Physical Separation in Modern Storage Systems

By

Lanyue Lu

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: December 1 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
Shan Lu, Associate Professor, Computer Sciences
Michael Swift, Associate Professor, Computer Sciences
Xinyu Zhang, Assistant Professor, ECE

Abstract

Digital data is essential to our daily lives and businesses. The amount of data generated each year grows exponentially. Various storage systems were developed to meet different data management requirements. For decades, researchers and practitioners have been trying to overcome a central challenge in these storage systems: how to reliably and efficiently access data on different storage devices.

In the first part of the dissertation, we respond to this challenge with a comprehensive study of modern Linux file systems to understand practical reliability and performance problems in existing file systems. We analyze eight years of Linux file-system changes across 5079 patches. We focus on file-system bugs by studying their fine-grained patterns, consequences and trends. We find that about 40% of patches are bugs, existing in both new and mature file systems; they do not diminish despite the stability. Most of the bugs lead to system crashes or data corruption. We also exam performance and reliability patches. The performance techniques used were relatively common and widespread. About a quarter of performance patches reduced synchronization overheads. Reliability techniques seemed to be added in a rather ad hoc fashion.

From the file system study, we find that file systems lack enough reliability isolation: a small fault can impact the whole file system. To solve this problem, we propose IceFS, a novel file system that separates physical structures of the file system for better isolation. A new abstraction, the cube, was provided to enable the grouping of files and directories inside a physically isolated container. We show three major benefits of cubes within IceFS: localized reaction to faults, fast recovery, and concurrent file-system updates. We demonstrate that IceFS is able to localize failures that were previously global, and recover quickly using localized online or offline fsck. IceFS can also provide specialized journaling to meet diverse application requirements for performance and consistency. Furthermore, we conduct two cases studies where IceFS is used to host multiple virtual machines and is deployed as the local file system for HDFS data nodes. IceFS achieves fault isolation and fast recovery in both scenarios, proving its usefulness in modern storage environments.

Motivated by physical separation techniques leading to better performance in IceFS, we continue to explore similar techniques in another important type of storage systems: key-value stores. We present WiscKey, a persistent LSM-tree-based key-value store with a novel performance-oriented data layout that separates keys and values to minimize I/O amplification. The data layout and I/O patterns of WiscKey are highly optimized for SSD devices. We solve a number of reliability and performance challenges introduced by the new key-value separation architecture. We propose a parallel range query design to leverage the SSD's internal parallelism for better range query performance on unordered datasets. We also introduce an online and light-weight garbage collector for WiscKey to reclaim the invalid key-value pairs without affecting the foreground workloads much. We demonstrate the advantages of WiscKey with both microbenchmarks and YCSB workloads. Microbenchmark results show that WiscKey is 2.5$\times$ - 111$\times$ faster than LevelDB for loading a database and 1.6$\times$ - 14$\times$ faster for random lookups. WiscKey is faster than both LevelDB and RocksDB in all six YCSB workloads, and follows a trend similar to the microbenchmarks.

To my family

Acknowledgments

Life is short, especially when you decide to pursue a PhD. How to spend time wisely always puzzles me. Looking into the mirror, I am not young anymore, but I feel very lucky for my stubborn decision and a humble dream several years ago. Now, my school life almost ends. An ending is also a new beginning. At this crossroad, let me thank those who have helped me both in work and life.

First, I would like to thank my advisors Remzi and Andrea Arpaci-Dusseau. Before I worked with them, I was a frustrating graduate student. They patiently helped me for many years to achieve my research dream, including how to find a great problem, how to get inspiration, how to think deeply, how to communicate clearly, how to write a good paper, and how to do a great presentation. I remember that they provided detailed suggestions slide by slide in numerous late afternoon group meetings. I remember many inspirational quotes from them: "Double your reading sources", "Such is life", "We should target a best paper". I remember that they met me at Google Madison and Palo Alto downtown to listen to my random ideas at random time. Remzi even taught me how to embed a joke in my conference presentation. Doing research with them becomes easy, fun and meaningful.

I also learned a lot from their personalities. I was seriously sick and rested at home for nearly one semester. I remember that Remzi emailed me every Monday morning to ask what he can help. He offered to talk to my doctor to make sure I was treated well. Remzi's great sense of humor urged me to buy a book about how to have a sense of humor. Andrea always smiles. She constantly encourages me and shares her confidence in my projects, especially during the hard time and the deadline rush. They share their first-class OS book online for free, making me rethink the value of money. Even though I decided to join industry, they helped me greatly by connecting me with their friends in many companies, and taught me how to negotiate offers wisely. What else can you ask for?

I would also like to thank my thesis committee. Shan Lu is also a coauthor of my first paper here. I remember that she sat with me to read many file system bug patches, and taught me different types of concurrency bugs. She is willing to help when I need a recommendation letter at various occasions. It is lucky to have Michael Swift as my prelim and thesis committee. He is very rigorous and his extensive questions, comments and feedback on different parts of my thesis helped make it much better. I also thank Xinyu Zhang for reading my thesis and giving me good suggestions.

I have been fortunate to do excellent internships throughout my PhD. I would like to thank the companies as well as my mentors there: Prasenjit Sarkar, Dinesh Subhraveti, Dean Hildebrand and Renu Tewari at IBM; Herodotos Herodotou, Sriram Rao and Raghu Ramakrishnan at Microsoft; Asit Desai and Sunil Satnur at VMWare. Working with these people have been a great and rewarding experience for me.

I am fortunate to work and discuss with a group of high quality students here: Yupu Zhang, Thanumalayan Pillai, Thanh Do, Jun He, Vijay Chidambaram, Yiying Zhang, Zev Weiss, Tyler Harter, Ao Ma, Asim Kadav, Suli Yang, Chris Dragga, Leo Arulraj, Ramnatthan Alagappan, Sriram Subramanian, and Swami Sundararaman. In particular, I would like to thank my coauthors Yupu, Thanu and Thanh. Yupu is always ready to discuss design and implementation details, help coding and write the paper. Thanu's expertise in crash consistency and hacking skills amazed me. Thanh gave me lots of great advice in research and life. I also want to thank my officemate Vijay, who is always full of motivation and curiosity. Having Thanu and Vijay as officemates definitely helped me a lot to be a better researcher.

I am grateful to have many friends at Madison to make my life more interesting. I thank Ji Liu for being a great roommate when I arrived Madison. I thank the Shell Friday basketball team for numerous fun games and great memories every Friday afternoon. I thank Wenbin Fang and Jun He for inviting me to many delicious dinners at their homes. I thank Yvonne Koh for practicing English speaking with me patiently in her free time. I also would like to thank many other friends at CS: Jia Xu, Yinan Li, Linhai Song, Guoliang Jin, Wenfei Wu, Qian Wan, Yimin Tan, Junming Xu, Yeye He, and Tan Zhang. With you all, the PhD journey is not lonely anymore.

In closing, I turn towards family. I would like to thank my parents for their unconditional love and support. They raised me in an environment with an emphasis on education and good personality. They set good examples for me in both work and life with their hard work and kindness. I have left home for study since I was 14. In these past years, my parents always supported me to pursue my dreams, no matter how far I will go and how long I will take. I remember that my mother cried every time when I left home for high school, college and graduate school. I remember my father woke me up and jogged with me in the hills nearly every morning when I was home. Their love enables me to go forward fearlessly. Finally, I thank my wife Xin for her love since we met. She took care of me in many ways even though she is also busy with her own research and study. I remember she cooked countless delicious meals for me when I work for various deadlines. She always supports me, cheers me up and helps me to be a better person. I dedicate this dissertation to my family: mother, father and Xin.

Contents

List of Figures and Tables

1 Introduction

Digital data is essential to our daily lives and businesses. Data-driven services are universal and crucial nowadays, including communication [68, 113, 150], travel [117, 231], healthcare [66, 101, 168], finance [79, 98], energy [61, 112], and entertainment [16, 154]. Meanwhile, data is generated at an unprecedented rate, growing 40% annually; the amount of total data is about 4.4 Zettabytes in 2013, and it is expected to skyrocket to 44 Zettabytes by 2020 [12].

Given such big data, various storage systems have been developed in last several decades. Typical examples include local and distributed file systems [77, 90, 94, 138, 139, 141, 175, 183, 191, 204], persistent and in-memory key-value stores [14, 21, 28, 48, 64, 71, 115, 152, 184] and numerous databases [9, 67, 86, 148, 155, 196, 202]. Storage platforms also evolve from traditional local storage servers [78, 91, 96, 160, 182, 207, 214, 220] to the cloud [18, 19, 22, 47, 87, 174, 177, 230] and mobile devices [110, 149, 186, 192, 194].

Furthermore, the hardware landscape changes fast too. New storage and computing hardware is at the rise. NAND-flash based solid state disks (SSD) [46, 82] and NVM devices [45, 58, 198] provide microsecond level latency and high internal data parallelism, which can greatly boost application performance. On the other hand, commodity servers contain an increasing number of computing cores. Servers with hundreds of cores are available already [11]. Trends indicate that the number of cores within a single machine will continue to increase in future [35]. The cache and memory capacity also increases with the number of cores for balanced performance; it is not uncommon that a single server machine contains over 100 GB of DRAM for high performance [56, 63].

For decades, researchers and practitioners have been solving two central problems of storage systems: how to *reliably and efficiently* store and retrieve data. As the workloads of the world move to the cloud and mobile devices, as the computing moves to virtual machines and containers, we are facing bigger challenges for high reliability and performance.

Unfortunately, reliability is hard to achieve. Data corruption and system crashes still happen in practice, leading to painful service disruption. Data corruption and system crashes can be caused by various hardware defects [24, 25, 118, 166], power loss [10, 144, 221], and software bugs in operating systems [54, 159, 200, 228], file systems [126, 165, 226, 227] and device drivers [104, 205, 206].

Another big challenge is that existing storage software may fail to fully utilize the (new) hardware, resulting in wasted resources and sub-optimal performance. In the past, a wide variety of techniques were proposed for better performance, such as data locality [44, 141, 176], request scheduling [93, 97, 132, 179], and caching [62, 145, 195, 222]. However, with new hardware (e.g., SSDs) and system abstraction (e.g., virtualization), old optimization techniques may not work well anymore [27, 32, 36, 45, 189, 232].

