBox ignores flush requests for increased performance [294].

In summary, the high cost of flushing has led to the situation where all the major actors in the storage world – drive manufacturers, file-system developers, and application developers – have stopped using flushes (and ordering primitives built upon flushes) correctly. As a result, the crash consistency of applications and file systems has been severely compromised.

3.3 Probabilistic Crash Consistency

Given the potential performance gains, practitioners sometimes forgo the safety provided by correct implementations that issue flushes and choose to disable flushes [58]. In this fast mode, a risk of file-system inconsistency is introduced; if a crash occurs at an untimely point in the update sequence, and blocks have been reordered across ordering points, crash recovery as run by the file system will result in an inconsistent file system.

In some cases, practitioners observed that skipping flush commands sometimes did *not* lead to observable inconsistency, despite the presence of (occasional) crashes. Such commentary led to a debate within the Linux community as to underlying causes. Long-time kernel developer Theodore Ts'o hypothesized why such consistency was often achieved despite the lack of ordering enforcement by the file system [286]:

> *I suspect the real reason why we get away with it so much with ext3 is that the journal is usually contiguous on disk, hence, when you write to the journal, it's highly unlikely that commit block will be written and the blocks before the commit block have not. ... The most important reason, though, is that the blocks which are dirty don't get flushed out to disk right away!*

What the Ts'o Hypothesis refers to specifically is two orderings: $J_M \rightarrow J_C$ and $J_C \rightarrow M$ (refer to Table 2.2 for terminology). In the first case, Ts'o notes that the disk is likely to commit J_C to disk after J_M even without an intervening flush (note that this is without the presence of transactional checksums) due to *layout* and *scheduling*; disks are simply unlikely to reorder two writes that are contiguous. In the second case, Ts'o notes that $J_C \rightarrow M$ often holds without a flush due to *time*; the checkpoint traffic that commits M to disk often occurs long after the transaction has been committed, and thus ordering is preserved without a flush.

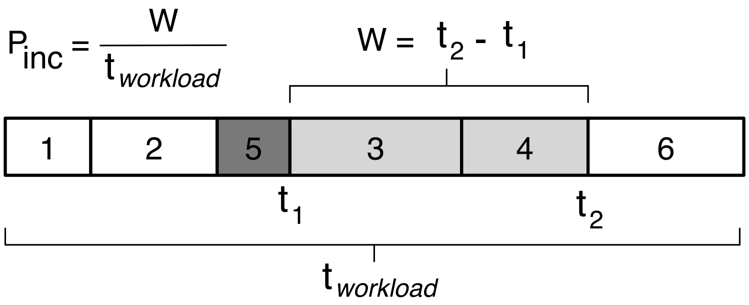


Figure 3.2: **The Probability of Inconsistency (P_{inc}).** *An example of a window of vulnerability is shown. Blocks 1 through 6 were meant to be written in strict order to disk. However, block 5 (dark gray) is written early. Once 5 is committed, a window of vulnerability exists until blocks 3 and 4 (light gray) are committed; a crash during this time will lead to observable reordering. The probability of inconsistency is calculated by dividing the time spent in such a window (i.e., , $W = t_2 - t_1$) by the total runtime of the workload (i.e., , $t_{workload}$).*

We refer to this arrangement as *probabilistic consistency*. In such a configuration, typical operation may or may not result in much reordering, and thus the disk is only sometimes in an inconsistent state. A crash may not lead to inconsistency despite a lack of enforcement by the file system via flush commands. Despite probabilistic crash consistency offering *no* guarantees on consistency after a crash, many practitioners are drawn to it due to large performance gains from turning off flushing.

3.3.1 Quantifying Probabilistic Consistency

Unfortunately, probabilistic consistency is not well understood. To shed light on this issue, we ran simulations to quantify how often inconsistency arises without flushing. To do so, we disabled flushing in Linux ext4 and extracted block-level traces underneath ext4 across a range of workloads. We analyzed the traces carefully to determine the chances of an inconsistency occurring due to a crash. Our analysis was done via a simulator built atop DiskSim [35], which enables us to model complex disk behaviors (*e.g.*, scheduling, caching).

The main output of our simulations is a determination of when a *window of vulnerability* (W) arises, and for how long such windows last. Such a window occurs due to reordering. For example, if A should be written to disk before B, but B is written at time t_1 and A written at t_2, the state of the system is vulnerable to inconsistency in the time period between, $W = t_2 - t_1$. If the system crashes during this window, the file system will be left in an inconsistent state; conversely, once the latter block (A) is written, there is no longer any concern.

Given a workload and a disk model, it is thus possible to quantify the *probability of inconsistency* (P_{inc}) by dividing the total time spent in windows of vulnerability by the total run time of the workload ($P_{inc} = \cup W_i / t_{workload}$); Figure 3.2 shows an example. Note that when a workload is run on a file system with cache-flushing enabled, P_{inc} is always zero.

3.3.2 Factors affecting P_{inc}

We explored P_{inc} in a systematic manner. Specifically, we determined sensitivity to workload and disk parameters such as the queue size and the placement of the journal relative to file-system structures. We used the validated Seagate Cheetah 15k.5 disk model [35] provided with DiskSim for our experiments. Table 3.3 provides the parameters of the disk model.

Workload

We first studied how the workload can impact P_{inc}. For this experiment, we used 6 different workloads described in the caption of Figure 3.4. We chose a mix of workloads with different characteristics: for example, there are read dominated workloads, workloads with different mixes of sequential and random writes, workloads with and without `fsync()` calls, and workloads using different number of threads.

Parameter	Value
Total Capacity	146.8 GB
Scheduling Policy	SPTF-OPT[1]
Spindle Speed	15K RPM
Average Latency	2 ms
Average Read Seek Time	3.5 ms
Average Write Seek Time	4.0 ms
Cache Size	16 MB
Number of discs	2
Number of heads	4
Number of buffer segments	32
Maximum number of write segments	11
Segment size (in blks)	1200
Queue length	8

Table 3.3: **Disk Model Parameters.** *The table lists the parameters for the Seagate Cheetah 15k.5 disk model used in our experiments.*

From Figure 3.4, we make the following observations. Most importantly, $\mathtt{P_{inc}}$ is workload dependent. For example, if a workload is mostly read oriented, there is little chance of inconsistency, as file-system state is not updated frequently (*e.g.*, Webserver). Second, for write-heavy workloads, the *nature* of the writes is important; workloads that write randomly or force writes to disk via `fsync()` lead to a fairly high chance of a crash leaving the file system inconsistent (*e.g.*, random writes, MySQL, Varmail). Third, there can be high variance in $\mathtt{P_{inc}}$; small events that change the order of persistence of writes can lead to large differences in chances of inconsistency. Fourth, although concurrent writes from multiple threads increases the probability of inconsistency, the increase is not significant; although the File Server workload uses multiple threads to write different files, the sequential nature of the writes results in low probability of inconsistency. Finally, even under extreme circumstances, $\mathtt{P_{inc}}$ never reaches 100% (the graph is cut off at 60%); there are many points in the lifetime of

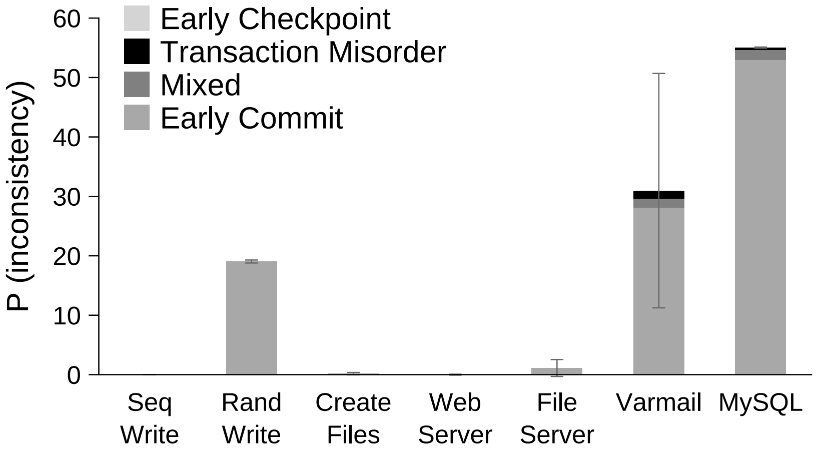


Figure 3.4: **Workload Study of P_{inc}.** *The figure shows P_{inc} for six workloads. The first two workloads are sequential and random writes to a 1 GB file. Createfiles uses 64 threads to create 1M files. Fileserver, Webserver, and Varmail are part of the Filebench benchmark suite [171]. Fileserver performs a sequence of creates, deletes, appends, reads, and writes. Webserver emulates a multi-threaded web host server, performing sequences of open-read-close on multiple files plus a log file append. Varmail emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory. MySQL represents the OLTP benchmark from Sysbench [12]. Each bar is broken down into the percent contribution of the different types of misordering. Standard deviations are shown as well.*

a workload when a crash will not lead to inconsistency.

Beyond the overall P_{inc} shown in the graph, we also break the probability further by the *type* of reordering that leads to a window of vulnerability. Specifically, assuming the following commit ordering ($D|J_M \rightarrow J_C \rightarrow M$), we determine when a particular reordering (*e.g.*, J_C before D) has resulted. The graph breaks down P_{inc} into fine-grained reordering categories, grouped into the following relevant cases: *early commit* (*e.g.*, $J_C \rightarrow J_M|D$), *early checkpoint* (*e.g.*, $M \rightarrow D|J_M|J_C$), *transaction misorder* (*e.g.*, $Tx_i \rightarrow Tx_{i-1}$), and *mixed* (*e.g.*, where more than one category could

[1] Shortest Positioning Time First

be attributed).

Our experiments show that early commit before data, ($J_C \rightarrow D$), is the largest contributor to P_{inc}, accounting for over 90% of inconsistency across all workloads, and 100% in some cases (Fileserver, random writes). This is not surprising, as in cases where transactions are being forced to disk (*e.g.*, due to calls to `fsync()`), data writes (D) are issued just before transaction writes (J_M and J_C); slight re-orderings by the disk will result in J_C being persisted first. Also, for some workloads (MySQL, Varmail), *all* categories might contribute; though rare, early checkpoints and transaction misordering can arise. Thus, any approach to provide reliable consistency mechanisms must consider all possible causes, not just one.

Queue Size

For the remaining studies, we focused on Varmail, as it exhibits the most interesting and varied probability of inconsistency. First, we studied how disk scheduler queue depth matters. Most storage drives can accept a fixed number of requests at the same time (until their queue is full) [308]. Storage drives re-order requests in the queue to increase performance. Thus, the queue size represents the amount of re-ordering the disk performs even if the cache is switched off; furthermore, large sequential requests (a disk write request may contain upto 128 MB of sequential data on modern SATA drives) will by-pass the write cache on many modern drives [226]. Figure 3.5 plots the results of our experiment. The left y-axis plots P_{inc} as we vary the number of outstanding requests to the disk; the right y-axis plots performance (overall time for all I/Os to complete).

From the figure, we observe the following three results. First, when there is no reordering done by the disk (*i.e.*, , queue size is 1), there is no chance of inconsistency, as writes are committed in order; we would find the same result if we used FIFO disk scheduling (instead of SPTF). Second, even with small queues (*e.g.*, 8), a great deal of inconsistency can arise;

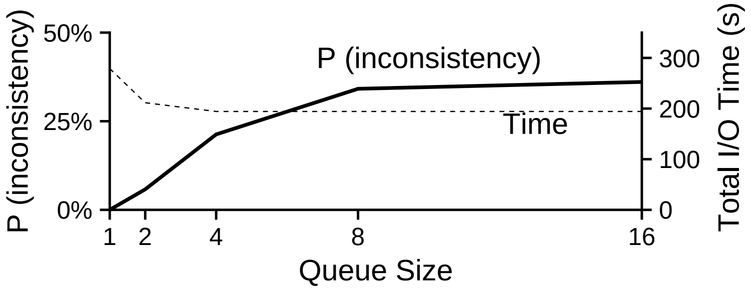


Figure 3.5: **The Effect of Queue Size.** *The figure shows* $\mathtt{P_{inc}}$ *(left y-axis) and total I/O completion time (right y-axis) as the queue size of the simulated disk varies (x-axis). For this experiment, we use the Varmail workload.*

one block committed too early to disk can result in very large windows of vulnerability. Finally, we observe that a modest amount of reordering does indeed make a noticeable performance difference; in-disk SPTF scheduling improves performance by about 30% with a queue size of 8 or more.

Journal Layout

We studied how distance between the main file-system structures and the journal affects $\mathtt{P_{inc}}$. Figure 3.6 plots the results of varying the location of Varmail's data and metadata structures (which are usually located in one disk area) from close to the journal (left) to far away.

From the figure, we observe distance makes a significant difference in $\mathtt{P_{inc}}$. Recall that one of the major causes of reordering is early commit (*i.e.,* , $\mathtt{J_C}$ written before D); by separating the location of data and the journal, it becomes increasingly unlikely for such reordering to occur. Secondly, we also observe that increased distance is not a panacea; inconsistency (10%) still arises for Varmail. Finally, increased distance from the journal can affect performance somewhat; there is a 14% decrease in performance when moving Varmail's data and metadata from right next to the journal to 140 GB away.

We also studied a number of other factors that might affect $\mathtt{P_{inc}}$, in-

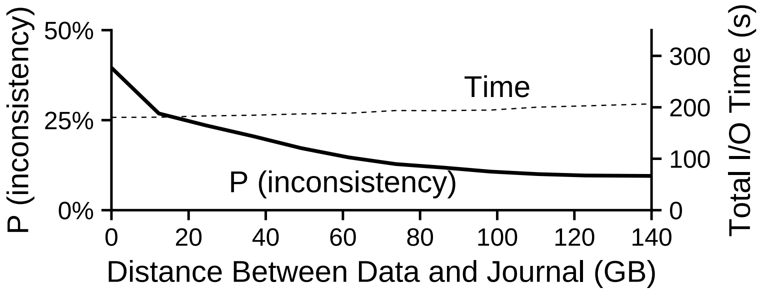


Figure 3.6: **The Effect of Distance.** *The figure shows* $\mathtt{P_{inc}}$ *(left y-axis) and total I/O completion time (right y-axis) as the distance (in GB) between the data region and the journal of the simulated disk is increased (x-axis). For this experiment, we use the Varmail workload, with queue size set to 8.*

cluding the disk size, the journal size, and the placement of the journal as it relates to track boundaries on the disk (since the SPTF policy may result in re-ordering among journal writes on different tracks to minimize rotational delay) . In general, these parameters did not significantly affect $\mathtt{P_{inc}}$ and thus are not included.

3.3.3 Summary

Classical journaling is overly pessimistic, forcing writes to persistent storage often when only ordering is desired. As a result, users have sometimes turned to probabilistic journaling, taking their chances with consistency in order to gain more performance. We have carefully studied which factors affect the consistency of the probabilistic approach, and shown that for some workloads, it works fairly well; unfortunately, for other workloads with a high number of random-write I/Os, or where the application itself forces traffic to disk, the probability of inconsistency becomes high. As devices become more sophisticated, and can handle a large number of outstanding requests, the odds that a crash will cause inconsistency increases. Thus, to advance beyond the probabilistic approach, a system must include machinery to either avoid situations that lead to inconsistency, or be able to detect and recover when such occurrences arise.

3.4 Required Solutions

The high cost of flushing has resulted in file-system and application crash-consistency being compromised. Flushing is an extremely inefficient ordering primitive because it provides ordering via durability of writes. To order write A before write B, the file system submits A to the drive, flushes the drive cache, and then submits B to the drive. If the application does not need to make either A or B durable (for now), flushing is an inefficient way to obtain the required ordering.

Crash consistency requires ordering updates. Making the updates durable is not required for maintaining consistency (although it may be required for usability). Thus, if a low-cost ordering primitive was available, which decoupled ordered from durability, applications and file systems could maintain crash consistency without suffering significant performance degradation.

In this dissertation, we design new crash-consistency techniques and ordering primitives to achieve this goal. Backpointer-Based Consistency System [47] maintains crash consistency without using any ordering primitives (Chapter 5). Optimistic Crash Consistency introduces new techniques that decouple ordering and durability, and a new primitive, `osync()`, that allows applications to order writes without making them immediately durable (Chapter 6).

3.5 Summary

We motivated the techniques and interfaces developed in this dissertation. We first presented the significant performance cost of flushing the disk cache, and discussed its far-reaching implications. We then described

Probabilistic Crash Consistency, and presented our analysis of various factors that affect whether a crash leads to file-system inconsistency. Finally, we discussed how this dissertation presents solutions to the different problems described in this chapter.

4 Studying Persistence Properties

Many important applications, including databases such as SQLite [263] and key-value stores such as LevelDB [93], are currently implemented on top of file systems such as ext4 instead of directly on raw disks. Such data-management applications must also be crash consistent, but achieving this goal atop modern file systems is challenging for two fundamental reasons.

The first challenge is that the exact guarantees provided by file systems are unclear and underspecified. Applications communicate with file systems through the POSIX system-call interface [278], and ideally, a well-written application using this interface would be crash-consistent on any file system that implements POSIX. Unfortunately, while the POSIX standard specifies the effect of a system call in memory, specifications of how disk state is mutated in the event of a crash are widely misunderstood and debated [3]. As a result, each file system persists application data slightly differently, leaving developers guessing.

To add to this complexity, most file systems provide a multitude of configuration options that subtly affect their behavior; for example, Linux ext3 provides numerous journaling modes, each with different performance and robustness properties [288]. While these configurations are useful, they complicate reasoning about exact file-system behavior in the presence of crashes.

The second challenge is that building a high-performance application-level crash-consistency protocol is not straightforward. Maintaining application consistency would be relatively simple (though not trivial) if all state were mutated synchronously. However, such an approach is prohibitively slow, and thus most applications implement complex *update protocols* to remain crash-consistent while still achieving high performance. Similar to early file system and database schemes, it is difficult to ensure that applications recover correctly after a crash [257, 275]. The protocols must handle a wide range of corner cases, which are executed rarely, relatively untested, and (perhaps unsurprisingly) error-prone.

In this chapter, we seek to answer the following question: what are the behaviors exhibited by modern file systems that are relevant to building crash-consistent applications? In section 4.1, we describe how application-level crash consistency is dependent on file-system behavior. We motivate why persistence properties are required in section 4.2. We define persistence properties in section 4.3. In section 4.4, we then describe the *Block Order Breaker* (Bob), a tool we built to analyze file-system persistence properties. Finally, we present the results of the study in section 4.5 and conclude in section 4.6. This chapter is based on the paper, *All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications*, published in OSDI 14 [216].[1]

4.1 Background

We begin by describing how applications use *update protocols* to safely update on-disk state. We then define crash states, and describe how crash states complicate writing update protocols that are both correct and efficient.

[1] Work done in collaboration with other students in Wisconsin Madison. This chapter represents my contribution to the paper.

[Protocol (a)]

```
lock(application_lock);
write(user.db);
unlock(application_lock);
```

[Protocol (b)]

```
lock(application_lock);
write(journal);
write(user.db);
unlock(application_lock);
```

[Protocol (c)]

```
lock(application_lock);
write(journal);
fsync(journal);
write(user.db);
fsync(user.db);
unlock(application_lock);
```

Figure 4.1: **Application Update Protocols.** *The figure shows pseudo-code for three different update protocols for SQLite (Version 3.7.17). Protocol (a) corresponds to using compilation option `SQLITE_ENABLE_ATOMIC_WRITE`. Protocol (b) corresponds to using option `PRAGMA database.synchronous = OFF`. Protocol (c) corresponds to using option `PRAGMA database.synchronous = FULL`. The first protocol is vulnerable to either a process or system crash. The second protocol is correct in the face of a process crash, but vulnerable in the event of a system crash. The last protocol is correct in the face of both process and system crashes.*

4.1.1 Application Update Protocols

Each application persists its state using a sequence of system calls. We term this sequence the *update protocol*. The update protocol determines whether the application is consistent across crashes and thus is critical for over-all application correctness.

Figure 4.1 shows three application update protocols. These protocols represent simplified versions of update protocols used by SQLite in different configurations. The protocols use different mechanisms to update state on storage, and differ in how vulnerable they are to crashes.

Protocol (a) updates the database file using a single write. This protocol is vulnerable to either a process or system crash: if the `write()` call is interrupted, the database could be partially updated (e.g., only the first 4096 bytes) leading to corruption.

Protocol (b) first writes to an application-level journal, and then to the actual database. If a process crash interrupts the journal write, the database in unaffected; if the process crashes in the middle of database write, SQLite can use the journal to restore the database to a consistent state. However, the protocol is not correct in the event of an inopportune system crash. Upon completion of the `write()` call, the data is stored in operating system buffers, and can be written to disk in any order. Consider that the operating system chooses to write out the database data first, and the system crashes in the middle of the write. Since the database is partially updated, it is in an inconsistent state. Furthermore, since the journal data was not written to disk, the journal cannot be used to restore the database to a consistent state.

Protocol (c) solves this problem by using `fsync()` to ensure that the journal is persisted *before* writing to the database file. A system crash can only lead to one of two states. In the first case the journal was not completely persisted. In this case the update failed and the database remains in the old, consistent state. In the second case the journal was completely persisted, but the database may not have been completely updated. The journal is read, and the database updated to the new consistent state. In either case, the database remains consistent. Thus the protocol is not vulnerable to either process or system crashes.

4.1.2 Crash States

We define the *crash state* as the on-disk state after a crash. It is comprised of the directory hierarchy and the data of all the files in the hierarchy. After a crash, an application uses the crash state to recover to a consistent state.

The consistency of an application update protocol depends upon the developer correctly handling all possible crash states, which is challenging for two reasons. First, there are a large numbers of possible crash states for even simple protocols. Second, the crash states that are possible vary from file system to file system.

Ideally, it should be possible to determine which crash states are possible by examining the system calls in the update protocol. Unfortunately, the POSIX standard defines how each system call should modify *in-memory* state; it does not specify what happens to the *on-disk* state in the event of a crash. Each file system *implements* system calls differently, leading to different behavior in the event of a crash.

4.1.3 Protocol Goals

Ideally, application update protocols should satisfy two goals: they must be *correct*, i.e., maintain application consistency in the face of a process or system crash at any point; and they must have *good performance*.

Naturally, there is a tension between these two requirements. For example, protocol (a) has the best performance of the three protocols in Figure 4.1; however, it is vulnerable to process and system crashes.

Crash-consistent update protocols are hard to write because programmers normally reason about *in-memory* application state. For an update protocol to be correct in the face of process and system crashes, the developer needs to reason about all the different possible *crash* states.

4.2 Motivation

To motivate persistence properties, we analyze the update protocol used by a simple application to update its state in a crash consistent manner. The protocol, shown in Listing 4.1, consists of only 4 operations. Since there

Listing 4.1: **Example Update Protocol.** *The listing shows part of the update protocol of HSQLDB.*

```
# File overwrite. Previously contains "old"
write(log, "new")

# New file create
creat(newfile)

# Append to the new file
write(newfile, "data")

# File rename
rename("newfile", "existing_file")
```

is no `fsync()` in the protocol, the updates are stored in the buffer cache when the protocol completes, and written to storage in the background.

Even this simple protocol can lead to many different on-disk states after a crash. Some of these crash states are listed in Table 4.2. State #0 is the initial state. States #1, #2, and #3 occur as the protocol proceeds in order. These states occur in all file systems.

States #4 and #5 can occur in the ordered and writeback mode of journaling file systems like ext3 and ext4. In these states, the overwritten data is not persisted before the crash, as these modes arbitrarily delay persisting overwrites that do not cause metadata changes.

States #6 and #7 can occur in file systems with delayed allocation (e.g., ext4, xfs, btrfs), where the append is persisted after the rename. State #8 represents the rename not being atomic – this can happen only in ext2. State #9, where `stmp` ends up with data from another file, can happen with file systems such as ext3, ext4, and reiserfs that have a writeback mode.

Thus, we have shown that the crash states possible for an update protocol vary widely by file system. We have ascertained that certain properties

#	log	newfile	`existing_file` points to:
0	`old`	ϕ	old contents
1	`new`	ϕ	old contents
2	`new`	`data`	old contents
3	`new`	`data`	new contents
4	`old`	`data`	old contents
5	`old`	`data`	new contents
6	`old`	ϕ	empty file
7	`new`	ϕ	empty file
8	`old`	ϕ	Missing!
9	`old`	`your_password`	data from `/etc/passwd`

Table 4.2: **Possible Crash States for Example Protocol.** *The table shows different crash states for the update protocol in Listing 4.1. The second and third columns show the contents of the files* `log` *and* `newfile`*. The last column represents what the directory entry* `existing_file` *is pointing to.*

of a system call's implementation (and how system calls interact with one another) determine the possible on-disk crash states.

Although application-level consistency depends upon how system calls are persisted, there are *no* standards that talk about how file systems should persist them. Indeed, before this work, there was no term to refer to the particular aspects of a file system that affected application crash recovery. We believe that defining and studying persistence properties is the first step towards standardizing them across file systems.

There are several ways of changing a file-system implementation that leads to weakening the same persistence properties. When thinking about the details of the implementation, it may be hard to see that this change weakens property X, which results in applications A, B and C breaking. For example, XFS introduced delayed allocation in 2004, breaking many applications [290]. XFS introduced code fixes in 2006 that supported applications who used `truncate()` for atomic updates [165]. Ext4 introduced delayed allocation in 2008, resulting in significant data loss [146]. Later in 2009, ext4 introduced changes similar to the ones XFS had introduced three years earlier [282]. By abstracting away implementation details, we hope that persistence properties allow developers of different file systems to communicate about the effect of changes to persistence properties.

4.3 Persistence Properties

Each file system implements system calls in a different manner. The crash consistency of applications is affected by some aspects of the implementation of each system call. Modelling the implementation as *persistence* properties allows us to abstract away the aspects of the implementation that do not affect application-level consistency, while capturing all the relevant aspects.

In this section, we first present the intuition behind persistence properties. We then formally define persistence properties and provide a few examples.

4.3.1 Intuition

Application-level consistency is affected by the on-disk state after a crash. Hence, aspects of the implementation that affect the on-disk state should be captured. For example, persistence properties should capture whether the changes made via a system call are all atomic, or whether a file is visible to applications after a crash. On the other hand, implementation aspects that relate to *how* on-disk changes are made are irrelevant: it is enough to know that a system call is atomic; it does not matter whether the atomicity is provided through journaling, copy-on-write, or some other mechanism. Two file-system implementations that have the same persistence properties are equivalent through the lens of application-level consistency, even if they differ in other aspects.

The on-disk state is modified by a stream of I/O resulting from system calls: hence it is enough for persistence properties to capture *what I/O becomes persistent on disk*.

4.3.2 Definition

A persistence property is an assertion about how I/O resulting from certain system calls is persisted. It has three components:

1. The group of system calls it operates on. For example, all I/O resulting from operations such as `rename()` might be persisted in a certain way in a file system.
2. The characteristic of I/O it refers to. The characteristic could either be atomicity or ordering. For example, `rename()` operations are persisted atomically in the given file system.
3. The arguments to the system call. For example, `rename()` might only be atomic if both source and target files are in the same directory.

We first describe the groups of system calls that we used when defining persistence properties. We then use an example to illustrate the atomicity and ordering characteristics. Finally, we provide examples of persistence properties.

System-Call Groups

For the sake of convenience, we broadly group system calls into three: *file* operations, *directory* operations, and *sync* operations.

File operations are concerned with only a single file and do not affect directories. The `write()` and `truncate()` system calls fall into this category. Writes to an `mmap()`-ed file are also considered as file operations.

Directory operations are only concerned with the directory hierarchy. They do not cause file data to be written. The following system calls

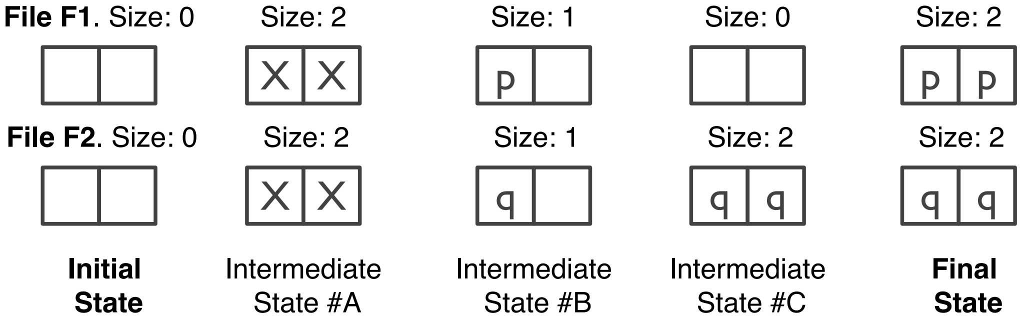


Figure 4.3: **Illustrating Atomicity and Ordering Properties.** *The figure shows the initial, final, and some of the intermediate crash states possible for the workload described in Section 4.3.2. X represents garbage data in the files. Intermediate states #A and #B represent different kinds of atomicity violations, while intermediate state #C represents an ordering violation.*

are directory operations: `link()`, `unlink()`, `creat()`, `mkdir()`, `rmdir()`, `mknod()`, `symlink()`, and `rename()`.

Sync operations persist the contents of a file or a directory. `fsync()`, `fdatasync()`, `sync_file_range()`, and `msync()` are treated as sync operations. Writes to file opened with `O_SYNC` are also treated as if they were followed by `sync_file_range()` for the write.

Illustrating Atomicity and Ordering

We consider the following pseudo-code snippet:

```
write(f1, "pp");
write(f2, "qq");
```

In this example, the application first appends the string `pp` to file descriptor `f1` and then appends the string `qq` to file descriptor `f2`. Note that we will sometimes refer to such a `write()` as an `append()` for simplicity. Figure 4.3 shows a few possible crash states that can result.

Atomicity. If the append is not *atomic*, for example, it would be possible for the *size* of the file to be updated without the new data reflected to disk;

in this case, the files could contain garbage, as shown in State A in the diagram. We refer to this as *size-atomicity*. A lack of atomicity could also be realized with only part of a write reaching disk, as shown in State B. We refer to this as *content-atomicity*.

Ordering. If the file system persists the calls out of order, another outcome is possible (State C). In this case, the second write reaches the disk first, and as a result only the second file is updated. Various combinations of these states are also possible.

Examples of Persistence Properties

Multi-block file writes are atomic. This atomicity persistence property indicates whether file operations such as `write()` that operate on more than a single block of data are atomic. This property has a general scope: a similar property with smaller scope would be *single sector over-writes are atomic*. Note that the second property has smaller scope for two reasons: it operates on a smaller set of operations (overwrites vs. all file operations) and only on single sector writes versus all writes larger than a block.

Directory operations are ordered. This ordering persistence property indicates whether all directory operations are persisted in program order in the file system. A similar property of narrow scope would be *directory operations on the same directory are ordered*.

4.4 Block Order Breaker

To study persistence properties, we built a tool named the *Block Order Breaker* (Bob). We first describe the goals of Bob (§4.4.1). We then explain our approach to studying persistence properties (§4.4.2). We describe how Bob tests file-system persistence properties. Finally, we discuss the limitations of Bob.

4.4.1 Goals

Having defined persistence properties (§4.3), we sought to answer three questions:

1. Which persistence properties are upheld by widely used file systems?
2. How do persistence properties vary among different file systems?
3. How do persistence properties vary among different configurations of the same file system?

The first question allows developers of applications targeted at specific file systems to know what properties are provided by each file system. The second question allows developers of portable applications (meant for multiple file systems) to infer whether their application will break when it is moved from one file system to another. The third question allows developers to know how sensitive their application is to the exact configuration of the underlying file system.

4.4.2 Approach

Unfortunately, it is extremely hard to identify the properties that each file system provides. One source of such knowledge could be the developers of the file system. Unfortunately, due to the complexity of modern file systems, and multiple developers working on different parts of the file system, it is hard to for developers to state with any confidence the properties provided by their system [50].

Another source of knowledge about persistence properties could be file-system documentation [135]. However, file-system documentation is typically extremely high-level, and does not provide details such as persistence properties provided. Moreover, the documentation may not be in sync with the implementation.

Persistence properties provided by a file system could be inferred from the source code itself. Although there have been a number of attempts at extracting the high-level architecture of software systems in a semi-automatic manner [31, 76], extracting lower-level details like persistence properties still remains a hard problem.

In this dissertation, as a first step towards studying persistence properties, we tackle a slightly different question: which persistence properties are *not* provided by a given file system? Unlike establishing the persistence property supported by a file system, finding properties that fail is significantly easier (as a single counter test-case is sufficient). Knowing which persistence properties are not supported helps us compare various file systems and different configurations of the same file system.

Identifying persistence properties that are not supported will only be useful if the properties that are examined are chosen carefully. To this end, we have used our knowledge of application-level crash consistency [275] to investigate persistence properties that are commonly assumed to be provided by many file systems [155].

4.4.3 Testing Persistence Properties with Bob

We first provide an overview of how Bob works, and describe how persistence properties are tested in detail. For the sake of convenience, we refer to the storage state as the "disk state". Bob is not specific to disks in any manner, and will work on any storage device.

Overview. Bob first runs a simple user-supplied workload designed to stress the persistence property tested (e.g., a number of writes of a specific size to test overwrite atomicity). Bob collects the block I/O generated by the workload, and then re-orders the collected blocks, selectively writing some of them to disk to generate a new *legal* disk state (disk barriers are obeyed). In this manner, Bob generates a number of unique disk images corresponding to possible on-disk states after a system crash. Bob then runs file-system recovery on each resulting disk image, and checks whether various persistence properties hold (e.g., if writes were atomic). If Bob finds even a single disk image where the checker fails, then we know that the property does not hold on the file system.

Test Workloads. Bob uses a different workload to test each persistence property. Figure 4.4 shows a sample of the code used in Bob. First, the disk state is initialized with a number of files containing well-known data. The workload then does a number of metadata operations, mixing synchronous and asynchronous operations. The block traffic generated by the file system (due to the workload) is captured. A number of crash states are generated, and specific invariants are checked (*e.g.*, is a later metadata operation persisted before an earlier operation?). If such a crash state is found, Bob reports that the file system does not support the specific persistence property being tested.

Generating Crash States. A crash state is generated by taking the initial disk state, and applying a subset of writes from the collected block trace. Given the block trace generated by the file system, Bob first looks for flush requests (*e.g.*, requests tagged with the `REQ_FLUSH` flag). It divides the trace into different *phases* delineated by flush requests. Within a phase, it is legal to re-order write requests: for example, if a phase contains writes A, B, and C (in that order), it is legal to generate a crash state with writes A and C. We call this "re-ordering" since B preceded C in program order, but C was persisted first. Bob does not re-order writes across flush requests or barriers.

Note that given a block trace, a large number of legal crash states can be generated. With Bob, we do not aim to exhaustively test all possible crash states. In preliminary experiments, we had observed that simple re-orderings lead to significant changes in observed disk state [275]. Thus, we generated crash states corresponding to the following simple re-orderings:

[Initialization]

```
def init_state():
   print("Initiating directory")
   disk_state = {}
   for fname in ["01","02","03","04","05",\
                 "11","12","13","21","22","23"]:
        disk_state = create_file( \
               fname, 6, "a", disk_state, False)
   # Store the disk state on file.
   pickle_file = open("init_pickle", 'wb+')
   pickle.dump(disk_state, pickle_file)
```

[Sample Workload]

```
UNLINK("01")
RENAME("02")
UNLINK("03")
RENAME("04")
UNLINK("05")

CREAT("0-end", True)

RENAME("21")
CREAT("2a", True)
RENAME("22")
CREAT("2b", True)
RENAME("23")

CREAT("1-end", True)
```

Figure 4.4: **BOB Workloads.** *The figure shows some of the code used in BOB Workloads. The first snippet shows that the directory is initialized with a number of files. The sample workloads does a number of metadata operations, both in synchronous (the True argument to CREAT) and asynchronous fashion. BOB would then generate different crash states and check if any of them contains re-ordered operations. For example, if the rename of file 02 is present but the unlink of 01 is not.*

1. Crash states with a prefix of the block trace applied. Note that the trace consists of write requests, and each request may consist of multiple blocks. We generated crash states with both a prefix of

write requests applied, and a prefix of blocks applied.

2. Crash states with a single legal write request applied.
3. Crash states with all but a single write request applied.

As our results show, even these simple crash states were sufficient to show that several persistence properties are supported by widely-used file systems.

Implementation. Bob uses Blktrace [34] to collect the block-level trace. It uses SystemTap [78] to interpose on the `blk_add_trace_rq_issue()` function call and collect block data. The utility `dd` is used to copy disk states and create crash states. The workloads and the verification checks are written in Python.

4.4.4 Limitations

Bob is meant to be a first step in the exploration of persistence properties. We do not intend to use Bob to identify all the persistence properties that fail on a given file system. Bob does not generate all the crash states given a block trace – a given persistence property could fail on the tested file system, but Bob may not identify this. Although Bob is sound (it does not falsely report a property as being violated), it is not complete.

4.5 Study of Persistence Properties

We study the persistence properties of six Linux file systems: ext2, ext3, ext4, btrfs, xfs, and reiserfs. A large number of applications have been written targeting these file systems. Many of these file systems also provide multiple configurations that make different trade-offs between performance and consistency: for instance, the data journaling mode of ext3

File System	Version	Mount Options
ext2	d211858837	None
ext2	d211858837	sync
ext3	d211858837	data=writeback
ext3	d211858837	data=ordered
ext3	d211858837	data=journal
ext4	5a0dc7365c	data=writeback
ext4	5a0dc7365c	data=ordered
ext4	5a0dc7365c	nodelalloc
ext4	5a0dc7365c	data=journal
btrfs	827fa4c762	None
xfs	be4f1ac828	wsync
reiserfs	bfe8684869	nolog
reiserfs	bfe8684869	data=writeback
reiserfs	bfe8684869	data=ordered
reiserfs	bfe8684869	data=journal

Table 4.5: **File-System Configurations.** *The table shows the file-system configurations tested using the Block-Order Breaker. All file-system versions correspond to the versions released with Linux 3.2. The Git commit number of the latest patch applied to each file system is also provided to identify the file-system version. All mount options other than those explicitly mentioned are set to their default values.*

provides the highest level of consistency, but often results in poor performance [220]. Between file systems and their various configurations, it is challenging to know or reason about which persistence properties are provided. Therefore, we examine different configurations of the file systems we study (a total of 16). Table 4.5 lists the file-system configurations that we examined.

Note that different system calls (e.g., writev(), write()) lead to the same file-system output. We group such calls together into a generic file-system update we term an *operation*. We have found that grouping all operations into three major categories is sufficient for our purposes here: file overwrite, file append, and directory operations (including rename, `link`, `unlink`, `mkdir`, etc.).

4.5.1 Persistence Properties Tested

Based on our experiments with investigating application-level crash consistency [216, 275], we select persist properties to test.

Atomicity. A number of applications (*e.g.*, PostgreSQL [279], LMDB [273], and ZooKeeper [16]) depend upon small, atomic, in-place writes to maintain crash consistency [219]. Hence, we first examine whether file systems provide this property (single sector overwrite and single block overwrite). Applications such as LevelDB [93] append to logs as part of their operations: in certain configurations, they assume the append will be atomic. Hence, we examine whether appends of various sizes are atomic on different file systems (Single sector append, single block append, multi-block appends). It is interesting to note that applications like HSQLDB [117], Mercurial [168], and LevelDB work correctly if a prefix of the append is persisted correctly. For example, if a crash leaves the appended portion filled with garbage or zeroes, the Mercurial repository is corrupted; no error is caused if the crash results in a prefix appended to the file [216]. Therefore, we test this persistence property as well (multi-block prefix append). Many applications (*e.g.*, GDBM [92], HSQLDB, LevelDB) depend on operations such as `rename()` being atomic; consequently, we include this property in our tests. Although applications do not directly use multi-block overwrite operations, it is a primitive that will be very useful [187]. Hence, we test whether file systems provide multi-block overwrites.

Ordering. Many applications use `rename()` or appends (with the `O_TRUNC` flag) to atomically update a files. For `rename()`, the sequence involving writing to a temporary file, then renaming it over the old file. The `O_TRUNC` flag involves a similar sequence. If the sequence is persisted out of order (*e.g.*, if the file is renamed without the new data being persisted), data could be lost; this has happened with XFS and ext4 in the past [146, 165]. Therefore, we test whether appends and renames are persisted in order on different file systems. Since file systems such as ext4, btrfs, and xfs have different heuristics about re-ordering operations [165], we test variations such as appends being followed by a `rename()` operation. For the sake of completeness, we test whether overwrite operations are re-ordered as well. POSIX does not guarantee that a new file is created until the parent directory is explicitly flushed [278]; since applications depend on this property [216], we included it in our tests.

4.5.2 Results

Table 4.6 lists the results of our study. The table shows, for each file system (and specific configuration) whether a particular persistence property has been found to *not* hold; such cases are marked with an ×. Note that the absence of an × does not mean that the property is provided by the tested file system; Bob simply may not have found a crash state where it fails.

We first describe how persistence properties are violated by file systems. For each property we tested, we describe the violation case found by Bob. We describe violations of atomicity and ordering persistence properties in turn. We then broadly discuss the results of our study.

Atomicity

For testing atomicity of different operations, we first store the initial disk state. We perform the operation, and capture the final disk state. We then test that for each crash state that Bob produces, the file system recovers to the initial state or the final state. For the multi-block prefix append test, there are a number of intermediate states that are also acceptable. Our initial disk state for these experiments consisted of ten empty files, a couple of directories, and four 1 GB files.

Persistence Property	File system															
	ext2	ext2-sync	ext3-writeback	ext3-ordered	ext3-datajournal	ext4-writeback	ext4-ordered	ext4-nodelalloc	ext4-datajournal	btrfs	xfs	xfs-wsync	reiserfs-nolog	reiserfs-writeback	reiserfs-ordered	reiserfs-datajournal
Atomicity																
Single sector overwrite																
Single sector append	×		×			×								×		
Single block overwrite	×	×	×	×		×	×	×			×	×	×	×	×	
Single block append	×		×			×							×	×		
Multi-block append/writes	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Multi-block prefix append	×		×			×							×	×		
Directory op	×	×											×			
Ordering																
Overwrite → Any op	×		×	×		×	×	×			×		×	×	×	
Append(file) → rename(file)	×		×			×					×		×	×		
`O_TRUNC` Append → close	×		×			×				×			×	×		
Append → Append (same file)	×		×			×							×	×		
Append → Dir op	×		×			×	×			×	×		×	×		
Dir op → Any op	×									×			×			

Table 4.6: **Persistence Properties.** *The table shows atomicity and ordering persistence properties that we empirically determined for different configurations of file systems. X → Y indicates that X is persisted before Y. [X,Y] → Z indicates that Y follows X in program order, and both become durable before Z. A × indicates that we have a reproducible test case where the property fails in that file system.*

Single sector overwrite. In this experiment, we overwrote a sector at an aligned position (*i.e.*, the 512 bytes do not cross disk sector boundaries) picked at random in a 1 GB file. We observe that all tested file systems seemingly provide atomic single-sector overwrites: in some cases (*e.g.*, ordered mode of ext3), this property arises because the underlying disk provides atomic sector writes. Note that if such file systems are run on top of new technologies (such as PCM) that provide only byte-level atomicity [56], single-sector overwrites will not be atomic.

Singe sector append. In this experiment, we appended 512 bytes of data to a file containing 40960 bytes (ten 4K blocks). On all the file systems we tested, allocating a new sector involves allocating a new 4K data block. Thus, there are at least two structures that must be updated on storage: the new data block, and the file inode. In this experiment, even though an entire block is being allocated, we only care about the first sector; the data in the other sectors in the block do not affect this persistence property.

Since ext2 relies on `fsck` [175] for recovery after a crash, it does not constrain the order of writes at run times. Hence, Bob finds a crash state where the inode is updated, but the data block is not. Since ext2 only provides metadata consistency (§2.6.2), recovery leaves the data block attached to the inode (potentially causing security problems if the block was previously part of a sensitive file such as `/etc/passwd`).

In the writeback modes of ext3, ext4, and reiserfs, metadata may be persisted before data blocks. Thus, Bob finds a crash state where the inode is persisted, while the data block is not.

Single block overwrite. In this experiment, we overwrote a block at an aligned position (*i.e.*, the 4096 bytes overwritten do not cross disk block boundaries) picked at random in a 1 GB file. Most disks have an atomic write unit of 512 bytes. Thus, writing a single 4K block requires writing eight sectors. The system may crash in the middle of writing these eight sectors; special machinery is required to guarantee the atomic write of even a single block. In the data journaling mode of ext3, ext4, and reiserfs, the overwritten data block is first written into the journal; thus, a crash while writing into the journal does not affect the atomicity of the overwrite. Similarly, btrfs writes a new copy of the data block and atomically switches pointers to the new block, thus providing an atomic block overwrite. For all other file-system configurations tested, Bob finds a crash state where some of the sectors (but not all) in the block were updated, violating the persistence property. Note that synchronously writing the sectors (as in ext2's `sync` mode) does not help.

Single block append. Similar to the single sector append experiment, we appended 4096 bytes of data to a file containing 40960 bytes (10 4K blocks). We tested only with the simple case of a new pointer being added to the inode; we did not test with cases where indirect blocks were allocated for the file. We believe that the relevant persistence behavior will be exposed even for the simple append case. The major difference of this test with the single-block overwrite is that partially writing a block is not a problem if the inode does not point to the block yet. In other words, as long as the inode write is ordered after the data block write, the fact that the block is written one sector at a time does not matter.

In the ordered mode of ext3, ext4, and reiserfs, the `nodelalloc` mode of ext4, and the `sync` mode of ext2, the data writes are ordered before metadata writes. Thus, the property is not violated in these configurations. In the `nolog` mode of reiserfs, journaling is turned off. In this configuration, and in writeback journaling modes of ext3, ext4, and reiserfs, and in ext2's default configuration, this persistence property is violated. Bob finds a crash state where some of the sectors (but not all) in the new data block were updated, violating the persistence property.

Multi-block append or writes. In this experiment, we overwrote/appended 50 MB to a 1 GB file. We chose 50 MB since this is above the maximum size of a transaction in the default configuration of ext4. None of the file-system configurations tested provided multi-block atomic appends or writes. Interfaces to request such atomic operations are currently not available to applications. In some cases, in the data journaling configurations, the data and metadata would be part of the same transaction, and thus the write/append would be atomic; however, this is not guaranteed, and Bob finds cases where the property is violated.

Multi-block prefix append. Similar to the multi-block append experiment, we appended 50 MB to a 1 GB file. We define a prefix append in the following manner: a crash results in the file containing a consistent prefix (*e.g.*, the first 2 blocks) of the append data. A prefix append does not result in the user reading garbage data at the tail of the appended data. File systems can provide this property by first persisting a prefix of the append data to storage, updating metadata to point to the newly persisted data, and then continuing with the rest of the append operation. File systems in most journaling modes (other than writeback) and copy-on-write file systems support this property. Thus, it is only violated in orderless file systems like ext2 (and reiserfs in `nolog` mode) and in the writeback modes of ext3, ext4, and reiserfs.

Directory operations. Bob tested that directory operations `creat`, `link`, `unlink`, and `rename` were atomic. `rename` is an interesting case because it involves changes to both the source and destination directories. In ext2 (default mode and in `sync` mode), Bob finds crash states where the file is missing in both directories. A similar crash state is found when journaling is turned off in reiserfs (`nolog` mode). In all other tested file-system configurations, the directory operations were found to be atomic.

Ordering

We tested whether operations were re-ordered in the following manner. We captured the initial state of the disk. After each operation was performed, we emulated a new disk state where that operation was persisted onto the previous disk state. Thus, after each operation, our list of acceptable disk states grew by one. We consider partial persistence of an operation equivalent to non-persistence of that operation. For each crash state created by Bob we check whether it is present in our list of acceptable crash states; otherwise we signal that the persistence property is violated. We used the same initial disk state as the atomicity tests.

Overwrite → any operation. We tested whether overwrites are re-ordered after other operations by first overwriting a block at an aligned position picked at random in a 1 GB file, and then performing data operations such as appends (to a different file) and directory operations (`creat`, `link`, `rename`, and `unlink`). Data overwrites are special operations because they don't necessarily cause metadata changes (unless options such as `atime`, `mtime` are enabled). Dirty data blocks are treated differently from metadata in most file-system configurations. The orderless configurations (ext2, reiserfs `nolog`) will persist the second operation before the overwrite. Bob finds crash states where both metadata and data operations are persisted before the overwrite in these file systems. Since the writeback configuration of journaling file systems does not order data writes before metadata operations, Bob finds crash states where a later metadata operation was persisted, but the over write was not. In the ordered mode (and the similar `nodelalloc`), data is persisted before metadata; however, this is only true for data blocks connected to the later metadata options. Overwritten data blocks are not connected to any other metadata blocks, and hence they are not persisted first. Thus, this persistence property is violated in writeback and ordered modes of ext3, ext4, reiserfs, and xfs.

Append(file) → rename(file). In this experiment, we first appended a 4K block to the one of the empty files in our initial disk state, and then renamed the file. Many applications use the technique of appending to a temporary file, and then renaming it over the old file to atomically update a file [146, 216]. On file systems with delayed allocation (such as the ordered mode of ext4), the appended data may be persisted after the rename, leading to an empty renamed file in the event of an inopportune crash. Therefore, some file systems recognize when the application is using this technique, and treat the `rename` similar to `fsync()`: both the append and the `rename` are persisted before any other operation [59]. Among file systems with delayed allocation, xfs is the only file system that does not treat this case specially [165]. Thus, in the case of ext2, the writeback configurations of ext3, ext4, and reiserfs, the `nolog` mode of reiserfs, and xfs, Bob finds crash states where the `rename` is persisted, but the appended data is not.

`O_TRUNC` Append → close. In this experiment, we open one of the 1 GB files in the initial disk state with the `O_TRUNC` flag, write ten 4K blocks to the file, and then close the file. Appending to a file opened with the `O_TRUNC` flag is another mechanism used by applications to atomically update a file. Similar to the append-rename case, most file-systems recognize this technique and handle it correctly by forcing the appended data to storage before the file is closed. In contrast to the append-rename case, xfs handles this correctly, but btrfs does not. The other violations are exactly the same as the append-rename case. In the case of violations, Bob finds crash states with empty files and later metadata operations persisted.

Append → Append (same file). In this experiment, we append ten 4K blocks (in separate operations) to one of the empty files in the initial disk state. When two appends to the same file are not ordered explicitly by `fsync()` calls, most file systems still order them internally. We believe this is because the same file metadata is being affected in both operations. Even on file systems with delayed allocations, appends were persisted in the correct order. Thus, only order-less configurations (ext2 and `nolog` reiserfs) and writeback configurations of ext3, ext4, and reiserfs violate this property. In case of violation, Bob finds a crash state with the tail of the append persisted, but some earlier part containing random data.

Append → Directory Operation. In this experiment, we first append a single 4K block to an empty file in the initial disk state, and follow it up with a mix of directory operations such as `link`, `mkdir`, and synchronous file creations. On file systems with delayed allocation, the append is persisted after other metadata operations. For example, if the append is followed by the synchronous creation of another file, Bob will find crash states with the new file created, but the appended data not persisted. This persistence property is violated on orderless configurations (ext2 and `nolog` reiserfs), writeback configurations, and journaling configurations where delayed allocation is active (*e.g.*, ext4 ordered mode). In addition, since btrfs persists metadata operations on the root directory before other operations, a `rename` operation on the root is persisted before the append in a crash state.

Directory operation → any operation. In this experiment, we first perform directory operations to a set of files in our initial disk state. We then perform overwrites and appends to a 1 GB file. Finally, we once again perform directory operations to a different set of files (in different directories). Directory operations are usually handled with care by most file systems, and persisted in order. Orderless file-system configurations (ext2 default mode and reiserfs `nolog` mode) re-order directory operations as well. Among modern file systems, btrfs re-orders directory operations to increase performance [50]. Bob found crash states where a `rename()` involving the root directory were persisted before an earlier metadata operation on a child of the root.

4.5.3 Discussion

From Table 4.6, we observe that persistence properties vary *widely* among file systems, and even among different configurations of the same file system. The order of persistence of system calls depends upon small details like whether the calls are to the same file or whether the file was renamed. From the viewpoint of an application developer, it is risky to assume that any particular property will be supported by all file systems.

It might seem strange that persistence properties vary so much between different file-system configurations. The primary motivation for file-system developers to design configurations with different persistence properties is performance. Providing strong persistence properties is costly in terms of performance. For example, a file system where all data and metadata writes are totally ordered is severely constrained in exploiting parallelism provided by the storage device, or allowing the device to come up with optimal I/O schedules. Hence, developers weaken persistence properties to increase performance.

There is a natural tension between providing strong persistence properties, and providing high performance. Strong persistence properties allow application developers to reason easily about crash states and application-level crash consistency. On the other hand, weakening persistence properties allow the file system to provide high performance to a range of different applications.

This tension is further compounded by the lack of communication between file-system developers and application developers. Both parties believe that their stance is correct, and want the burden of changing the software (either file systems or applications) to fall on the other group. Historically, it has been the file-system developers who have changed the file system to accommodate application developers [59]. There are hundreds of thousands of applications, and tens of file systems; hence, it makes sense that the file-system code change when there is a mismatch of expectations. Furthermore, Linus Torvalds, who functions as the "benevolent dictator" of the Linux Kernel development, strongly believes that file systems should "just do the right thing" [59].

4.6 Conclusion

In this chapter, we show how application-level consistency is dangerously dependent upon file system *persistence properties*, i.e., how file systems persist system calls. We develop Bob, a tool to test persistence properties among different file systems. We use Bob to study a total of sixteen configurations of six widely-used file systems, and show that such properties vary widely.

Our results show that applications should not be written assuming the underlying file system will have specific persistence properties. Unfortunately, this is often the case; applications are developed and tested mainly on a single file system. We have taken the first steps towards solving this problem: we built Alice, a framework that pin-points application code that would work incorrectly on different file systems [216]. Others are working towards building tools that will allow applications to be crash-consistent on any given file system [301].

5 The No-Order File System

In Chapter 3, we described how disk-cache flushes are used to order writes, and the high performance cost of flushing the disk cache. In this chapter, we attempt to answer the question: *can a file-system maintain crash consistency without ordering writes?* We present a new crash-consistency protocol, *Backpointer-Based Consistency*, that does not require any ordering among writes to storage. Backpointer-Based Consistency embeds a *backpointer* into each file-system object, and builds consistency by mutual agreement between objects. We demonstrate the power of Backpointer-Based Consistency by designing and implementing the *No-Order File System*. This chapter is based on the paper, *Consistency Without Ordering* [47], published in FAST 12.

In the rest of this chapter, we first describe the goals of NoFS and its assumptions about the storage device (§5.1). We then present the high-level design of NoFS (§5.2), and discuss its implementation (§5.3). We present an evaluation of NoFS (§5.4), and describe its limitations, uses cases, and implementation challenges (§5.5). Finally, we present a formal proof that NoFS provides data consistency (§5.6), and conclude (§5.7).

5.1 Goals and Assumptions

We motivate why we designed the No-Order File System (NoFS). We then describe the goals of NoFs, and our assumptions about the storage device.

The problem with ordering points. Current file systems use a number of techniques to ensure that the file system is consistent in the face of a crash. Almost all of these techniques require carefully ordering writes to storage. We term as an ordering point every point in the file-system update protocol where write X must be persisted before write Y.

In the event of a crash, ordering points allow the file system to reason about which writes reached the disk and which did not, enabling the file system to take corrective measures, such as replaying the writes, to recover. Unfortunately, ordering points are not without their own set of problems:

1. By their very nature, ordering points introduce waiting into the file-system code, thus potentially lowering performance. They constrain the scheduling of disk writes, both at the operating system level and at the disk driver level.

2. They introduce complexity into the file-system code, which leads to bugs and lower reliability [220, 221, 311, 312].

3. The use of ordering points also forces file systems to ignore the end-to-end argument [237], as the support of lower-level systems and disk firmware is required to implement imperatives such as the disk cache flush. When such imperatives are not properly implemented [247], file-system consistency is compromised [226]. In today's cloud computing environment [19], the operating system runs on top of a tall stack of virtual devices, and only one of them needs to neglect to enforce write ordering [294] for file-system consistency to fail.

Although file systems that use the file-system check [175] for crash consistency do not need to use ordering points, they lead to another problem: unavailability. On today's high-capacity disks, a crash results in the file-system being un-available for hours or days while the file-system check is running. Businesses cannot afford to lose access to data for such extended periods of time. Furthermore, file-system checkers are often complex, and often contain bugs leading to corruption and data loss [102, 311, 312].

Goals. Our goals for the No-Order File System follow naturally from the problems outline above:

1. Remove *all* ordering points in the file-system protocol. Disk flushes or barriers should not be issued to maintain file-system crash consistency. Thus, the file system does not depend on lower-level storage correctly implementing the disk-cache flush [226].

2. Provide strong crash-consistency guarantees, equivalent to ext3's ordered journaling mode. Specifically, we seek to provide data consistency (§2.6.2).

3. Provide access to files immediately upon mounting. A long file-system check should not make the file-system unavailable.

4. Create a crash-consistency protocol that is simple, understandable, and hopefully easy to implement. Our hope was that this would reduce bugs in the implementation.

Assumptions. The key assumption on which the consistency of NoFS rests is that the storage device allows the file system to write an object and its 8-byte backpointer together in an atomic fashion. Current SCSI drives allow a 520-byte atomic write to enable checksums along with each 512-byte sector [274]; we envision that future drives with 4-KB blocks will provide similar functionality.

We also assume that the file system has a fixed data layout that is well-known. For example, when the file system is mounted, inodes should be located at a well-known location. For copy-on-write file systems such as WAFL [114] or btrfs [166], this assumption does not hold, and other mechanisms to find the location of inodes and other metadata is required.

5.2 Design

We present the design of the *No-Order file system (NoFS)*, a lightweight, consistent file system with no ordering points in its update protocol. NoFS provides access to files immediately upon mounting, with no need for a file-system check or journal recovery.

In this section, we introduce *backpointer-based consistency (BBC)*, the technique used in NoFS for maintaining consistency. We use a logical framework to prove that BBC provides data consistency in NoFS. We discuss how BBC can be used to detect and recover from inconsistencies, and elaborate on why allocation structures are not persisted to non-volatile storage in NoFS.

5.2.1 Overview

The main challenge in NoFS is maintaining file-system crash consistency without ordering points. Crash consistency is closely tied to logical identity in file systems. Inconsistencies arise due to confusion about an object's identity; for example, after a crash, two files may each claim to own a data block. If the block's true owner is known, such inconsistencies could be resolved. Associating each object with its logical identity is the crux of the backpointer-based consistency technique.

Employing backpointer-based consistency allows NoFS to detect inconsistencies on-the-fly, upon user access to corrupt files and directories. The presence of a corrupt file does not affect access to other files in any way. This property enables immediate access to files upon mounting, avoiding the downtime of a file-system check or journal recovery. A read is guaranteed to never return garbage data, though stale data (data that belonged to the file at some point in the past) may be returned.

We intentionally avoided using complex rules and dependencies in NoFS. We simplified the update protocols, not persisting allocation structures to disk. We maintain in-memory versions of allocation structures and discover data and metadata allocation information in the background while the file system is running.

5.2.2 Backpointer-based consistency

Backpointer-based consistency is built around the logical identity of file-system objects. The logical identity of a data block is the file it belongs to, along with its position inside the file. The logical identity of a file is the list of directories that it is linked to. This information is embedded inside each object in the form of a *backpointer*. Upon examining the backpointer of an object, the parent file or directory can be determined instantly. Blocks have only one owner, while files are allowed to have multiple parents. Figure 5.1 illustrates how backpointers link file-system objects in NoFS. As each object in the file system is examined, a consistent view of the file-system state can be incrementally built up.

Although conceptually simple, backpointers allow detection of a wide range of inconsistencies. Consider a block that is deleted from a file, and then assigned to another file and overwritten. If a crash happens at any point during these operations, some subset of the data structures on disk may not be updated, and both files may contain pointers to the block. However, by examining the backpointer of the block, the true owner of the block can be identified.

In designing NoFS, we assume that the write of a block along with its backpointer is atomic (§5.1). This assumption is key to our design, as we infer the owner of the data block by examining the backpointer. Backpointers are similar to checksums in that they verify that the block pointed to by the inode actually belongs to the inode. However, a checksum does not identify the owner of a data block; it can only confirm that the correct block is being pointed to. Consistency and recovery require identification of the owner.

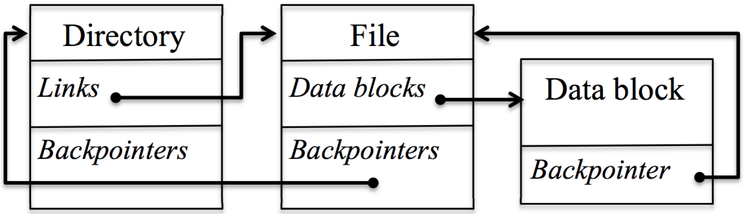


Figure 5.1: **Backpointers.** *The figure shows a conceptual view of the backpointers present in NoFS. The file has a backpointer to the directory that it belongs to. The data block has a backpointer to the file it belong to. Files and directories have many backpointers while data blocks have a single backpointer.*

Intuition

We briefly provide some intuition about the correctness of using the backpointer-based consistency technique to ensure data consistency. We first consider what *data consistency* and *version consistency* mean (§2.6.2), and the file-system structures required to ensure each level of consistency.

Data consistency provides the guarantee that all the data accessed by a file belongs to that file; it may not be garbage data or belong to another file. This guarantee is obtained when a backpointer is added to a data block. Consider a file pointing to a data block. Upon reading the data block, the backpointer is examined. If the backpointer matches the file, then the data block must have belonged to the file, since the backpointer and the data inside the block were written together. If the data block was reallocated to another file and written, it would be reflected in the backpointer. Hence, no ordering is required between writes to data and metadata since the data block's backpointer would disagree in the event of a crash. Note that the data block could have belonged to the file at some point in the past; the backpointer does not provide any information about when the data block belonged to the file. Thus, the file might be pointing to an old version of the data block, which is allowed under data consistency.

Version consistency is a stricter form of data consistency which requires that in addition to belonging to the correct file, all accessed data must be the correct version. Stale data is not allowed in this model. Backpointers are not sufficient to enforce version consistency, as they contain no information about the version of a data block. Hence more information needs to be added to the file system. Specifically, every object and forward pointer in the file system needs to include a time stamp. For example, each data block has a timestamp indicating when it was last updated. This timestamp is also stored in the inode containing the data block. When a block is accessed, the timestamp in the inode and data block must match. Since timestamps are a way to track versions, the versions in the inode and data block can be verified to be the same, thereby providing version consistency.

We decided against including timestamps in NoFS backpointers because updating timestamps in backpointers and metadata reduces performance and induces a considerable amount of storage overhead. Timestamps need to be stored with every object and its parent. Every update to an object involves an update to the parent object, the parent's parent, and so on all the way up to the root. Furthermore, doing so works against our goal of keeping the file system simple and lightweight; hence, NoFS provides data consistency, but not version consistency.

Using Checksums. A natural question about the design is whether checksums can be used instead of backpointers to achieve the required properties. Backpointers are more powerful than checksums for the following reasons. A backpointer allows mutual verification; the data block and the file both can verify they have the correct pointer. Two checksums (at the data block and the file) would be needed to achieve the same effect – in the case of backpointers, we take advantage of the fact that forward pointers are already present in the system. Furthermore, while checksums allow verification, they do not provide identification; a backpointer in a data block allows us to identify who the parent file potentially is.

Detection and Recovery

In NoFS, detection of an inconsistency happens upon access to corrupt files or data. When a data or metadata block is accessed, the backpointer is checked to verify that the parent metadata block has the same information. If a file is not accessed, its backpointer is not checked, which is why the presence of corrupt files does not affect access to other files: checking is performed on-demand.

This checking happens both at the file level and the data block level. When a file is accessed, it is checked to see whether it has a backpointer to its parent directory. This check allows identification of deleted files where the directory did not get updated, and files which have not been properly updated on disk.

NoFS is able to recover from inconsistencies by treating the backpointer as the true source of information. When a directory and a file disagree on whether the file belongs to the directory or not, the backpointer in the file is examined. If the backpointer to the directory is not found, the file is deleted from the directory. Issues involving blocks belonging to files are similarly handled.

5.2.3 Non-persistent allocation structures

In an order-less file system, allocation structures like bitmaps cannot be trusted after a crash, as it is not known which updates were applied to the allocation structures on disk at the time of the crash. Any allocation structure will need to be verified before it can be used. In the case of global allocation structures, all of the data and metadata referenced by the structure will need to be examined to verify the allocation structure.

Due to these complexities, we have simplified the update protocols in NoFS, making the allocation structures non-persistent. The allocation structures are kept entirely in-memory. NoFS starts out with empty allocation structures and allocation information is discovered in the background, while the file system is online. NoFS can verify whether a block is in use by checking the file that it has a backpointer to; if the file refers to the data block, the data block is considered to be in use. Similarly, NoFS can verify whether a file exists or not by checking the directories in its backpointers. Thus NoFS can incrementally (and in the background) learn allocation information about files and blocks. NoFS starts with zero free data blocks; it learns of new free blocks as the scan proceeds in the background. Thus, although NoFS allows reading of files immediately upon mounting, it requires some time to find free data blocks before allowing writes. Optimizations such as maintaining a list of free blocks (that need not necessarily be correct) can be implemented at the cost of extra complexity; we have not done so in our implementation.

5.3 Implementation

We now present the implementation of NoFS. We first describe the operating system environment, and then discuss the implementation of the two main components of NoFS: backpointers and non-persistent allocation structures. We describe the backpointer operations that NoFS performs for each file-system operation.

5.3.1 Operating system environment

NoFS is implemented as a loadable kernel module inside Linux 2.6.27.55. We developed NoFS based on ext2 file-system code. Since NoFS involves changes to the file-system layout, we modified the `e2fsprogs` tools 1.41.14 [285] used for creating the file system.

Linux file systems cache user data in a unified page cache [62]. File reads (except direct I/O) are always satisfied from the page cache. If the page is not up-to-date at the time of read, the page is first filled with data

Action	Backpointer operations
Create	Write backlink into new inode
Read	Translate offset Verify block backpointer in data block
Write	Translate offset Verify block backpointer in data block
Append	Translate offset Write block backpointer into data block
Truncate	No backpointer operations
Delete	No backpointer operations
Link	Write backlink into inode
Unlink	Remove backlink from inode
mkdir	Write directory entry backpointer into directory block
rmdir	No backpointer operations

Table 5.2: **NoFS backpointer operations.** *The table lists the operations on backpointers caused by common file system operations. Note that all checks are done in memory.*

from the disk and then returned to the user. File writes cause pages to become dirty, and an I/O daemon called `pdflush` periodically flushes dirty pages to disk. Due to this tight integration between the page cache and the file system, NoFS involves modifications to the Linux page cache.

5.3.2 Backpointers

NoFS contains three types of backpointers. We describe each of them in turn, pointing out the objects they conceptually link, and how they are implemented in NoFS. Figure 5.3 illustrates how various objects are linked by different backpointers. Every file-system operation that involves the creation or access of a file, directory, or data block involves an operation on backpointers. These operations are listed in Table 5.2.

Block backpointers

Block backpointers are {*inode number, block offset*} pairs, embedded inside each data block in the file system. The first 8 bytes of every data block are reserved for the backpointer. Note that we need to embed the backpointer inside the data block since disks currently do not provide the ability to store extra data along with each 4K block atomically. The first 4 bytes denote the inode number of the file to which the data block belongs. The second 4 bytes represent the logical block offset of the data block within the file. Given this information, it is easy to check whether the file contains a pointer to the data block at the specified offset. Indirect blocks contain backpointers too, since they belong to a particular file. However, since the indirect block data is not logically part of a file, they are marked with a negative number for the offset.

Our implementation depends on the `read` and `write` system calls being used; data is modified as it is passed from the page cache to the user buffer and back during these calls. When these calls are by-passed (via `mmap`) or the page cache itself is by-passed (via direct I/O mode), verifying each access becomes challenging and expensive. We do not support `mmap` or direct I/O mode in NoFS.

Insertion: The data from a `write` system call goes through the page cache before being written to disk. We modified the page cache so that when a page is requested for a disk write, the backpointer is written into the page first and then returned for writing. The block offset translation was modified to take the backpointer into account when translating a logical offset into a block number.

Verification: Once a page is populated with data from the disk, the page is checked for the correct backpointer. If the check fails, an I/O error is returned, and the inode's attributes (size and number of blocks) are updated. Note that the page is not checked on every access, but only the first time that it is read from disk. Assuming memory corruption does not

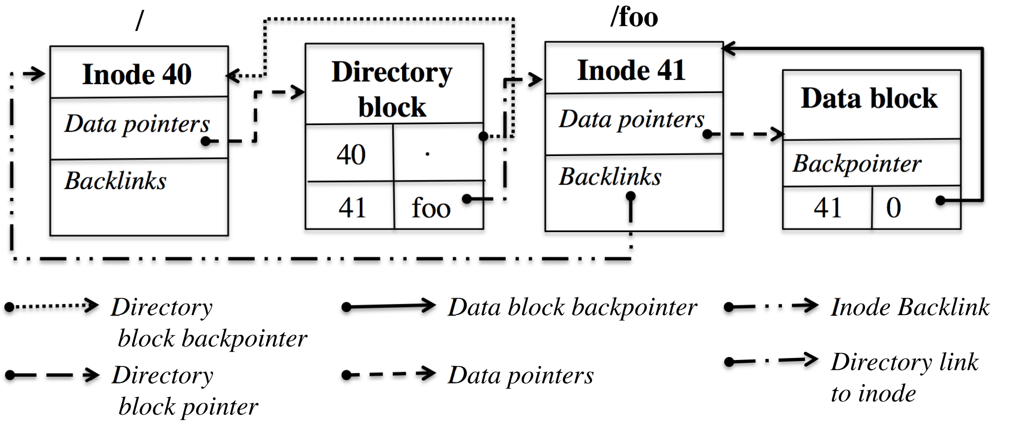


Figure 5.3: **Implementation of backpointers.** *The figure shows the different kinds of backpointers present in NoFS. foo is a child of the root inode /. This link is represented by a backlink from foo to /. Similarly, the data block is a part of foo, and hence has a backpointer to foo. Directory blocks also contain backpointers, in the form of dot entries to their owner's inode.*

occur [314], this level of checking is sufficient.