In this dissertation, our central goal is to *understand and improve* both reliability and performance of modern storage systems. In the first part of this dissertation, we conduct a comprehensive study of popular file systems to understand the practical problems existing in modern file systems for years. In the second part, we build on the insights gained from our study to propose physical separation techniques for both local file systems and persistent key-value stores that can achieve significantly better failure behaviors, recovery speed and I/O performance.

1.1 File System Study

To better understand the real problems in modern storage systems, we begin our research with a practical study of modern Linux file systems. Open-source local file systems, such as Linux Ext4 [139], XFS [204], and Btrfs [138, 175], remain a critical component in the world of modern storage. Many recent distributed file systems, such as Google GFS [77] and Hadoop DFS [191], replicate data objects (and associated metadata) across local file systems. On smart phones, most user data is managed by a local file system, such as Ext4 [110, 192] on Android phones and HFSX [149] on Apple's iOS devices. Finally, many desktop users still do not backup their data regularly [102, 137]; in this case, the local file system clearly plays a critical role as sole manager of user data.

Open-source local file systems remain a moving target. Developed by different teams with different goals, these file systems evolve rapidly to add new features, fix bugs, and improve performance and reliability, as one might expect in the open-source community [171]. Major new file systems are introduced every few years [34, 138, 141, 176, 204]; with recent technology changes (e.g., Flash [33, 82] and NVM [169, 197]), we can expect even more flux in this domain.

However, despite all the activity in local file system development, there is little *quantitative* understanding of their code bases. For example, where does the complexity of such systems lie? What types of bugs are common? Which performance features exist? Which reliability features are utilized? These questions are important for different communities: for developers, so that they can improve current designs and create better systems; for tool builders, so that they can improve their tools to match reality (e.g., by finding the types of bugs that plague existing systems).

In the first part of this dissertation, we garner insight into these questions by studying the artifacts themselves. The fact that every version of Linux is available online, including a detailed set of patches which describe how one version transforms to the next, enables us to carefully analyze how file systems have changed over time. A new type of “systems software archeology” is now possible. We perform the first comprehensive study of the evolution of Linux file systems, focusing on six major and important ones: Ext3 [209], Ext4 [139], XFS [204], Btrfs [138, 175], ReiserFS [38], and JFS [30]. These file systems represent diverse features, designs, implementations and even groups of developers. We examine every file-system patch in the Linux 2.6 series over a period of eight years including 5079 patches. By carefully studying each patch to understand its intention, and then labeling the patch accordingly along numerous important axes, we can gain deep quantitative insight into the file-system development process. We can then answer questions such as “what are most patches for?”, “what types of bugs are common?”, and in general gain a new level of insight into the common approaches and issues that underlie current file-system development and maintenance.

We make the following high-level observations. A large number of patches (nearly 50%) are maintenance patches, reflecting the constant refactoring work needed to keep code simple and maintainable. The remaining dominant category is bugs (just under 40%), showing how much effort is required to slowly inch towards a “correct” implementation. Interestingly, the number of bugs does not die down over time (even for stable file systems), rather ebbing and flowing.

Breaking down the bug category further, we find that semantic bugs are the dominant bug category (over 50% of all bugs). These types of bugs are vexing, as most of them are hard to detect via generic bug detection tools [29, 158]; Concurrency bugs are the next most common (about 20% of bugs), more prevalent than in user-level software [120, 181, 201]. The remaining bugs are split evenly across memory bugs and error handling. We also categorize bugs along other axes to gain further insight. For example, when broken down by consequence, we find that most of the bugs we studied lead to crashes or corruption, and hence are quite serious. When categorized by data structure, we find that B-trees, present in many file systems for scalability, have relatively few bugs per line of code. When classified by whether bugs occur on normal or failure-handling paths, we make the following important discovery: nearly 40% of all bugs occur on failure-handling paths. Future system designs need better tool or language support to make these rarely-executed failure paths correct.

Finally, while bug patches comprise most of our study, performance and reliability patches are also prevalent, accounting for 8% and 7% of patches respectively. The performance techniques used are relatively common and widespread (e.g., removing an unnecessary I/O, or downgrading a write lock to a read lock). About a quarter of performance patches reduce synchronization overheads; thus, while correctness is important, performance likely justifies the use of more complicated and time saving synchronization schemes. In contrast to performance techniques, reliability techniques seem to be added in a rather ad hoc fashion (e.g., most file systems apply sanity checks non-uniformly). Inclusion of a broader set of reliability techniques could harden all file systems.

1.2 Physical Disentanglement in IceFS

From the file system study, we know that there are many bugs in file systems, and most of bugs may lead to serious consequences (e.g., data corruption or system crashes). One natural following question is that is there enough *isolation* within a file system to cope with data corruption, system crashes and performance problems.

Isolation is central to increased reliability and improved performance of modern computer systems. Researchers and practitioners alike have developed a host of techniques to provide isolation in various computer subsystems: isolating performance of CPU, memory, and disk bandwidth in SGI's IRIX operating system [212]; isolating the CPU across different virtual machines [85]; sharing storage cache and I/O bandwidth [214]. Others have designed isolation schemes for device drivers [37, 205, 233], CPU and memory resources [2, 7, 26, 156], and security [81, 99, 105].

Unfortunately, we find that one aspect of current file-system design has remained devoid of isolation: the physical on-disk structures of file systems. This can lead to poor reliability, poor performance, or both. As a simple example, consider a bitmap, used in historical systems such as FFS [141] as well as many modern file systems [44, 139, 209] to track whether inodes or data blocks are in use or free. When blocks from different files are allocated from the same bitmap, aspects of their reliability are now *entangled*, i.e., a failure in that bitmap block can affect otherwise unrelated files. Similar entanglements exist at all levels of current file systems; for example, Linux Ext3 includes all current update activity into a single global transaction [164], leading to painful and well-documented performance problems [4, 5, 8].

Our remedy to this problem is realized in a new file system we call *IceFS*. IceFS provides users with a new basic abstraction in which to co-locate logically similar information; we call these containers *cubes*. IceFS then works to ensure that files and directories within cubes are physically distinct from files and directories in other cubes; thus data and I/O within each cube is *disentangled* from data and I/O outside of it.

To realize disentanglement, IceFS is built upon three core principles. First, there should be no shared physical resources across cubes. Structures used within one cube should be distinct from structures used within another. Second, there should be no access dependencies. IceFS separates key file system data structures to ensure that the data of a cube remains accessible regardless of the status of other cubes; one key to doing so is a novel *directory indirection* technique that ensures cube availability in the file system hierarchy despite loss or corruption of parent directories. Third, there should be no bundled transactions. IceFS includes novel *transaction splitting* machinery to enable concurrent updates to file system state, thus disentangling write traffic in different cubes.

One of the primary benefits of cube disentanglement is *localization*. We demonstrate three key benefits. First, we show how cubes enable localized *micro-failures*; crashes and read-only remounts that normally affect the entire system are now constrained to the faulted cube. Second, we show how cubes permit localized *micro-recovery*; instead of an expensive file-system wide repair, the disentanglement found at the core of cubes enables IceFS to fully (and quickly) repair a subset of the file system (and even do so online). Third, we illustrate how transaction splitting allows the file system to commit transactions from different cubes in parallel, greatly increasing performance (by a factor of 2x–5x) for some workloads.

Interestingly, the localization that is innate to cubes also enables a new benefit: *specialization* [42]. Because cubes are independent, it is natural for the file system to tailor the behavior of each. We realize the benefits of specialization by allowing users to choose different journaling modes per cube; doing so creates a performance/consistency knob that can be set as appropriate for a particular workload, enabling higher performance.

We further show the utility of IceFS in two important modern storage scenarios. In the first, we use IceFS as a host file system in a virtualized VMware [213] environment, and show how it enables fine-grained fault isolation and fast recovery as compared to the state of the art. In the second, we use IceFS beneath HDFS [191], and demonstrate that IceFS provides failure isolation between clients. Overall, these two case studies demonstrate the effectiveness of IceFS as a building block for modern virtualized and distributed storage systems.

1.3 Key-Value Separation in WiscKey

Besides file systems, persistent key-value stores also play a critical role in a variety of modern data-intensive applications. After demonstrating that physical separation of structures in file systems can provide great reliability and performance advantages, we are also curious to explore physical separation techniques in key-value stores.

For write-intensive workloads, key-value stores based on Log Structured Merge Trees (LSM-trees) [157] have become the state of the art. Various distributed and local stores built on LSM-trees are widely deployed in large-scale production environments, such as BigTable [48] and LevelDB [184] at Google, Cassandra [115], HBase [88] and RocksDB [71] at Facebook, PNUTS [59] at Yahoo!, and Riak [14] at Basho.

To deliver high write performance, LSM-trees batch key-value pairs and write them out sequentially. Subsequently, to enable efficient lookups (for both individual keys as well as range queries), LSM-trees continuously read, sort, and write out key-value pairs in the background, thus maintaining keys and values in sorted order. As a result, the same data is read and written multiple times throughout its lifetime; this I/O amplification in typical LSM-trees can reach a factor of 50x or higher.

The success of LSM-based technology is tied closely to its usage upon classic hard-disk drives (HDDs). In HDDs, random I/Os are over $100\times$ slower than sequential ones [157]; thus, performing additional sequential reads and writes to continually sort keys and enable efficient lookups represents an excellent trade-off. However, the storage landscape is quickly changing, and modern solid-state storage devices (SSDs) are supplanting HDDs in many important use cases. As compared to HDDs, SSDs are fundamentally different in their performance and reliability characteristics; when considering key-value storage system design, we believe the following three differences are of paramount importance. First, the difference between random and sequential performance is not nearly as large as with HDDs; thus, an LSM-tree that performs a large number of sequential I/Os to reduce later random I/Os may be wasting bandwidth needlessly. Second, SSDs have a large degree of internal parallelism; an LSM built atop an SSD must be carefully designed to harness said parallelism. Third, SSDs can wear out through repeated writes [116, 147]; the high write amplification in LSM-trees can significantly reduce device lifetime. As we will show in the paper (§4.3), the combination of these factors greatly impacts LSM-tree performance on SSDs, reducing throughput by 90% and increasing write load by a factor over 10. While replacing an HDD with an SSD underneath an LSM-tree does improve performance, with current LSM-tree technology, the SSD's true potential goes largely unrealized.

We present WiscKey, an SSD-conscious persistent key-value store derived from the popular LSM-tree implementation, LevelDB. The central idea behind WiscKey is the separation of keys and values [153]; only keys are kept sorted in the LSM-tree, while values are stored separately in a log. This simple technique can significantly reduce write amplification by avoiding the unnecessary movement of values while sorting. Furthermore, the size of the LSM-tree is also noticeably decreased, leading to better caching and fewer device reads during lookups. WiscKey retains the benefits of LSM-tree technology, including excellent insert and lookup performance, but without excessive I/O amplification.

Separating keys from values introduces a number of challenges and optimization opportunities. First, range query (scan) performance may be affected because values are not stored in sorted order anymore. WiscKey solves this challenge by using the abundant internal parallelism of SSD devices. Second, WiscKey needs garbage collection to reclaim the free space used by invalid values. WiscKey proposes an online and lightweight garbage collector which only involves sequential I/Os and impacts the foreground workload minimally. Third, separating keys and values makes crash consistency challenging; WiscKey leverages an interesting property in modern file systems, that appends never result in garbage data on a crash. WiscKey optimizes performance while providing the same consistency guarantees as found in modern LSM-based systems.

We compare the performance of WiscKey with LevelDB [184] and RocksDB [71], two popular LSM-tree key-value stores. For most workloads, WiscKey performs significantly better. With LevelDB's own microbenchmark, WiscKey is 2.5×–111× faster than LevelDB for loading a database, depending on the size of the key-value pairs; for random lookups, WiscKey is 1.6×–14× faster than LevelDB. WiscKey's performance is not always better than standard LSM-trees; if small values are written out in random order, and a large dataset is range-queried sequentially, WiscKey performs worse than LevelDB. However, this workload does not reflect real-world use cases (which primarily use shorter range queries) and can be improved by log reorganization. Under YCSB macrobenchmarks [60] that reflect real-world use cases, WiscKey is faster than both LevelDB and RocksDB in all six YCSB workloads, and follows a trend similar to the load and random lookup microbenchmarks.

1.4 Summary of Contributions and Overview

Below is a summary of our contributions, which also serves as an overview for the rest of the dissertation.

- **File System Study**. In Chapter 2, we begin presenting our first contribution, which describes a comprehensive study of Linux file systems [126–128]. By analyzing eight years of Linux file-system changes across over 5000 patches, we derive numerous new insights into file systems' reliability and performance problems. Our analysis includes on of the largest studies of bugs to date (nearly 1800 bugs).

- **Physical Disentanglement in IceFS**. In Chapter 3, we presents a solution for problems we found in the file system study. We introduce IceFS [125, 130], a novel file system that separates physical structures of the file system. A new abstraction, the *cube*, is provided to enable the grouping of files and directories inside a physically isolated container. We show three major benefits of cubes within IceFS: localized reaction to faults, fast recovery, and concurrent file-system updates.

- **Key-Value Separation in WiscKey**. We continue to explore physical structure separation technique in key-value stores in Chapter 4. We present WiscKey [129], a persistent LSM-tree based key-value store with key-value separation and SSD-conscious optimization. WiscKey can achieve significantly better performance with minimal I/O amplification.

- **Related Work**. In Chapter 5, we discuss various research efforts and systems that are related to this dissertation. First, we describe previous system studies over the years. Then, we describe reliability isolation and performance improvement techniques in file systems and key-value stores.

- **Future Work, Lessons Learned, and Conclusions**. In Chapter 6, we first summarize our work and highlight the lessons learned from three projects in this dissertation. Then, we discuss various directions for future work motivated by our research. We believe our contributions will be valuable to a broad range of storage systems running on new hardware in the future.

2 Study of Linux File Systems

Open-source local file systems are a critical component in the world of modern storage. Developed by different teams with different goals, local file systems evolve rapidly to add new features, fix bugs, and improve performance and reliability. Major new file systems are introduced frequently [34, 138, 141, 176, 204]; with recent new storage hardware (Flash [33, 82] and NVM [169, 197]), we can expect even more flux in this domain.

However, despite all the activity in local file system development, there is little *quantitative* understanding of their code bases. For example, where does the complexity of such systems lie? What types of bugs are common? Which performance features exist? Which reliability features are utilized? These questions are important for different communities: for developers, so that they can improve current designs and create better systems; for tool builders, so that they can improve their tools to match reality (e.g., by finding the types of bugs that plague existing systems).

In this chapter, we garner insight into these questions by studying the artifacts themselves. The fact that every version of Linux is available online, including a detailed set of patches which describe how one version transforms to the next, enables us to carefully analyze how file systems have changed over time. A new type of "systems software archeology" is now possible. We perform the first comprehensive study of the evolution of Linux file systems, focusing on six major and important ones: Ext3, Ext4, XFS, Btrfs, ReiserFS, and JFS. These file systems represent diverse features, designs, implementations and even groups of developers. We examine every file-system patch in the Linux 2.6 series over a period of eight years including 5079 patches. By carefully studying each patch to understand its intention, and then labeling the patch accordingly along numerous important axes, we can gain deep quantitative insight into the file-system development process. We can then answer questions such as “what are most patches for?”, “what types of bugs are common?”, and in general gain a new level of insight into the common approaches and issues that underlie current file-system development and maintenance.

The rest of this chapter is organized as follows. We discuss our methodology (§2.1) first, followed by an overall discussion of patches (§2.2). We then present a detailed analysis of bugs (§2.3) and performance and reliability patches (§2.4.1). Finally, we conclude with a case study (§2.5) and conclusions (§2.6).

2.1 Methodology

In this section, we first give a brief description of our target file systems. Then, we illustrate how we analyze patches with a detailed example. Finally, we discuss the limitations of our methodology.

2.1.1 Target File Systems

Our goal in selecting a collection of disk-based file systems is to choose the most popular and important ones. The selected file systems should include diverse reliability features (e.g., physical journaling, logical journaling, checksumming, copy-on-write), data structures (e.g., hash tables, indirect blocks, extent maps, trees), performance optimizations (e.g., asynchronous thread pools, scalable algorithms, caching, block allocation for SSD devices), advanced features (e.g., pre-allocation, snapshot, resize, volumes), and even a range of maturity (e.g., stable, under development). For these reasons, we selected six file systems and their related modules: Ext3 with JBD [209], Ext4 with JBD2 [139], XFS [204], Btrfs [138, 175], ReiserFS [38], and JFS [30].

Ext3, JFS, ReiserFS and XFS were all introduced in Linux 2.4, and were considered as stable file systems in Linux 2.6. Ext4, as a direct successor of Ext3, was introduced in Linux 2.6.19 and was marked as stable in Linux 2.6.28. In contrast, Btrfs was born in Linux 2.6.29, which is still under active development. Ext3 was a popular default file system for many Linux distributions in the last decade; recently, Ext4 has became the default file systems in most modern Linux distributions, such as Red Hat, SUSE and Ubuntu. JFS (by IBM), ReiserFS (by Namesys), and XFS (by Silicon Graphics) serve as good candidates of enterprise file systems.

Btrfs is a copy-on-write (COW) file system with modern features comparable to ZFS [34], while the remaining file systems are all journaling file systems. JFS and XFS only provide metadata journaling, while Ext3, Ext4 and ReiserFS provide both metadata and file data journaling. B-tree is widely used in JFS, ReiserFS, Btrfs and XFS to scale their metadata and file data. Advanced features are also common in several file systems. For example, Ext4 implements delayed allocation, pre-allocation, flexible block group, online defragmentation and journal checksumming [165]. Btrfs supports snapshot, volume management, data checksumming, compression, and even optimized block allocation for SSD devices. XFS is great at scalability, and it provides guaranteed-rate I/O for realtime applications, hierarchical storage management, and custom buffer cache.

2.1.2 Classification of File-System Patches

For each file system, we conduct a comprehensive study of its evolution by examining all patches from Linux 2.6.0 (Dec '03) to 2.6.39 (May '11). These are Linux mainline versions, which are released every three months with aggregate changes included in change logs. Patches consist of all formal modifications in each new kernel version, including new features, code maintenance, and bug fixes, and usually contain clear descriptions of their purpose and rich diagnostic information. On the other hand, Linux Bugzilla [41] and mailing lists [76, 124] are not as well organized as final patches, and may only contain a subset or superset of final changes merged in kernel.

To better understand the evolution of different file systems, we conduct a broad study to answer three categories of fundamental questions:

- *Overview*: What are the common types of patches in file systems and how do patches change as file systems evolve? Do patches of different types have different sizes?
- *Bugs*: What types of bugs appear in file systems? Do some components of file systems contain more bugs than others? What types of consequences do different bugs have?
- *Performance and Reliability*: What techniques are used by file systems to improve performance? What common reliability enhancements are proposed in file systems?

To answer these questions, we manually analyzed each patch to understand its purpose and functionality, examining 5079 patches from the selected Linux 2.6 file systems. Each patch contains a patch header, a description body, and source-code changes. The patch header is a high-level summary of the functionality of the patch (e.g., fixing a bug). The body contains more detail, such as steps to reproduce the bug, system configuration information, proposed solutions, and so forth. Given these details and our knowledge of file systems, we categorize each patch along a number of different axes, as described later.

Figure 2.1 shows a real Ext3 patch. We can infer from the header that this patch fixes a null-pointer dereference bug. The body explains the

```
[PATCH] fix possible NULL pointer in fs/ext3/super.c.

In fs/ext3/super.c::ext3_get_journal() at line 1675
'journal' can be NULL, but it is not handled right
(detect by Coverity's checker).

---   /fs/ext3/super.c
+++   /fs/ext3/super.c
@@  -1675,6 +1675,7 @@ journal_t *ext3_get_journal()

    if (!journal){
       printk(KERN_ERR "EXT3: Could not load ... ");
       iput(journal_inode);
  +    return NULL;
    }
    journal->j_private = sb;
```

Figure 2.1: **An Example Patch.** *This figure shows an Ext3 patch in Linux 2.6.7.*

cause of the null-pointer dereference and the location within the code. The patch also indicates that the bug was detected with Coverity [29].