Directory backpointers

The dot directory entry serves as the backpointer for directory blocks, as it points to the inode which owns the block. However, the dot entry is only present in the first directory block. We modified ext2 to embed the dot entry in every directory block, thus allowing the owner of any directory block to be identified using the dot entry.

Although the block backpointer could have been used in directory blocks as well, we did not do so for two reasons. First, the structured content of the directory block enables the use of the dot entry as the backpointer, simplifying our implementation. Second, the offset part of the block backpointer is unnecessary for directory blocks since directory blocks are unordered and appending a directory block at the end suffices for recovery.

Insertion: When a new directory entry is being added to the inode, it is determined whether a new directory block will be needed. If so, the dot entry in added in the new block, followed by the original directory entry.

Verification: Whenever the directory block is accessed, such as in `readdir`, the dot entry is cross-checked with the inode. If the check fails, an I/O error is returned and the directory inode's attributes (size and block count) are updated.

Backlinks

An inode's backlinks contain the inode numbers of all its parent directories. Every valid inode must have at least one parent. Hard linked inodes may have multiple parents.

We modified the file-system layout to add space for backlinks inside each inode. The inode size is increased from the default 128 bytes to 256 bytes, enabling the addition of 32 backlinks, each of size 4 bytes. The `mke2fs` tool was modified to create a backlink between the `lost+found` directory and the root directory when the file system is created.

Insertion: When a child inode is linked to a parent directory during system calls such as `create` or `link`, a backlink to the parent is added in the child inode.

Verification: At each step of the iterative inode lookup process, we check that the child inode contains a backlink to the parent. A failed check stops the lookup process, reduces the number of links for the inode, and returns an I/O error.

Detection

Every data block is checked for a valid backpointer when it is read from the disk into the page cache. We assume that neither memory nor on-disk corruption happens; hence, it is safe to limit checking to when a data block is first brought into main memory. It is this property that leads to the high performance of NoFS; because disk I/O is several orders of magnitude slower than in-memory operations, the backpointer check can be performed on disk blocks with low overhead.

Inode backlink checking occurs during directory path resolution. The child inode's backlink to the parent inode is checked. Since both inodes are typically in memory during directory path resolution, the backlink check is a quick in-memory check, and does not degrade performance significantly, since a disk read is not performed to obtain the parent or child inode.

Note that the detection of inconsistency happens at the level of a single resource, such as an inode or a data block. Verifying that a data block belongs to an inode can be done without considering any other object in the file system. The presence of corrupt files or blocks does not affect the reads or writes to other non-corrupt files. As long as corrupt blocks are not accessed, their presence can be safely ignored by the rest of the system. This feature contributes to the high availability of NoFS: a file-system check or recovery protocol is not needed upon mount. Files can be immediately accessed, and any access of a corrupt file or block will return an error. This feature also allows NoFS to handle concurrent writes and deletes. Even if many writes and deletes were going on at the time of a crash, NoFS can still detect inconsistencies by considering each inode and data block pair in isolation.

Examples. We illustrate how backpointers help in detecting inconsistencies with two examples. Figure 5.4 shows three failure scenarios during the rename of a file. The file-system state before the rename, and the in-memory changes that happen due to the rename are shown. In Scenario #1, a crash happens before the file is unlinked in the old directory; as a result, both new and old directories claim the file. Using the inode backlink, the true owner of the file (the new directory) is determined. In Scenario #2, the crash happens before both the old directory and the file inode are updated; once again, both directories claim the file. But in this case the

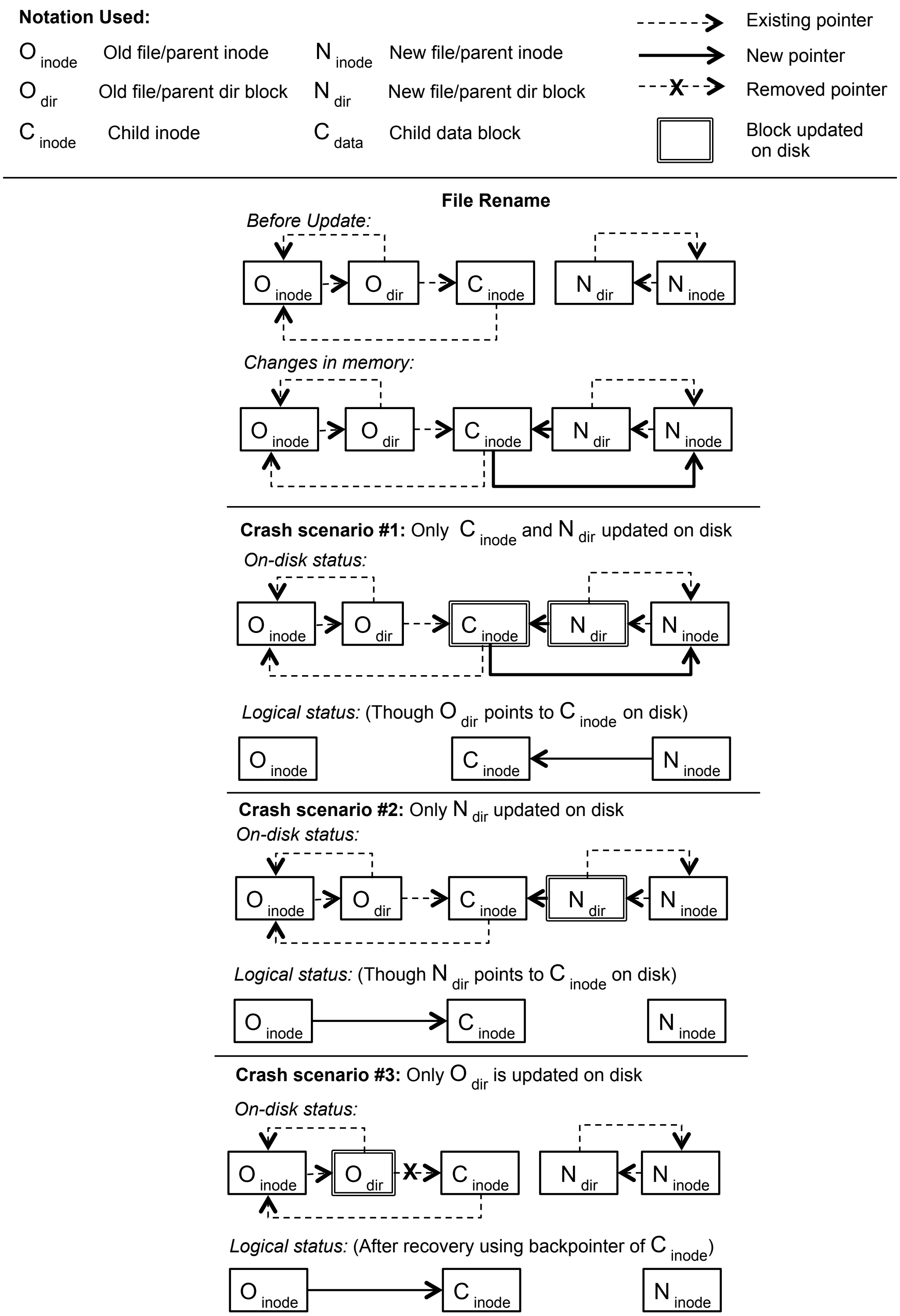

Notation Used:
O inode Old file/parent inode
N inode New file/parent inode
O dir Old file/parent dir block
N dir New file/parent dir block
C inode Child inode
C data Child data block
Existing pointer
New pointer
X Removed pointer
Block updated on disk
File Rename
Before Update:
O inode
O dir
C inode
N dir
N inode
Changes in memory:
Crash scenario #1: Only C inode and N dir updated on disk
On-disk status:
Logical status: (Though O dir points to C inode on disk)
Crash scenario #2: Only N dir updated on disk
On-disk status:
Logical status: (Though N dir points to C inode on disk)
Crash scenario #3: Only O dir is updated on disk
On-disk status:
X
Logical status: (After recovery using backpointer of C inode)

Figure 5.4: **Failure Scenario: Rename.** *The figure presents three failure scenarios during the rename of a file. In each scenario, employing backpointers allows us to detect inconsistencies such as both the old and new parent directories claiming the renamed file.*

true owner is determined to be the old directory using the inode backlink. In Scenario #3, only the old directory is updated before the crash. As a result, the renamed file is missing from both directories. Using the inode backlink, the file can be restored to the old directory.

Figure 5.5 illustrates the detection of inconsistencies due to a crash while creating a single byte file. The figure shows the file-system state before and after the operation, and three crash scenarios. In Scenario #1, the directory and file inode are updated on-disk before the crash; since the data block has not been updated, the block backpointer is missing, and this is detected when the data block is accessed. In Scenario #2, the inode and the data block are updated on-disk before the crash; as a result, the directory does not point to the inode. Since the inode backlink is present, the file system can determine the directory the file is supposed to belong to; the file-system policy can determine if the directory is updated to point to the file in cases like this. In Scenario #3, the directory and the data block are updated before the crash. Since the inode is not updated, an error is returned when the file is accessed via the directory. The data block remains unallocated.

Recovery

Having backlinks and backpointers allows recovery of lost files and blocks. Files can be lost due to a number of reasons. A rename operation consists of a unlink and a link operation. An inopportune crash could leave the inode not linked to any directory. A crash during the create operation could also lead to a lost file. Such a lost file can be recovered in NoFS, due to the backlinks inside each inode. Each such inode is first checked for access to all its data blocks. If all the data blocks are valid, it is a valid subtree in the file system and can be inserted back into the directory hierarchy (using the backlinks information) without compromising the consistency of the file system. When adding a directory entry for the

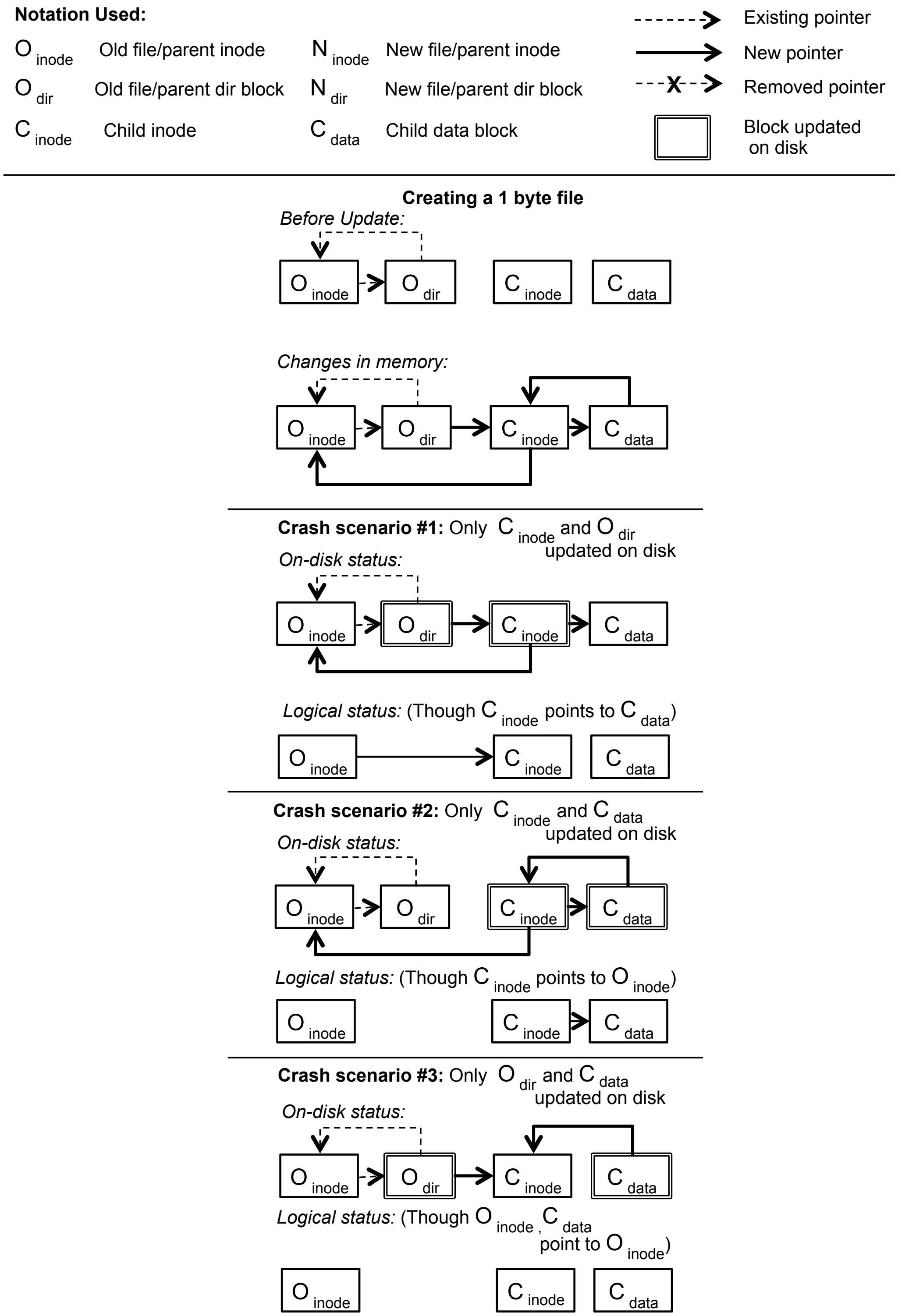

Notation Used:
O inode Old file/parent inode
N inode New file/parent inode
O dir Old file/parent dir block
N dir New file/parent dir block
C inode Child inode
C data Child data block
Existing pointer
New pointer
X Removed pointer
Block updated on disk
Creating a 1 byte file
Before Update:
O inode
O dir
C inode
C data
Changes in memory:
O inode
O dir
C inode
C data
Crash scenario #1: Only C inode and O dir updated on disk
On-disk status:
O inode
O dir
C inode
C data
Logical status: (Though C inode points to C data)
O inode
C inode
C data
Crash scenario #2: Only C inode and C data updated on disk
On-disk status:
O inode
O dir
C inode
C data
Logical status: (Though C inode points to O inode)
O inode
C inode
C data
Crash scenario #3: Only O dir and C data updated on disk
On-disk status:
O inode
O dir
C inode
C data
Logical status: (Though O inode , C data point to O inode)
O inode
C inode
C data

Figure 5.5: **Crash Scenario: File Create.** *The figure presents three failure scenarios during the creation of a file with 1 byte of data. In each scenario, employing backpointers allows us to detect inconsistencies such as the new file pointing to a data block that hasn't been updated.*

recovered inode, it is correct to append the directory entry at the end of the directory, since directory entries are an unordered collection; there is no meaning attached to the exact offset inside a directory block where a directory entry is added. Note that before a file can be inserted back into a directory, the directory needs to be valid; if the directory has not already been checked, then the path has to be checked all the way to the root. We believe that in the common case, most of the directory hierarchy will be present in memory.

In a similar fashion, it it possible to recover data blocks lost due to a crash before the inode is updated. A data block, once it has been determined to belong to an inode, cannot be embedded at an arbitrary point in the inode data. It is for this reason that the *offset* of a data block is embedded in the data block, along with the inode number. The offset allows a data block to be placed exactly where it belongs inside a file. Indirect blocks of a file do not have the offset embedded, as they do not have a logical offset within the file. Indirect blocks are not required to reconstruct a file; only data blocks and their offsets are needed.

Using reconstruction of files from their blocks on disk, files can be potentially "undeleted", provided that the blocks have not been reused for another file. We have not implemented undelete in NoFS. Block allocation would need to be tweaked to not reuse blocks for a certain amount of time, or until a certain free-space threshold is reached. Undelete might turn up stale data because NoFS does not support version consistency; the data block might have been part of an older version of the inode.

5.3.3 Non-persistent allocation structures

The allocation structures in ext2 are bitmaps and group descriptors. These structures are not persisted to disk in NoFS. In-memory versions of these data structures are built using the *metadata scanner* and *data scanner*. Statistics usually maintained in the group descriptors, such as the number of free blocks and inodes, are also maintained in their in-memory versions.

Upon file-system mount, in-memory inode and block bitmaps are initialized to zero, signifying that every inode and data block is free. Since every block and inode has a backpointer, it can be determined to be in use by examining its backlink or backpointer, and cross-checking with the inode mentioned in the backpointer. As every object is examined, consistent file-system state is built up and eventually complete knowledge of the system is achieved.

In the file system, a block or inode that is marked free could mean two things: it is free, or it has not been examined yet. Since all blocks and inodes are marked free at mount time, inodes need to be examined to check that they are indeed free; hence blocks or inodes that have not been examined yet cannot be allocated. In order to mark which inodes or blocks have been examined, we added a new in-memory bitmap each for inodes and data blocks called the *validity* bitmap. If a block or inode has been examined and marked as free, it is safe to use it. Blocks not marked as valid could actually be used blocks, and hence must not be used for allocation. The examination of inodes and blocks are carried out by two background threads called the metadata scanner and data scanner. The two threads work closely together in order to efficiently find all the used inodes and blocks on disk. File writes can occur while the two threads are in operation (as long as free inodes and data blocks are available).

Metadata Scan

Each inode needs to be examined to find out if it is in use or not. The backlinks in the inode are found, and the directory blocks of the referred inodes are searched for a directory entry to this inode. Note that the directory hierarchy is not used for for the scan. The disk order of inodes is used instead, as this allows for fast sequential reads of the inode blocks.

Once an inode is determined to be in use, its data blocks have to verified.

This information is communicated to the data scanner by adding the data blocks of the inode to a list of data blocks to be scanned. The inode information is also attached to the list so that the data scanner can simply compare the backpointer value to the attached value to determine whether the block is used. However, if the inode has indirect blocks, the inode data blocks are explored and verified immediately. An inode with indirect blocks may contain thousands of data blocks, and it would be cumbersome to add all those data blocks to the list and process them later; hence inode data is verified immediately by the metadata scanner. Each inode is marked valid after it has been scanned, allowing inode allocation to occur concurrently with the metadata scan.

Data Scan

Observe that a data block is in use only if it is pointed to by a valid inode which is in use; hence only data blocks that belong to a valid inode need to be checked, thus reducing the set of blocks to be checked drastically.

The data block scanner works off a list of data blocks that the metadata scanner provides. Each list item also includes information about the inode that contained the data block. Therefore, the data scanner simply needs to read the inode off the disk and compare the backpointer inode to the inode information in the list item. The data block is marked valid after the examination is complete.

Since the data scanner only looks at blocks referred to by inodes, there may be plenty of unexamined blocks which are not referred and potentially free. These blocks cannot be marked as valid and free until the end of the data scan, when all valid inodes have been examined. While the scan is running, the file system may indicate that there are no free blocks available, even if there are many free blocks in the system. In order to fix this, we implemented another scanner called the sequential block scanner which reads data blocks in disk order and verifies them one by one. This thread is only started if no free blocks are found for an application write, and the data scanner is still running. In the future, we plan to start the data block scanner when the number of data blocks available for allocation falls below a minimum user-configurable threshold.

5.4 Evaluation

We now evaluate NoFS along two axes: reliability and performance. For reliability testing, we artificially prevent writes to certain sectors from reaching the disk, and then observe how NoFS handles the resulting inconsistency. For performance testing, we evaluate the performance of NoFS on a number of micro and macro-benchmarks. We compare the performance of NoFS to ext2, an order-less file system with weak consistency guarantees, and ext3 (in ordered mode), a journaling file system with metadata consistency.

5.4.1 Reliability

Methodology. We test whether NoFS can handle inconsistencies caused by a file-system crash. When a crash happens, any subset of updates involved in a file-system operation could be lost. We emulate different system-crash scenarios by artificially restricting blocks from reaching the disk, and restarting the file-system module. The restarted module will see the results of a partially completed update on disk.

We use a pseudo-device driver to prevent writes on target blocks and inodes from reaching the disk drive. We interpose the pseudo-device driver in-between the file system and the physical device driver, and all writes to the disk drive go via the pseudo-device driver. The file system and the device driver communicate through a list of sectors. In the file system, we calculate the on-disk sectors of target blocks and inodes and add them to the black list of sectors. All writes to these sectors are ignored by the device driver. Thus, we are able to target inodes and blocks in a fine grained manner.

Note that we do not test writes dropped over a long period of operation or multiple re-ordered file-system operations. Since our reliability measures are not affected by either the time when writes occurred or other concurrent writes at the time of the crash, we believe our tests reflect all the relevant cases.

Example. Consider the `mkdir` operation. It involves adding a directory entry to the parent directory, updating the new child inode, and creating a new directory block for the child inode. We do not consider updates to the access time of the parent inode. In the reliability test, we would drop writes to different combinations of these blocks, access the files or data blocks involved in the test, and observe the actions taken by the file system. For instance, if the write to the new child inode is dropped, it creates a bad directory entry in the parent directory, and orphans the directory block of the new child inode. We observe whether the file system detects this corrupt directory entry, and whether the orphan block is reclaimed.

Results. Table 5.6 shows the results of our reliability experiments. Each row in the table corresponds to a different experiment, with one or more blocks in the file-system update being dropped (*i.e*, is not updated on disk). The table rows which have two system calls denote the second system call happening after the first system call. These particular combinations were selected because they share a common resource. For example, truncate-write explores the case when a data block is deleted from a file and reassigned to another file. If the write to the truncated file inode fails, both files now point to the same data block, leading to an inconsistency. Similarly unlink-link and delete-create may share the same inode.

The table lists the errors that result from the dropped blocks in each experiment. The *bad directory entry (BD)* error indicates that a directory

System call	Blocks dropped	Error	ext2		NoFS	
			Detected?	Action?	Detected?	Action?
mkdir	C^{inode}	P^{BD}, C^{OB}	×	–	√	R, C^{EI}
mkdir	C^{dir}	C^{BD}	√	C^{ED}	√	C^{ED}
mkdir	P^{dir}	C^{OI}, C^{OB}	×	–	√	R
mkdir	C^{inode}, C^{dir}	P^{BD}, C^{BD}	×	–	√	C^{EI}
mkdir	C^{inode}, P^{dir}	C^{OB}	×	–	√	R
mkdir	C^{dir}, P^{dir}	C^{OI}	×	–	√	R
link	C^{inode}	C^{HL}	×	–	√	C^{EN}
link	P^{dir}	C^{OI}	×	–	√	R
unlink	C^{inode}	C^{HL}	×	–	√	C^{EO}
unlink	O^{dir}	P^{BD}	×	–	√	C^{EI}
rename	N^{dir}	O^{BD}	×	–	√	C^{EI}
rename	O^{dir}	C^{OI}	×	–	√	R
write	C^{data}	C^{GD}	×	–	√	C^{EB}
write	C^{ind}	C^{GD}	×	–	√	C^{EB}
write	C^{inode}, C^{data}	C^{OB}	×	–	√	R
write	C^{inode}, C^{ind}	C^{OB}	×	–	√	R
write	C^{data}, C^{ind}	C^{GD}	×	–	√	C^{EB}
delete-create	O^{dir}	O^{BD}	×	–	√	C^{EO}
truncate-write	O^{inode}	O^{TP}	×	–	√	O^{EB}
unlink-link	O^{dir}	O^{BD}	×	–	√	C^{EO}

General Key

C	Child	*inode*	File inode
P	Parent	*dir*	Directory block
O	Old file/parent	*data*	Data block
N	New file/parent	*ind*	Indirect block

Key for Error		Key for Action	
BD	Bad dir entry	*R*	Block/inode reclaimed on scan
OB	Orphan block	*EI*	Error on inode access
OI	Orphan inode	*ED*	Error on data access
HL	Wrong hard link count	*EN*	Error on access via new path
GD	Garbage data	*EO*	Error on access via old path
TP	2 inodes refer to 1 block	*EB*	Error on block access

Table 5.6: **Reliability testing.** *The table shows how NoFS and ext2 react to various inconsistencies that occur due to updates not reaching the disk. NoFS detects all inconsistencies and reports an error, while ext2 lets most of the errors pass by undetected.*

points to an file that has been deleted or is not initialized. The *orphan (OB, OI)* block and inode errors indicate resources that are marked used in the file system but are not actually in use (*e.g.*, marking a data block as used when the file it was allocated to is not modified to point to the data block). The *wrong hard link count (HL)* indicates that the inode was not updated in accordance with the directory entries pointing to the inode. The *garbage data (GD)* error indicates that the file points to an uninitialized data block. The *TP* error indicates that two inodes point to the same data block.

The table also lists the action taken by the file system in response to the error. Action *R* indicates that file system reclaims the orphan block or inode on the next scan. Actions *EI*, *ED*, and *EB* indicate that the file system returns an error when the inode is accessed (*e.g.*, via `ls`), when the directory block is accessed (*e.g.*, via `ls`), or when the data block is accessed on a file read. Actions *EN* and *EO* indicate that the file system returns an error when the file is accessed via the new or old directory paths respectively.

Only one inconsistency, a corrupt directory block, is detected by ext2 (due to the internal structure of a directory block). Other inconsistencies, such as reading garbage data, are not detected by ext2. All inconsistencies are detected by NoFS, and an error is returned to the user. When blocks and inodes are orphaned due to a crash, they are reclaimed by NoFS when the file system is scanned for allocation information upon reboot. Some of the inconsistencies could lead to potential security holes: for example, linking a sensitive file for temporary access, and removing the link later. If the directory block is not written to disk, the file could still be accessed, providing a way to read sensitive information. NoFS detects these security holes upon access and returns an error.

5.4.2 Performance

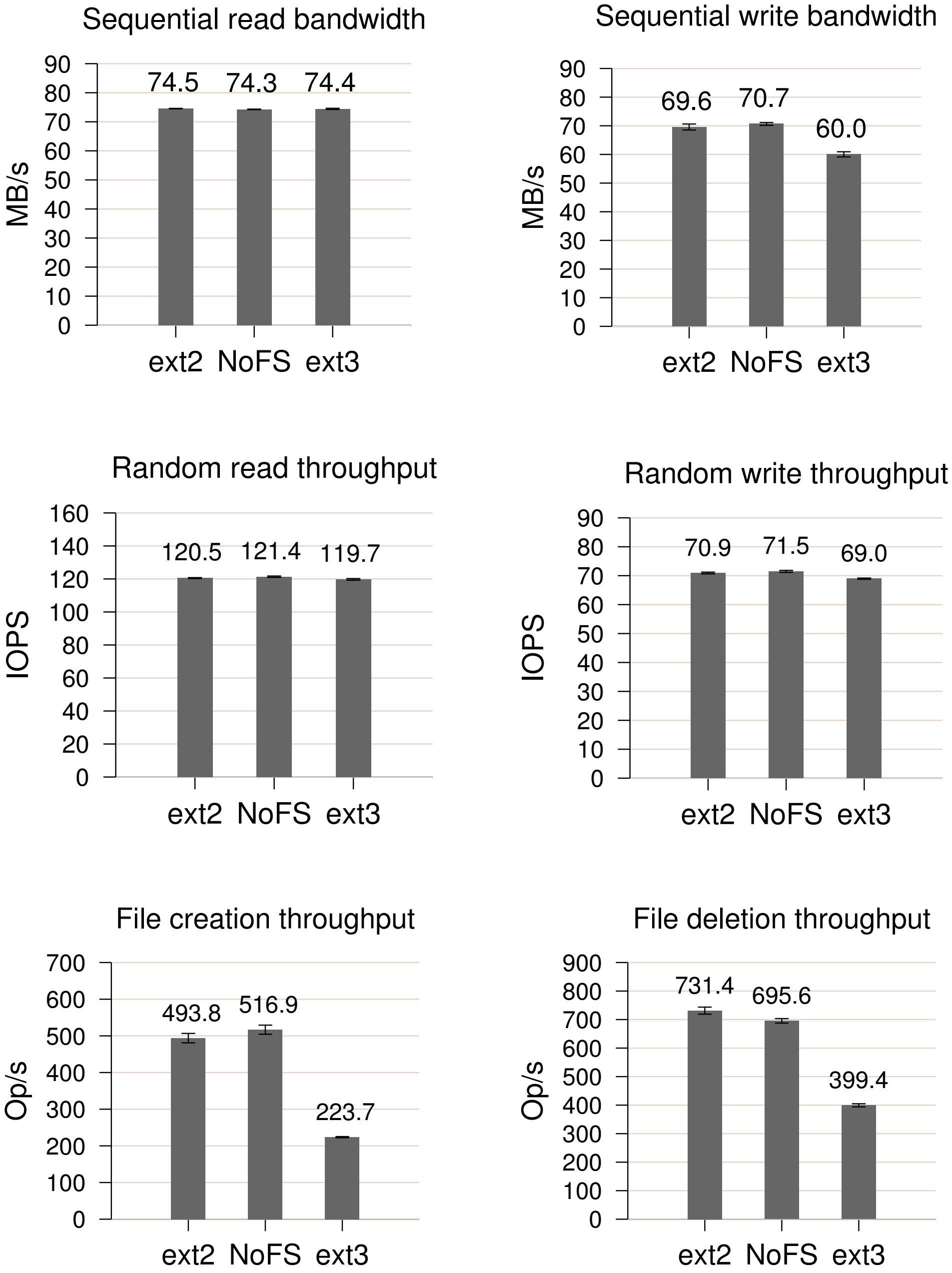
To evaluate the performance of NoFS, we run a series of micro-benchmark and macro-benchmark workloads. We also observe the performance of NoFS at mount time, when the scan threads are still active. We show that NoFS has comparable performance to ext2 in most workloads, and that the performance of NoFS is reasonable when the scan threads are running in the background. We also measure the scan running time when the file system is populated with data, the rate at which NoFS scans data blocks to find free space, and the performance cost incurred when the stat system call is run on unverified inodes.

The experiments were performed on a machine with a AMD 1 GHz Opteron processor, 1 GB of memory, and a Seagate Barracuda 160 GB drive. The operating system was Linux 2.6.27.55. The Seagate drive provides a maximum of 75 MB/s read throughput and 70 MB/s write throughput. The experiments were performed on a cold file-system cache, and were stable and repeatable. The numbers reported are the average over 10 runs.

Micro-benchmarks

We run a number of micro-benchmarks, focusing on different operations like sequential write and random read. Figure 5.7 illustrates the performance of NoFS on these workloads. We observe that NoFS has minimal overhead on the read and write workloads. For the sequential write workload, the performance of ext3 is worse than ext2 and NoFS due to the journal writes that ext3 performs.

The creation and deletion workloads involve doing a large number of creates/deletes of small files followed by fsync. This workload clearly brings out the performance penalty due to ordering points. The throughput of NoFS is twice that of ext3 on the file creation micro-benchmark, and 70% higher than ext3 on the file deletion benchmark.



Sequential read bandwidth

Sequential write bandwidth

Random read throughput

Random write throughput

File creation throughput

File deletion throughput

Figure 5.7: **Micro-benchmark performance.** *This figure compares file-system performance on various micro-benchmarks. The sequential benchmarks involve reading and writing a 1 GB file. The random benchmarks involve 10K random reads and writes in units of 4088 bytes (4096 bytes - 8 byte backpointer) across a 1 GB file, with a fsync after 1000 writes. The creation and deletion benchmarks involve 100K files spread over 100 directories, with a fsync after every create or delete.*

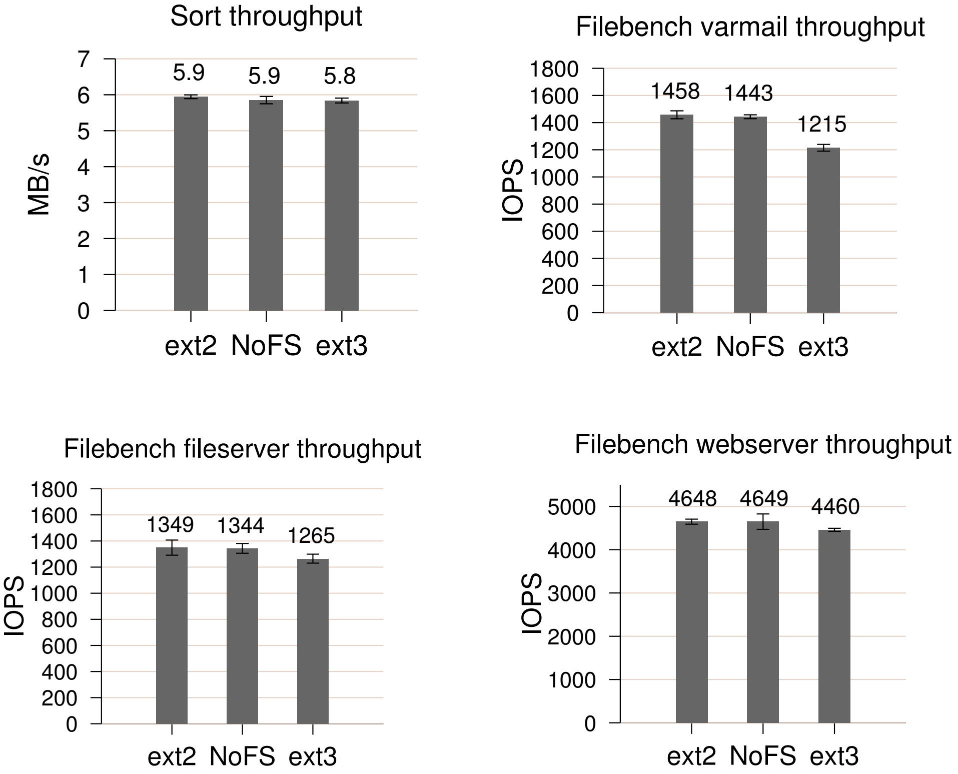


Figure 5.8: **Macro-benchmark performance.** *The figure shows the throughput achieved on various application workloads. The sort benchmark is run on 500 MB of data. The varmail benchmark was run with parameters 1000 files, 100K mean dir width, 16K mean file size, 16 threads, 16K I/O size and 16K mean append size. The file and webserver benchmarks were run with the parameters 1000 files, 20 dir width, 1 MB I/O size and 16K mean append size. The mean file size was 128K for the fileserver benchmark and 16K for the webserver benchmark. Fileserver benchmark used 50 threads while webserver used 100 threads.*

Macro-benchmarks

We run the sort and Filebench [82] macro-benchmarks to assess the performance of NoFS on application workloads. Figure 5.8 illustrates the performance of the three file systems on this macro-benchmark. We selected the sort benchmark because it is CPU intensive. It sorts a 500 MB file generated by the *gensort* tool [198], using the command-line sort utility. The performance of NoFS is similar to that of ext2 and ext3, demonstrating that NoFS has minimal CPU overhead.