This patch is classified as a bug (type=bug). The size is 1 (size=1) as one line of code is added. From the related source file (super.c), we infer the bug belongs to Ext3's superblock management (data-structure=super). A null-pointer access is a memory bug (pattern=memory,nullptr) and can lead to a crash (consequence=crash).

However, some patches have less information, making our analysis harder. In these cases, we sought out other sources of information, including design documents, forum and mailing-list discussions, and source-code analysis. Most patches are analyzed with high confidence given all the available information and our domain knowledge. Examples are shown throughout to give more insight as to how the classification is performed.

Limitations: Our study is limited by the file systems we chose, which may not reflect the characteristics of other file systems, such as other non-Linux file systems and flash-device file systems. We only examined kernel patches included in Linux 2.6 mainline versions, thus omitting patches for Ext3, JFS, ReiserFS, and XFS from Linux 2.4. As for bug representativeness, we only studied the bugs reported and fixed in patches, which is a biased subset; there may be (many) other bugs not yet reported. A similar study may be needed for user-space utilities, such as mkfs and fsck [143].

2.2 Patch Overview

File systems evolve through patches. A large number of patches are discussed and submitted to mailing lists, bug-report websites, and forums. Some are used to implement new features, while others fix existing bugs. In this section, we investigate three general questions regarding file-system patches. First, what are file-system patch types? Second, how do patches change over time? Lastly, what is the distribution of patch sizes?

2.2.1 Patch Type

We classify patches into five categories (Table 2.2): bug fixes (*bug*), performance improvements (*performance*), reliability enhancements (*reliability*), new features (*feature*), and maintenance and refactoring (*maintenance*). Each patch usually belongs to a single category.

Figure 2.3(a) shows the number and relative percentages of patch types for each file system. Note that even though file systems exhibit significantly different levels of patch activity (shown by the total number of patches), the percentage breakdowns of patch types are relatively similar.

Maintenance patches are the largest group across all file systems (except Btrfs, a recent and not-yet-stable file system). These patches include changes to improve readability, simplify structure, and utilize cleaner abstractions; in general, these patches represent the necessary costs of

Type	Description
Bug	Fix existing bugs
Performance	Propose more efficient designs or implementations to improve performance (e.g., reducing synchronization overhead or use tree structures)
Reliability	Improve file-system robustness (e.g., data integrity verification, user/kernel pointer annotations, access-permission checking)
Feature	Implement new features
Maintenance	Maintain the code and documentation (e.g., adding documentation, fix compiling error, changing APIs)

Table 2.2: **Patch Type.** *This table describes the classification and definition of file-system patches. There are five categories: bug fixes (bug), performance improvements (performance), reliability enhancements (reliability), new features (feature), and maintenance and refactoring (maintenance).*

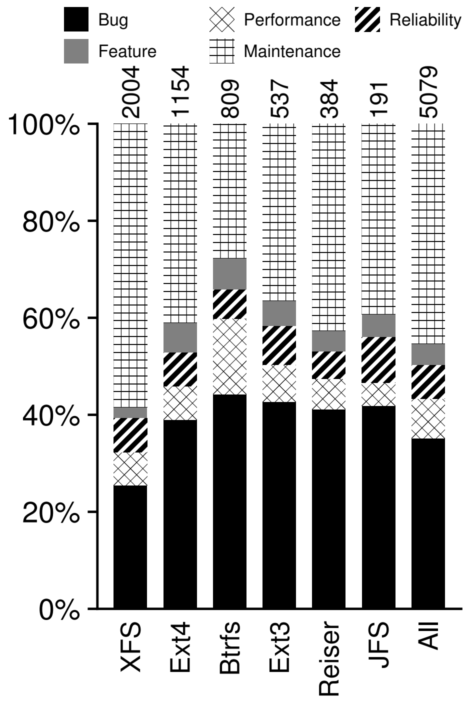
keeping a complex open-source system well-maintained. Because maintenance patches are relatively uninteresting, we do not examine them further in this chapter.

Bug patches have a significant presence, comprising nearly 40% of patches in total. Not surprisingly, the Btrfs has a larger percentage of bug patches than others; however, stable and mature file systems (such as Ext3) also have a sizable percentage of bug patches, indicating that bug fixing is a constant in a file system's lifetime (Figure 2.10). Because this class of patch is critical for developers and tool builders, we characterize them in detail later (§2.3).

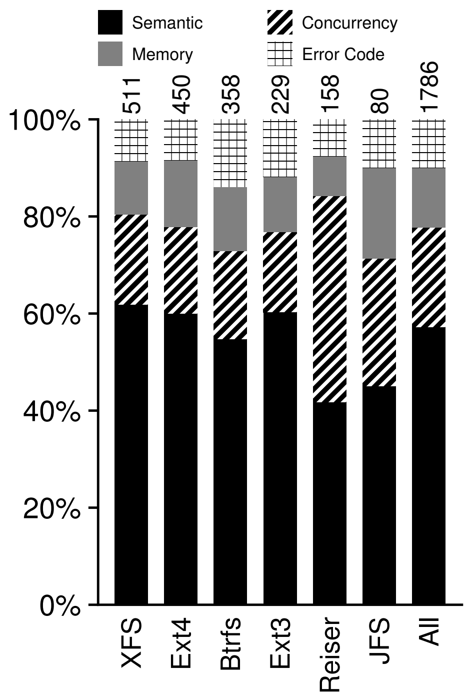
Both *performance* and *reliability* patches occur as well, although with less frequency than maintenance and bug patches. They reveal a variety of techniques used by different file systems, motivating further study (§2.4.1).

Finally, *feature* patches account for a small percentage; as we will see, most of *feature* patches contain more lines of code than other patches.

Summary: Nearly half of total patches are for code maintenance and



(a) Patch Type



(b) Bug Pattern

Figure 2.3: **Patch Type and Bug Pattern.** *This figure shows the distribution of patch types and bug patterns. The total number of patches is on top of each bar.*

documentation; a significant number of bugs exist in not only new file systems, but also stable file systems; all file systems make special efforts to improve their performance and reliability; feature patches account for a relatively small percentage of total patches.

2.2.2 Patch Trend

File systems change over time, integrating new features, fixing bugs, and enhancing reliability and performance. Does the percentage of different patch types increase or decrease with time?

We studied the changes in patches over time and found few interesting changes. While the number of patches per version increased in general,

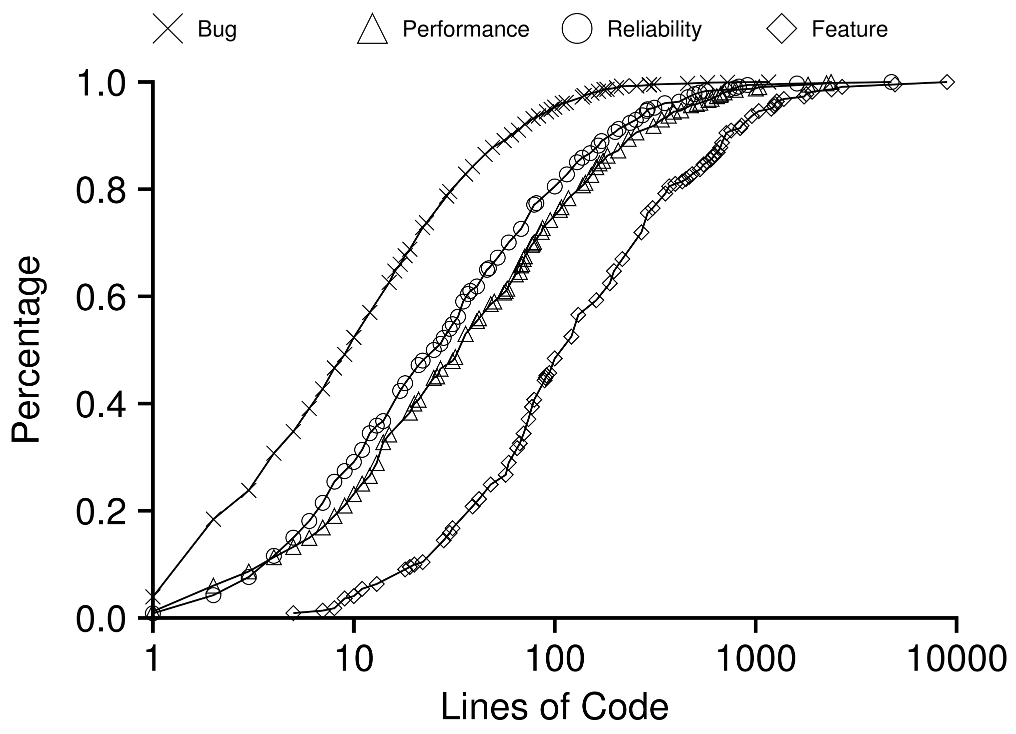


Figure 2.4: **Patch Size.** *This figure shows the size distribution for different patch types, in terms of lines of modifications. The x-axis shows the lines of code in log scale; the y-axis shows the percentage of patches.*

the percentage of maintenance, bug, reliability, performance, and feature patches remained relatively stable. Although there were a few notable exceptions (e.g., Btrfs had a time where a large number of performance patches were added), the statistics shown in the previous section are relatively good summaries of the behavior at any given time. Perhaps most interestingly, bug patches do not decrease over time; living code bases constantly incorporate bug fixes (see §2.3).

Summary: The patch percentages are relatively stable over time; newer file systems (e.g., Btrfs) deviate occasionally; bug patches do not diminish despite stability.