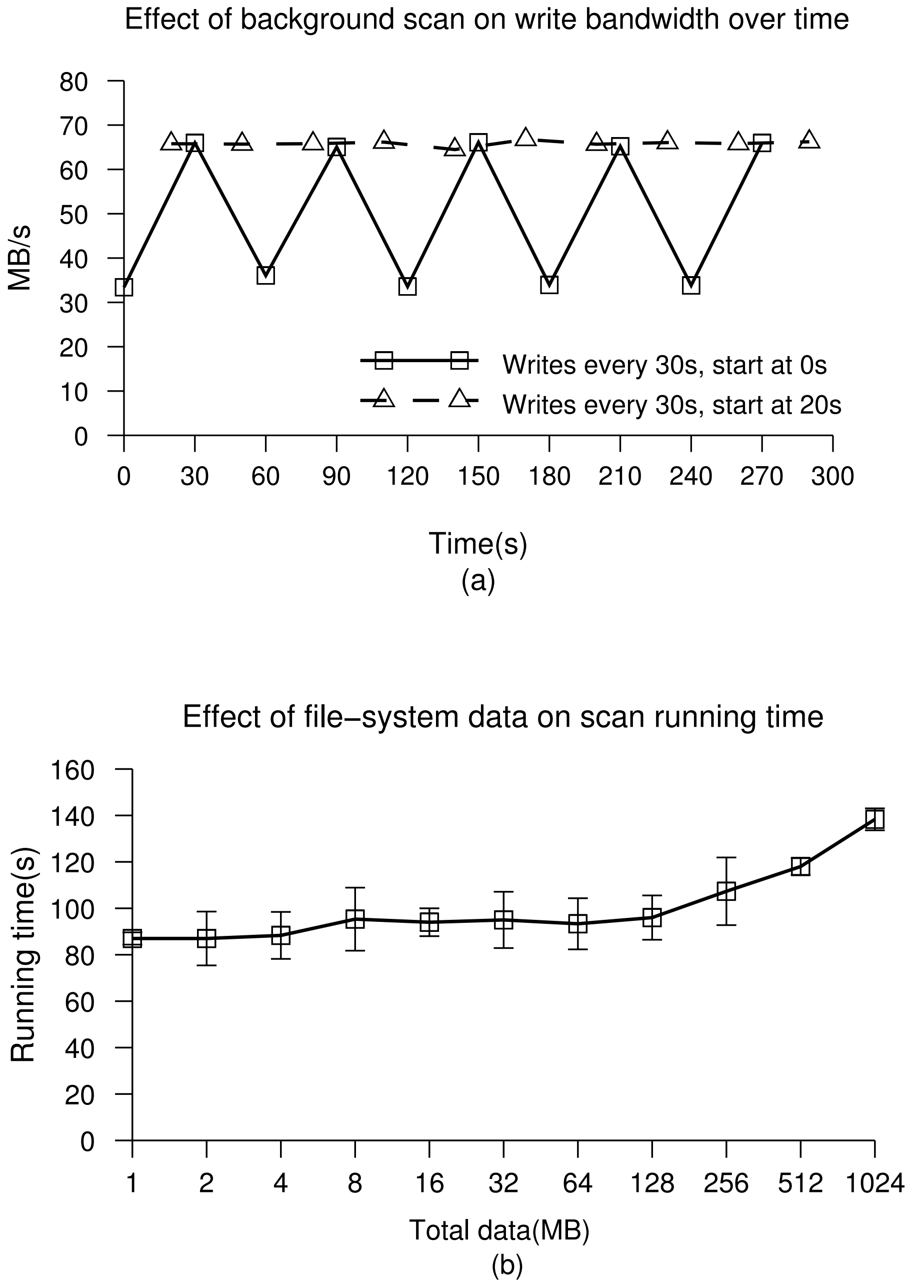
We run three Filebench workloads. The fileserver workload emulates file-server activity, performing a sequence of creates, deletes, appends, reads, and writes. The webserver workload emulates a multi-threaded web host server, performing sequences of open-read-close on multiple files plus a log file append, with 100 threads. The varmail workload emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory.

We believe these benchmarks are representative of the different kind of I/O workloads performed on file systems. The performance of NoFS matches ext2 and ext3 on all three workloads. NoFS outperforms ext3 by 18% on the varmail benchmark, demonstrating the performance degradation in ext3 due to ordering points.

Scan performance

We evaluate the performance of NoFS at mount time, when the scanner is still scanning the disk for free resources. The scanner is configured to run every 60 seconds, and each run lasts approximately 16 seconds. In order to understand the performance impact due to scanning, we do two experiments involving 10 sequential writes of 200 MB each. The writes are spaced 30 seconds apart.

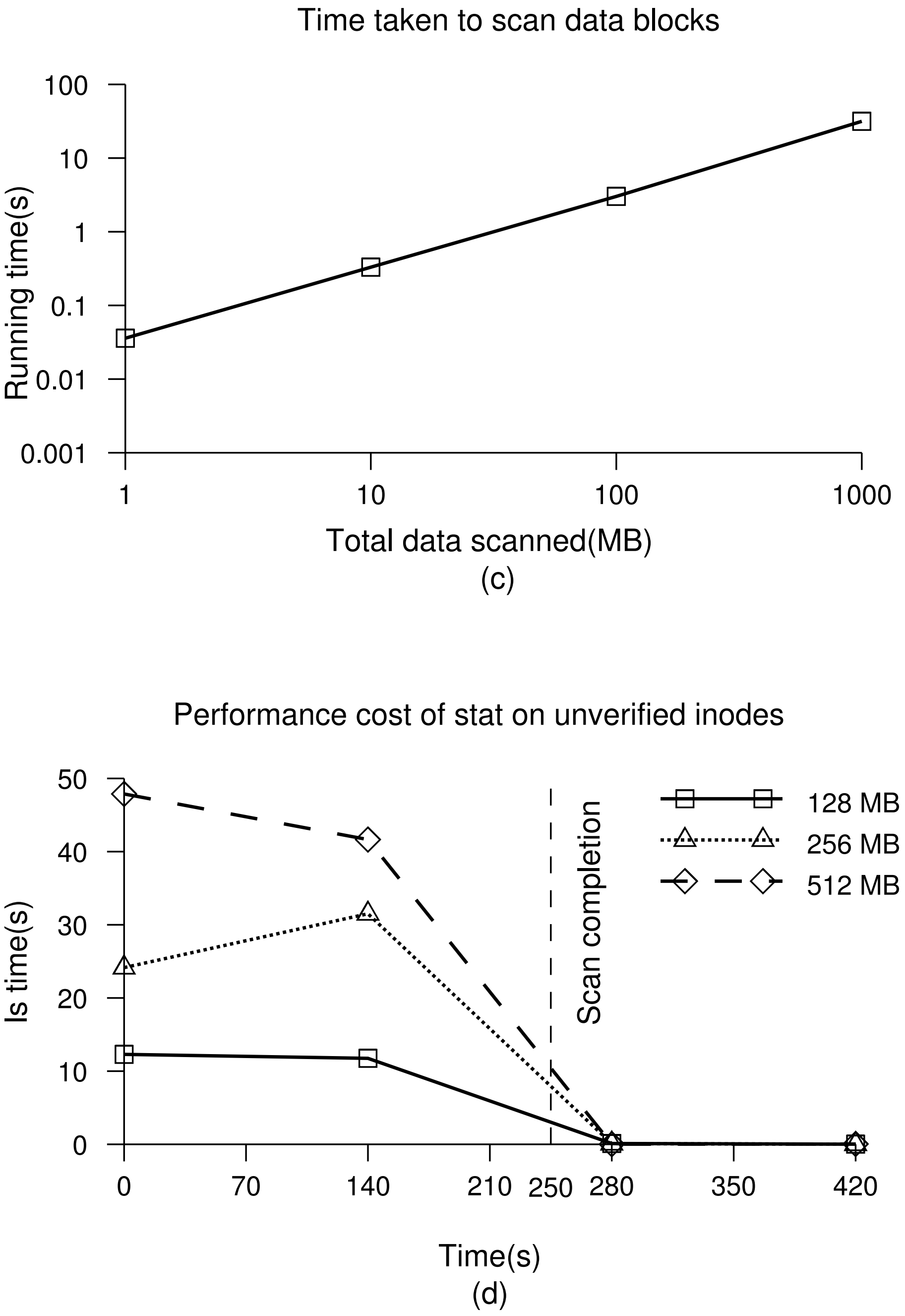
In the first experiment, we start the writes at mount time. The scanning of the disk and the sequential write is interleaved at 0s, 60s, 120s, and so on, leading to the write bandwidth dropping to half. When the sequential writes are run at 30s, 90s, 150s, and so on, the writes achieve peak bandwidth. In the second experiment, the writes were once again spaced 30s apart, but were started at 20s, after the end of the first scan run. In this experiment, the writes are never interleaved with the scan



(a)

(b)

Figure 5.9: **Interaction of Background Scan and File-System Activity.** *Figure (a) depicts the reduction in write bandwidth when sequential writes interleave with the background scan. Figure (b) shows that the running of the scan increases slowly with the amount of data in the file system.*



(c)

(d)

Figure 5.10: **Cost of the Background Scan.** *Figure (c) illustrates the rate at which data blocks are scanned. Figure (d) demonstrates the performance cost incurred when the stat system call is run on unverified inodes.*

reads, and hence suffer no performance degradation. Graph (a) in Figure 5.9 illustrates these results.

Once the scan finishes, writes will once again achieve peak bandwidth. Running the scan runs without a break causes the scan to finish in around 90 seconds on an empty file system. Of course, one can configure this trade-off as need be; the larger the interval between scans, the smaller the performance impact during this phase, but the longer it takes to fully discover the free blocks of the system.

Graph (b) in Figure 5.9 depicts the time taken to finish the scan (both metadata and data) when the file system is increasingly populated with data. In this experiment, the scan is run without a break upon file-system mount. All the data in the file system are in units of 1 MB files. The running time of the scan increases slowly when the amount of data in the file system is increased, reaching about 140s for 1 GB of data. The trend indicates that the running time for multi-terabyte systems will be tens of hours. The long runtime is not a problem since the file-system is available for reading and writing while the scan completes in the background; only in the case of a almost-full file system will the user encounter a problem. We also performed an experiment where we created a variable number of empty files in the file systems and measured the time for the scan to run. We found that the time taken to finish the scan remained the same irrespective of the number of empty files in the system. Since every inode in the system is read and verified, irrespective of whether it is actively used in the file system or not, the scan time remains constant.

During a file write, if there are no free blocks, the sequential block scanner is invoked in order to scan data blocks and find free space. The write will block until free space is found. Graph (c) in Figure 5.10 illustrates the performance of the sequential block scanner. The latency to scan 100 MB is around 3 seconds, and 1 GB of data is scanned in around 30 seconds. The throughput is currently around 30 MB/s, so there is opportunity for optimizing its performance.

As mentioned in Section 5.5.1, when `stat` is run on an unverified inode, NoFS first verifies the inode by checking all its data blocks. We ran an experiment to estimate the cost of such verification. We created four identical directories, each filled with a number of 1 MB files. Every 140 seconds, `ls -li` was run on one directory, leading to a `stat` on each inode in the directory. The background scan started at file-system mount and finished at approximately 250 seconds. We varied the number of files from 128 to 512 and measured the time taken for `ls -li` in each experiment. Graph (d) in Figure 5.10 illustrates the results. As expected, the time taken for `ls` to complete increases with the total data in the directory. After the scan completion at 250 seconds, all the inodes are verified, and hence `ls` finishes almost instantly.

5.5 Discussion

Although NoFS provides strong consistency guarantees and good performance on many workloads, its unique design affects file-system users in several ways. We first discuss the limitations of the design. We then describe appropriate use-cases for NoFS. Finally, we discuss challenges faced in implementing NoFS.

5.5.1 Limitations

The design of NoFS involves a number of trade-offs. We describe the limitations that arise from our design choices.

Recovery: NoFS was designed to be as lightweight as possible, avoiding heavy machinery for logging or copy-on-write. As a result, file-system recovery is limited. For example, consider a file that is truncated, and later written with new data. After a crash in the middle of these updates (and subsequent remount), the file may point to a block that it does not own.

This inconsistency is detected upon access to the data block. However, the version of the file which pointed to its old data cannot be recovered easily. By utilizing logging, a file system like ext3 provides the ability to preserve data in the event of a crash.

Rename: Atomic rename is critical in the following update sequence: 1. create temporary file; 2. write temporary file with the entire contents of the old file, plus updates; 3. persist temporary file via `fsync()`; 4. atomically rename temporary file over old file. At the end of this sequence, the file should exist in either the old state or the new state with all the updates. If `rename()` is not atomic, a crash could result in the file being lost [59]. Since `rename()` is not atomic in NoFS, many applications cannot be run correctly on top of NoFS.

Note that NoFS could be modified to remove this problem. Two changes would be required. The first is modifying the namespace from the current hierarchical structure to a flat structure like the Google File System [238]. The second change is eliminating the centralized directory structure, and storing directory information directly inside inodes (similar to ReconFS [160]). In this new design, a rename only affects a single inode. Similar to how allocation information needs to be built up in memory on boot, directory information will also have to be scanned into memory in the background. We have not implemented this design.

The ABA problem: Since NoFS provides data consistency, a file can contain a data block that belonged to the file at some point in the past; similarly, a directory may contain a file that was part of the directory at some point in the past. NoFS is susceptible to the ABA problem [245, 284]: if the existence of a file in directory is used by an application to denote that some external event has not occurred, the application may behave incorrectly. Such applications require version consistency to behave correctly; NoFS needs to be augmented with time-stamps to achieve version consistency.

`mmap()` support: NoFS intercepts all `read()` calls, and hence is able to verify that files are accessing the correct data off storage. NoFS does not intercept all `mmap()` calls, and hence an application using `mmap()` (such as older versions of LevelDB [93]) cannot be run correctly on top of NoFS.

Accessing unverified objects: For large disks, it is possible that an object is accessed before the scan has verified it. Accessing such unverified objects involves a performance cost. The performance cost is felt during different system calls for inodes and data blocks.

Running the `stat` system call on an unverified inode may result in invalid information, as the number of blocks recorded in the inode may not match the actual number of blocks that belong to the inode on disk. In order to handle this, NoFS checks the inode status upon a `stat` call, and verifies the inode immediately if required, and then allows the system call to proceed. Since verification involves checking every data block referred to by the inode, the verification can take a lot of time. Running `ls -l` on a large directory of unverified files involves a large performance penalty arising from reading every file. For verified inodes, the `stat` will always return valid data, as the inode's attributes are updated whenever an error is encountered on block access. Note that NoFS does not check directory entries for correctness.

In the case of an unverified data block, no additional I/O is incurred during reads and partial writes since both involve reading the block off the disk anyway. However, in the case of a block overwrite, the block has to be read first to verify that it belongs to the inode before overwriting it. As a result, a write in ext2 is converted into a read-modify-write in NoFS, effectively cutting throughput in half. It should be noted that this happens only on the *first* overwrite of each unverified block. After the first overwrite, the block has been verified, and hence the backpointer no longer needs to be checked.

Thus it can be seen that accessing unverified objects involves a large performance hit. However, these costs are only incurred during the window between file-system mount and scan completion.

5.5.2 Use Cases

Given its current design, we feel an excellent use-case for NoFS would be as the local file system of a distributed file system such as the Google File System [238] or the Hadoop File System [254]. In such a distributed file system, reliable detection of corruption is all that is required, since redundant copies of data would be stored across the system. If the master controller is notified that a particular block has been corrupted in the local file system of a particular node, it can make additional copies of the data in order to counter the corruption of the block. Furthermore, such distributed file systems typically have large chunk sizes. As shown in section 5.4, NoFS provides very good performance on large sequential reads and writes, and is well suited for such workloads.

It should be noted that backpointer-based consistency could also be used to help ensure integrity in a conventional file system against bugs or data corruption. The simplicity and low overhead of backpointers makes such an addition to an existing file system feasible.

By eliminating ordering, backpointer-based consistency allows the file system to maintain consistency without depending upon lower-layer primitives such as the disk cache flush. Previous research has shown that SATA drives do not always obey the flush command [226, 247], which is essential for file systems to implement ordering. IDE drives have also been known to disobey flush commands [225, 262]. Using backpointer-based consistency allows a file system to run on top of such misbehaving disks and yet maintain consistency.

Potential users of NoFS should note two things. One, any application which requires strict ordering among file creates and writes should not use NoFS. Two, if there are corrupt files in the system, NoFS will only detect them upon access and not upon file-system mount. Some users may prefer to find out about corruption at mount time rather than when the file system is running. Such a use case aligns better with a file system such as ext3.

5.5.3 Implementation Challenge: Widespread Assumptions About Block Size

NoFS requires that the backpointer and the data block are written together atomically to storage. While implementing NoFS, we emulated this by reducing the amount of data stored in each disk block from 4096 bytes to 4088 bytes (to have space for an 8-byte backpointer). Changing the amount of data stored in a data block proved surprisingly tricky.

There were two main areas where changes were needed: the file-system code, and the broader page-cache code. We describe each in turn.

File-System Code. The file-system has to translate a logical byte in a given file to a byte inside a specific block on disk. The translation requires using the disk block size. Changing the code to work with a 4088 byte block involves extensive changes across the code; bit-operations at several places had to changed into arithmetic operations to work with the 4088 block size. Simply changing the header `#define` statements was not enough.

Page-Cache Code. With a 4088-byte block, a 4096 page includes data from two disk blocks. Implementing this also proved tricky; the kernel code assumes that the data from a disk block goes exclusively into a single page. Reading data from two 4088 byte blocks into a 4096 page similarly involved changing bit operations extensively into arithmetic operations.

The assumption of the block size being a power of two may not hold true for all storage systems. Several devices allow data (such as checksums) to be stored in out-of-band areas [223, 274]: if the block and the out-of-band data could be read together in one operation, it would make the design of storage systems simpler. Rewriting kernel code so that it does not make implicit assumptions about the block/page size would improve the reliability and flexibility of storage systems.

5.6 Proof

Sivathanu *et al.* provided a logical framework for modeling file systems, and reasoning about the correctness of new features that are added [256]. We extend that framework to show that that adding backpointers to data and inode blocks in a file system ensures data consistency. Morever, by further adding timestamps, we can achieve version consistency.

5.6.1 Notation

The main entities are *containers, pointers, and generations*. A file system is simply a collection of containers, which can be freed and reused. They are linked to each other through pointers. The epoch of a container is incremented and its timestamp is changed every time the contents of a container change in memory. Thus, the epoch represents the version of a container. The generation of a container is incremented after each reallocation.

A state of the file system in memory or disk is represented by a belief. Beliefs denoted as $\{\}_M$ and $\{\}_D$ are memory beliefs and disk beliefs respectively. For example, $\{ A \dashrightarrow B\}_D$ indicates a belief that container A physically points to container B on disk.

Operators. We use a special ordering operator called *precedes* ($\prec$). Only a belief can appear to the left of a $\prec$ operator. $A \prec B$ means that belief A occurs before B. The $\Rightarrow$ operator indicates the belief to the left of the operator *implies* the belief on the right.

Table 5.11: Notations on containers

Symbol	Description
&A	set of entities that point to container A
A^x	the x^{th} version of container A
A_y	The y^{th} generation of container A
$g(A^x)$	the generation of the xth version of container A
$\{A^x\}_M$	the x^{th} version of A in memory
$\{A^x\}_D$	the x^{th} version of A on disk
$A \dashrightarrow B$	denotes that A has a physical pointer to B
$A \dashleftarrow B$	denotes that B has a backpointer to A
$A \rightarrow B$	denotes that A logically points to B
t(A)	the time that A was last updated
$ts(B, A)$	the timestamp for A that is stored in B

5.6.2 Axioms

In this subsection, we present the axioms that govern the transition of beliefs across memory and disk.

- If a version of a container exists on disk, it must first have existed in memory, followed by a write to disk.
$$\{A^x\}_M \prec write(A) \Rightarrow \{A^x\}_D \tag{5.1}$$

- B points to A logically in memory (or disk) only if B has a pointer to A, and A has a backpointer to B in memory (or disk).

$$\{B \rightarrow A\}_M \iff \{B \dashrightarrow A\}_M \wedge \{B \dashleftarrow A\}_M \tag{5.2}$$
$$\{B \rightarrow A\}_D \iff \{B \dashrightarrow A\}_D \wedge \{B \dashleftarrow A\}_D \tag{5.3}$$

- If A does not belong to any container in memory (or disk), it's backpointer does not point to any valid container in memory (or disk).

$$\{\&A = \phi\}_M \Rightarrow \forall c \neg \{c \dashleftarrow A\}_M \tag{5.4}$$
$$\{\&A = \phi\}_D \Rightarrow \forall c \neg \{c \dashleftarrow A\}_D \tag{5.5}$$

- Two versions of container B are different only if their timestamps are different.

$$x \neq y \iff t(B^x) \neq t(B^y) \tag{5.6}$$

5.6.3 Data Consistency

Data consistency indicates that the data blocks of a file will not contain garbage data after a crash. Since all blocks that logically belong to a file contain a backpointer to the file, it is trivially true that they belong to the file (now or at some point in the past) and that they do not contain garbage data. We now prove that if the block belonged to the file at some point in the past, it is the same version of the block that is now logically part of the file.

Formally, if the x^{th} version of B logically points to the z^{th} version of A on disk, and previously the same version of B pointed to the k^{th} generation of A, then the in-memory generation and on-disk generation should match. In other words, the version pointed to in memory is the version stored on disk.

$$\left(\{B^x \to A_k\}_M \prec \{B^x \to A^z\}_D\right) \Rightarrow (g(A^z) = k)$$

We assume that $g(A^z) \neq k$ and prove that this leads to a contradiction.

$$\left(\{B^x \to A_k\}_M \prec \{B^x \to A^z\}_D\right) \wedge (g(A^z) \neq k)$$

Since the epoch of B (x) is the same on disk and memory, B has not been changed until the disk write happened. Since B logically points to A, A has a backpointer to B. $g(A^z) \neq k$ can only happen if block A was freed and written to disk. After the free, it was re-allocated to B again.

This leads to two cases where the free could have happened - before the write of B (in memory) or after the write of B (on disk).

Case 1: Block A was freed and written to disk before Block B was written. However, since A has a backpointer to B on disk, the in-memory version of A must have had a backpointer to B. This contradicts the initial assumption that A was freed (and hence has no backpointers).

$$\begin{aligned}
\Rightarrow \quad & \Big((\{B^x \rightarrow A_k\}_M \;\wedge\; (\&A = \phi) \wedge \mathit{write}(A)) \\
& \prec \{B^x \rightarrow A^z\}_D\Big) \;\wedge\; (g(A^z) \neq k) \\
\Rightarrow \quad & \Big((\{B^x \dashleftarrow A_k\}_M \;\wedge\; (\&A = \phi) \wedge \mathit{write}(a)) \\
& \prec \{B^x \rightarrow A^z\}_D\Big) \;\wedge\; (g(A^z) \neq k) \text{ (Using 5.2)} \\
\Rightarrow \quad & \Big((\{B^x \dashleftarrow A_k\}_M \wedge \neg\{B^x \dashleftarrow A_k\}_M) \\
& \prec \{B^x \rightarrow A^z\}_D\Big) \;\wedge\; (g(A^z) \neq k) \\
& (\mathit{Since}, \{\&A = \phi\}_M \Rightarrow \forall c \neg\{c \dashleftarrow A\}_M) \\
\Rightarrow \quad & \Big(\mathit{false} \prec \{B^x \rightarrow A^z\}_D\Big) \;\wedge\; (g(A^z) \neq k)
\end{aligned}$$

We have arrived at a contradiction (i.e a false belief), and hence this case cannot occur.

Case 2: Block A was freed and written to disk after Block B was written. If block A was freed and written to disk, it should not have a valid backpointer. However, A has a backpointer to B. Thus we arrive at a contradiction.

$$\begin{aligned}
\Rightarrow \quad & \Big(\{B^x \rightarrow A_k\}_M \prec (\{B^x \rightarrow A^z\}_D \\
& \wedge(\&A = \phi) \wedge \mathit{write}(a))\Big) \;\wedge\; (g(A^z) \neq k) \\
\Rightarrow \quad & \Big(\{B^x \rightarrow A_k\}_M \prec (\{B^x \dashleftarrow A^z\}_D \\
& \wedge(\&A = \phi) \wedge \mathit{write}(a))) \wedge (g(A^z) \neq k) \text{ (Using (2))} \\
\Rightarrow \quad & \Big(\{B^x \rightarrow A_k\}_M \prec (\{B^x \dashleftarrow A^z\}_D \\
& \wedge\neg\{B^x \dashleftarrow A^z\}_D)\Big) \wedge (g(A^z) \neq k) \\
& (\text{ Since } \{\&A = \phi\}_D \Rightarrow \forall c \neg\{c \dashleftarrow A\}_D) \\
\Rightarrow \quad & \Big(\{B^x \rightarrow A_k\}_M \prec \mathit{false}\Big) \wedge (g(A^z) \neq k)
\end{aligned}$$

We have arrived at a contradiction, and hence this case cannot occur.

Thus we have shown that data consistency holds given that the file system has the backpointer property.

5.6.4 Version Consistency

Version consistency is a stricter version of data consistency. In version consistency, each file-system object and forward pointer has a timestamp associated with it. Each data block has a timestamp indicating when it was last updated. A corresponding timestamp is also stored in the inode, with the pointer to the data block. When a block is accessed, the timestamp in the inode and the data block must match. This helps us detect lost updates to data blocks. This is reflected in the rule:

$$\{B^x \to A^y\}_D \Rightarrow \{B^x \to A^y\}_M \prec \{B^x \to A^y\}_D$$

For the L.H.S to hold on disk, writes to both B and A need to have happened. This could have happened in two ways, considering the two possible orderings of the writes to A and B:

$$\begin{aligned}&\{B^x \to A^y\}_D \Rightarrow (\{B^x \to A^b\}_M \prec write(B))\\&\vee (\{B^a \to A^y\}_M \prec write(A))\end{aligned}$$

where a and b are arbitrary epochs of containers B and A.

Consider the first case:

$$\{B^x \to A^y\}_D \Rightarrow (\{B^x \to A^b\}_M \prec write(B))$$

Now, for the memory and on-disk copies of A to match, we need to prove that b = y:

$$\begin{aligned}&\{B^x \to A^b\}_M \Rightarrow ts(B^x, A) = t(A^b)\\&\{B^x \to A^y\}_D \Rightarrow ts(B^x, A) = t(A^y)\\&\Rightarrow t(A^b) = t(A^y)\\&\Rightarrow b = y\end{aligned}$$

Similarly, for Case 2, we can prove that a = x. Hence, when the file system uses back pointers with timestamps, we have shown that version consistency holds.

5.7 Conclusion

Every modern file system uses ordering points to ensure consistency. However, ordering points have many disadvantages including lower performance, higher complexity in file-system code, and dependence on lower layers of the storage stack to enforce ordering of writes.

In this chapter, we describe how to build an order-less file system, NoFS, that provides consistency without sacrificing simplicity, availability or performance. NoFS allows immediate data access upon mounting, without file-system checks. We show that NoFS has excellent performance on many workloads, outperforming ext3 on workloads that frequently flush data to disk explicitly.

Although potentially useful for the desktop, we believe NoFS may be of special significance in cloud computing platforms, where many virtual machines are multiplexed onto a physical device. In such cases, the underlying host operating system may try to batch writes together for performance, potentially ignoring ordering requests from virtual machines. NoFS allows virtual machines to maintain consistency without depending on the numerous lower layers of software and hardware. Removing such trust is key to building more robust and reliable storage systems.

The source code for NoFS can be obtained at: `http://www.cs.wisc.edu/adsl/Software/nofs`. We hope that this will encourage adoption of backpointer-based consistency.

6 The Optimistic File System

In this chapter, we present Optimistic Crash Consistency [46], a new crash consistency protocol that improves performance for several workloads significantly while providing strong crash-consistency guarantees. Central to the performance benefits of OptFS is the separation of ordering and durability. By allowing applications to order writes without incurring a disk flush, and request durability when needed, OptFS enables application-level consistency at high performance. OptFS introduces two new file-system primitives: `osync()`, which ensures ordering between writes but only *eventual* durability, and `dsync()`, which ensures immediate durability as well as ordering. Optimistic Crash Consistency tries to obtain most of the benefits of Backpointer-Based Consistency (presented in Chapter 5), while still supporting an API like `rename()` that is critical for today's applications. This chapter is based upon the paper, *Optimistic Crash Consistency* [46], published in SOSP 13.

First, we present our goals for Optimistic Crash Consistency (§6.1). We then describe the design of the *Optimistic File System (OptFS)*, which builds upon the principles of optimistic crash consistency (§6.2). We describe how we implemented OptFS (§6.3), and evaluate its performance (§6.4). We show that `osync()` provides a useful base on which to build higher-level application consistency semantics (§6.5), and finally conclude (§6.6).

6.1 Goals

While the No-Order File System (NoFS) (§5) establishes that a file system can maintain crash consistency without requiring any ordering barriers or flushes, it loses out on certain features. For example, atomic primitives such as `rename()` are not available on NoFS.

A number of applications use primitives such as `rename()` to atomically update their files. Indeed, `rename()` is the only primitive available in many systems to atomically update files. Thus, although NoFS provides strong file-system crash consistency, it does not enable application-level crash consistency.

We wanted to overcome this limitation in the Optimistic File System. At the same time, we maintain the No-Order File System's goal of eliminating or reducing disk-cache flushes required for crash consistency. Thus, we end up with two goals for Optimistic Crash Consistency:

1. Eliminate disk-cache flushes in the common case. If the user requests durability (via `fsync()`), data should be forced to disk. In most other cases, disk flushes should not be issued.

2. Provide primitives that can be used to build efficient application-level crash consistency. From Chapter 3, we know that applications require a means to cheaply order their writes. We seek to satisfy this need in Optimistic Crash Consistency.

6.2 Optimistic Crash Consistency

Given that journaling with probabilistic consistency often gives consistent results even in the presence of system crashes (§3.3), we note a new opportunity. The goal of *optimistic crash consistency*, as realized in an *optimistic journaling* system, is to commit transactions to persistent storage in a manner that maintains consistency to the same extent as pessimistic journaling, but with nearly the same performance as with probabilistic consistency. Optimistic journaling requires minimal changes to current disk interfaces and the journaling layer; in particular, our approach does not require changes to file-system structures outside of the journal (*e.g.*, backpointers [47]).

To describe optimistic crash consistency and journaling, we begin by describing the intuition behind optimistic techniques. Optimistic crash consistency is based on two main ideas. First, checksums can remove the need for ordering writes. Optimistic crash consistency eliminates the need for ordering during transaction commit by generalizing metadata transactional checksums [222] to include data blocks. During recovery, transactions are discarded upon checksum mismatch.

Second, *asynchronous durability notifications* are used to delay checkpointing a transaction until it has been committed durably. Fortunately, this delay does not affect application performance, as applications block until the transaction is committed, not until it is checkpointed. Additional techniques are required for correctness in scenarios such as block reuse and overwrite.

We first propose an additional notification that disk drives should expose. We then explain how optimistic journaling provides different properties to preserve the consistency semantics of ordered journaling. We show that these properties can be achieved using a combination of optimistic techniques. We also describe an additional optimistic technique which enables optimistic journaling to provide consistency equivalent to data journaling.

6.2.1 Asynchronous Durability Notification

The current interface to the disk for ensuring that write operations are performed in a specified order is pessimistic: the upper-level file system tells the lower-level disk when it must flush its cache (or certain blocks) and the disk must then promptly do so. However, the actual ordering and durability of writes to the platter does not matter, unless there is a crash. Therefore, the current interface is overly constraining and limits I/O performance.

Rather than requiring the disk to obey ordering and durability commands from the layer above, we propose that the disk be freed to perform reads and writes in the order that optimizes its scheduling and performance. Thus, the performance of the disk is optimized for the common case in which there is no crash.

Given that the file system must still be able to guarantee consistency and durability in the event of a crash, we propose a minimal extension to the disk interface. With an *asynchronous durability notification* the disk informs the upper-level client that a specific write request has completed and is now guaranteed to be durable. Thus there will be two notifications from the disk: first when the disk has received the write and later when the write has been persisted. Some interfaces, such as Forced Unit Access (FUA), provide a single, *synchronous* durability notification: the drive receives the request and indicates completion when the request has been persisted [137, 304]. Tagged Queuing allows a limited number of requests to be outstanding at a given point of time [140, 304]. Unfortunately, many drives do not implement tagged queuing and FUA reliably [174]. Furthermore, a request tagged with FUA also implies urgency, prompting some implementations to force the request to disk immediately. While a correct implementation of tagged queuing and FUA may suffice for optimistic crash consistency, we feel that an interface that decouples request acknowledgement from persistence enables higher levels of I/O concurrency and thus provides a better foundation on which to build OptFS.

6.2.2 Optimistic Consistency Properties

As described in Section 2.8, ordered journaling mode involves the following writes for each transaction: data blocks, D, to in-place locations on disk; metadata blocks to the journal, J_M; a commit block to the journal, J_C; and finally, a checkpoint of the metadata to its in-place location, M. We refer to writes belonging to a particular transaction i with the notation $: i$; for example, the journaled metadata of transaction i is denoted $J_M : i$.

Ordered journaling mode ensures several properties. First, metadata written in transaction $Tx: i + 1$ cannot be observed unless metadata from transaction $Tx: i$ is also observed. Second, it is not possible for metadata to point to invalid data. These properties are maintained by the recovery process and how writes are ordered. If a crash occurs after the transaction is properly committed (*i.e.*, , D, J_M, and J_C are all durably written), but before M is written, then the recovery process can replay the transaction so that M is written to its in-place location. If a crash occurs before the transaction is completed, then ordered journaling ensures that no in-place metadata related to this transaction was updated.

Optimistic journaling allows the disk to perform writes in any order it chooses, but ensures that in the case of a crash, the necessary consistency properties are upheld for ordered transactions. To give the reader some intuition for why particular properties are sufficient for ordered journaling semantics, we walk through the example in Figure 6.1. For simplicity, we begin by assuming that data blocks are not reused across transactions (*i.e.*, , they are not freed and re-allocated to different files or overwritten); we will remove this assumption later (§6.2.3).

In Figure 6.1, four transactions are in progress: $Tx: 0$, $Tx: 1$, and $Tx: 2$, and $Tx: 3$. At this point, the file system has received notification that $Tx: 0$ is durable (*i.e.*, , $D: 0$, $J_M : 0$, and $J_C : 0$) and so it is in the process of checkpointing the metadata $M: 0$ to its in-place location on disk (note that $M: 0$ may point to data $D: 0$). If there is a crash at this point, the recovery

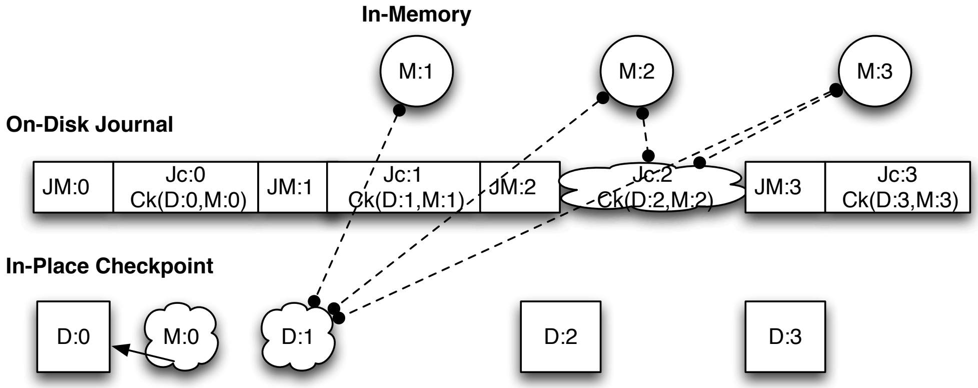


Figure 6.1: **Optimistic Journaling.** *The figure shows four transactions in progress, involving writes to main memory, the on-disk journal, and to in-place checkpoints on disk. A rectangle block indicates that the file system has been notified that the write has been durably completed. Cloud-shaped blocks indicate that the write has been initiated, but the file system has not yet been notified of its completion and it may or may not be durable. Circles indicate dirty blocks in main memory that cannot be written until a previous write is durable; a dashed line indicates the write it is dependent on. Finally, the solid arrow indicates that the meta-data may refer to the data block.*

mechanism will properly replay Tx: 0 and re-initiate the checkpoint of M: 0. Since Tx: 0 is durable, the application that initiated these writes can be notified that the writes have completed (*e.g.*, if it called `fsync()`). Note that the journal entries for Tx: 0 can finally be freed once the file system has been notified that the in-place checkpoint write of M: 0 is durable.

The file system has also started transactions Tx: 1 through Tx: 3; many of the corresponding disk writes have been initiated, while others are being held in memory based on unresolved dependencies. Specifically, the writes for D: 1, J_M: 1, and J_C: 1 have been initiated; however, D: 1 is not yet durable, and therefore the metadata (M: 1), which may refer to it, cannot be checkpointed. If M: 1 were checkpointed at this point and a crash occurred (with M: 1 being persisted and D: 1 not), M: 1 could be left pointing to garbage values for D: 1. If a crash occurs now, before D: 1 is durable, checksums added to the commit block of Tx: 1 will indicate a mismatch with D: 1; the recovery process will not replay Tx: 1, as desired.

Tx: 2 is allowed to proceed in parallel with Tx: 1; in this case, the file system has not yet been notified that the journal commit block J_C: 2 has completed; again, since the transaction is not yet durable, metadata M: 2 cannot be checkpointed. If a crash occurs when J_C: 2 is not yet durable, then the recovery process will detect a mismatch between the data blocks and the checksums and not replay Tx: 2. Note that D: 2 may be durable at this point with no negative consequences because no metadata is allowed to refer to it yet, and thus it is not reachable.

Finally, Tx: 3 is also in progress. Even if the file system is notified that D: 3, J_M: 3, and J_C: 3 are all durable, the checkpoint of M: 3 cannot yet be initiated because essential writes in Tx: 1 and Tx: 2 are not durable (namely, D: 1 and J_C: 2). Tx: 3 cannot be made durable until all previous transactions are guaranteed to be durable; therefore, its metadata M: 3 cannot be checkpointed.

6.2.3 Optimistic Techniques

The behavior of optimistic journaling described above can be ensured with a set of optimistic techniques: in-order journal recovery and release, checksums, background writes after notification, reuse after notification, and selective data journaling. We now describe each.

In-Order Journal Recovery

The most basic technique for preserving the correct ordering of writes after a crash occurs during the journal recovery process itself. The recovery process reads the journal to observe which transactions were made durable and it simply discards or ignores any write operations that occurred out of the desired ordering.

The correction that optimistic journaling applies is to ensure that if any part of a transaction Tx: i was not correctly or completely made durable, then neither transaction $\mathsf{Tx}\colon i$ nor any following transaction $\mathsf{Tx}\colon j$ where $j > i$ is left durable. Thus, journal recovery must proceed in-order, sequentially scanning the journal and performing checkpoints in-order, stopping at the first transaction that is not complete upon disk. The in-order recovery process will use the checksums described below to determine if a transaction is written correctly and completely.

In-Order Journal Release

Given that completed, durable journal transactions define the write operations that are durable on disk, optimistic journaling must ensure that journal transactions are not freed (or overwritten) until all corresponding checkpoint writes (of metadata) are confirmed as durable.

Thus, optimistic journaling must wait until it has been notified by the disk that the checkpoint writes corresponding to this transaction are durable. At this point, optimistic journaling knows that the transaction need not be replayed if the system crashes; therefore, the transaction can be released. To preserve the property that $\mathsf{Tx}\colon i+1$ is made durable only if $\mathsf{Tx}\colon i$ is durable, transactions must be freed in order.

Checksums

Checksums are a well-known technique for detecting data corruption and lost writes [210, 265]. A checksum can also be used to detect whether or not a write related to a specific transaction has occurred. Specifically, checksums can optimistically "enforce" two orderings: that the journal commit block (J_C) persists only after metadata to the journal (J_M) and after data blocks to their in-place location (D). This technique for ensuring metadata is durably written to the journal in its entirety has been referred to as transactional checksumming [222]; in this approach, a checksum is calculated over J_M and placed in J_C. If a crash occurs during the commit process, the recovery procedure can reliably detect the mismatch between

J_M and the checksum in J_C and not replay that transaction (or any transactions following). To identify this particular instance of transactional checksumming we refer to it as *metadata transactional checksumming*.

A similar, but more involved, version of *data transactional checksumming* can be used to ensure that data blocks D are written in their entirety as part of the transaction. Collecting these data blocks and calculating their checksums as they are dirtied in main memory is more involved than performing the checksums over J_M, but the basic idea is the same. With the data checksums and their on-disk block addresses stored in J_C, the journal recovery process can abort transactions upon mismatch. Thus, data transactional checksums enable optimistic journaling to ensure that metadata is not checkpointed if the corresponding data blocks were not durably written.

Background Write after Notification

An important optimistic technique ensures that the checkpoint of the metadata (M) occurs after the preceding writes to the data and the journal (*i.e.*, , D, J_M, and J_C). While pessimistic journaling guaranteed this behavior with a flush after J_C, optimistic journaling explicitly postpones the checkpoint write of metadata M until it has been notified that all previous transactions have been durably completed. Note that it is not sufficient for M to occur after only J_C; D and J_M must precede M as well since optimistic journaling must ensure that the entire transaction is valid and can be replayed if any of the in-place metadata M is written. Similarly, $M\colon i+1$ must be postponed until all transactions $Tx\colon i$ have been durably committed to ensure that $M\colon i+1$ is not durable if $M\colon i$ cannot be replayed. We note that $M\colon i+1$ does not need to wait for $M\colon i$ to complete, but must instead wait for the responsible transaction $Tx\colon i$ to be durable.

Checkpointing is one of the few points in the optimistic journaling protocol where the file system must wait to issue a particular write until a specific set of writes have become durable. However, this particular waiting is not likely to impact performance because checkpointing occurs in the background. Subsequent writes by applications will be placed in later transactions and these journal updates can be written independently of any other outstanding writes; journal writes do not need to wait for previous checkpoints or transactions. Therefore, even applications waiting for journal writes (*e.g.*, by calling `fsync()`) will not observe the checkpoint latency. Checkpointing can occur in the foreground, and potentially block applications, if the journal is full or if there is memory pressure. Both these situations are uncommon given that today's systems have plentiful disk storage and memory.

For this reason, waiting for the asynchronous durability notification before a background checkpoint is fundamentally more powerful than the pessimistic approach of sending an ordering command to the disk (*i.e.*, , a cache flush). With a traditional ordering command, the disk is not able to postpone checkpoint writes across multiple transactions. On the other hand, the asynchronous durability notification command does not artificially constrain the ordering of writes and gives more flexibility to the disk so that it can best cache and schedule writes; the command also provides the needed information to the file system so that it can allow independent writes to proceed while appropriately holding back dependent writes.

Reuse after Notification

The preceding techniques were sufficient for handling the cases where blocks were not reused across transactions. The difficulty occurs with ordered journaling because data writes are performed to their in-place locations, potentially overwriting data on disk that is still referenced by durable metadata from previous transactions. Therefore, additional optimistic techniques are needed to ensure that durable metadata from earlier transactions never points to incorrect data blocks changed in later transactions. Block reuse can lead to a security issue: if user A deletes their file, and then the deleted block becomes part of user B's file, a crash should not lead to user A being able to view user B's data. Correct block reuse is one of the *update dependency* rules required for Soft Updates [251], but optimistic journaling enforces this rule with a different technique: reuse after notification.

To understand the motivation for this technique, consider the steps when a data block D_A is freed from one file M_A and allocated to another file, M_B and rewritten with the contents D_B. Depending on how writes are re-ordered to disk, a durable version of M_A may point to the erroneous content of D_B.

This problem can be fixed with transactions as follows. First, the freeing of D_A and update to M_A, denoted $M_{A'}$, is written as part of a transaction $J_{M_{A'}}: i$; the allocation of D_B to M_B is written in a later transaction as $D_B: i+1$ and $J_{M_B}: i+1$. Pessimistic journaling ensures that $J_{M_{A'}}: i$ occurs before $D_B: i+1$ with a traditional flush between every transaction. The optimistic techniques introduced so far are not sufficient to provide this guarantee because the writes to $D_B: i+1$ in their in-place locations cannot be recovered or rolled back if M'_A is lost (even if there is a checksum mismatch and transactions $Tx: i$ or $Tx: i+1$ are found to be incomplete).

Optimistic journaling guarantees that $J_{M_{A'}}: i$ occurs before $D_B: i+1$ by ensuring that data block D_A is not re-allocated to another file until the file system has been notified by the disk that $J_{M_{A'}}: i$ has been durably written; at this point, the data block D_A is "durably free." When the file M_B must be allocated a new data block, the optimistic file system allocates a "durably-free" data block that is known to not be referenced by any other files; finding a durably-free data block is straight-forward given the proposed asynchronous durability notification from disks.

Performing reuse only after notification is unlikely to cause the file system to wait or to harm performance. Unless the file system is nearly 100% full, there should always exist a list of data blocks that are known to be durably free; under normal operating conditions, the file system is unlikely to need to wait to be informed by the disk that a particular data block is available.

Selective Data Journaling

Our final optimistic technique selectively journals data blocks that have been overwritten. This technique allows optimistic journaling to the strong consistency semantics of data journaling instead of the weaker (but default) ordered-journaling semantics.

A special case of an update dependency occurs when a data block is overwritten in a file and the metadata for that file (*e.g.*, size) must be updated consistently. Optimistic journaling could handle this using reuse after notification: a *new* durably-free data block is allocated to the file and written to disk ($D_B : j$), and then the new metadata is journaled ($J_{M_B} : j$). The drawback of this approach is that the file system takes on the behavior of a copy-on-write file system and loses some of the locality benefits of an update-in-place file system [215]; since optimistic journaling forces a durably-free data block to be allocated, a file that was originally allocated contiguously and provided high sequential read and write throughput may lose its locality for certain random-update workloads.

If update-in-place is desired for performance, a different technique can be used: *selective* data journaling. Data journaling places both metadata and data in the journal and both are then updated in-place during checkpointing. Data journaling is attractive because in-place data blocks are not overwritten until the transaction is checkpointed; therefore, data blocks can be reused if their metadata is also updated in the same transaction. The disadvantage of data journaling is that every data block is written twice (once in the journal, J_D, and once in its checkpointed in-place location, D)

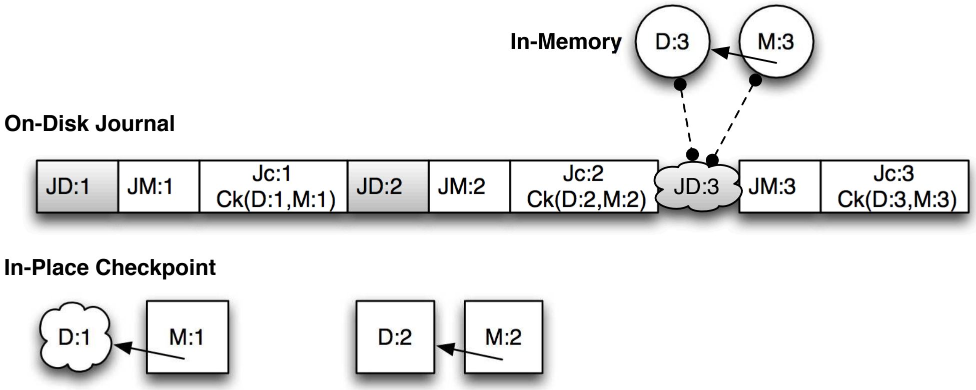


Figure 6.2: **Optimistic Journaling: Selective Data Journaling.** *The figure shows that selective data journaling may be used when transactions involve overwriting in-place data. Data blocks are now placed in the journal and checkpointed after the transaction is committed.*

and therefore often has worse performance than ordered journaling [220].

Selective data journaling allows ordered journaling to be used for the common case and data journaling only when data blocks are repeatedly overwritten within the same file and the file needs to maintain its original layout on disk. In selective data journaling, the checkpoint of both D and M simply waits for durable notification of all the journal writes (J_D, J_M, and J_C).

Figure 6.2 shows an example of how selective data journaling can be used to support overwrite, in particular the case where blocks are reused from previous transactions without clearing the original references to those blocks. In this example, data blocks for three files have been overwritten in three separate transactions.

The first transaction illustrates how optimistic ordering ensures that durable metadata does not point to garbage data. After the file system has been notified of the durability of $Tx\colon 1$ (specifically, of $J_D\colon 1$, $J_M\colon 1$, and $J_C\colon 1$), it may checkpoint both $D\colon 1$ and $M\colon 1$ to their in-place locations. Because the file system can write $M\colon 1$ without waiting for a durable notification of $D\colon 1$, in the case of crash it is possible for $M\colon 1$ to refer to garbage values in D: 1; however, the recovery process will identify this situation due to the checksum mismatch and replay Tx: 1 with the correct values for D: 1.

The second and third transactions illustrate how optimistic ordering ensures that later writes are visible only if all earlier writes are visible as well. Specifically, D: 2 and M: 2 have been checkpointed, but only because both Tx: 2 and Tx: 1 are both durable; therefore, a client cannot see new contents for the second file without seeing new contents for the first file. Furthermore, neither D: 3 nor M: 3 (or any later transactions) can be checkpointed yet because not all blocks of its transaction are known to be durable. Thus, selective data journaling provides the desired consistency semantics while allowing overwrites.

6.2.4 Durability vs. Consistency

Optimistic journaling uses an array of novel techniques to ensure that writes to disk are properly ordered, or that enough information exists on disk to recover from an untimely crash when writes are issued out of order; the result is file-system consistency and proper ordering of writes, but without guarantees of durability. However, some applications may wish to force writes to stable storage for the sake of durability, not ordering. In this case, something more than optimistic ordering is needed; the file system must either wait for such writes to be persisted (via an asynchronous durability notification) or issue flushes to force the issue. To separate these cases, we believe two calls should be provided; an "ordering" sync, `osync()`, guarantees ordering between writes, while a "durability" sync, `dsync()`, ensures when it returns that pending writes have been persisted.

We now define and compare the guarantees given by `osync()` and `dsync()`. Assume the user makes a series of writes $W_1, W_2, ..., W_n$. If no `osync()` or `dsync()` calls are made, there is no guarantee as to file-system state after a crash: any or all of the updates may be lost, and updates may be applied out of order, *i.e.*, , W_2 may be applied without W_1.

Now consider when every write is followed by `dsync()`, *i.e.*, , W_1, $\mathtt{d}_1$, W_2, $\mathtt{d}_2$, ..., W_n, $\mathtt{d}_n$. If a crash happens after $\mathtt{d}_i$, the file system will recover to a state with $W_1, W_2, ..., W_i$ applied.

If every write was followed by `osync()`, *i.e.*, , $W_1, \mathtt{o}_1, W_2, \mathtt{o}_2, ..., W_n, \mathtt{o}_n$, and a crash happens after $\mathtt{o}_i$, the file system will recover to a state with $W_1, W_2, ..., W_{i-k}$ applied, where the last k writes had not been made durable before the crash. We term this *eventual* durability. Thus `osync()` provides *prefix semantics* [302], ensuring that users always see a consistent version of the file system, though the data may be stale. Prior research indicates that prefix semantics are useful in many domains [52].

6.3 Implementation of OptFS

We have implemented the Optimistic File System (OptFS) inside Linux 3.2, based on the principles outlined before (§6.2), as a variant of the ext4 file system, with additional changes to the JBD2 journaling layer and virtual memory subsystem.

6.3.1 Asynchronous Durability Notifications

Since current disks do not implement the proposed asynchronous durability notification interface, OptFS uses an approximation: *durability timeouts*. Durability timeouts represent the maximum time interval that the disk can delay committing a write request to the non-volatile platter. When a write for block A is received at time T by the disk, the block must be made durable by time $T + T_D$. The value of T_D is specific to each disk, and must be exported by the disk to higher levels in the storage stack.

Upon expiration of the time interval T_D, OptFS considers the block to be durable; this is equivalent to the disk notifying OptFS after T_D seconds. Note that to account for other delays in the I/O subsystem, T_D is measured from the time the write is acknowledged by the disk, and not from the time the write is issued by the file system.

This approximation is limiting; T_D might overestimate the durability interval, leading to performance problems and memory pressure; T_D might underestimate the durability interval, comprising consistency. OptFS errs towards safety and sets T_D to be 30 seconds.

6.3.2 Handling Data Blocks

OptFS does not journal *all* data blocks: newly allocated data blocks are only checksummed; their contents are not stored in the journal. This complicates journal recovery as data-block contents may change after the checksum was recorded. Due to selective data journaling, a data block that is not journaled (as it is newly allocated) in one transaction might be overwritten in the following transaction and therefore journaled. For example, consider data block D with content A belonging to Tx_1. The checksum A will be recorded in Tx_1. D is overwritten by Tx_2, with content B. Although this sequence is valid, the checksum A in Tx_1 will not match the content B in D.

This necessitates individual block checksums, since checksum mismatch of a single block is not a problem if the block belongs to a later valid transaction. In contrast, since the frozen state of metadata blocks are stored in the journal, checksumming over the entire set is sufficient for metadata transactional checksums. We explain how OptFS handles this during recovery shortly.

OptFS does not immediately write out checksummed data blocks; they are collected in memory and written in large batches upon transaction commit. This increases performance in some workloads.

6.3.3 Optimistic Techniques

We now describe the implementation of the optimistic journaling techniques. We also describe how OptFS reverts to more traditional mechanisms in some circumstances (*e.g.*, when the journal runs out of space).

In-Order Journal Recovery: OptFS recovers transactions in the order they were committed to the journal. A transaction can be replayed only if all its data blocks belong to valid transactions, and the checksum computed over metadata blocks matches that in the commit block.

OptFS performs recovery in two passes: the first pass linearly scans the journal, compiling a list of data blocks with checksum mismatches and the first journal transaction that contained each block. If a later valid transaction matches the block checksum, the block is deleted from the list. At the end of the scan, the earliest transaction in the list is noted. The next pass performs journal recovery until the faulting transaction, thus ensuring consistency.

OptFS journal recovery might take longer than ext4 recovery since OptFS might need to read data blocks off non-contiguous disk locations while ext4 only needs to read the contiguous journal to perform recovery.

In-Order Journal Release: When the virtual memory (VM) subsystem informs OptFS that checkpoint blocks have been acknowledged at time T, OptFS sets the transaction *cleanup* time as $\mathsf{T}+\mathsf{T}_\mathsf{D}$, after which it is freed from the journal.

When the journal is running out of space, it may not be optimal to wait for the durability timeout interval before freeing up journal blocks. Under memory pressure, OptFS may need to free memory buffers of checkpoint blocks that have been issued to the disk and are waiting for durability timeouts. In such cases, OptFS issues a disk flush, ensuring the durability of checkpoint blocks that have been acknowledged by the disk. This allows OptFS to clean journal blocks belonging to some checkpointed transactions and free associated memory buffers.

Checksums: OptFS checksums data blocks using the same CRC32 algorithm used for metadata. A tag is created for each data block which stores the block number and its checksum. Data tags are stored in the descriptor blocks along with tags for metadata checksums.

Background Write after Notification: OptFS uses the VM subsystem to perform background writes. Checkpoint metadata blocks are marked as dirty and the `expiry` field of each block is set to be $\mathsf{T}+\ \mathsf{T_D}$ (the disk acknowledged the commit block at T). T will reflect the time that the entire transaction has been acknowledged because the commit is not issued until the disk acknowledges data and metadata writes. The blocks are then handed off to the VM subsystem.

During periodic background writes, the VM subsystem checks if each dirty block has expired: if so, the block is written out; otherwise the VM subsystem rechecks the block on its next periodic write-out cycle.

Reuse after Notification: Upon transaction commit, OptFS adds deleted data blocks to a global in-memory list of blocks that will be freed after the durability time-out, $\mathsf{T_D}$. A background thread periodically frees blocks with expired durability timeouts. Upon file-system unmount, all list blocks are freed.

When the file system runs out of space, if the reuse list contains blocks, a disk flush is issued; this ensures the durability of transactions which freed the blocks in the list. These blocks are then set to be free. We expect that these “safety” flushes will be used infrequently.

Selective Data Journaling: Upon a block write, the block allocation information (which is reflected in `New` state of the `buffer_head`) is used to determine whether the block was newly allocated. If the write is an overwrite, the block is journaled as if it was metadata.

		Crash points	
Workload	**Delayed Blocks**	**Total**	**Consistent**
Append	Data	50	50
	J_M, J_C	50	50
	Multiple blocks	100	100
Overwrite	Data	50	50
	J_M, J_C	50	50
	Multiple blocks	100	100

Table 6.3: **Reliability Evaluation.** *The table shows the total number of simulated crashpoints, and the number of crashpoints resulting in a consistent state after remounting the file system and running recovery.*

6.4 Evaluation

We now evaluate our prototype implementation of OptFS on two axes: reliability and performance. Experiments were performed on an Intel Core i3-2100 CPU with 16 GB of memory, a Hitachi DeskStar 7K1000.B 1 TB drive, and running Linux 3.2. The experiments were repeatable; numbers reported are the average over 10 runs.

6.4.1 Reliability

To verify OptFS's consistency guarantees, we build and apply a crash-robustness framework to it under two synthetic workloads. The first appends blocks to the end of a file; the second overwrites blocks of an existing file; both issue `osync()` calls frequently to induce more ordering points and thus stress OptFS machinery.

Crash simulation is performed using the following steps:

1. Capture an image of the file-system in its initial state.

2. Run workload and trace all writes performed. The trace also includes write data.

3. Re-order the writes in the trace (while obeying flush semantics).
4. Pick a random point `P` in the trace.
5. Drop all writes after point `P` from the trace.
6. Apply writes from trace to initial file-system image to obtain a *crash* image
7. Mount file system from crash image and allow it to recover.

The resulting file-system state emulates file-system after a crash at a random point of time; at the point of the crash, some writes would have become durable (and therefore available after the crash), while other writes would be in the volatile disk cache (and therefore lost due to the crash).

Table 6.3 shows the results of 400 different crash scenarios. OptFS recovers correctly in each case to a file system with a consistent prefix of updates (§6.2.4).

6.4.2 Performance

We first present a figure that depicts how write requests pass through different layers in the storage stack for ext4 and OptFS. We then analyze the performance of OptFS and ext4 over several micro- and macro-benchmarks.

Illustrating OptFS I/O Handling. Figure 6.4 shows how write requests are processed in ext4 and OptFS, while running the Varmail workload. This figure allows us to intuitively understand why using `osync()` on OptFS is faster than using `fsync()` on ext4. ext4 issues a `FLUSH` to ensure that D and J_M are persisted before J_C – persisting all those writes requires 51.3 milliseconds. ext4 also issues a `FLUSH` to ensure checkpointing happens after J_C is persisted; this results in an additional 8.5 milliseconds. Overall, the application waits a total of 61.8 milliseconds before it can issue the

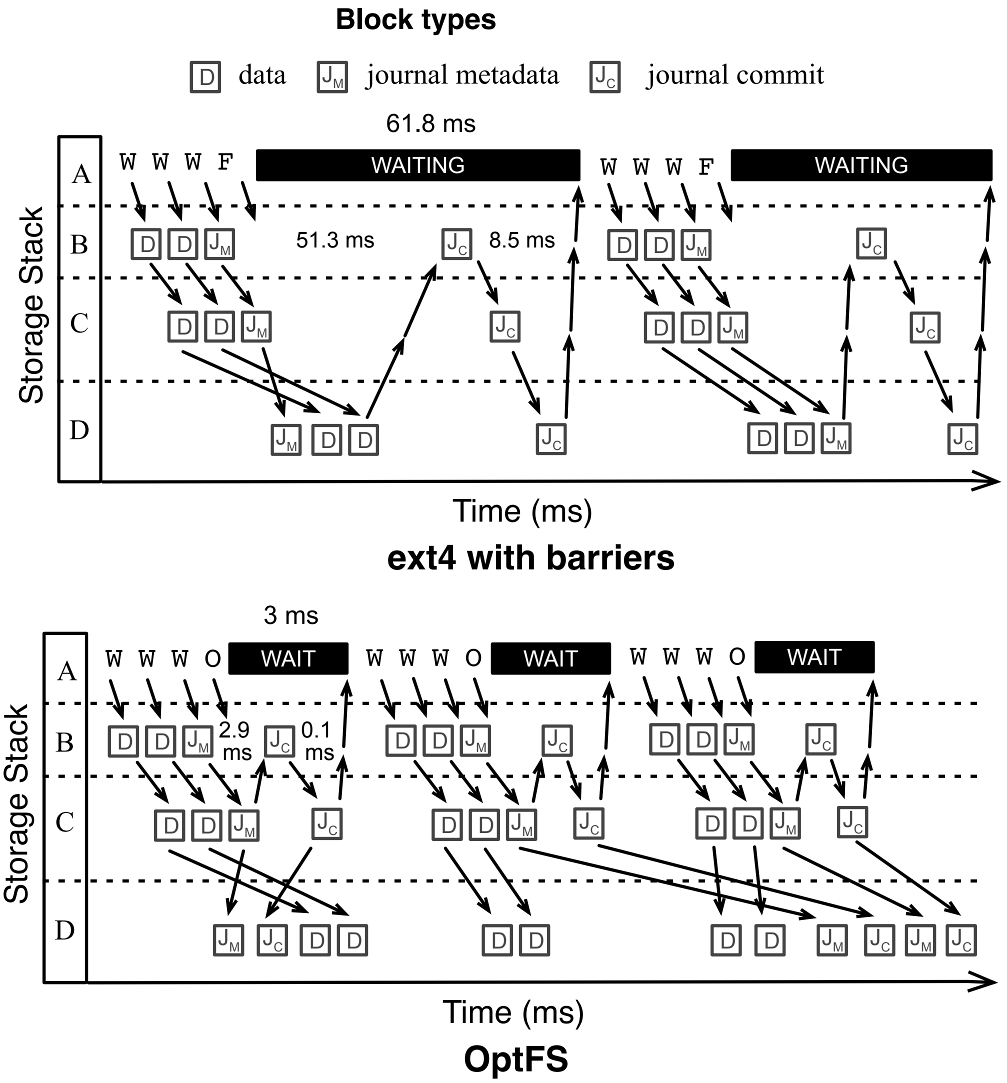


Figure 6.4: **I/O Timeline.** *The figure illustrates how I/O are handled in ext4 with flushes and OptFS. Legend: A – application; B – buffer cache; C – disk cache; D – disk platter; W – `write()` system call; F – `fsync()` system call; O – `osync()` system call. The workload is the Filebech Varmail benchmark that emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory. In ext4, the application is blocked while blocks are flushed to the disk platter; in OptFS, the application is blocked only until I/O hits the disk cache.*

next set of writes. In contrast, OptFS does not issue flushes when the application calls `osync()`; this results in the wait time reduced from 61.8 ms to a mere 3 ms (the time for the writes to propagate down the storage stack to the disk cache).

We analyze the performance of OptFS under a number of micro- and macro-benchmarks. Figure 6.5 illustrates OptFS performance under these workloads; details of the workloads are found in the caption.

Micro-benchmarks: OptFS sequential-write performance is similar to ordered mode; however, sequential overwrites cause bandwidth to drop to half of that of ordered mode as OptFS writes every block twice. Random writes on OptFS are 3x faster than on ext4 ordered mode, as OptFS converts random overwrites into sequential journal writes (due to selective data journaling). If the journal size is set to 1 GB, and 2 GB of random writes are issued, OptFS performs 15% worse than ordered mode due to journal checkpointing overhead.

On the Createfiles benchmark, OptFS performs 2x better than ext4 ordered mode, as ext4 writes dirty blocks in the background for a number of reasons (*e.g.*, hitting the threshold for the amount of dirty in-memory data), while OptFS consolidates the writes and issues them upon commit. When we modified ext4 to stop the background writes, its performance was similar to OptFS.

Macro-benchmarks: We run the Filebench Fileserver, Webserver, and Varmail workloads [171]. OptFS performs similarly to ext4 ordered mode without flushes for Fileserver and Webserver. Varmail's frequent `fsync()` calls cause a significant number of flushes, leading to OptFS performing 7x better than ext4. ext4, even with flushes disabled, does not perform as well as OptFS since OptFS delays writing dirty blocks, issuing them in large batches periodically or on commit; in contrast, the background `pdflush` threads issue writes in small batches so as to not affect foreground activity.

Finally, we run the MySQL OLTP benchmark from Sysbench [12] to

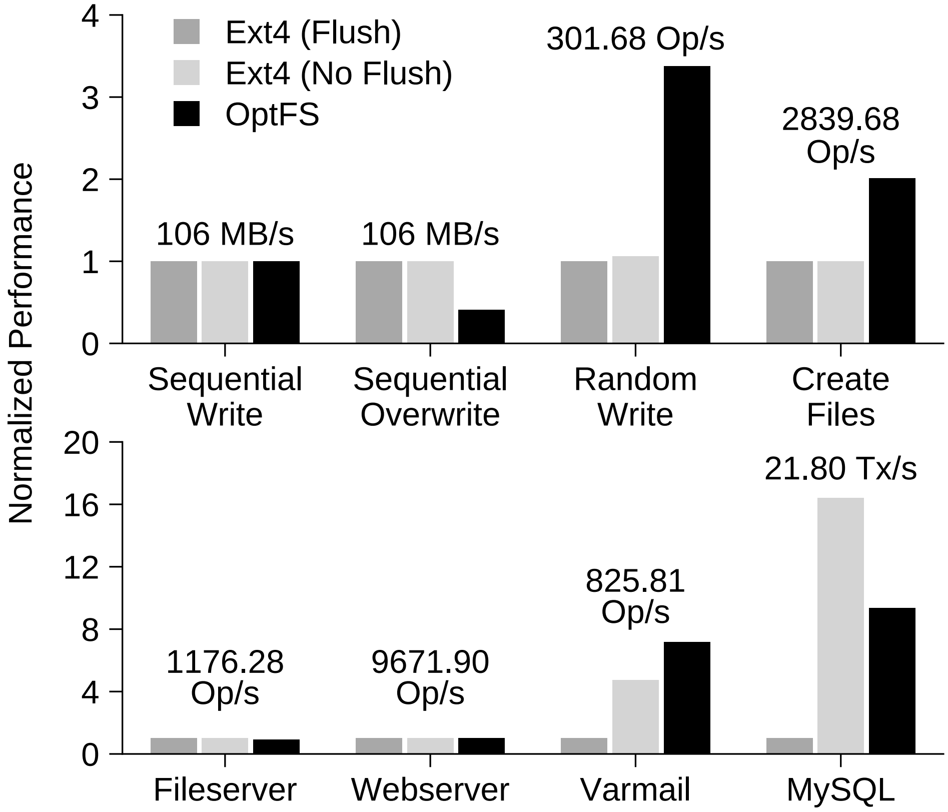


Figure 6.5: **Performance Comparison.** *Performance is shown normalized to ext4 ordered mode with flushes. The absolute performance of ordered mode with flushes is shown above each workload. Sequential writes are to 80 GB files. 200K random writes are performed over a 10 GB file, with an `fsync()` every 1K writes. The overwrite benchmark sequentially writes over a 32 GB file. Createfiles uses 64 threads to create 1M files. Fileserver emulates file-server activity, using 50 threads to perform a sequence of creates, deletes, appends, reads, and writes. Webserver emulates a multi-threaded web host server, performing sequences of open-read-close on multiple files plus a log file append, with 100 threads. Varmail emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory. Each workload was run for 660 seconds. MySQL OLTP benchmark performs 200K queries over a table with 1M rows.*

investigate the performance on database workloads. OptFS performs 10x better than ordered mode with flushes, and 40% worse than ordered mode without flushes (due to the many in-place overwrites of MySQL, which result in selective data journaling).

Background Writes: On the Createfiles and Varmail benchmark, we notice that even after disabling flushes, ext4 does not perform as well as OptFS. This is due to a difference in how dirty data in the buffer cache is written out to disk. The `pdflush` daemon threads are responsible for writing out dirty data. Data writeout is affected by a number of system-wide parameters in `/proc/sys/vm`. Reproducing the three most significant parameters from the kernel documentation:

dirty_ratio. Contains, as a percentage of total available memory that contains free pages and reclaimable pages, the number of pages at which a process which is generating disk writes will itself start writing out dirty data. The ratio could also be expressed as a number of bytes with *dirty_bytes*. The default is 10 percent.

dirty_background_ratio. Contains, as a percentage of total available memory that contains free pages and reclaimable pages, the number of pages at which the background kernel flusher threads will start writing out dirty data. The default is 5 percent.

dirty_expire_centisecs. This tunable is used to define when dirty data is old enough to be eligible for writeout by the kernel flusher threads. It is expressed in 100'ths of a second. Data which has been dirty in-memory for longer than this interval will be written out next time a flusher thread wakes up. The default is 30 seconds.

Table 6.6 shows how file-system performance is affected when these parameters are tuned. The workload used is Filebench Createfiles. As `dirty_ratio` and `dirty_background_ratio` are increased to make the background threads lazier, ext4 performance gets closer to OptFS performance. With values of 90 for both these parameters, ext4 performance

Setup			Ratio:
DBR	DR	DEC (s)	OptFS / ext4
5	10	5	2.02 (default)
50	50	90	1.55
70	70	270	1.29
80	80	540	1.15
90	90	270	0.92

Table 6.6: **Dirty Block Background Writeout.** *The table shows how tuning dirty block writeout by `pdflush` threads affects file-system performance for the Filebench Createfiles workload. OptFS writes out dirty blocks lazily, collecting them into big batches. ext4 dirty data is written out eagerly (when 5 percent of memory is dirty) by background threads. When the background threads are tuned to behave more lazily, ext4 performance is similar to OptFS. Ext4 performs better when the background threads batch writes more than OptFS. Legend: DBR – Dirty Background Ratio. DR – Dirty Ratio. DEC – Dirty Expire Centiseconds (shown here in seconds).*

is similar to OptFS performance. Although `dirty_expire_centisecs` affects background writeout behavior (blocks are not written out until they have been dirty for as long as this parameter indicates), it is not the determining factor for file-system performance. We have not explored the relation between these parameters and their effect on file-system performance deeply.

Summary: OptFS significantly outperforms ordered mode with flushes on most workloads, providing the same level of consistency at considerably lower cost. On many workloads, OptFS performs as well as ordered mode without flushes, which offers no consistency guarantees. OptFS may not be suited for workloads which consist mainly of sequential overwrites.