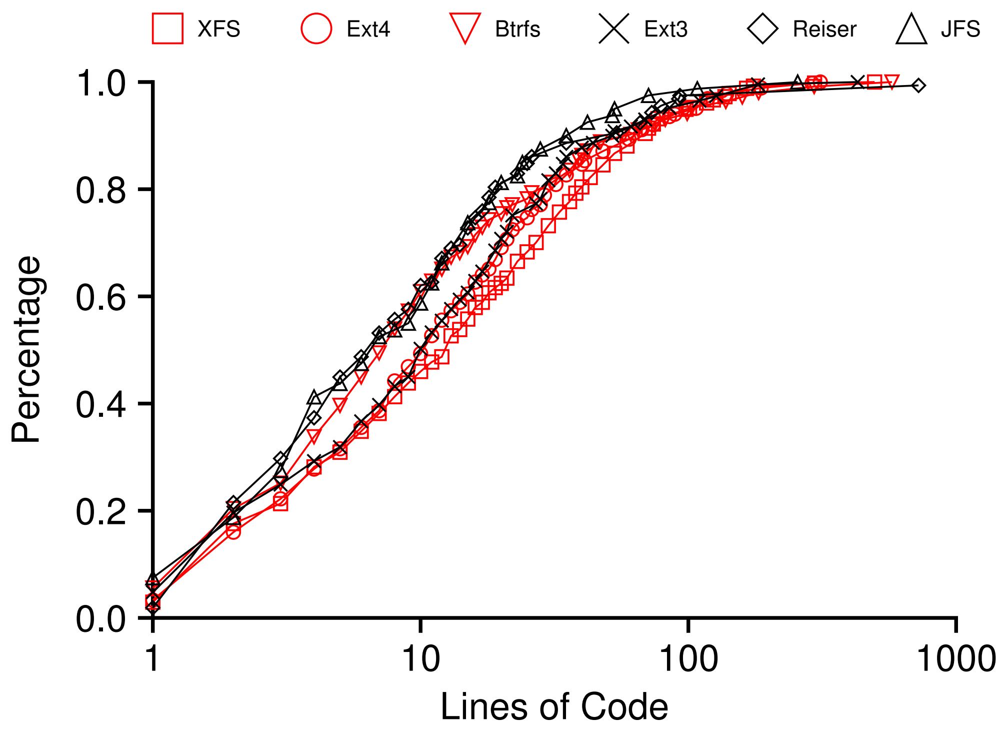


Figure 2.5: **Bug Patch Size.** *This figure shows the size distribution of bug patches for different file systems, in terms of lines of modifications. The x-axis shows the lines of code in log scale; the y-axis shows the percentage of patches.*

2.2.3 Patch Size

Patch size is one approximate way to quantify the complexity of a patch, and is defined here as the sum of lines of added and deleted by a patch. Figure 2.4 displays the size distribution of bug, performance, reliability, and feature patches. Most *bug* patches are small; 50% are less than 10 lines of code. However, more complex file systems tend to have larger bug patches due to their internal complexity and large code bases. Figure 2.5 shows that XFS, Ext4 and Btrfs have larger bug patches than ReiserFS and JFS. Interestingly, feature patches are significantly larger than other patch types. Over 50% of these patches have more than 100 lines of code; 5% have over 1000 lines of code.

Summary: Bug patches are generally small; complicated file systems have larger bug patches; reliability and performance patches are medium-sized; feature patches are significantly larger than other patch types.

2.3 File-System Bugs

In this section, we study file-system bugs in detail to understand their patterns and consequences comprehensively. First, we examine the code evolution for each file system over 40 versions. Second, we show the distribution of bugs in file-system logical components. Third, we describe our bug pattern classification, bug trends, and bug consequences. Finally, we analyze each type of bug with a more detailed classification and a number of real examples.

2.3.1 Code Evolution

FFS had only 1200 lines of code [141]; modern systems are notably larger, including Ext4, Btrfs, and XFS. How does the code of these major file systems change across time? Do all file systems increase their code bases over time?

Figure 2.6 displays lines of code (LOC) for six file systems. First, XFS has the largest code base for all Linux 2.6 versions; its code size is significantly larger than Ext4, Ext3, ReiserFS and JFS. Interestingly, the XFS code base has been significantly reduced over time, from nearly 80K LOC to 64K LOC. As Figure 2.3(a) in §2.2.1 shows, about 60% of XFS's patches are maintenance patches, which are mainly used to simplify its structures, refactor redundant code, and utilize more generic functions.

Second, new file systems Ext4 and Btrfs continuously increase their code sizes by adding new features, improving performance and reliability.

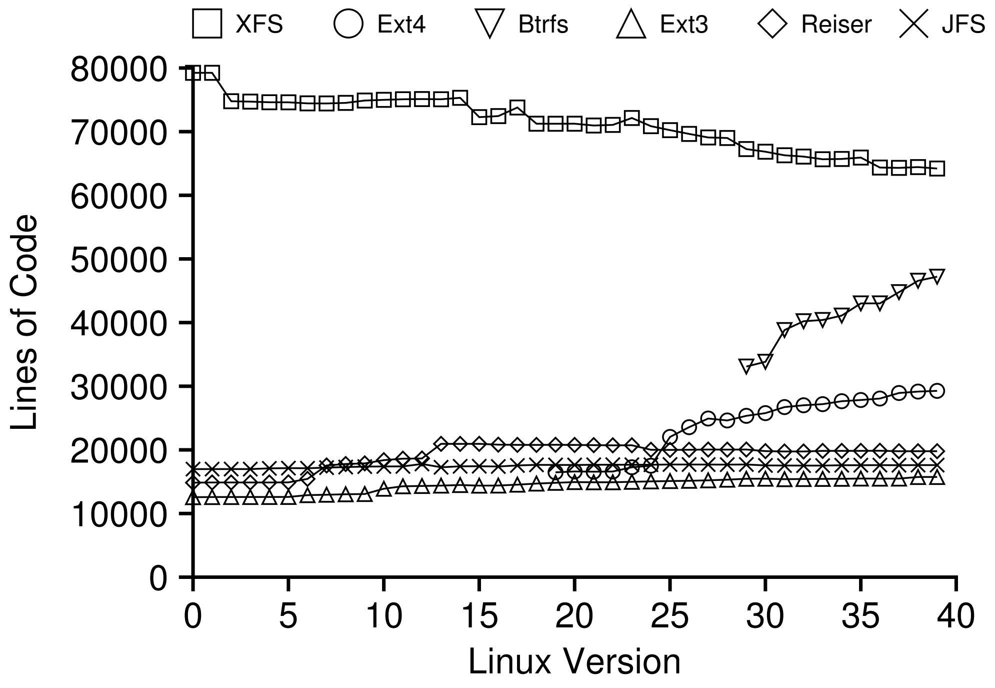


Figure 2.6: **File-System Code Evolution.** *This figure shows the lines of code for each file system in Linux 2.6 series. The x-axis shows 40 versions of Linux 2.6; the y-axis shows lines of code (LOC). Note that LOC of Ext3 includes the code of JBD and LOC of Ext4 includes the code of JBD2.*

Ext4 nearly doubles its size compared with the initial copy of Ext3. Btrfs grows aggressively by implementing many modern advanced features. Ext4 and Btrfs seem to continue to grow linearly.

Third, the remaining mature file systems (Ext3, ReiserFS and JFS) keep relatively stable code size across versions. Changes happen occasionally when new features are added or major structures are modified. For example, Ext3's code size slightly increased when the block reservation algorithm was added at 2.6.10. On the other side, ReiserFS tried to simplify its structure by removing its own custom file read/write functions at 2.6.24, which reduced its code size accordingly. JFS has fewest changes

Name	Description
balloc	Data block allocation and deallocation
dir	Directory management
extent	Contiguous physical blocks mapping
file	File read and write operations
inode	Inode-related metadata management
trans	Journaling or other transactional support
super	Superblock-related metadata management
tree	Generic tree structure procedures
other	Other supporting components (e.g., xattr, ioctl, resize)

Table 2.8: **Logical Components.** *This table shows the classification and definition of file-system logical components. There are nine components for all file systems.*

due to its smaller developer and user communities.

Summary: XFS has the most lines of code, but it has reduced its size and complexity over time; new file systems (EXT4 and Btrfs) keep growing by adding new features; mature file systems are relatively stable while small changes arise due to major features or structure changes.

2.3.2 Correlation Between Code and Bugs

The code complexity of file systems is changing over time as discussed in the previous section. Several fundamental questions are germane: How is the code distributed among different logical components? Where are the bugs? Does each logical component have an equal degree of complexity?

File systems generally have similar logical components, such as inodes, superblocks, and journals. To enable fair comparison, we partition each file system into nine logical components (Table 2.8).

Figure 2.7 shows the percentage of bugs versus the percentage of code for each of the logical components across all file systems and versions. Within a plot, if a point is above the $y = x$ line, it means that a logical component (e.g., inodes) has more than its expected share of bugs, hinting