6.4.3 Resource consumption

Table 6.8 compares the resource consumption of OptFS and ext4 for a 660 second run of Varmail. OptFS consumes 10% more CPU than ext4

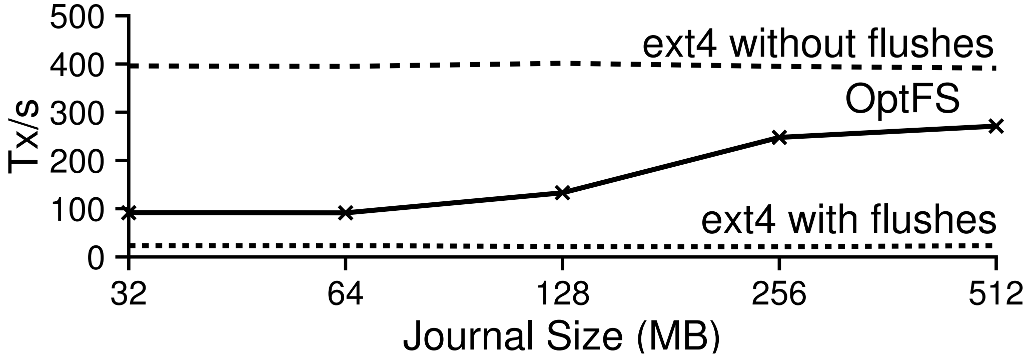


Figure 6.7: **Performance with Small Journals.** *The figure shows the variation in OptFS performance on the MySQL OLTP benchmark as the journal size is varied. When OptFS runs out of journal space, it issues flushes to safely checkpoint transactions. Note that even in this stress scenario, OptFS performance is 5x better than ext4 ordered mode with flushes.*

ordered mode without flushes. Some of this overhead in our prototype can be attributed to CRC32 data checksumming and the background thread which frees data blocks with expired durability timeouts, though further investigation is required.

OptFS delays checkpointing and holds metadata in memory longer than ext4, thereby increasing memory consumption. Moreover, OptFS delays data writes until transaction commit time, increasing performance for some workloads (*e.g.*, Filebench Createfiles), but at the cost of additional memory load.

6.4.4 Journal size

When OptFS runs out of journal space, it issues flushes in order to checkpoint transactions and free journal blocks. To investigate the performance of OptFS in such a situation, we reduce the journal size and run the MySQL OLTP benchmark. The results are shown in Figure 6.7. Note that due to selective data journaling, OptFS performance will be lower than that of ext4 without flushes, even with large journals. We find that OptFS performs well with reasonably sized journals of 512 MB and greater; only with smaller journal sizes does performance degrade near the level of ext4 with flushes.

6.5 Case Studies

We now show how to use OptFS ordering guarantees to provide meaningful application-level crash consistency. Specifically, we study atomic updates in a text editor (gedit) and logging within a database (SQLite).

6.5.1 Atomic Update within Gedit

Many applications atomically update a file with the following sequence: first, create a new version of the file under a temporary name; second, call `fsync()` on the file to force it to disk; third, rename the file to the desired file name, replacing the original atomically with the new contents (some applications issue another `fsync()` to the parent directory, to persist the name change). The gedit text editor, which we study here, performs this sequence to update a file, ensuring that either the old or new contents are in place in their entirety, but never a mix.

To study the effectiveness of OptFS for this usage, we modified gedit to use `osync()` instead of `fsync()`, thus ensuring order is preserved. We then take a block-level I/O trace when running under both ext4 (with and without flushes) and OptFS, and simulate a large number of crash points and I/O re-orderings in order to determine what happens during a crash. Specifically, we create a disk image that corresponds to a particular subset of writes taking place before a crash; we then mount the image, run recovery, and test for correctness.

Table 6.9 shows our results. With OptFS, the saved filename always points to either the old or new versions of the data in their entirety; atomic

File system	**CPU %**	**Memory (MB)**
ext4 ordered mode with flushes	3.39	486.75
ext4 ordered mode without flushes	14.86	516.03
OptFS	25.32	749.4

Table 6.8: **Resource Consumption.** *The table shows the average resource consumption by OptFS and ext4 ordered mode for a 660 second run of Filebench Varmail. OptFS incurs additional overhead due to optimistic techniques such as data transactional checksumming.*

	gedit			**SQLite**		
	ext4 w/o flush	**ext4 w/ flush**	**OptFS**	**ext4 w/o flush**	**ext4 w/ flush**	**OptFS**
Total crashpoints	50	50	50	100	100	100
Inconsistent	7	0	0	73	0	0
Old state	26	21	36	8	50	76
New state	17	29	14	19	50	24
Time per op (ms)	1.12	39.4	0.84	23.38	152	15.3

Table 6.9: **Case Study: Gedit and SQLite.** *The table shows the number of simulated crashpoints that resulted in a consistent or inconsistent application state after remounting. It also shows the time required for an application operation.*

update is achieved. In a fair number of crashes, old contents are recovered, as OptFS delays writing updates; this is the basic trade-off OptFS makes, increasing performance but delaying durability. With ext4 (without flush), a significant number of crashpoints resulted in inconsistencies, including unmountable file systems and corrupt data. As expected, ext4 (with flush) did better, resulting in new or old contents exactly as dictated by the `fsync()` boundary.

The last row of Table 6.9 compares the performance of atomic updates in OptFS and ext4. OptFS delivers performance similar to ext4 without flushes, roughly 40x faster per operation than ext4 with flushes.

6.5.2 Temporary Logging in SQLite

We now investigate the use of osync() in a database management system, SQLite. To implement ACID transactions, SQLite first creates a temporary log file, writes some information to it, and calls fsync(). SQLite then updates the database file in place, calls fsync() on the database file, and finally deletes the log file. After a crash, if a log file is present, SQLite uses the log file for recovery (if necessary); the database is guaranteed to be recovered to either the pre- or post-transaction state.

Although SQLite transactions provide durability by default, its developers assert that many situations do not require it, and that "sync" can be replaced with pure ordering points in such cases. The following is an excerpt from the SQLite documentation [262]:

> As long as all writes that occur before the sync are completed before any write that happens after the sync, no database corruption will occur. [...] *the database will at least continue to be consistent, and that is what most people care about* (emphasis ours).

We now conduct the same consistency and performance tests for SQLite. With a small set of tables (≈30KB in size), we create a transaction to move records from one half the tables to the other half. After a simulated disk image that corresponds to a particular crash-point is mounted, if consistent, SQLite should recover the database to either the pre-transaction (old) or post-transaction (new) state. The results in Table 6.9 are similar to the gedit case-study: OptFS always results in a consistent state, while ext4 (without flushes) does not. OptFS performs 10x better than ext4 with flushes, while providing the same level of consistency.

6.6 Conclusion

We presented Optimistic Crash Consistency, a new approach to maintaining crash consistency in file systems. In contrast to standard pessimistic consistency techniques (such as journaling), which always flush the disk cache to order writes, optimistic crash consistency takes advantage of the fact that crashes are rare, and that ordering can be achieved by several light-weight mechanisms (such as checksums).

Optimistic Crash Consistency allows the file system to write to storage in any order (with the exception of checkpoint writes). Since the storage controller is no longer constrained by write orders, it can write to the storage in the manner that maximizes performance. By speculatively persisting writes, and making writes visible to the user only at well-defined points, Optimistic Crash Consistency allows the file system to obtain excellent performance and strong crash-consistency guarantees.

We implemented the principles of Optimistic Crash Consistency in the Optimistic File System (OptFS), a variant of the ext4 journaling file system. We emulate random crashes and show that OptFS provides strong consistency guarantees. We evaluate OptFS performance using a variety of micro- and macro-benchmarks, and show that OptFS performance is significant higher (an order of magnitude in some cases) than the default mode of ext4.

We introduce two new file-system primitives, `osync()` and `dsync()`, which decouple ordering from durability. We study two applications, `Gedit` and `SQLite`, and show that both applications can employ the new primitives to obtain high performance and meaningful semantics. We believe that such decoupling holds the key to resolving the constant tension between consistency and performance in file systems.

The source code for OptFS can be obtained at: `http://www.cs.wisc.edu/adsl/Software/optfs`. We hope that this will encourage adoption of the optimistic approach to consistency.

7 Discussion

We now discuss how the systems built as part of this dissertation compare with each other, and how techniques presented in this dissertation can be used in other contexts. First, we compare NoFS and OptFS among several axes (§7.1). We then describe how optimistic techniques could be used in other contexts (such as on new storage technologies) (§7.2). We then discuss the challenges in using the `osync()` system call (§7.3). Finally, we describe our interactions with industry about the new interfaces proposed in this dissertation (§7.4).

7.1 Comparing NoFS and OptFS

NoFS and OptFS were designed to satisfy different goals: NoFS was designed to separate crash consistency from I/O ordering, while OptFS was designed to separate crash consistency from I/O durability. Thus, they involve different design decisions. We now compare them along several axes: hardware requirements, usability, and performance.

7.1.1 Hardware Requirements

NoFS and OptFS require different hardware support. NoFS requires that the storage device provides the ability to write an 8-byte backpointer and a 4K block atomically. OptFS requires that the drive informs the file system when submitted writes have become durable.

SCSI drives provide the Data Integrity Field, atomically writing 8 bytes of extra data with each 512 byte sector [274]. Seagate offers drives that store 4K block plus a few bytes of additional data integrity information in an atomic fashion [246]. However, the extra 8 bytes are usually used by the device driver to write checksums or similar redundant information; current device drivers do not allow the developer to write arbitrary content into the 8 bytes.

Although we have described the most general and ideal version of asynchronous durability notifications in this work, there are several ways in which durability notifications could be realized on storage systems. Implementing durability notifications does not require that the hardware be modified; for example, one could envision a service that keeps track of the durability of in-flight writes, issues flushes periodically, and sends durability notifications to the file system. We have not explored how best to implement durability notifications in software.

In summary, we believe it will be possible to run both OptFS and NoFS on today's hard drives. NoFS will require writing new device drivers, while OptFS will require writing new software layers.

7.1.2 Usability

An important feature missing in NoFS is the atomic `rename()` system call that is used to atomically update files. Atomic rename is critical in the following update sequence:

1. Create temporary file
2. Write temporary file with the entire contents of the old file, plus updates
3. Persist temporary file via `fsync()`
4. Atomically rename temporary file over old file.

At the end of this sequence, the file should exist in either the old state or the new state with all the updates. If `rename()` is not atomic, or operations can be re-ordered, the entire file could be lost due to an inopportune crash (something that has been observed in deployment [59]). Thus, hundreds of applications will have to rewritten to achieve atomicity in a different manner on NoFS.

OptFS supports `rename()` since journal transactions are applied atomically to the file system (all `rename()` changes end up in the same transaction). By providing both `osync()` and `dsync()`, applications can choose at different points to achieve ordering or durability. Applications will still have to rewritten to take advantage of `osync()`, but current applications will run correctly on top of OptFS.

7.1.3 Performance

A straight-forward comparison of the performance is NoFS and OptFS is challenging as they provide different semantics. For example, a user cannot order writes in NoFS. The best way to compare is consider a use case where the application does not need to order any writes, and hence never calls `osync()` (or `dsync()` for ordering purposes). Furthermore, consider that OptFS uses extremely large transactions. Thus, all application writes in OptFS will be part of one big journal transaction, with no additional transactions for ordering.

In this example, the primary difference between NoFS and OptFS is that all writes go to the journal in OptFS, whereas they are written in-place in NoFS. This will cause performance differences if the checkpoint traffic in OptFS interferes with the journal writes. For example, for the MySQL OLTP benchmark, selective data journaling causes two writes every time a file block is overwritten. Such double writes result in OptFS performance being significantly lower than NoFS.

NoFS performance will always be equivalent or higher than that of

OptFS, since NoFS does not worry about write ordering or atomicity of file-system operations.

7.1.4 Summary

NoFS and OptFS are meant for different use cases. NoFS demonstrates that it is possible to achieve file-system crash consistency without ordering writes. However, it is intended for applications that do not require `rename()`(§5.5.2). For these applications, NoFS delivers high performance.

OptFS demonstrates that decoupling ordering and durability significantly improves performance. While OptFS is backward-compatible with existing applications, it improves significantly for applications that use the `osync()` primitive.

7.2 Optimistic Techniques in Other Contexts

In Section 6.2, optimistic techniques were described in a specific context: the ext4 journaling file system. We now describe how these optimistic techniques can be applied in other contexts.

7.2.1 Optimistic Techniques in Other Crash-Consistency Mechanisms

Optimistic Crash Consistency has a core requirement: the file system should be able to query the dirty/clean status of writes in the cache (*e.g.*, via asynchronous durability notifications). Given this primitive, it is then possible to convert any crash-consistency mechanism that uses ordering points into an optimistic protocol. We now briefly discuss different crash-consistency mechanisms and the challenges in making them optimistic.

Journaling. The main difference between ext4 and other journaling file systems such as XFS [272], ReiserFS [229], and JFS [27] is that only ext4 provides data journaling. Optimistic Journaling uses Selective Data Journaling (§6.2.3) to provide guarantees equivalent to data journaling mode of ext4. Hence, without modifications to add data journaling, the optimistic versions of the other file systems will not be able to provide the strong crash-consistency guarantees of ext4. The other differences between the file systems (such as using extent-based allocation instead of block-based allocation) do not significantly affect crash consistency.

Copy-on-Write. Copy-on-write file systems such as WAFL [114], ZFS [29] and Btrfs [166] lend themselves nicely to optimistic techniques. Since all data and metadata is versioned, writing to a new subtree replaces writing to the journal, with the root pointer change replacing the transaction commit. The checkpointing phase is done away with entirely. However, new challenges arise due to garbage collection, and new optimistic techniques specific to copy-on-write file systems will need to be developed.

Soft Updates. Soft Updates depends entirely on carefully ordering writes for crash consistency. Ordering writes is usually done via disk cache flushes. Given a primitive such as asynchronous durability notifications, Soft Updates could be modified to maintain crash consistency without issuing flushes, thus increasing performance significantly.

File-System Check. File systems such as ext2 [38], which depend upon the file-system checker [175] to restore consistency after a crash, will not benefit from employing optimistic principles. Such file systems do not need to issue flushes to maintain crash consistency. However, from a user's point of view, introducing `osync()` in such file systems will make it easier to build crash-consistent applications.

Technology	Read	Write
DRAM	60 ns	60 ns
NAND FLash	25 µs	20–500 µs
PCM	115 ns	120 µs
Disk	2–15 ms	2–15 ms

Table 7.1: **Access Latency of Different Storage Media.** *The table shows the read and write latency for small, random I/O on different storage media. These numbers are based on demonstrated prototypes [243, 298].*

7.2.2 Optimistic Techniques in Other Media

We now discuss whether optimistic techniques would be relevant or useful on other storage media such as Flash [9] or non-volatile memory such as phase-change memory [149], Spin-Transfer Torque RAM [116], and memristors [270]. Table 7.1 lists the access latencies of some of these technologies.

We discuss the performance of OptFS on Flash, and then discuss how optimistic techniques could be applied in non-volatile memory storage systems.

Optimistic Techniques on Flash

Most Flash SSDs available (as of July 2015) are equipped with DRAM write caches. Several high-end SSDS available on the market today are equipped with super-capacitors [55, 122, 259]. These super-capacitors ensure that all the dirty data in the volatile cache is made persistent in the event of power loss. However, recent work has shown that despite these super-capacitors, power loss leads to corruption and data loss on many SSDs [150, 317]. Hence, flushing dirty data from the caches is still recommended to ensure durability.

Therefore, it seems worthwhile to investigate the performance cost of flushing on Flash SSDs. We ran experiments to compare the performance

System	**Op/s**
ext4 (with flush)	13018.89
ext4 (without flush)	13835.74
OptFS	6794.84
OptFS (without selective journaling)	13321.56

Table 7.2: **ext4 and OptFS performance on SSDs.** *The table shows the performance of ext4 and OptFS in different modes for the Filebench Varmail workload running on top of an SSD.*

of ext4 and OptFS on the Filebench Varmail workload running on top of an SSD. The SSD was Intel SSD 520 series (120 GB). The results are presented in Table 7.2.

We make several observations based on the results. First, the performance difference between enabling and disabling flushes on SSDs (6%) is significantly lower than on disks. We suspect that this results from storage-stack latencies: our experiments revealed that a write takes around one millisecond to propagate down the Linux kernel stack. Given that the stack propagation delay is 2X–50X the latency of a write, software latencies dominate the hardware access latencies. On flash-optimized storage system such as DFS [130], Onyx [10] Moneta [41], the results may be different.

Second, OptFS performance is about 50% that of ext4 for this workload. When selective journaling is turned off in OptFS, OptFS performance is similar to ext4. We have not analyzed why selective journaling leads to different performance on disks and SSDs.

The current implementation of OptFS is not optimized for SSDs. Since an SSD write only requires a few hundred microseconds [298], any additional CPU processing performed by OptFS significantly affects system performance. For example, our preliminary test results show that the CRC32 checksum computation is a significant part of the total run-time of the Filebench Varmail workload. We could optimize this by using a computationally lighter checksum. However, our results indicate that unless the time taken to propagate a write down the storage stack is optimized, such optimizations will have limited effect on performance.

7.2.3 Optimistic Techniques on NVM

Recent work on non-volatile memory place non-volatile memory directly on the memory bus, side-by-side with DRAM [56, 159, 190]. The reasoning behind this decision is that NVM has low access latencies (in the hundreds of nanoseconds for PCM [298]. STT-RAM is expected to be even faster); putting NVM behind an I/O bus would waste the performance benefits of NVM while also forcing block-based access [56].

However, putting NVM on the memory bus leads to crash-consistency challenges that are similar to ones discussed in this dissertation. A write is first stored in volatile CPU caches, and later written back to non-volatile memory (similar to how a write is stored first in the page cache and later written to the disk). The order in which non-volatile memory is updated is important for crash consistency.

Thus, storage systems are required to enforce ordering on the writes to non-volatile memory. Similar to the situation for disks, flushing the CPU caches is an extremely expensive way to order writes to non-volatile memory. Different research groups have come up with different ways to tackle this problem.

BPFS [56] introduces a new hardware primitive called the *epoch barrier* to order writes to non-volatile memory. Using the epoch barrier allows BPFS to decouple ordering and durability, similar to OptFS. However, while OptFS required hardware changes only to the storage device, BPFS requires changes to the processor core, the caches, and memory controller, and the non-volatile memory chips.

Mnemosyne [298] builds on Intel's fence and flush instructions [123] to provide barriers and flushes without requiring additional hardware modifications. Volos *et al.* show that it is possible to build higher-level interfaces such as atomic regions and transactions on top of these low-level primitives.

We believe that asynchronous durability notifications are more powerful than fence instructions, and that they will lead to better performance in the common case. Given a complex graph of dependencies, figuring out the order of writes in which the minimum number of barriers are required is not trivial. In the case of asynchronous durability notifications, each layer in the system that is concerned about ordering among writes will take care to issue its own writes in the correct order, not worrying about writes of other layers. In contrast, we believe that using barriers in a multi-layer storage stack will lead to a proliferation of barriers. As of 2010, the Linux kernel no longer supports barriers, instead using Forced Unit Access commands [60].

Work done at CMU by Moraru *et al.* [190] and Lu *et al.* [159] is closely related to OptFS. In both works, the hardware is modified so that the dirty status of writes in the caches can be queried by software. Instead of notifications when a write becomes durable, the tag associated with the write is changed in each cache level. Similar to optimistic crash consistency, Lu *et al.* speculatively write to new locations on non-volatile memory, and make the new writes visible upon commit.

Thus, optimistic principles are relevant and applicable to non-volatile memory. Introduction of new non-volatile technologies such as STT-RAM and memristors will create new challenges for storage-system designers who seek to employ optimistic principles.

7.2.4 Optimistic Techniques in Distributed Systems

The principles of optimistic crash consistency are not limited to single-node systems. Distributed systems that employ disks with volatile caches face crash-consistency problems similar to those of single-node systems; optimistic techniques can be applied in such distributed systems.

In collaboration with Microsoft Research, we employed optimistic techniques in a distributed storage system, *Blizzard*[1]. The Blizzard work was published as a paper titled *Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications* in NSDI 2014 [180]. We now briefly describe the problem Blizzard was trying to solve, its relation to OptFS, and how it differed from OptFS. We describe only the aspects of Blizzard that relate to optimistic crash consistency – please refer to the paper [180] for complete details.

Motivation

With the advent of cloud computing, enterprises are looking to move their applications onto the cloud. Given that cloud service providers such as Amazon have scale-out storage services, enterprises reasonably expect that their application performance will improve drastically. POSIX applications have two characteristics that make them a poor fit for scale-out storage:

1. POSIX applications issue small, random I/Os that are typically 32–128 KB in size [151, 297]. Many scale-out file systems, such as the Google File System [238], are built to optimize large, sequential I/Os.

2. POSIX applications order their writes using `fsync()` to ensure consistency [216]. Such `fsync()` calls serialize writes and block the application, preventing applications from exploiting I/O parallelism.

Due to these characteristics, simply running POSIX applications on scale-out file systems does not provide the required performance for POSIX applications. This poses a problem since there are hundreds of POSIX applications that customers would like to run on the cloud, yet most

[1] Work done as an intern at Microsoft Research, Redmond

customers do not have the technical knowledge and skills to rewrite their applications for cloud platforms.

Blizzard aims to solve this problem. Blizzard allows *unmodified, cloud-oblivious* POSIX applications to run on fast, scalable cloud storage [195] and obtain 2–10× performance. Blizzard is a high-performance block store that is exposed to Windows applications as a virtual SATA drive. However, Blizzard translates block reads and writes to parallel I/Os on remote drives.

Using Optimistic Principles in Blizzard

Let us assume that the client application can generate enough I/O to saturate the combined I/O bandwidth of the storage cluster. Ideally, Blizzard would like to write to all remote disks in parallel. Unfortunately, many applications use `fsync()` to serialize writes to ensure consistency [216]. Each `fsync()` causes a SATA `FLUSH` command at the drive. Some applications, such as databases, interact with the SATA drive directly, and issue SATA `FLUSH` commands to serialize writes. Each `FLUSH` command acts as a write barrier, preventing writes issued after the command from becoming durable until all writes issued before the command have become durable. This prevents Blizzard from making writes durable in parallel.

Storage Substrate. Blizzard uses Flat Datacenter Storage (FDS) as its low-level storage substrate [195]. FDS is a datacenter-scale blob store that connects all clients and disks using a network with full-bisection bandwidth, i.e., no oversubscription [99]. FDS also provisions each storage server with enough network bandwidth to match its aggregate disk bandwidth. For example, a single physical disk has roughly 128 MB/s of maximum sequential access speed. 128 MB/s is 1 Gbps, so if a storage server has ten disks, FDS provisions that server with a 10 Gbps NIC; if the server has 20 disks, it receives two 10 Gbps NICs. The resulting storage substrate provides a locality-oblivious storage layer; any client can access any remote disk at maximum disk speeds. A Blizzard virtual disk is backed by a single FDS blob.

Flushes as Ordering Barriers. Blizzard solves this problem by issuing writes in a way that respects flush-order semantics. Each `FLUSH` request starts a new *flush epoch*. When Blizzard's virtual disk receives a `FLUSH` request, it immediately acknowledges the flush to the client application, even though Blizzard has not made writes from that flush epoch durable. If the client or the virtual disk crashes, the disk will always recover to a consistent state in which writes from different flush epochs will never be intermingled – all writes up to some epoch $\mathrm{N} - 1$ will be durable; some writes from epoch N will be durable; and all writes from subsequent epochs are lost. Note that this is similar to the semantics offered by OptFS, with epochs defined by `FLUSH` commands instead of `osync()` or `fsync()` commands. Similar to OptFS, Blizzard provides eventual durability.

Design Overview. Blizzard provides eventual durability by using techniques similar to OptFS. When a write becomes durable at a remote drive, it is not immediately visible to the application. The Blizzard client makes prefixes of durable writes visible to the application. Similar to asynchronous durability notifications, a process running at each remote drive sends a message to the Blizzard client when a write has become durable. Thus, Blizzard can issue writes in parallel and out-of-order, and based on notifications from remote drives, can make a prefix visible to the application. This allows Blizzard to exploit I/O parallelism to significantly increase performance.

To maximize the rate at which writes are issued, Blizzard defines a scheme that allows writes to be acknowledged immediately and issued immediately, regardless of their flush epoch. This means that writes may become durable out-of-order. However, Blizzard enforces eventual durability using two mechanisms. First, Blizzard uses a log structure to avoid updating blocks in place; thus, if a particular write fails to become durable, Blizzard can recover a consistent version of the target virtual disk block. Second, even though Blizzard issues each new write immediately, Blizzard uses a deterministic permutation to determine which log entry (i.e., which <tract,offset>) should receive the write. To recover to a consistent state after a crash, the client can start from the last checkpointed epoch and permutation position, and roll the permutation forward, examining log entries and determining the last epoch which successfully retired.

Data Structures. Let there be V blocks in the virtual disk, where each block is of equal size, a size that reflects the average I/O size for the client (say, 64 KB or 128 KB). The V virtual blocks are backed by `P` physical blocks (where `P` $>$ V) in the underlying FDS blob. Blizzard treats the physical blocks as a log structure. Blizzard maintains a `blockMap` that tracks the backing physical block for each virtual block. Blizzard also maintains an `allocationBitMap` that indicates which physical blocks are currently in use. When the client issues a read to a virtual block, Blizzard consults the `blockMap` to determine which physical block contains the desired data. Handling writes is more complicated, as explained below. Blizzard maintains a counter called `currEpoch`; this counter is incremented for each flush request, and all writes are tagged with `currEpoch`. Blizzard also maintains a counter called `lastDurableEpoch` which represents the last epoch for which all writes are retired.

Virtual-to-Physical translation. When Blizzard initializes the virtual disk, it creates a deterministic permutation of the physical blocks. This permutation represents the order in which Blizzard will update the log. For example, if the permutation begins 18, 3,. . . , then the first write, regardless of the virtual block target, would go to physical block 18, and the second write, regardless of the virtual block target, would go to physical block 3. Importantly, Blizzard can represent a permutation of length P in O(1) space, not O(P) space. Using a linear congruential generator [138], Blizzard only needs to store three integer parameters `(a, c, and m)`, and another integer representing the current position in the permutation. As we will describe later, the serialized permutation will go into the checkpoints that Blizzard creates.

Handling Reads and Writes. Handling reads is simple: when the client wants data from a particular virtual block, Blizzard uses the `blockMap` to find which physical block contains that data; Blizzard then fetches the data.

Handling writes requires more bookkeeping. When a write arrives, Blizzard issues the write to the next unallocated physical block in the deterministic permutation. Blizzard then places the write in a queue. Blizzard uses the write queue to satisfy reads to byte ranges with in-flight (but possibly non-durable) writes.

Note that once the write is issued, Blizzard does not update `blockMap` or `allocationBitMap` – those structures are reflected into checkpoints, so they can only be updated in a way that respects eventual durability.

When the write becomes durable, Blizzard checks whether, according to the permutation order, the write was the oldest un-retired write in `lastDurableEpoch+1`. If so, Blizzard removes the relevant write queue entry, and updates `blockMap` and `allocationBitMap`. Otherwise, Blizzard waits for older writes to commit first. Once all writes in the associated epoch are durable, Blizzard increments `lastDurableEpoch`.

Writing Expanded Blocks. When Blizzard issues a write to FDS, it actually writes an expanded block. This expanded block contains the raw data from the virtual block, as well as the virtual block id, the write's epoch number, and a CRC over the entire expanded block. As we explain below, Blizzard will use this information during crash recovery.

If the client issues a write that is smaller than the size of a virtual block, Blizzard must read the remaining parts of the virtual block before calculating the CRC and then writing the new expanded block. This read-before-write penalty is similar to the one suffered by RAID arrays that use parity bits. This penalty is suffered for small writes, or for the bookends of a large write that straddles multiple blocks. For optimal performance, Blizzard's virtual block size should match the expected I/O size of the client. For example, POSIX applications like databases and email servers often have a configurable "page size"; these applications try to issue reads and writes that are integral multiples of the page size, so as to minimize disk seeks. For these applications, Blizzard's virtual block size should be set to the application-level page size.

Checkpointing. Periodically, the client checkpoints the `blockMap`, the `allocationBitMap`, the four permutation parameters, `lastDurableEpoch`, and a CRC over the preceding quantities. For a 500 GB virtual disk, the checkpoint size is roughly 16 MB. Blizzard does not update the checkpoint in place; instead, it reserves enough space on the FDS blob for two checkpoints, and alternates checkpoint writing between the two locations for reliability [234].

Recovery. To recover from a crash, Blizzard inspects the two serialized checkpoints and initializes itself using the latest correct checkpoint (as indicated by `lastDurableEpoch`). Blizzard then scans physical blocks in the order in which they appear in the virtual log, starting from the last virtual block indicated in the checkpoint. Upon reading a current physical block, one of three things can happen:

- If the `allocationBitMap` says that the current physical block is in use, Blizzard inspects the next physical block.
- If the block belongs to a previous epoch, Blizzard terminates recovery. If the CRCs of the block replicas don't match, Blizzard terminates recovery.
- If the block is legitimate, Blizzard updates the block's allocation status in `allocationBitMap` and translation in `blockMap`. Allocation

and translation status for the previous physical block backing this virtual block are also updated. The value of `lastDurableEpoch` is updated to match the block's epoch. The last virtual block in the checkpoint is also updated.

How Blizzard Differs From OptFS

Blizzard demonstrates how Optimistic Crash Consistency can be applied in a distributed systems setting. While the Optimistic File System applies Optimistic Principles at the file-system level, Blizzard applies it at the block-device level, showing that the principles can be applied in a file-system agnostic manner. While OptFS was developed for the Linux operating system, Blizzard runs on Windows, showing that the ideas are not limited to a specific operating system.

7.3 Using Osync

As implemented in OptFS, an `osync()` call will end the currently running transaction, and begin a new one. Since all writes go into a single currently-running global transaction, this means that `osync()` is effectively a global ordering barrier. Since `osync()` does not return until the entire transaction is present in the disk cache, every `osync()` call has a delay associated with it ($\approx$ 1 millisecond). Therefore, `osync()` should not be called indiscriminately.

Unfortunately, it is not trivial to figure out where `osync()` should be called, especially if the developer did not write the code originally. Modern applications have their update protocols (*i.e.*, the sequence of writes they use to update their on-disk state) spread across multiple files [216]. They heavily make use of libraries and frameworks that have their own ordering requirements. Some applications call `fsync()` excessively to order writes, while others do not call fsync() even if required for crash consistency, fearing its performance cost [216].

Given the code for a crash-consistent application (*i.e.*, the application enforces ordering where required using fsync()), the task then becomes one of figuring out which of the fsync() calls are meant for ordering, and replacing those fsync() calls with osync() calls.

In collaboration with other students at UW Madison, we have developed ALICE [216], a tool that can be used to figure out where osync() calls should be inserted in an application. The developer provides ALICE with an input workload, and a checker. The checker can be run on the application state (after a crash) to verify whether certain invariants hold. For example, if one inserts a row into a SQLite [263] database, regardless of when a crash happens, the database should not contain a corrupt row. ALICE then constructs different on-disk states that could occur after a crash, and uses the checker to detect when an invariant is violated. ALICE can determine if the violation was due to application writes being re-ordered; if so, ALICE can determine where osync() should be inserted.

ALICE has certain limitations. It is not complete; it figures out where osync() should be inserted in a given trace of the application. It does not guarantee that an fsync() call can be safely replaced by osync() in all runs of the application. The developer should use ALICE to obtain hints about where osync() calls may be inserted, and use their expertise to confirm that the replacement is sound.

7.4 Interactions with Industry

We now describe our experience talking with developers in industry about the new interfaces proposed in this dissertation.

ADNs. We interacted with developers at Seagate and other storage companies about supporting asynchronous durability notifications (ADNs).

Developers were concerned about the performance cost of ADNs as presented in this dissertation (§6.2.1). They suggested variations where the notifications are batched together based on block address range (*e.g.*, notifications for blocks in range 1000–2000), or time (*e.g.*, notifications for blocks submitted in last 5 seconds). Samsung is interested in a variation of ADNs for their mobile systems.

osync. We presented the ideas behind `osync()` at the Linux FAST Summit 2014 [291] that brought together Linux kernel developer and researchers in file-system and storage community. Developers were interested in the performance impact of the `osync()` call. They were concerned that using `osync()` would imply that full data journaling mode would be used by the file system; they were interested in how selective data journaling works. The developers were resistant to the idea of introducing a new system call; this is usually done only when a number of applications are strongly requesting the new system call.

Kernel developers are extremely wary about accepting new code from researchers. The developers are concerned that researchers will "patch-and-vanish", introduce new code into the Linux kernel, and then disappear without providing proper support over the years. Due to this reason, ideas moving from academia to the Linux kernel happens rarely: either because the idea is simple to understand and implement [221], or because the researcher is a long-time kernel developer with sufficient standing in the community [63].

7.5 Summary

In this chapter, we discussed how NoFS and OptFS compare along several axes. In terms of hardware requirements, both systems require specific hardware support. The atomic write required by NoFS is available on today's drives. Durability notifications can be built in software, thus enabling OptFS to be run on commodity drives. Although NoFS provides strong crash-consistency, the lack of atomic `rename()` restricts usability; in contrast, OptFS enables applications to build efficient update protocols. Both OptFS and NoFS eliminates flushes in the common case; however, since OptFS provides additional semantics (via `osync()`), its performance is lower than that of NoFS.

We described how optimistic techniques can be used in other contexts. File-system crash-consistency techniques such as copy-on-write can be adapted to operate in an optimistic manner. Removing ordering constraints has been shown to increase performance in other media such as non-volatile memory; such storage systems will benefit from using optimistic techniques. We described how Blizzard, a distributed storage system, has been modified to take advantage of optimistic techniques.

Finally, we discussed the challenges application developers face in using `osync()` in application update protocols. We described our efforts in encouraging industry to adopt asynchronous durability notifications and the `osync()` primitive.

8 Related Work

In this chapter, we discuss research and systems that are related to this dissertation. First, we describe efforts over the years to provide safe recovery from crashes (§8.1). Then, we describe work related to the specific techniques used in our work (§8.2). We discuss efforts at testing file systems similar to work on BOB (§8.3). Finally, we discuss different interfaces used to obtain consistency and durability (§8.4).

8.1 Crash Recovery

Designing systems to recover safely after crashes has been the focus of systems researchers for many years. We first describe how databases provided crash consistency, and then explain how file systems and applications built on top of these techniques to build crash consistency protocols.

Crash Recovery in Databases. The database community pioneered the transaction abstraction [97, 98, 188] for atomic updates and introduced two techniques to optimize how transactions are persisted: logging and checkpointing [96, 105, 188], and group commit [71, 89, 188]. Mohan *et al.* brought these techniques together in the ARIES algorithm [188] that has been implemented in a number of industrial and research systems [39, 43, 64, 70, 109, 239, 244].

Crash Recovery in File Systems. File systems adapted techniques such as logging to update data and metadata in a consistent fashion. Section 2.6.2

describes the different approaches taken by file systems. Logical disks [65, 100] sought to separate out file management from crash recovery; atomic updates would be provided by a logical disk that works with different file-system structures. In a similar vein, systems such as the Inversion File System [200, 268] layer file systems on top of databases to exploit their transactional capabilities; however, such systems are not widely adopted due to their complexity and poor performance. TableFS [230, 231] builds a file system on top of LevelDB [93]; TableFS takes advantage of the sequential layout of LevelDB to increase performance. However, the LevelDB files need to be stored on another file system, so while TableFS increases performance, it relies upon both LevelDB and the underlying file system to recover correctly from a crash; the complex interactions between LevelDB and the file system make this a challenge [216].

Crash Recovery in Applications. Stasis [248] builds on write-ahead logging [106] to provide transactional storage for applications. The crash recovery mechanisms of Stasis are built around the same principles as journaling file systems like OptFS. However, Stasis transactions provide properties such as isolation which applications may not need (and may not be willing to be pay for).

A number of modern applications use databases such as SQLite [263] to atomically update their on-disk state. The heavy performance cost associated with SQLite [126] has led to many applications creating ad-hoc protocols for achieving crash consistency [216]. The ad-hoc nature of these protocols lead to data loss and corruption on some file-system configrations [216].

Recent work builds on transactional flash storage to allow applications to update state in an atomic fashion [187]. By using transactions, Min *et al.* hope to avoid the bugs that plague ad-hoc protocols. Their CFS file system requires that the underlying storage provide multi-page atomic writes. Providing such atomic writes in software (via techniques like atomic recovery units [65]) will incur additional performance overhead and complexity.

8.2 Reliability Techniques

We describe work related to the various techniques used in this dissertation: using embedded information (§8.2.1), incremental fsck in the background (§8.2.2), ordering updates to storage (§8.2.3), and trading durability for performance (§8.2.4).

8.2.1 Using Embedded Information

The idea of using information inside or near the block to detect errors has been used in several systems. The Cambridge File Server [72] used certain bits in each cylinder (cylinder map) to store the allocation status of blocks in that cylinder. The Cedar File System [106] used *labels* inside pages to check their allocation status. Embedding logical identity of blocks (inode number + offset) has been done in RAID to recover from lost and misdirected writes [141]. Transactional flash [223] embeds commit records inside every page to provide transactions and recovery. However, NoFS is the first work that we know of that clearly defines the level of consistency that such information provides and uses such information alone to provide consistency.

The design of the Pilot file system [228] is similar to that of NoFS. Pilot employs self identifying pages and uses a scavenger to reconstruct the file system metadata upon crash. However, like the file-system check, the scavenger needs to finish running before the file system can be accessed. In NoFS, the file system is made available upon mount, and can be accessed while the scan is running in the background.

Pangaea [236] uses backpointers for consistency in a distributed wide area file system. However, its use of backpointers is limited to directory entry backpointers that are used to resolve conflicting updates on directories. Similar to NoFS, Pangaea also uses the backpointer as the true source of information, letting the backpointers of child inodes dictate whether they belong to a directory or not.

Btrfs [166] supports back references that allow it to obtain the list of the extents that refer to a particular extent. Although back references are conceptually similar to NoFS backpointers, the main purpose of btrfs back references is supporting efficient data migration, rather than providing consistency. Other mechanisms such as checksums are used to ensure that the data is not corrupt in btrfs. Another key difference is that btrfs does not always store the back reference inside the allocated extent: sometimes the back references are stored as separate items close to the extent allocation records.

Backlog [163] also uses explicit back references in order to manage migration of data in write anywhere file systems. The back references in Backlog are stored in a separate database, and are designed for efficient querying of usage information rather than consistency. Backlog's back references are not used for incremental file-system checking or resolving ownership disputes.

The Selfie virtual disk format [310] embeds metadata into data blocks to allow data blocks to be written without requiring an associated metadata write, thus increasing performance significantly for data writes. Similar to the non-persistent allocation structures of NoFS, Selfie's lookup table in maintained in memory, and reconstructed from the on-disk version after a crash. While NoFS uses an out-of-band area for storing backpointers, Selfie depends on the data being compressed to make room for metadata inside a block. Thus, although Selfie does not depend upon hardware characteristics, it is dependent on workload characteristics.

The ideas behind ReconFS [161] are similar to those of NoFS. ReconFS makes the directory structure volatile, and recovers it from the on-disk structure by embedding extra information into the on-disk structures to make them recoverable. While NoFS maintains both forward and backward pointers, ReconFS maintains only backward pointers (inverted indices). NoFS is concerned with the crash consistency of the whole file system, while ReconFS focuses on the consistency of the name-space alone. ReconFS and Selfie are good testaments to our design choices of not persisting certain metadata structures and allowing I/O to be re-ordered.

8.2.2 Incremental FSCK in the Background

NoFS performs the checks done by fsck incrementally in the background. There have been previous work that modified fsck in a similar manner. McKusick's background fsck [173] could repair simple inconsistencies such as lost resources by running fsck on snapshots of a running system. Chunkfs [112] is similar to NoFS, providing incremental, online file-system checking. Chunkfs differs from NoFS in that the minimal unit of checking is a chunk whereas it is a single file or block in NoFS. Chunkfs does not offer online repair of the file system, while it is possible in NoFS, due to backpointers and non-persistent allocation structures.

8.2.3 Ordered Updates

Soft Updates [87] shows how to carefully order disk updates so as to never leave an on-disk structure in an inconsistent form. In contrast with OptFS, FreeBSD Soft Updates issues flushes to implement `fsync()` and ordering (although the original work modified the SCSI driver to avoid issuing flushes).

Given the presence of asynchronous durability notifications, Soft Updates could be modified to take advantage of such signals. We believe doing so would be more challenging than modifying journaling file systems; while journaling works at the abstraction level of metadata and data, Soft Updates works directly with file-system structures, significantly increasing its complexity.

OptFS is similar to that of Frost *et al.*'s work on Featherstitch [84], which provides a generalized framework to order file-system updates, in either a soft-updating or journal-based approach. OptFS instead focuses on delayed ordering for journal commits; some of our techniques could increase the journal performance observed in their work.

Lu *et al.*'s work on loose-ordering consistency [159, 190] has a similar flavor to OptFS. They increase the performance of writes to persistent memory by allowing speculative writes and only making the writes visible at transaction commit. The dirty status of writes in the cache are identified using extra tag bits in the cache.

Apart from file systems, write order is important in a different (but related) context. A number of storage systems today use SSD or RAM devices as caches to increase performance [115, 240]. For such storage systems, the policy controlling how data is transferred from the cache to the backing device is crucial for performance and reliability. The setup in these systems is similar to the buffer cache and the disk storage device.

Koller *et al.* propose writeback policies (ordered write-back and journaled writeback) to ensure consistency at the storage level (either point-in-time or epoch-based) [139]. Qin *et al.* take advantage of the fact that applications do not have consistency expectations in between two barrier events (*e.g.*, `fsync()`) to further increase performance [224]. OptFS writeback is similar to the journaled writeback policy of Koller *et al.*, with `fsync()` calls or time-duration triggers defining epochs.

Write ordering is also important in non-volatile memory systems. Recent work by Pelley *et al.* [214] explores how memory consistency models relates to persistent writes in non-volatile memory, and how the write orders may be relaxed while maintaining consistency.

8.2.4 Delaying Durability

The work of Nightingale *et al.* on “rethinking the sync” [196] has a similar flavor to OptFS. In that work, the authors cleverly note that disk writes only need to become durable when some external entity can observe said durability; thus, by delaying persistence until such externalization occurs, huge gains in performance can be realized. OptFS is complimentary, in that it reduces the number of such durability events, instead enforcing a weaker (and higher performance) ordering among writes, but avoiding the complexity of implementing dependency tracking within the OS. In cases where durability is required (*i.e.*, , applications use `dsync()` and not `osync()`), optimistic journaling does not provide much gain; thus, Nightingale *et al.*’s work still can be of benefit therein.

The work of Keeton *et al.* on disaster recovery argues that durability can be traded for increased performance for many applications [134]. Snapmirror [212], Seneca [128], and the Smoke and Mirrors File System (SMFS) [303] are asynchronous (or semi-synchronous) mirroring systems that trade off durability for performance, similar to OptFS. All three systems use a modified form of log-structured file systems [234] and employ techniques similar to journaling. Using the terminology of Seneca [128], OptFS provides asynchronous out-of-order atomic updates. The major difference between OptFS and these systems is that OptFS can gain performance without any data loss. If we define data loss as committed data that becomes unavailable, the performance gain of OptFS comes from using `osync()` – where users have no expectations about the durability of data. In contrast, Snapmirror, Seneca, and SMFS can lose committed data if there is a disaster.

8.3 Testing File Systems

There have been several efforts to test the reliability of storage systems. We discuss the work most closely related to BOB (§4).

Zheng *et al.* developed a framework for testing whether commercial databases violate ACID properties under power failure scenarios [316]. They trace the writes resulting from one of four hand-crafted workloads, and replay different sequences of those writes to detect ACID violations. Their framework differs from BOB in two significant ways. First, they do root-cause analysis on ACID violations to help engineers diagnose the bug. Since BOB was not developed for the purpose of diagnosing bugs, it does not contain tools for root-cause analysis. Second, their framework is limited to SCSI devices (as they modify the iSCSI driver); BOB, on the other hand, works above the device-driver level, and therefore works on any storage device.

The EXPLODE framework [311] systematically checks storage systems for errors. It drives the system into rare corner cases and tests system behavior. EXPLODE is a powerful framework that can be used to test applications, file systems, and software at any level of the storage stack for errors. However, for the simple task for testing persistence properties of file systems, we feel it is needlessly complex. For example, EXPLODE requires developers to carefully annotate complex file systems using *choose()* calls. BOB, on the other hand, does not require any annotation or specialized knowledge of the file system being tested.

There has been interest in the Linux kernel community towards more effective power-failure testing. Josef Basik, a software engineer at Facebook, has been developing a tool based on the device mapper to easily reproduce power failures [24, 25]. Similar to BOB, the tool logs writes and then creates new disk images based on the traces. However, the focus of the tool is on whether file systems properly make acknowledged writes durable, and not on testing persistence properties.

Zheng *et al.* have tested the reliability of SSDs and disk drives under power failure [317]. Unlike the other work discussed here, they developed hardware to cut power to the devices being tested and examined device behavior. Hardware testing is essential for testing storage devices, since the firmware is closed-source, and we cannot emulate the effects of power loss, as we do for software layers in the storage stack. Thus, Zheng *et al.*'s work is complementary to tools such as BOB.

We note that none of the tools discussed in this section examine or test the persistence properties of file systems. File-system persistence behavior was largely unexplored before our work, and BOB represents the first step in defining and examining persistence properties.

8.4 Interfaces for Consistency and Durability

We now describe interfaces used in various systems to achieve consistency and durability.

Transactions. Several file systems have provided a transactional interface, allowing developers to update application state with ACID guarantees [199, 218, 252, 261]. However, either due to the performance cost (14% in TxOS [218]), or high complexity [199], transactional file systems have not been adopted widely.

Several storage devices provide support for the transactional interface [48, 54, 85, 133, 157, 158, 206, 209, 223]. Applications or file systems can build on top of these devices to allow the user to atomically update application state [187].

Atomic Update. Several recent systems offer an interface for users to obtain atomicity (without isolation) [164, 208, 293]. Park *et al.* offer an atomic version of `msync()` [208], while Verma *et al.* provides failure-atomic `writev()`, `msync()`, and `syncv()` [293]. The MariaDB database builds on top of the atomic write offered by FusionIO SSDs [164].

At the application level, atomic update is usually carried using the rename() system call [101]. There are subtle problems with using rename(), depending upon the file-system configuration [59, 216]. Specialized interfaces such as exchangedata() [17] and replacefile() [181] are also used to achieve atomic updates.

Barriers and Flushes. The osync() and dsync() primitives are similar to the fence and flush memory primitives in the Intel architecture [123]. The key difference is that a dsync() issues a flush at the end of osync(), and thus is a superset of osync(); on the other hand, flushes do not guarantee the ordering provided by barriers. Mnemosyne [298], a lightweight system for exposing storage-class memory to user-mode programs, builds upon the Intel primitives to offer the persistent-memory version of flush and fence primitives to users (among other interfaces such as transactions). Since Mnemosyne is only concerned with data in processor caches and storage-class memory, Intel's primitives are sufficient, and no additional hardware support is required. In contrast, since current disks do not offer the fence primitive, hardware modifications like asynchronous durability notifications are required for OptFS.

BPFS [56] is another file system built on top of phase-change memory. BPFS introduces two new hardware primitives: 8-byte atomic writes and epoch barriers. Similar to osync(), epoch barriers order writes without affecting durability. Note that while BPFS uses epoch barriers internally, it does not present an interface such as osync() to the user. While OptFS requires modifications to the storage device, BPFS requires modifications to the processor core, the cache, the memory controller, and the phase-change memory chips.

Featherstitch [84] provides similar primitives for ordering and durability: pg_depend() is similar to osync(), while pg_sync() is similar to dsync(). The main difference lies in the amount of work required from application developers: Featherstitch requires developers to explicitly encapsulate sets of file-system operations into units called *patchgroups* and define dependencies between them. Since `osync()` builds upon the familiar semantics of `fsync()`, we believe it will be easier for application developers to use.

We believe that barriers and flushes are the most fundamental of these interfaces; other interfaces can be implemented using barriers and flushes. Mnemosyne [298] builds transactions and atomic updates using barriers and flushes; it also exposes the low-level primitives to users so that they can build their own consistency mechanisms. We advocate a similar approach for applications using file systems to store state.

9

Future Work

In this chapter, we outline directions in which our work could be extended in the future. These fall into three main categories: removing a limitation of our work such as requiring special hardware support (§9.1), enabling developers and users to more easily benefit from our work (§9.2, §9.3), and applying the principles learnt in our work in different contexts (§9.4, §9.5).

9.1 Software Async Durability Notifications

We developed the notion of Asynchronous Durability Notifications (ADNs) for Optimistic Crash Consistency [46]. The drawback of ADNs is that they are not supported on any current hardware. Hardware manufacturers are reluctant to introduce new hardware features that will not directly increase performance. Once a manufacturer is convinced, developing and manufacturing hardware that support ADNs will take a number of years.

Software ADNs can be developed to address this challenge. We propose a new system that acts as an intermediary between file systems and traditional storage devices. The system communicates with traditional storage devices and issues flushes and FUA requests as required. The file system submits writes to the new system and gets software ADNs indicating when different writes have become durable.

Without hardware support, the system cannot query the storage device for the durability status of I/O. Instead, the system forces writes to be durable using either flushes or flags such as FUA. The system keeps track of writes that have been submitted to the device; a flush would make all of the submitted writes durable. The system would then send software ADNs to the file system (or a database) indicating that certain writes have become durable.

The system could be configured to issue flushes based on different policies. For example, flushes could be issued periodically, depending on the window of vulnerability that client applications are comfortable with. A flush could also be issued once a certain amount of data has queued up in the system's buffers; this policy would be determined by the maximum amount of data (rather than time) that the client applications would be comfortable losing.

Tracking the durability status of all in-flight writes at a fine granularity without excessive space overheads or CPU consumption will be a challenge. Allowing some writes to be un-ordered and un-tracked (*e.g.*, writes to debug files) may help address this challenge. The semantics between ordered writes and un-ordered writes will have to be carefully defined.

Hardware ADNs allow the storage device to make I/O durable in the order that maximizes performance (*e.g.*, by minimizing seeks [308]). The absence of flushes essentially allow a bigger "working set" of I/O requests that the device can pick from, when choosing the next request to service. Flushes reduce the size of the working set by forcing I/O to be serviced at various points. Thus, by using flushes, storage performance would be reduced from the maximum attainable. We expect that this performance reduction would be compensated by enabling optimistic crash consistency to be employed on traditional storage devices.

9.2 Automatic Osync Substitution

In Section 7.3, we discussed how it is not trivial to figure out where `osync()` calls should be inserted in an application. Although ALICE provides hints as to where `osync()` should be called [216], it cannot do automatic substitution of `osync()` for `fsync()` calls. Given the large number of applications that use `fsync()` for only ordering their writes [216], there is a large opportunity for transparently increasing the performance of applications by doing automatic, sound replacements of a number of `fsync()` calls with `osync()` calls.

We can identify `fsync()` calls used for ordering using this intuition: if there is a `fsync()` call before the application process communicates externally (via I/O to an external device, messages to another process, etc.), that `fsync()` call was meant for durability; otherwise, the `fsync()` call was used for ordering. This intuition is a process-specific form of external synchrony [196].

The idea behind the rule is simple: processes usually make state durably before telling an external party that they have performed an action. For example, a bank needs to make your transaction durable before informing you that it has your money. Application-update protocols that we examined in the ALICE project supports this intuition [216].

It is easy to see that `fsync()` calls that occur in the middle of an application's update protocols (with the end defined as external communication) can be safely replaced by `osync()` calls, as a later `fsync()` call will ensure durability before anyone external can observe it.

The question is whether this rule is too conservative. If we define external parties to include the kernel (as two processes may communicate via the kernel), then all `fsync()` calls may be flagged as calls for durability. We need to experiment with several applications to find a definition of external parties that is not too restrictive.

Combined with software ADNs (§9.1), transparently replacing `fsync()` calls with `osync()` calls has the potential to increase the performance of thousands of applications running on commodity hardware. External synchrony [196] is hard to apply since it involves modifications to the kernel, and requires tracking all dependencies inside the operating system. By contrast, software ADNs and `osync()` replacement can be done without changing the kernel, or even involving the original developers of applications.

9.3 Tools for Testing File-System Crash Consistency

As part of this dissertation, we developed tools to test whether file-systems could reliably recover after crashes. As part of the NoFS project, we developed a pseudo-device driver that drops specified writes. The pseudo-device driver allows us to do targeted testing. As part of the OptFS project, we developed a tool that allows us to emulate random crashes.

There are no tools publicly available at this time to robustly test the crash consistency of file systems. Currently, slow power-cycle testing is used to test crash reliabilty [208, 317]. Zheng *et al.* have developed tools to test the crash reliability of databases [316], but the workloads and testing methodology is specific to the ACID properties of databases. While KVM XFS provides some crash-consistency tests [281], it is heavyweight, requiring a virtual machine to be booted up using KVM. Our tools can test crash-consistency completely in user-space without requiring a virtual-machine setup.

Our tools were built specifically for NoFS and OptFS, and hence require some development to work correctly with all file systems. We believe modifying our tools in such a manner and making them open-source would be of significant benefit to the file-system community.

9.4 Verifying Crash Consistency for Complex Stacks

Modern storage stacks are comprised of a large number of layers. For example, the Windows I/O stack has 18 layers between the application and the storage [283]. As we show in this dissertation, crash consistency depends upon each layer correctly handling flags like FUA and passing down flush requests down to the storage device. Unfortunately, there are many instances where a layer does not pass along flush requests to the lower layer [153, 294]. In such cases, the application (and perhaps the file system) loses consistency if there is a power loss or a crash.

With the advent of software-defined storage, we believe applications will soon be able to request and automatically obtain customized storage stacks [79, 119, 193]. Amazon EC2 already does this at a coarse level by allowing users to specify the storage they require for each virtual machine [1]. The day is not far off when storage stacks will be constructed on the fly, mixing and matching different layers like block re-mappers, logical volume managers, and file systems [295, 296].

How does one test if an application is crash-consistent on a given storage stack? Developers test their applications comprehensively on one storage stack, and deploy their applications on storage stacks that offer the same API. Several vendors provide API compatibility to facilitate such testing and deployment [4, 5].

Unfortunately, while a common API guarantees that applications will execute on different stacks, it does not guarantee that they will do so correctly. Most API specifications simply describe what operations are offered by the storage stack. The specifications do not describe the semantics offered by the system: for example, whether two operations are persisted in order or whether an operation is persisted atomically. Yet, recent work has shown that application correctness hinges on storage-stack semantics [216, 316]. For example, LevelDB [93] required the rename of a file to be persisted before the unlink of another file. If the storage stack does not order these operations, it results in data corruption [2, 216].

Testing whether an application will be crash-consistent on a storage stack is challenging for two reasons. First, the guarantees that an application requires from storage are not well-specified; if the developer has only been testing on one platform, they may not even realize that their application depends on certain features of the platform. Our work on application crash vulnerabilities suggests that the required guarantees are complex, and cannot be expressed in simple binary checks or numeric limits [216]. Second, modern storage stacks are composed of many layers [283]. Each layer builds upon the guarantees given by lower layers to provide guarantees to higher layers. To identify the guarantees given by a dynamically composed stack, we have to examine the guarantees given by each layer in the stack.

We tackle each challenge by borrowing techniques from the programming languages community. First, we propose to specify complex storage guarantees in a formal language (such as Isar [305]). We suggest that the same language could be used to specify the high-level design of each layer of the storage stack. Second, proof assistants (such as Isabelle [197]) can be used to prove that the stack provides the guarantees required by the application. Just as a statement could be proved given a collections of axioms and theorems, we propose that guarantees required by applications could be proved given the guarantees offered by each storage-stack layer.

In collaboration with colleagues at UW Madison, we used Isar to specify the design of a simple two-layer storage stack, and used Isabelle to prove that the `put` operation in a simple key-value store is failure atomic [11].

We believe such verification will be essential for software-defined storage in clouds and datacenters. When storage stacks are constructed on the fly, the corresponding high-level specifications for different layers can be retrieved, and the guarantees of the resulting stack can then be compared with application requirements. Such checking can be used to construct the optimal storage stack (in terms of resource utilization or other metrics) that will satisfy the given application requirements.

9.5 OptFS for SSDs

Given the prevalence of SSDs, it is interesting to examine the behavior of OptFS when running on SSDs instead of disk drives. Our preliminary tests show that OptFS does not increase performance significantly (compared to ext4 with flushes) when run on top of flash SSDs (§7.2.2).

We found that employing selective data journaling cuts performance by half for the Filebench Varmail workload. Initially, we believed that the performance degradation was due to the double writes of data journaling. However, if that is the case, Varmail on hard drives should also experienced similar performance degradation; instead, OptFS (with selective data journaling enabled) increases Varmail performance significantly on hard drives. More investigation is required to understand the cause of the performance degradation.

We observed that checksum calculation time was a significant percentage of the total run time for the Varmail workload. We plan to investigate whether using lightweight checksums increases performance significantly for a range of workloads.

Finally, we need to re-architect the storage stack so that the time taken for a write to propagate down the stack is minimal. Currently, the propagation delays is around one millisecond; given that the write delay for an SSDs is in the order of hundreds of microseconds, stack propagation delay dominates the write latency of small writes.

9.6 Summary

In this chapter, we described how our dissertation work can be extended in the future. First, we describe how asynchronous durability notifications can be implemented as a software layer between the file system and the storage device. We then discuss how application code can be analyzed to automatically insert `osync()` calls, by using a variation of external synchrony. We explain the need for tools to inject targeted and random crash failures in file systems. We describe how storage stacks have become so complex that automated methods are required to check that an application is crash-consistent on a given storage stack. Finally, we discuss the challenges in running OptFS on top of Flash SSDs.

10 Lessons Learned and Conclusions

Everyday, increasing amounts of data vital to our well-being are being stored digitally [95, 127, 162, 169, 306]. Safely updating this data is a challenge due to interruptions by power loss or crashes [53, 131, 176, 178, 192, 292, 313]. While a number of solutions exist, ranging from the file-system check [175] to journaling [106], they result in significant performance degradation. Many practitioners turn off consistency solutions, risking data loss and corruption rather lose so much performance. In this dissertation, we presented solutions that offer both strong consistency guarantees and excellent performance.

We started by analyzing how crashes lead to file-system inconsistency. We also analyzed how file systems persisted dirty writes to storage in the absence of `fsync()` calls in the application; file systems doing this in the right order is crucial for application-level crash consistency. Our analysis leads to two conclusions. First, there is a need for new crash-consistency techniques that provide strong guarantees without significantly degrading performance. Second, applications require new primitives they can use to explicitly order their on-disk updates.

In the second part of our dissertation, we presented our solutions to these problems. We introduced Backpointer-Based Consistency, a new crash-consistency techniques that does not required writes to storage to be ordered. We presented Optimistic Crash Consistency, a new crash-consistency protocol that eliminates disk-cache flushes in the common case. Both these techniques increase performance significantly while providing strong crash-consistency guarantees. Finally, we introduced the `osync()` primitive, which allows applications to order writes without making them durable.

In this chapter, we first summarize our analysis and solutions (§10.1). We then describe a set of lessons we have learned in the course of this dissertation work (§10.2). Finally, we conclude (§10.3).

10.1 Summary

This dissertation is comprised of two parts. In the first part, we analyzed the behavior of file systems when flushes are disabled, and when applications don't explicitly call `fsync()`. In the second part, we built upon the insights from the first part to develop new crash-consistency mechanisms, and new interfaces to help applications maintain crash consistency. We summarize each part in turn.

10.1.1 Crash-Consistency Analysis

The first part of this dissertation is about analyzing how crashes lead to file-system inconsistency, and how file systems affect application-level crash consistency. First, we studied the factors that affect whether a crash leads to file-system inconsistency. Some practitioners had observed that crashes don't lead to file-system inconsistency for certain workloads [286]. We termed this Probabilistic Crash Consistency [46], and built a framework to investigate the probability of file-system inconsistency for various workloads under different conditions. We found that for some workloads, such as large sequential writes or static web servers, a crash would rarely leave the file-system inconsistent. For other workloads, such as email servers or database update queries, there is a significant risk of inconsistency upon crash. Thus, for practitioners who seek to obtain high performance and strong crash consistency, simply turning off flushes is not enough.

Second, we studied file-system behavior that affects application-level crash consistency. For applications to be consistent after a crash, their updates to on-disk state need to be persisted in a specific order. Due to the high cost of `fsync()` calls, applications do not enforce this order; instead, they depend upon the file system persisting writes in the correct order. We defined the aspects of file-system behavior that affect application-level consistency as persistence properties [216]. We built a tool, the Block-Order Breaker (Bob), and studied how persistence properties varied across sixteen configurations of six widely-used file systems. We found that persistence properties varied widely among different file systems, and even among different configurations of the same file system. Thus, applications cannot depend upon the file-system persistence behavior to persist writes in the right order, and must instead do so via an explicit interface.

10.1.2 Crash-Consistency Solutions

In the second part of this dissertation, we presented solutions to the problems we discovered in the first part. First, we presented Backpointer-Based Consistency [47], a new crash-consistency solution that does not require disk writes to be ordered. Backpointer-Based Consistency embeds a backpointer into each object in the file system, and builds consistency based upon mutual agreement between objects. We implemented Backpointer-Based Consistency in the No-Order File System (NoFS), a variant of the ext2 file system. We showed that NoFS provides performance equivalent to that of ext2, and that it provides strong crash-consistency guarantees (similar to ext3). While NoFS provides excellent performance, it does not support atomic primitives such as `rename()`. A large number of applications use such primitives to safely update their on-disk state. Hence, the usability of NoFS was limited.

To address this limitation of NoFS, we designed Optimistic Crash Consistency, a new crash-consistency protocol [46]. Optimistic Crash Consistency takes advantage of the fact that ordering file-system updates could be accomplished by means significantly cheaper than disk-cache flushes. For example, atomicity and ordering are duals of each other: thus, making a group of updates atomic removes the need for ordering among them. We implemented the principles of Optimistic Crash Consistency in the Optimistic File System (OptFS). OptFS introduces two new interfaces: asynchronous durability notifications at the storage level, and the `osync()` primitive at the application level. OptFS decouples the ordering of writes from durability. Applications can use `osync()` to order their writes (without making them durable), thus building correct, efficient application update protocols. We show that two applications, Gedit [91] and SQLite [263], can utilize `osync()` effectively to maintain application-level consistency at high performance.

Working on this dissertation allowed us to understand the relative importance of different file-system design goals. While crash consistency is a basic requirement for a file system, it is not sufficient for many applications. Thus, while NoFS provides a consistent file system, many applications will need to be modified to run correctly on top of NoFS. Atomically updating on-disk state is an important concern for many applications. Such atomic updates require either a transaction-like atomicity primitive (that can include multiple files) or a combination of a single-file atomicity primitive (such as `rename()`) and a primitive to order file-system operations. With OptFS, we fulfill such application requirements in addition to providing file-system crash consistency.

10.2 Lessons Learned

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

10.2.1 The Importance of Interface Design

In this dissertation, we studied existing crash-consistency problems in file systems and applications. Most of these problems can be directly traced to bad interfaces provided by storage devices and file systems.

Storage Interface. Storage devices at the lower end of the market (such as SATA and IDE drives) expose the flush command for managing the disk cache. There are two problems with the flush interface: coarseness, and lack of visibility. First, the flush interface is too coarse: there is no option to selectively flush a few blocks to non-volatile storage. If the cache is filled with dirty data, out of which the file system only wants to make a few blocks durable, the current interface forces a lot of inefficient waiting. Second, file systems do not have visibility into the durability status of blocks in the disk cache. Combined with the coarseness of the flush interface, this leads to a lot of un-necessary waiting for the file system.

File-System Interface. File systems implement the POSIX interface [278] that includes the `fsync()` system call for making dirty writes durable. The `fsync()` call results in a flush command at the storage level, and therefore inherits the performance problems associated with the flush interface. As mentioned in Section 6.2.4, the `fsync()` interface couples together ordering and durability; applications desiring only ordering between writes are forced to pay the cost of durability.

These interface flaws have led to widespread problems: disk drives that lie about flushing [153, 226], system administrators who deliberately put their systems at risk for performance [59], systems that do not honor flush requests [294], and finally, application developers who do not use `fsync()` fearing the performance cost [280].

The importance of open interfaces. We believe that the closed nature of storage-device interfaces have resulted in significant additional complexity and loss of performance. Much of file-system research can be traced back to working around the pitfalls of badly-designed interfaces. We believe that making the device interface open will greatly aid the storage community. Several groups are working towards this with open-channel SSDs [205, 253, 300].

10.2.2 The Importance of Asynchrony

The benefits of asynchrony have long been known in the software world [8, 179]. For example, when asynchronous Javascript was introduced, it enabled the creation of exciting, responsive web applications such as Google Suggest and Google Maps [88].

Yet, the role of asynchrony in storage systems has been limited. Until the 1990s, storage systems were essentially synchronous, performing one operation at time [172, 250]. With the introduction of tag queuing in SCSI disks [15, 81, 304], disks could accept sixteen simultaneous requests. Unfortunately, many devices do not implement tag queuing correctly [174].

In this dissertation, we have shown that asynchronous, orderless I/O has significant advantages. First, not constraining the order of I/O allows large performance gains, especially on today's multi-tenant systems [283]. Recent work on non-volatile memory has shown that removing ordering constraints on I/O can increase performance by 30× [214].

Second, using interfaces such as asynchronous durability notifications allows each layer to introduce optimizations such as delaying, batching, or re-ordering I/O, without affecting the correctness of the file system or the application. Increasing the independence of each layer in the storage stack leads to a more robust storage system.

10.2.3 The Need for Better Tools

During the course of this dissertation work, we realized that there is a need for better frameworks for analyzing, testing, and verifying file-system crash consistency. Despite decades of research on file systems, there are few open-source tools available for inspecting file-system consistency.

Tools for Analyzing Crash Consistency. While working on Probabilistic Crash Consistency, we had to build a framework to analyze the different ordering relationships that need to hold for a journaling file system to remain consistent in the event of a crash. Analyzing these relationships eventually led us to discover Selective Data Journaling [46], the precise circumstances in which not journaling data blocks provides the same crash-consistency guarantees as when data blocks are journaled. Specifically, if the data block is not already part of a file, journaling it does not provide any additional crash guarantees. Selective Data Journaling improves performance significantly for workloads where large files are updated atomically via mechanisms such as `rename()`(*e.g.*, in text editors such as Gedit [91]).

In multiple interactions, researchers and developers in the file-system community have reported that the Selective Data Journaling idea was obvious once explained. However, this insight has eluded file-system researchers and practitioners for many years (although it is known in the software transactional memory community [7, 75, 108]). We believe this is due to two reasons: the inherent complexity of protocols like journaling, and the complexity associated with implementations such as ext4 [194]. Without using a framework to analyze the guarantees provided by these systems and the requirements for those guarantees, even simple insights are hard to obtain. Just as practitioners in distributed systems formally specify the design of their systems and verify optimizations [145, 194], we believe a similar effort is required in the realm of file systems.

Tools for Testing Crash Consistency. While building the No-Order File System and the Optimistic File System, we needed to test the reliability of the file system. We found that there were no open-source tools available to inject either targeted failures or random failures. As a consequence, we had to develop our own tools for this purpose. We have received requests from other universities to share these tools. Although the current versions of these tools are specific to NoFS and OptFS, we believe generalized versions of these tools would be greatly aid file-system development and research. We strongly advocate for better tools that aid file-system research to made open-source (*e.g.*, the Filebench suite [171]).

Tools for Verifying Crash Consistency. Modern storage stacks are complex and consist of many layers [283]. Crash consistency of the application on top of the stack depends on every layer between the application and storage handling requests such as flushes and FUA correctly. Configuration options at each layer (*e.g.*, journaling mode of file system) affects the crash consistency of the application.

We believe that storage stacks have grown so complex that it is not feasible for humans to observe the composition of a storage stack and reason about whether a given application will obtain crash consistency on top of that stack. Furthermore, with customized on-demand software-defined storage [79, 193, 295, 296], a human verifying a composed storage stack is not practical.

We believe that the design of each layer (the aspects that relate to crash consistency) must be formally defined, and that we should develop tools to automatically verify the crash guarantees that a storage stack claims to provide. We have undertaken initial efforts in this direction [11].

10.3 Closing Words

With the world becoming increasingly digitized, the safety of data is paramount. The threat of power loss and crashes still remains; thus, it is important to design systems that can safely recover in the face of interruptions. Given that most applications access storage via a file system, file-system crash consistency is an important problem.

However, existing solutions degrade performance to such a large extent that many practitioners believe that systems *cannot* achieve both strong crash consistency and high performance at the same time. In this dissertation, we show that this dichotomy is false; we present solutions that offer *both* high performance and strong crash-consistency guarantees.

Most of the crash-consistency problems described in this dissertation arise from poorly-designed interfaces. In this dissertation, we design new interfaces for both file systems and storage, and show that these interfaces allow richer functionality and better performance.

The storage landscape is set to change drastically, with the coming of new storage technologies such as 3D XPoint Storage [83], and the advent of software-defined storage [79, 193, 295]. We urge the designers of these systems to craft the storage-system interfaces carefully, keeping in mind that interfaces are extremely resistant to change. As we show in this dissertation, with well-designed interfaces, these new technologies could offer both correctness and performance.

Bibliography

[1] Amazon Instance Storage. http://docs.aws.amazon.com/AWSEC-2/latest/UserGuide/InstanceStorage.html.

[2] LevelDB - Issue 189: Possible bug: fsync() required after calling rename(). https://code.google.com/p/leveldb/issues/detail?id=189.

[3] Necessary step(s) to synchronize filename operations on disk. http://austingroupbugs.net/view.php?id=672.

[4] OpenStack EC2 Compatibility API. http://docs.openstack.org/admin-guide-cloud/content/instance-mgmt-ec2compat.html.

[5] Swift API Feature Comparison. https://wiki.openstack.org/wiki/Swift/APIFeatureComparison.

[6] Vijay Kumar Adhikari, Yang Guo, Fang Hao, Matteo Varvello, Volker Hilt, Moritz Steiner, and Zhi-Li Zhang. Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery. In *The 31st Annual IEEE International Conference on Computer Communications (IEEE INFOCOM 2012)*, pages 1620–1628. IEEE, 2012.