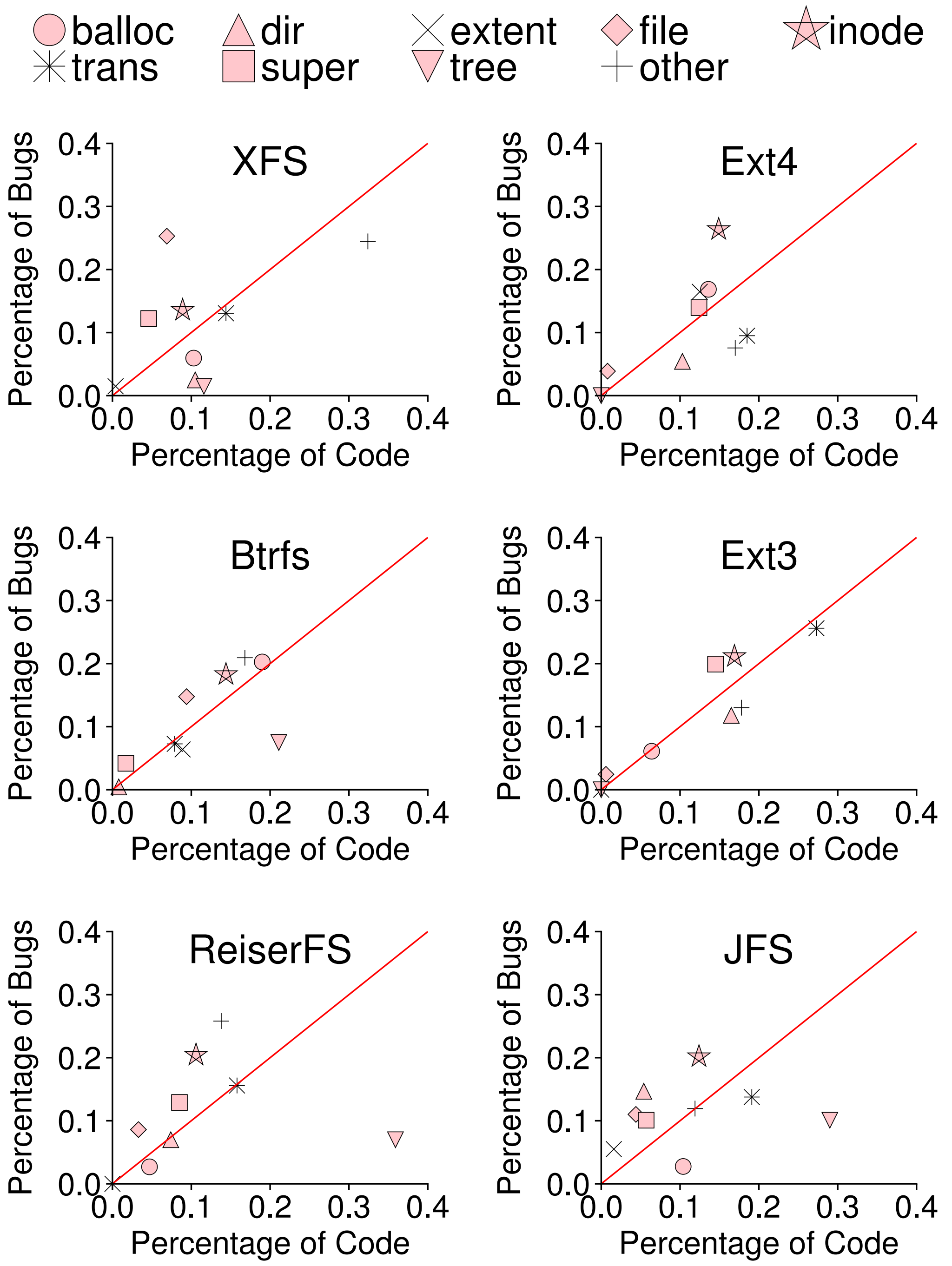


Figure 2.7: **File-System Code and Bug Correlation.** *This figure shows the correlation between code and bugs. The x-axis shows the average percent of code of each component (over all versions); the y-axis shows the percent of bugs of each component (over all versions).*

at its complexity; a point below said line indicates a component (e.g., a tree) with relatively few bugs per line of code, thus hinting at its relative ease of implementation.

We make the following observations. First, for all file systems, the *file*, *inode*, and *super* components have a high bug density. The file component is high in bug density either due to bugs on the fsync path (Ext3) or custom file I/O routines added for higher performance (XFS, Ext4, ReiserFS, JFS), particularly so for XFS, which has a custom buffer cache and I/O manager for scalability [204]. The inode and superblock are core metadata structures with rich and important information for files and file systems, which are widely accessed and updated; thus, it is perhaps unsurprising that a large number of bugs arise therein (e.g., forgetting to update a time field in an inode, or not properly using a superblock configuration flag).

Second, transactional code represents a substantial percentage of each code base (as shown by the relatively high x-axis values) and, for most file systems, has a proportional amount of bugs. This relationship holds for Ext3 as well, even though Ext3 uses a separate journaling module (JBD); Ext4 (with JBD2) has a slightly lower percentage of bugs because it was built upon a more stable JBD from Linux 2.6.19. In summary, transactions continue to be a double-edged sword in file systems: while transactions improve data consistency in the presence of crashes, they often add many bugs due to their large code bases.

Third, the percentage of bugs in *tree* components of XFS, Btrfs, ReiserFS, and JFS is surprisingly small compared to code size. One reason may be the care taken to implement such trees (e.g., the tree code is the only portion of ReiserFS filled with assertions). File systems should be encouraged to use appropriate data structures, even if they are complex, because they do not induce an inordinate amount of bugs.

Although bug patches also relate to feature patches, it is difficult to correlate them precisely. Code changes partly or totally overlap each other

Type	Sub-Type	Description
Semantic	State	Incorrectly update or check file-system state
	Logic	Wrong algorithm/assumption/implementation
	Config	Missed configuration
	I/O Timing	Wrong I/O requests order
	Generic	Generic semantic bugs: wrong type, typo
Concurrency	Atomicity	The atomic property for accesses is violated
	Order	The order of multiple accesses is violated
	Deadlock	Deadlock due to wrong locking order
	Miss unlock	Miss a paired unlock
	Double unlock	Unlock twice
	Wrong lock	Use the wrong lock
Memory	Resource leak	Fail to release memory resource
	Null pointer	Dereference null pointer
	Dangling Pt	Dereference freed memory
	Uninit read	Read uninitialized variables
	Double free	Free memory pointer twice
	Buf overflow	Overrun a buffer boundary
Error Code	Miss Error	Error code is not returned or checked
	Wrong Error	Return or check wrong error code

Table 2.9: **Bug Pattern Classification.** *This table shows the classification and definition of file-system bugs. There are four big categories based on root causes and further sub-categories for detailed analysis.*

over time. A bug patch may involve code changes of both old code and recent feature patches.

Summary: The file, inode, and superblock components contain a disproportionally large number of bugs; transactional code is large and has a proportionate number of bugs; tree structures are not particularly error-prone, and should be used when needed without much worry.

2.3.3 Bug Patterns

To build a more reliable file system, it is important to understand the type of bugs that are most prevalent and the typical patterns across file systems. Since different types of bugs require different approaches to detect and fix, these fine-grained bug patterns provide useful information to developers and tool builders alike.

We partition file-system bugs into four categories based on their root causes as shown in Table 2.9. The four major categories are *semantic* [120, 199], *concurrency* [74, 131], *memory* [54, 120, 199], and *error code* bugs [84, 178].

Figure 2.3(b) (page 19) shows the total number and percentage of each type of bug across file systems. There are about 1800 total bugs, providing a great opportunity to explore bug patterns at scale. Semantic bugs dominate other types (except for ReiserFS). Most semantic bugs require file-system domain knowledge to understand, detect, and fix; generic bug-finding tools (e.g., Coverity [29]) may have a hard time finding these bugs. Concurrency bugs account for about 20% on average across file systems (except for ReiserFS), providing a stark contrast to user-level software where fewer than 3% of bugs are concurrency-related [120, 181, 201]. ReiserFS stands out along these measures because of its transition, in Linux 2.6.33, away from the Big Kernel Lock (BKL), which introduced a large number of concurrency bugs. There are also a fair number of memory-related bugs in all file systems; their percentages are lower than that reported in user-level software [120, 201]. Many research and commercial tools have been developed to detect memory bugs [29, 158], and some are used to detect file-system bugs. Error code bugs account for only 10% of total bugs.

Summary: Beyond maintenance, bug fixes are the most common patch type; over half of file-system bugs are semantic bugs, likely requiring domain knowledge to find and fix; file systems have a higher percentage of concurrency bugs compared with user-level software; memory and error code bugs arise but in smaller percentages.

2.3.4 Bug Trends

File systems mature from the initial development stage to the stable stage over time, by applying bug-fixing, performance and reliability patches. Various bug detection and testing tools are also proposed to improve file-system stability. A natural question arises: do file-system bug patterns change over time, and in what way?

Our results (Figure 2.10) show that within bugs, the relative percentage of semantic, concurrency, memory, and error code bugs varies over time, but does not converge; a great example is XFS, which under constant development goes through various cycles of higher and lower numbers of bugs. Interesting exceptions occasionally arise (e.g., the BKL removal from ReiserFS led to a large increase in concurrency bugs in 2.6.33). JFS does experience a decline in bug patches, perhaps due to its decreasing usage and development [219]. JFS and ReiserFS both have relatively small developer and user bases compared to the more active file systems XFS, Ext4 and Btrfs.

Summary: Bug patterns do not change significantly over time, increasing and decreasing cyclically; large deviations of bug patterns arise due to major structural changes.

2.3.5 Bug Consequences

As shown in Figure 2.3(b) (on page 19), there are a significant number of bugs in file systems. But how serious are these file-system bugs? We now categorize each bug by impact; such *bug consequences* include severe ones (data corruption, system crashes, unexpected errors, deadlocks, system

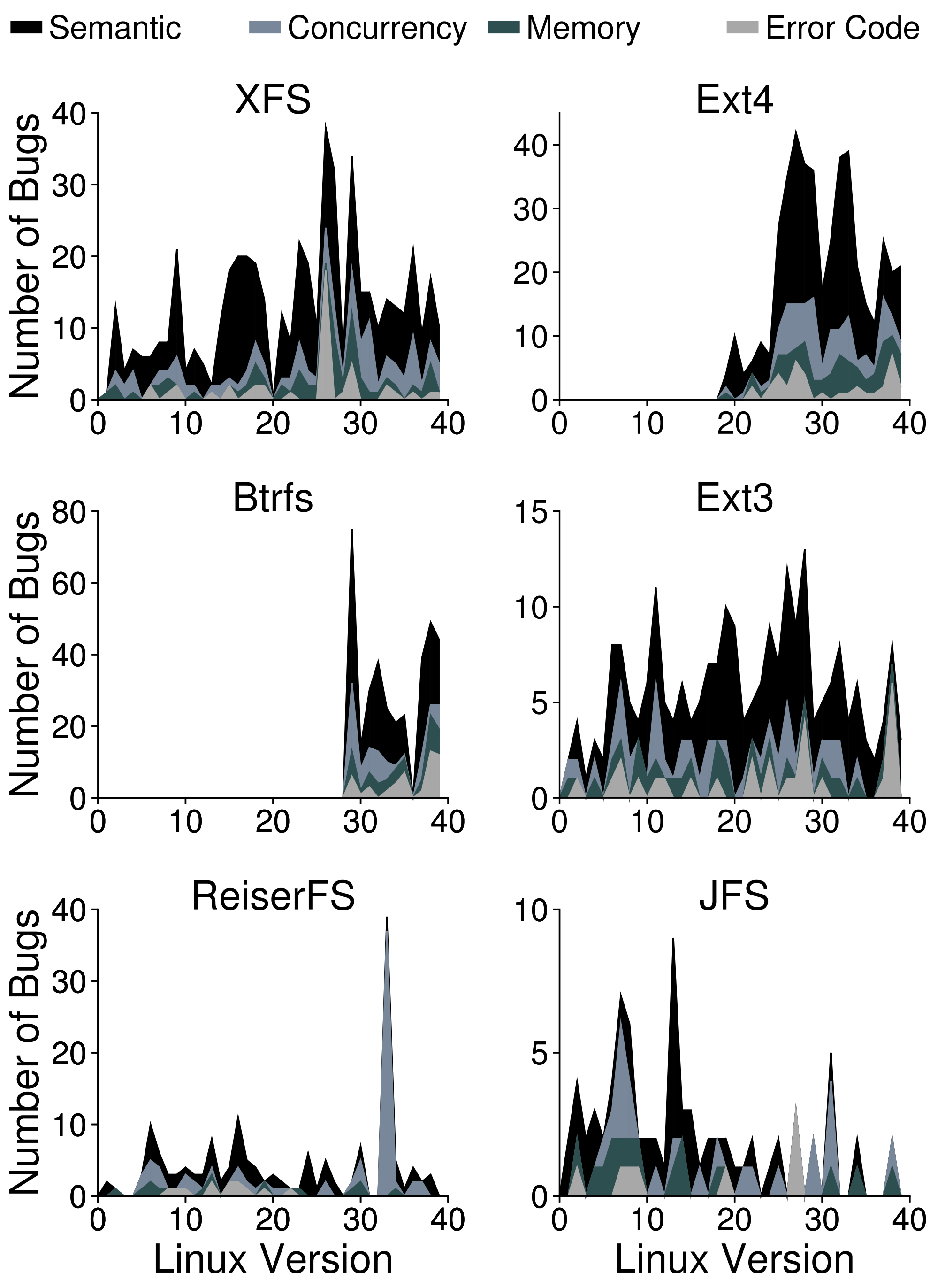


Figure 2.10: **Bug Pattern Evolution.** *This figure shows the bug pattern evolution for each file system over all versions.*

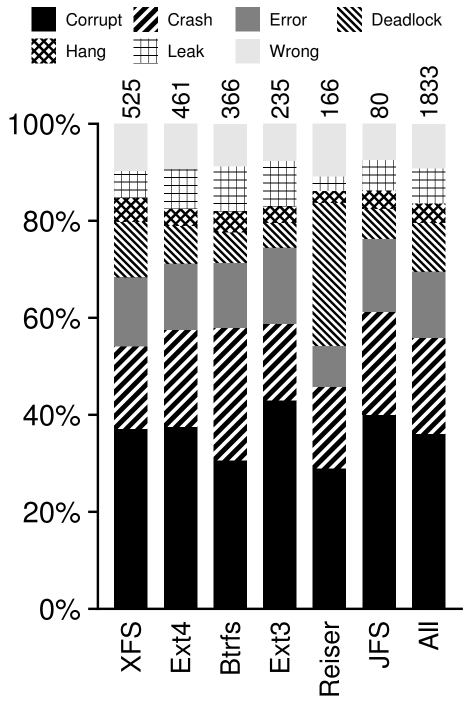
Type	Description
Corruption	On-disk or in-memory data structures are corrupted (e.g., file data or metadata corruption, wrong statistics)
Crash	File system becomes unusable (e.g., dereference null pointer, assertion failures, panics)
Error	Operation failure or unexpected error code returned (e.g., failed write operation due to ENOSPC error)
Deadlock	Wait for resources in circular chain
Hang	File system makes no progress (e.g., infinite loop, live lock)
Leak	System resources are not freed after usage (e.g., forget to free allocated file-system objects)
Wrong	Diverts from expectation, excluding the above ones (e.g., undefined behavior, security vulnerability)

Table 2.11: **Bug Consequence Classification.** *This table shows the definitions of various bug consequences. There are seven categories based on impact: data corruption, system crashes, unexpected errors, deadlocks, hangs, resource leaks, and wrong behaviors.*

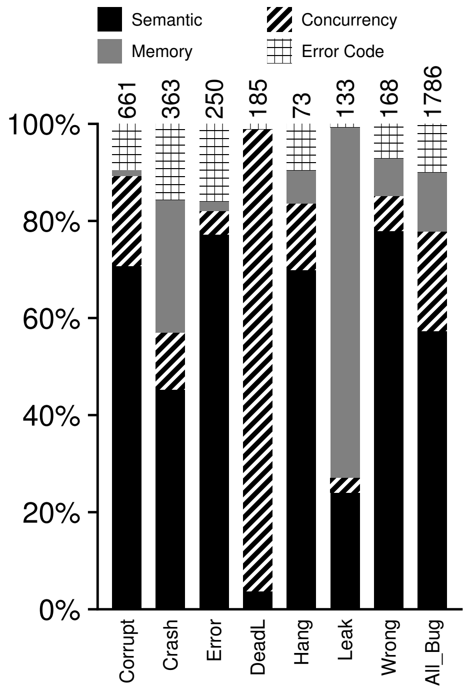
hangs and resource leaks), and other wrong behaviors. Table 2.11 provides more detail on these categories.

Figure 2.12(a) shows the per-system breakdowns. Data corruption is the most predominant consequence (40%), even for well-tested and mature file systems. Crashes account for the second largest percentage (20%); most crashes are caused by explicit calls to `BUG()` or `Assert()` as well as null-pointer dereferences. If the patch mentions that the crash also causes corruption, then we classify this bug with multiple consequences. Unexpected errors and deadlocks occur quite frequently (just under 10% each on average), whereas other bug consequences arise less often. For example, exhibiting the wrong behavior without more serious consequences accounts for only 5-10% of consequences in file systems, whereas it is dominant in user applications [120].

Given that file-system bugs are serious bugs, we were curious: do certain bug types (e.g., semantic, concurrency, memory, or error code)



(a) By File Systems



(b) By Bug Patterns

Figure 2.12: **Bug Consequences.** *This figure displays the breakdown of bug consequences for file systems and bug patterns. The total number of consequences is shown on top of each bar. A single bug may cause multiple consequences; thus, the number of consequences instances is slightly higher than that of bugs in Figure 2.3(b).*

exhibit different levels of severity? Figure 2.12(b) shows the relationship between consequences and bug patterns. Semantic bugs lead to a large percentage of corruptions, crashes, errors, hangs, and wrong behaviors. Concurrency bugs are responsible for nearly all deadlocks (almost by definition) and a fair percentage of corruptions and hangs. Memory bugs lead to many memory leaks (as expected) and a fair amount of crashes. Finally, error code bugs lead to a relatively small percentage of corruptions, crashes, and (unsurprisingly) errors.

Summary: File-system bugs cause severe consequences; corruptions and crashes are most common; wrong behavior is uncommon; semantic bugs can lead to significant amounts of corruptions, crashes, errors, and hangs; all bug types have severe consequences.

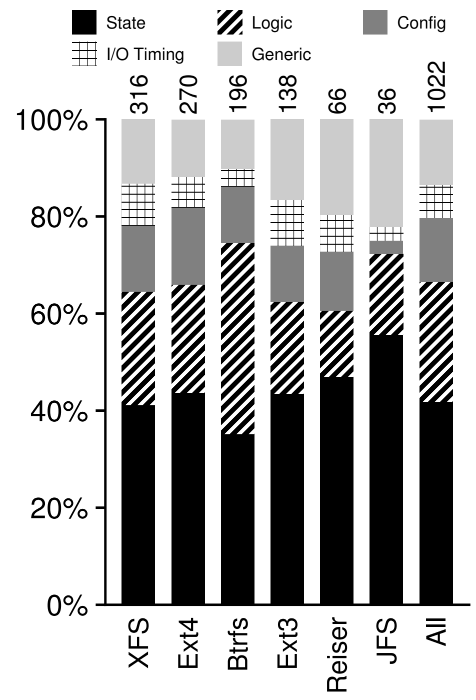
2.3.6 Bug Pattern Examples and Analysis

To gain further insight into the different classes of bugs, we now describe each class in more detail. We present examples of each and further break down each major class (e.g., memory bugs) into smaller sub-classes (e.g., leaks, null-pointer dereferences, dangling pointers, uninitialized reads, double frees, and buffer overflows).

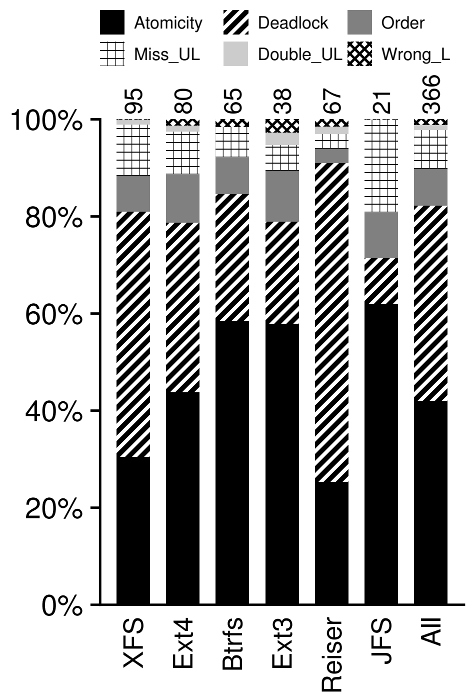
Semantic Bugs

Semantic bugs are dominant in file systems, as shown in Figure 2.3(b). Understanding these bugs often requires file-system domain knowledge. Semantic bugs usually are difficult to categorize in an informative and general way. We are the first to identify several common types of file-system specific semantic bugs based on extensive analysis and careful generalization of many semantic bugs across file systems. These common types and typical patterns provide useful guidelines for analysis and detection of file-system semantic bugs. We partition the semantic bugs into five categories as described in Table 2.9, including *state*, *logic*, *config*, *I/O timing* and *generic*. Figure 2.13(a) shows the percentage breakdown and total number of semantic bugs; each is explained in detail below.