[7] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, New York, 2006. ACM.

[8] Atul Adya, Jon Howell, Marvin Theimer, Bill Bolosky, and John Douceur. Cooperative Task Management without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[9] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[10] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: a Protoype Phase Change Memory Storage Array. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*. USENIX Association, 2011.

[11] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Aws Albarghouthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Beyond Storage APIs: Provable Semantics for Storage Stacks. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[12] Alexey Kopytov. SysBench: a system performance benchmark. `http://sysbench.sourceforge.net/index.html`, 2004.

[13] Amazon. Amazon Elastic Block Store (EBS). `http://aws.amazon.com/ebs/`, 2012.

[14] Amazon. Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3/`, 2012.

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

[16] Apache. Apache Zookeeper. `http://zookeeper.apache.org/`.

[17] Apple. exchangedata (2) Mac OS X Developer Tools Manual Page. `https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/exchangedata.2.html`.

[18] Apple. Technical Note TN1150. http://dubeiko.com/development/FileSystems/HFSPLUS/tn1150.html, March 2004.

[19] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A View of Cloud Computing, 2010.

[20] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.

[21] Jens Axboe. Linux IO block – present and future. In *Ottawa Linux Symposium*, pages 51–61, 2004.

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

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

[24] Josef Basik. dm: add dm-power-fail target. https://lwn.net/Articles/623834/.

[25] Josef Basik. Power-Failure Testing: Making Filesystems More Robust Even When Power Goes Out at the Worst Possible Time. In *The 2015 Linux Storage, Filesystem, and Memory Management Summit*, 2015.

[26] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, February 2010.

[27] Steve Best. JFS Overview. http://jfs.sourceforge.net/project/pub/jfs.pdf, 2000.

[28] Michael Blennerhassett and Robert G. Bowman. A Change In Market Microstructure: The Switch To Electronic Screen Trading on The New Zealand Stock Exchange. *Journal of International Financial Markets, Institutions and Money*, 8(3):261–276, 1998.

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

[30] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2006.

[31] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux As A Case Study: Its Extracted Software Architecture. In *Proceedings of the 21st International Conference on Software Engineering*, pages 555–563. ACM, 1999.

[32] Box. Box | Secure Content & Online File Sharing for Businesses. `https://www.box.com/`, 2015.

[33] Thomas C. Bressoud, Tom Clark, and Ti Kan. The Design and Use of Persistent Memory on the DNCP Hardware Fault-Tolerant Platform. In *International Conference on Dependable Systems and Networks (DSN)*, pages 487–492. IEEE, 2001.

[34] Alan D. Brunelle and Jens Axboe. Blktrace User Guide, 2007.

[35] John S. Bucy, Jiri Schindler, Steven W. Schlosser, and Gregory R. Ganger. The DiskSim Simulation Environment Version 4.0 Reference Manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University, May 2008.

[36] Ed Burnette. *Hello, Android: Introducing Google's Mobile Development Platform*. Pragmatic Bookshelf, 2009.

[37] Pei Cao, Swee Boon Lin, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP Parallel RAID Architecture. *ACM Transactions on Computer Systems (TOCS)*, 12(3):236–269, 1994.

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

[39] Michael J. Carey, David J DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 91–100. Morgan Kaufmann Publishers Inc., 1986.

[40] J. D. Carothers, R. K. Brunner, J. L. Dawson, M. O. Halfhill, and R. E. Kubec. A New High Density Recording System: the Ibm 1311 Disk Storage Drive with Interchangeable Disk Packs. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, pages 327–340. ACM, 1963.

[41] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*, Atlanta, Georgia, December 2010.

[42] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, Frank W. King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, et al. A History and Evaluation of System R. *Communications of the ACM*, 24(10):632–646, 1981.

[43] Philip Y. Chang and William W Myre. OS 2 EE Database Manager Overview And Technical Highlights. *IBM Systems Journal*, 27(2):105, 1988.

[44] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. Mime: A High Performance Parallel Storage Device with Strong Recovery Guarantees. Technical Report HPL-CSP-92-9rev1, HP Laboratories, November 1992.

[45] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[46] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[47] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, California, February 2012.

[48] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *ACM Transactions on Storage (TOS)*, 4(4):14, 2009.

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

[50] Chris Mason. Btrfs Mailing List. Re: Ordering Of Directory Operations Maintained Across System Crashes In Btrfs? `http://www.spinics.net/lists/linux-btrfs/msg32215.html`, 2014.

[51] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode File System. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 43–60, San Francisco, California, January 1992.

[52] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules, and Alistair Veitch. LazyBase: Trading Freshness for Performance in a Scalable Database. In *Proceedings of the EuroSys Conference (EuroSys '12)*, pages 169–182, Bern, Switzerland, April 2012.

[53] CNN. Manufacturer Blames Super Bowl Outage on Incorrect Setting. `http://www.cnn.com/2013/02/08/us/superdome-power-outage/`, 2013.

[54] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction Support for Next-Generation, Solid-State Drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 197–212. ACM, 2013.

[55] ComputerWorld. Viking Combines DRAM Module with Flash for Auto Backup. `http://www.computerworld.com/article/2499047/data-center/viking-combines-dram-module-with-flash-for-auto-backup.html`.

[56] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[57] George Copeland, Tom W. Keller, Ravi Krishnamurthy, and Marc G. Smith. The Case for Safe RAM. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB 15)*, pages 327–335, Amsterdam, The Netherlands, August 1989.

[58] Jonathan Corbet. Barriers and Journaling Filesystems. `http://lwn.net/Articles/283161`, May 2008.

[59] Jonathan Corbet. That massive filesystem thread. `http://lwn.net/Articles/326471/`, March 2009.

[60] Jonathan Corbet. The end of block barriers. `https://lwn.net/Articles/400541/`, August 2010.

[61] Craig Balding. A Question of Integrity: To MD5 or Not to MD5. `http://cloudsecurity.org/blog/2008/06/25/a-question-of-integrity-to-md5-or-not-to-md5.html`, June 2008.

[62] Charles D. Cranor and Gurudatta M. Parulkar. The UVM Virtual Memory System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '99)*, Monterey, California, June 1999.

[63] Christoffer Dall and Jason Nieh. KVM/ARM: the Design and Implementation of the Linux ARM Hypervisor. *ACM SIGARCH Computer Architecture News*, 42(1):333–348, 2014.

[64] Chris J. Date and Colin J. White. *A Guide to DB2*. Addison Wesley Publishing Company, 1989.

[65] Wiebren de Jonge, Frans M. Kaashoek, and Wilson 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)*, pages 15–28, Asheville, North Carolina, December 1993.

[66] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[67] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-Based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 25–36. ACM, 2011.

[68] Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

[69] Catherine M. DesRoches, Eric G. Campbell, Sowmya R. Rao, Karen Donelan, Timothy G. Ferris, Ashish Jha, Rainu Kaushal, Douglas E. Levy, Sara Rosenbaum, Alexandra E. Shields, et al. Electronic Health Records in Ambulatory Care – A National Survey of Physicians. *New England Journal of Medicine*, 359(1):50–60, 2008.

[70] David J. DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-I Hsiao, Rick Rasmussen, et al. The Gamma Database Machine Project. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):44–62, 1990.

[71] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD Conference on the Management of Data (SIGMOD '84)*, pages 1–8, Boston, Massachusetts, June 1984.

[72] Jeremy Dion. The Cambridge File Server. *SIGOPS Operating Systems Review*, 14:26–35, October 1980.

[73] Linux Documentation. fsync(2) - Linux Man Page. `http://linux.die.net/man/2/fsync`.

[74] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.

[75] Aleksandar Dragojevic, Yang Ni, and Ali-Reza Adl-Tabatabai. Optimizing Transactions for Captured Memory. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, pages 214–222, New York, 2009.

[76] Stéphane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[77] Ray Duncan. *Advanced MS-DOS Programming*. Microsoft Press Redmond, WA, 1988.

[78] Frank C. Eigler, Vara Prasad, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. Architecture of Systemtap: A Linux Trace/Probe Tool. `http://sourceware.org/systemtap/archpaper.pdf`, July 2005.

[79] EMC. Rethink Storage: Transform the Data Center with EMC ViPR Software-Defined Storage. `http://pages.cs.wisc.edu/~vijayc/thesis-refs/emc-sds.pdf`.

[80] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[81] Gary Field, Peter M. Ridge, John Lohmeyer, Gerhard Islinger, and Stephen Grall. *The Book of SCSI: I/O for the New Millennium*. No Starch Press, 2000.

[82] Stony Brook University File System Storage Lab (FSL). Filebench Benchmark. `http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench`, 2011.

[83] Forbes. Intel And Micron Announce Breakthrough Faster-Than-Flash 3D XPoint Storage Technology. `http://onforb.es/1Mt49VW`, July 2015.

[84] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 307–320, Stevenson, Washington, October 2007.

[85] E. Gal and S. Toledo. A Transactional Flash File System for Microcontrollers. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, Anaheim, California, April 2005.

[86] Gregory R. Ganger, Marshall Kirk McKusick, Craig A.N. Soules, and Yale N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2), May 2000.

[87] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[88] Jesse James Garrett et al. Ajax: A New Approach To Web Applications. 2005.

[89] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8(2):3–10, 1985.

[90] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: An Active Distributed Key-Value Store. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 323–336, Vancouver, Canada, December 2010.

[91] GNOME. Apps/Gedit - GNOME Wiki! `https://wiki.gnome.org/Apps/Gedit`.

[92] GNU. GNU Database Manager (GDBM). `http://www.gnu.org.ua/software/gdbm/gdbm.html`, 1979.

[93] Google. LevelDB. `https://code.google.com/p/leveldb/`, 2011.

[94] Google. Google Chrome Browser. `http://google.com/chrome/`, 2013.

[95] Catherine Gowthorpe and Oriol Amat. External Reporting of Accounting and Financial Information via the Internet in Spain. *European Accounting Review*, 8(2):365–371, 1999.

[96] James N. Gray. *Notes on Data Base Operating Systems*. Springer, 1978.

[97] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB 7)*, Cannes, France, September 1981.

[98] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–242, 1981.

[99] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable And Flexible Data Center Network. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 51–62. ACM, 2009.

[100] Robert Grimm, Wilson C. Hsieh, Frans M. Kaashoek, and Wiebren De Jonge. Atomic Recovery Units: Failure Atomicity for Logical Disks. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 26–36. IEEE, 1996.

[101] The Open Group. Rename: The Open Group Base Specifications Issue 6. `http://pubs.opengroup.org/onlinepubs/009695399/functions/rename.html`, 2004.

[102] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[103] Haryadi Sudirman Gunawi. *Towards Reliable Storage Systems*. The University of Wisconsin-Madison, 2009.

[104] Donald J. Haderle and Robert D. Jackson. IBM Database 2 Overview. *IBM Systems Journal*, 23(2):112–125, 1984.

[105] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[106] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[107] John M. Harker, Dwight W. Brede, Robert E. Pattison, George R. Santana, and Lewis G. Taft. A Quarter Century Of Disk File Innovation. *IBM Journal of Research and Development*, 25(5):677–690, 1981.

[108] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing Memory Transactions. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 14–25, New York, NY, USA, 2006. ACM.

[109] Rober Haskin, Yoni Malachi, and Gregory Chan. Recovery management in quicksilver. *ACM Transactions on Computer Systems (TOCS)*, 6(1):82–108, 1988.

[110] Kristiina Häyrinen, Kaija Saranto, and Pirkko Nykänen. Definition, Structure, Content, Use and Impacts Of Electronic Health Records: A Review of the Research Literature. *International Journal of Medical Informatics*, 77(5):291–304, 2008.

[111] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. Reducing Fsck Time For Ext2 File Systems. In *Ottawa Linux Symposium (OLS '06)*, Ottawa, Canada, July 2006.

[112] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using Divide-And-Conquer to Improve File System Reliability and Repair. In *IEEE 2nd Workshop on Hot Topics in System Dependability (HotDep '06)*, Seattle, Washington, November 2006.

[113] Maurice Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Transactions on Database Systems (TODS)*, 15(1):96–124, 1990.

[114] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[115] David A. Holland, Elaine Lee Angelino, Gideon Wald, and Margo I. Seltzer. Flash Caching on the Storage Client. In *USENIX ATC'13 Proceedings of the 2013 USENIX conference on Annual Technical Conference*. USENIX Association, 2013.

[116] Yiming Huai. Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects. *AAPPS Bulletin*, 18(6):33–40, 2008.

[117] HyperSQL. HSQLDB. `http://www.hsqldb.org/`.

[118] IBM. IBM 350 Disk Storage Unit. http://www-03.ibm.com/ibm/history/exhibits/storage/storage_350.html, September 1956.

[119] InfoStor. Emerging Trends in Software Defined Storage. http://www.infostor.com/storage-management/virtualization/emerging-trends-in-software-defined-storage-1.html.

[120] InfoWorld. Microsoft Loses Sidekick Users' Personal Data. http://www.infoworld.com/article/2629952/smartphones/microsoft-loses-sidekick-users--personal-data.html, October 2009.

[121] Geoff Ingram. *High-Performance Oracle: Proven Methods for Achieving Optimum Performance and Availability*. John Wiley & Sons, 2002.

[122] Intel. More Power-Loss Data Protection with Intel SSD 320 Series. http://web.archive.org/web/20140207071838/http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/ssd-320-series-power-loss-data-protection-brief.pdf.

[123] Intel. Intel 64 and IA-32 Architectures Software Developers Manual Volume 1: Basic Architecture. http://download.intel.com/design/processor/manuals/253665.pdf, March 2010.

[124] D. M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical Report HPL-CSP-91-7, Hewlett Packard Laboratories, 1991.

[125] Pankaj K. Jain. Financial Market Design and the Equity Premium: Electronic Versus Floor Trading. *The Journal of Finance*, 60(6):2955–2985, 2005.

[126] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *Proceedings of the USENIX Annual Technical Conference*, pages 309–320, 2013.

[127] Ashish K. Jha, Catherine M. DesRoches, Eric G. Campbell, Karen Donelan, Sowmya R. Rao, Timothy G. Ferris, Alexandra Shields, Sara Rosenbaum, and David Blumenthal. Use of Electronic Health Records in US Hospitals. *New England Journal of Medicine*, 360(16):1628–1638, 2009.

[128] Minwen Ji, Alistair C Veitch, John Wilkes, et al. Seneca: Remote Mirroring Done Write. In *Proceedings of the USENIX Annual Technical Conference*, pages 253–268, 2003.

[129] Rob Johnson and David Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Proceedings of the 13th USENIX Security Symposium (Sec '04)*, San Diego, California, August 2004.

[130] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: a File System for Virtualized Flash Storage. *ACM Transactions on Storage (TOS)*, 6(3):14, 2010.

[131] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating Hardware Device Failures in Software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[132] Asim Kadav and Michael M. Swift. Understanding Modern Device Drivers. *ACM SIGARCH Computer Architecture News*, 40(1):87–98, 2012.

[133] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 97–108. ACM, 2013.

[134] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for Disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.

[135] Kernel.Org. Index of /doc/Documentation/filesystems. `https://www.kernel.org/doc/Documentation/filesystems/`.

[136] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI '07)*, Cambridge, Massachusetts, April 2007.

[137] KnowledgeTek. Serial ATA Specification Rev. 3.0 Gold. `http://www.knowledgetek.com/datastorage/courses/SATA_3.0-8.14.09(CD).pdf`, 2009.

[138] Donald E. Knuth. The Art of Computer Programming (Volume 2), 1981.

[139] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write Policies for Host-Side Flash Caches. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, pages 45–58, San Jose, California, February 2013.

[140] Charles M. Kozierok. Overview and History of the SCSI Interface. `http://www.pcguide.com/ref/hdd/if/scsi/over-c.html`, 2001.

[141] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[142] Hsiang-Tsung Kung and John T Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[143] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600. ACM, 2010.

[144] Stefan Lai. Current Status of the Phase Change Memory and Its Future. In *International Electron Devices Meeting (IEDM'03) Technical Digest*. IEEE, 2003.

[145] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[146] Ubuntu Bugs LaunchPad. Bug #317781: Ext4 Data Loss. `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781?comments=all`.

[147] Karen Layne and Jungwoo Lee. Developing Fully Functional E-Government: a Four Stage Model. *Government Information Quarterly*, 18(2):122–136, 2001.

[148] A. Leach. Level 3's UPS Burnout Sends Websites Down In Flames. `http://www.theregister.co.uk/2012/07/10/datacentrepowercut/`, 2012.

[149] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[150] Luke Kenneth Casson Leighton. Analysis of SSD Reliability During Power-Outages. `http://lkcl.net/reports/ssd_analysis.html`.

[151] Andrew W. Leung, Shankar Pasupathy, Garth R. Goodson, and Ethan L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 213–226, Boston, Massachusetts, June 2008.

[152] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[153] Mac Developer Library. fsync(2) Mac OS X Developer Tools Manual Page. `https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/fsync.2.html`.

[154] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[155] Linus Torvalds. Git Mailing List. Re: what's the current wisdom on git over NFS/CIFS? `http://marc.info/?l=git&m=124839484917965&w=2`, 2009.

[156] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[157] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. LightTx: A Lightweight Transactional Design in Flash-Based SSDs to Support Flexible Transactions. In *Proceedings of the 31st International Conference on Computer Design (ICCD)*, pages 115–122. IEEE, 2013.

[158] Youyou Lu, Jiwu Shu, and Long Sun. Blurred Persistence in Transactional Persistent Memory. In *Proceedings of the 31st IEEE Conference on Massive Data Storage (MSST '15)*, Santa Clara, California, May 2015.

[159] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-Ordering Consistency for Persistent Memory. In *Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD)*, pages 216–223. IEEE, 2014.

[160] Youyou Lu, Jiwu Shu, and Wei Wang. Reconfs: a Reconstructable File System on Flash Storage. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, pages 75–88, Santa Clara, California, February 2014.

[161] Youyou Lu, Jiwu Shu, and Wei Wang. ReconFS: a Reconstructable File System on Flash Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 75–88, Santa Clara, CA, 2014. USENIX.

[162] Peter Lyman and Hal Varian. How Much Information? 2004.

[163] Peter Macko, Margo Seltzer, and Keith A. Smith. Tracking Back References in a Write-Anywhere File System. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[164] MariaDB. Fusion-io NVMFS Atomic Write Support. `https://mariadb.com/kb/en/mariadb/fusion-io-nvmfs-atomic-write-support/`, April 2013.

[165] Martin F. Krafft. XFS and Zeroed Files. http://madduck.net/blog/2006.08.11:xfs-zeroes/, August 2006.

[166] Chris Mason. The Btrfs Filesystem. oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf, September 2007.

[167] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilge, Alex Tomas, and Laurent Vivier. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[168] Matt Mackall. Mercurial. http://mercurial.selenic.com/, 2005.

[169] Marissa Mayer. The Physics of Data. http://www.parc.com/event/936/innovation-at-google.html, August 2009.

[170] Kirby McCoy. *VMS file system internals*. Digital Press, 1990.

[171] Richard McDougall and Jim Mauro. Filebench. http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench, 2005.

[172] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[173] Marshall Kirk McKusick. Running 'fsck' in the Background. In *Proceedings of BSDCon 2002 (BSDCon '02)*, San Fransisco, California, February 2002.

[174] Marshall Kirk McKusick. Disks from the Perspective of a File System. *Communications of the ACM*, 55(11):53–55, 2012.

[175] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[176] R. McMillan. Amazon Blames Generators For Blackout That Crushed Netflix. http://www.wired.com/wiredenterprise/2012/07/amazonexplains/, 2012.

[177] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '91)*, pages 33–43, Dallas, Texas, January 1991.

[178] Michael M. Swift and Brian N. Bershad and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[179] Brenda M. Michelson. Event-Driven Architecture Overview. *Patricia Seybold Group*, 2, 2006.

[180] James Mickens, Edmund B. Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, Cloud-Scale Block Storage for Cloud-Oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 257–273, Seattle, WA, April 2014. USENIX Association.

[181] Microsoft. ReplaceFile function (Windows). `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365512%28v=vs.85%29.aspx`.

[182] Microsoft. Microsoft OneDrive. `https://onedrive.live.com/about/en-us/`, 2015.

[183] R. Miller. Human Error Cited In Hosting.Com Out-Age. `http://www.datacenterknowledge.com/archives/2012/07/28/human-error-cited-hosting-com-outage/`, 2012.

[184] R. Miller. Power Outage Hits London Data Center. `http://www.datacenterknowledge.com/archives/2012/07/10/power-outage-hits-london-data-center/`, 2012.

[185] R. Miller. Data Center Outage Cited In Visa Downtime Across Canada. `http://www.datacenterknowledge.com/archives/2013/01/28/data-center-outage-cited-in-visa-downtime-across-canada/`, 2013.

[186] R. Miller. Power Outage Knocks Dreamhost Customers Offline. `http://www.datacenterknowledge.com/archives/2013/03/20/power-outage-knocks-dreamhost-customers-offline/`, 2013.

[187] Changwoo Min, Woon-Hak Kang, T. Kim, and S. W. Lee. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, 2015.

[188] 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.

[189] M. Jae Moon. The Evolution of E-Government Among Municipalities: Rhetoric or Reality? *Public Administration Review*, 62(4):424–433, 2002.

[190] Iulian Moraru, David G. Andersen, Michael Kaminsky, Nathan Binkert, Niraj Tolia, Reinhard Munz, and Parthasarathy Ranganathan. Persistent, protected and cached: Building blocks for main memory data stores. Technical Report CMU-PDL-11-114, Carnegie Mellon University Parallel Data Lab, 2011.

[191] Rajeev Nagar. *Windows NT File System Internals: A Developer's Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[192] NBC. Power Outage Affects Thousands, Including Columbus Hospital. `http://columbus.gotnewswire.com/news/power-outage-affects-thousands-including-columbus-hospital`, 2014.

[193] NetApp. NetApp Software-Defined Storage. `http://www.netapp.com/us/technology/software-defined-storage/`.

[194] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, 2015.

[195] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 1–15, Hollywood, CA, 2012. USENIX.

[196] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 1–16, Seattle, Washington, November 2006.

[197] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a Proof Assistant for Higher-Order Logic*. Springer Science & Business Media, 2002.

[198] Chris Nyberg. Gensort Data Generator. `http://www.ordinal.com/gensort.html`, 2009.

[199] Jason Olson. Enhance Your Apps With File System Transactions. `http://msdn.microsoft.com/en-us/magazine/cc163388.aspx`, July 2007.

[200] Michael A. Olson. The Design and Implementation of the Inversion File System. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '93)*, San Diego, California, January 1993.

[201] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[202] Oracle. Oracle Berkeley DB C Version. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`, 1994.

[203] John K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, CA, June 1990.

[204] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, Washington, December 1985.

[205] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *ACM SIGPLAN Notices*, volume 49, pages 471–484. ACM, 2014.

[206] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA-11)*, San Antonio, Texas, February 2011.

[207] Mike Owens and Grant Allen. *SQLite*. Springer, 2010.

[208] Stan Park, Terence Kelly, and Kai Shen. Failure-Atomic Msync (): a Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 225–238. ACM, 2013.

[209] Sunhwa Park, Ji Hyun Yu, and Seong Yong Ohm. Atomic Write FTL for Robust Flash File System. In *Proceedings of the Ninth International Symposium on Consumer Electronics (ISCE 2005)*, pages 155–160. IEEE, 2005.

[210] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok. I^{3}FS: An In-kernel Integrity Checker and Intrusion detection File System. In *Proceedings of the 18th Annual Large Installation System Administration Conference (LISA '04)*, pages 69–79, Atlanta, Georgia, November 2004.

[211] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[212] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. Snapmirror®: File System Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. USENIX Association, 2002.

[213] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast Consistency Checking for the Solaris File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, pages 77–89, New Orleans, Louisiana, June 1998.

[214] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture*, pages 265–276. IEEE Press, 2014.

[215] Zachary N. J. Peterson. Data Placement for Copy-on-write Using Virtual Contiguity. Master's thesis, U.C. Santa Cruz, 2002.

[216] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

[217] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash consistency. *Queue*, 13(7):20:20–20:28, July 2015.

[218] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating Systems Transactions. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[219] Postgres. PostgreSQL: Documentation: 9.1: WAL Internals. `http://www.postgresql.org/docs/9.1/static/wal-internals.html`.

[220] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[221] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[222] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[223] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[224] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable Write-back for Client-Side Flash Caches. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, pages 451–462. USENIX Association, 2014.

[225] R1Soft. Disk Safe Best Practices. `http://wiki.r1soft.com/display/CDP3/Disk+Safe+Best+Practices`, December 2011.

[226] Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Coerced Cache Eviction and Discreet-Mode Journaling: Dealing with Misbehaving Disks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'11)*, Hong Kong, China, June 2011.

[227] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems (Third Edition)*. McGraw-Hill, 2004.

[228] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C.Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[229] Hans Reiser. ReiserFS. `www.namesys.com`, 2004.

[230] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, San Jose, CA, 2013. USENIX.

[231] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance With Stateless Caching and Bulk Insertion. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, pages 237–248. IEEE, 2014.

[232] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. In *Proceedings of the 4th ACM Symposium on Operating Systems Principles (SOSP '73)*, Yorktown Heights, New York, October 1973.

[233] Rick Rogers, John Lombardo, Zigurd Mednieks, and Blake Meike. *Android Application Development: Programming With the Google SDK*. O'Reilly Media, Inc., 2009.

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

[235] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[236] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik 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.

[237] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[238] Sanjay Ghemawat and Howard Gobioff and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

[239] Mahadev Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems (TOCS)*, 12(1):33–57, 1994.

[240] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: a Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 267–280. ACM, 2012.

[241] Bill N. Schilit, Marvin M. Theimer, and Brent B. Welch. Customizing Mobile Applications. In *USENIX Symposium on Mobile and Location-independent Computing*, pages 129–138, Cambridge, Massachusetts, 1993.

[242] Eric Schrock. UFS/SVM vs. ZFS: Code Complexity. `http://blogs.sun.com/eschrock/`, November 2005.

[243] Greg Schulz. Part II: How Many IOPS Can a HDD, HHDD or SSD Do With VMware? `http://storageioblog.com/part-ii-iops-hdd-hhdd-ssd/`, 2013.

[244] P. Schwarz, Walter Chang, Johann Christoph Freytag, G. Lohman, John McPherson, C. Mohan, and Hamid Pirahesh. Extensibility in the Starburst Database System. In *Proceedings on the 1986 International Workshop on Object-Oriented Database Systems*, pages 85–92. IEEE Computer Society Press, 1986.

[245] Michael L Scott. Shared-Memory Synchronization. *Synthesis Lectures on Computer Architecture*, 8(2):1–221, 2013.

[246] Seagate. Transition to Advanced Format 4K Sector Hard Drives. `http://www.seagate.com/tech-insights/advanced-format-4k-sector-hard-drives-master-ti/`, 2010.

[247] Seagate Forums. ST3250823AS (7200.8) ignores FLUSH CACHE in AHCI mode. `http://bit.ly/xcSAUV`, September 2011.

[248] Russell Sears and Eric Brewer. Stasis: Flexible Transactional Storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 29–44, Seattle, Washington, November 2006.

[249] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '93)*, pages 307–326, San Diego, California, January 1993.

[250] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '90)*, pages 313–324, Washington, D.C, January 1990.

[251] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 71–84, San Diego, California, June 2000.

[252] Margo I. Seltzer and Michael Stonebraker. Transaction Support in Read Optimized and Write Optimized File Systems. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB 16)*, pages 174–185, Brisbane, Australia, August 1990.

[253] Woong Shin, Myeongcheol Kim, Kyudong Kim, and Heon Y. Yeom. Providing QoS Through Host Controlled Flash SSD Garbage Collection and Multiple SSDs. In *International Conference on Big Data and Smart Computing (BigComp)*, pages 111–117. IEEE, 2015.

[254] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[255] Dick Sites. How Fast Is My Disk? Systems Seminar at the University of Wisconsin-Madison, January 2013. `http://www.cs.wisc.edu/event/how-fast-my-disk`.

[256] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A Logic of File Systems. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 1–15, San Francisco, California, December 2005.

[257] Stewart Smith. Eat My Data: How Everybody Gets File I/O Wrong. In *OSCON*, Portland, Oregon, July 2008.

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

[259] OCZ Storage Solutions. Vertex 2 Pro. `http://ocz.com/consumer/vertex-2-pro-sata-2-ssd`.

[260] Jon A. Solworth and Cyril U. Orji. Write-Only Disk Caches. In *Proceedings of the 1990 ACM SIGMOD Conference on the Management of Data (SIGMOD '90)*, pages 123–132, Atlantic City, New Jersey, June 1990.

[261] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, pages 29–42, San Francisco, California, February 2009.

[262] SQLite. How To Corrupt Your Database Files. `http://www.sqlite.org/howtocorrupt.html`.

[263] SQLite. SQLite Transactional Sql Database Engine. `http://www.sqlite.org/`.

[264] Marting Steigerwald. Imposing Order. Linux Magazine, May 2007.

[265] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, pages 79–90, Boston, Massachusetts, June 2001.

[266] Curtis E. Stevens. SATA IO NCQ. 2006.

[267] Louis D. Stevens. The Evolution of Magnetic Storage. *IBM Journal of Research and Development*, 25(5):663–676, 1981.

[268] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[269] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. In *IEEE Transactions on Knowledge and Data Engineering*, pages 340–355, 1986.

[270] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The Missing Memristor Found. *Nature*, 453:80–83, 2008.

[271] Sun Microsystems. MySQL White Papers, 2008.

[272] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[273] Symas. Lightning Memory-Mapped Database (LMDB). `http://symas.com/mdb/`, 2011.

[274] Technical Committee T10. T10 Data Integrity Field standard. `http://www.t10.org/`, 2009.

[275] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Jooyoung Hwang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Towards Efficient, Portable Application-Level Consistency. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep '13)*, Farmington, PA, November 2013.

[276] The ADvanced Systems Laboratory (ADSL). The No-Order File System. `http://research.cs.wisc.edu/adsl/Software/nofs/`.

[277] The ADvanced Systems Laboratory (ADSL). The Optimistic File System. `http://research.cs.wisc.edu/adsl/Software/optfs/`.

[278] The Open Group. POSIX.1-2008 IEEE Std 1003.1. http://pubs.opengroup.org/onlinepubs/9699919799/, 2013.

[279] The PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org/.

[280] Theodore Ts'o. Don't fear the fsync! http://thunk.org/tytso/blog/2009/03/15/dont-fear-the-fsync/.

[281] Theodore Ts'o. An Automated XFStests Infrastructure Using KVM. http://lwn.net/Articles/592783, March 2014.

[282] Theodore Ts'o (tytso). Comment #45 : Bug #317781 : Bugs : linux package : Ubuntu. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781/comments/45, March 2009.

[283] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: a Software-Defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.

[284] R Kent Treiber. *Systems Programming: Coping with Parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[285] Theodore Ts'o. http://e2fsprogs.sourceforge.net, June 2001.

[286] Theodore Tso. Re: [PATCH 0/4] (RESEND) ext3[34] barrier changes. Linux Kernel Mailing List. http://article.gmane.org/gmane.comp.file-systems.ext4/6662, May 2008.

[287] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

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

[289] Stephen C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[290] Ubuntu LaunchPad. XFS Leaves Garbage in File if App Does Write-New-Then-Rename Without f(fata)sync. https://bugs.launchpad.net/ubuntu/+source/linux-source-2.6.15/+bug/37435.

[291] Usenix. Usenix Research in Linux File and Storage Technologies Summit. https://www.usenix.org/conference/linuxfastsummit14, Feb 2014.

[292] J. Verge. Internap Data Center Outage Takes Down Livestream And Stackexchange. http://www.datacenterknowledge.com/archives/2014/05/16/internap-data-center-outage-takes-livestream-stackexchange/, 2014.

[293] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Mannarswamy, Terence Kelly, and Charles B. Morrey. Failure-Atomic Updates of Application Data in a Linux File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 203–211. USENIX Association, 2015.

[294] VirtualBox Manual. Responding to Guest IDE/SATA Flush Requests. http://www.virtualbox.org/manual/ch12.html.

[295] VMWare. Software-Defined Storage (SDS) and Storage Virtualization. http://www.vmware.com/software-defined-datacenter/storage.

[296] VMWare. The VMware Perspective on Software-Defined Storage. http://www.vmware.com/files/pdf/solutions/VMware-Perspective-on-software-defined-storage-white-paper pdf.

[297] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, December 1999.

[298] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[299] An-I A. Wang, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[300] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014.

[301] Xi Wang. Building Crash-Safe Applications with Crakit. Personal Communication, 2015.

[302] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus Scalable Block Store. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 357–370, Lombard, Illinois, April 2013.

[303] Hakim Weatherspoon, Lakshmi Ganesh, Tudor Marian, Mahesh Balakrishnan, and Ken Birman. Smoke and Mirrors: Reflecting Files at a Geographically Remote Location Without Loss of Performance. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, pages 211–224, San Francisco, California, February 2009.

[304] Ralph O. Weber. SCSI Architecture Model - 3 (SAM-3). `http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf`, September 2004.

[305] Markus Wenzel. Isar–a Generic Interpretative Approach to Readable Formal Proof Documents. In *Theorem Proving in Higher Order Logics*, pages 167–183. Springer, 1999.

[306] Richard Winter. Why Are Data Warehouses Growing So Fast? `http://www.b-eye-network.com/view/7188`, 2008.

[307] R. S. V Wolffradt. Fire In Your Data Center: No Power, No Access, Now What? `http://www.govtech.com/state/Fire-in-your-Data-Center-No-Power-No-Access-Now-What.html`, 2014.

[308] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '94)*, pages 241–251, Nashville, Tennessee, May 1994.

[309] Michael Wu and Willy Zwaenepoel. eNVy: a Non-Volatile, Main Memory Storage System. In *ACM SigPlan Notices*, volume 29, pages 86–97. ACM, 1994.

[310] Xingbo Wu, Zili Shao, and Song Jiang. Selfie: Co-Locating Metadata and Data to Enable Fast Virtual Block Devices. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 2. ACM, 2015.

[311] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

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

[313] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How Do Fixes Become Bugs? – A Comprehensive Characteristic Study on Incorrect Fixes in Commercial and Open Source Operating Systems. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the Acm Sigsoft Symposium on the Foundations of Software Engineering (FSE11)*, 2011.

[314] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[315] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. ViewBox: Integrating Local File Systems with Cloud Storage Services. In *Proceedings of the 12th Conference on File and Storage Technologies (FAST '14)*, Santa Clara, California, February 2014.

[316] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[317] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the Robustness of SSDs Under Power Fault. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, pages 271–284, San Jose, California, February 2013.

A

Appendix A

A.1 Source Code

We strongly believe that releasing research prototypes helps in reproducing results, building on prior research, and increasing impact. To this end, we have made both the major systems developed as part of this dissertation work publicly available.

The source code for NoFS can be obtained at: `http://www.cs.wisc.edu/adsl/Software/nofs`. The source code for OptFS can be obtained at: `http://www.cs.wisc.edu/adsl/Software/optfs`. To further aid researchers interested in OptFS, we have also released virtual-machine images with the file system (and modified kernel) installed: users only need a hypervisor such as VirtualBox installed to try out OptFS.