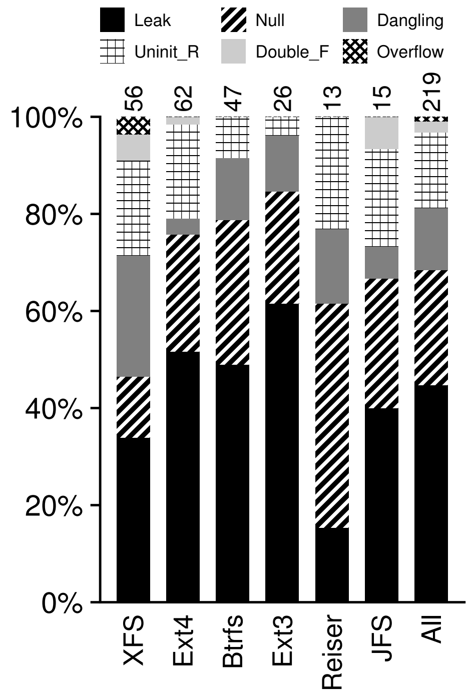
File systems maintain a large amount of in-memory and on-disk state. Generally, operations transform the file system from one consistent state to another; a mistaken state update may lead to serious consequences. As shown in Figure 2.13(a), these *state* bugs contribute to roughly 40% of semantic bugs. An example of a *state* bug is shown in S1 of Table 2.14, which misses an inode-field update. Specifically, the buggy version of



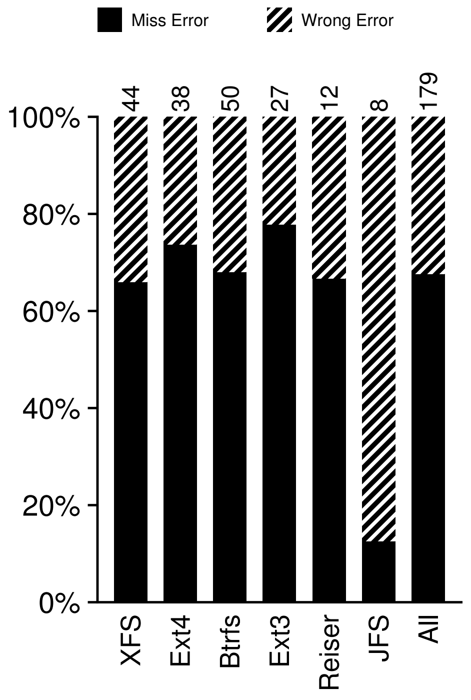
(a) Semantic Bugs



(b) Concurrency Bug



(c) Memory Bugs



(d) Error Code Bugs

Figure 2.13: **Detailed Bug Patterns.** *The detailed classification for each bug pattern; total number of bugs is shown on top of each bar.*

`ext3_rename()` does not update the `mtime` and `ctime` of the directory into which the file is moved, leaving metadata in an incorrect state.

There are also numerous *logic* bugs, which arise via the use of wrong algorithms, bad assumptions, and incorrect implementations. An example of a wrong algorithm is shown in S2 of Table 2.14: `find_group_other()` tries to find a block group for inode allocation, but does not check all candidate groups; the result is a possible `ENOSPC` error even when the file system has free inodes.

File-system behavior is also affected by various configuration parameters, such as mount options and special hardware support. Unfortunately, file systems often forget or misuse such configuration information (about 10% to 15% of semantic bugs are of this flavor). A semantic *configuration* bug is shown in S3 of Table 2.14; when Ext4 loads the journal from disk, it forgets to check if the device is read-only before updating the superblock.

Correct I/O request ordering is critical for crash consistency in file systems. The *I/O timing* category contains bugs involving incorrect I/O ordering. For example, in ordered journal mode, a bug may flush metadata to disk before the related data blocks are persisted. We found that only a small percentage of semantic bugs (3-9%) are I/O timing bugs; however, these bugs can lead to potential data loss or corruption. An I/O timing bug is shown in S4 of Table 2.14. ReiserFS is supposed to wait for the submitted transaction to commit synchronously instead of returning immediately.

A fair amount of *generic* bugs also exist in all file systems, such as using the wrong variable type or simple typos. These bugs are general coding mistakes (such as comparing unsigned variable with zero [216]), and may be fixed without much file-system knowledge. A generic semantic bug is shown in S5 of Table 2.14. In Ext3, `s_group_desc` is an array of pointers to `buffer_head`. However, at line 3, `memcpy` uses the address of `buffer_head`, which will corrupt the file system.

Summary: Incorrect state update and logic mistakes dominate seman-

ext3/namei.c, 2.6.26 **State (S1)**

```
   ext3_rename(...){
+    new_dir->i_ctime = CURRENT_TIME_SEC;
+    new_dir->i_mtime = CURRENT_TIME_SEC;
+    ext3_mark_inode_dirty(handle, new_dir);
```

ext3/ialloc.c, 2.6.4 **Logic (S2)**

```
   find_group_other(...){
-    group = parent_group + 1;
-    for (i = 2; i < ngroups; i++) {
+    group = parent_group;
+    for (i = 0; i < ngroups; i++) {
```

ext4/super.c, 2.6.37 **Configuration (S3)**

```
   ext4_load_journal(...){
-    if (journal_devnum && ...)
+    if (!read_only && journal_devnum ...)
       es->s_journal_dev = devnum;
```

reiserfs/super.c, 2.6.6 **I/O Timing (S4)**

```
   reiserfs_write_super_lockfs(...){
     journal_mark_dirty(&th, s, ...);
     reiserfs_block_writes(&th);
-    journal_end(&th, s, 1);
+    journal_end_sync(&th, s, 1);
```

ext3/resize.c, 2.6.17 **Generic (S5)**

```
   setup_new_group_blocks(...){
     lock_buffer(bh);
-    memcpy(b_data, s_group_desc[i], b_size);
+    memcpy(b_data, s_group_desc[i]->b_data, b_size);
     unlock_buffer(bh);
     ext3_journal_dirty_metadata(handle, gdb);
```

Table 2.14: **Semantic Bug Code Examples.** *This table shows five code examples of semantic bugs in different file systems. One example for each category:* state, logic, *config, I/O timing and generic.*

tic bug patterns; configuration errors are also not uncommon; incorrect I/O orderings are rare (but can have serious consequences); generic bugs require the least file-system knowledge to understand.

Concurrency Bugs

Concurrency bugs have attracted a fair amount of attention in the research community as of late [74, 103, 131, 217, 225]. To better understand file-system concurrency bugs, we classify them into six types as shown in Table 2.9 (on page 27): *atomicity violations*, *deadlocks*, *order violations*, *missed unlocks*, *double unlocks*, and *wrong locks*.

Figure 2.13(b) shows the percentage and total number of each category of concurrency bugs. Atomicity violation bugs are usually caused by a lack of proper synchronization methods to ensure exclusive data access, often leading to data corruption.

An example of an *atomicity* violation bug in Ext4 is shown in C1 of Table 2.15. For this bug, when two CPUs simultaneously allocate blocks, there is no protection for the `i_cached_extent` structure; this atomicity violation could thus cause the wrong location on disk to be read or written. A simple spin-lock resolves the bug.

There are a large number of *deadlocks* in file systems (about 40%). Two typical causes are the use of the wrong kernel memory allocation flag and calling a blocking function when holding a spin lock. These patterns are not common in application-level deadlocks, and thus knowledge of these patterns is useful to both developers (who should be wary of such patterns) and tool builders (who should detect them). Many deadlocks are found in ReiserFS, once again due to the BKL. The BKL could be acquired recursively; replacing it introduced a multitude of locking violations, many of which led to deadlock.

A typical memory-related deadlock is shown in C2 of Table 2.15. Btrfs uses `extent_readpages()` to read free space information; however, it *ext4/extents.c, 2.6.30* **Atomicity (C1)**

```
+    spin_lock(i_block_reservation_lock);
     i_cached_extent->ec_start = start;
+    spin_unlock(i_block_reservation_lock);
```

btrfs/extent_io.c, 2.6.39 **Deadlock (C2)**

```
     if (!add_to_page_cache_lru(page, mapping,
-    page->index, GFP__KERNEL)) {
+    page->index, GFP__NOFS)) {
       __extent_read_full_page(...);
```

`ext4/mballoc.c, 2.6.31` **Order (C3)**

```
+    rcu_barrier();
     kmem_cache_destroy(ext4_pspace_cachep);
```

ext3/resize.c, 2.6.17 **Miss Unlock (C4)**

```
   lock_super(sb);
   if (input->group != sbi->s_groups_count){
+    unlock_super(sb);
     goto exit_journal;
```

xfs/xfs_dfrag.c, 2.6.30 **Double Unlock (C5)**

```
     xfs_iunlock(ip, ...);
-  out_unlock:
-    xfs_iunlock(ip, ...);
   out:
     return error;
+  out_unlock:
+    xfs_iunlock(ip, ...);
+   goto out;
```

ext3/resize.c, 2.6.24 **Wrong Lock (C6)**

```
-    lock_buffer(bh);
+    lock_buffer(gdb);
     memcpy(gdb->b_data, b_data, b_size);
-    unlock_buffer(bh);
+    unlock_buffer(gdb);
```

Table 2.15: **Concurrency Bug Code Examples.** *This table shows six code examples of concurrency bugs. One example for each category:* atomicity violations, deadlocks, order violations, missed unlocks, double unlocks, *and* wrong locks.

should not use `GFP_KERNEL` flag to allocate memory, since the VM memory allocator `kswapd` will recursively call into file-system code to free memory. The fix changes the flag to `GFP_NOFS` to prevent VM re-entry into file-system code.

A noticeable percentage of *order* violation bugs exist in all file systems. An example of an order concurrency bug is shown in C3 of Table 2.15. The kernel memory cache `ext4_pspace_cachep` may be released by `call_rcu()` callbacks. Thus, Ext4 must wait for the release to complete before destroying the structure.

Missing unlocks happen mostly in exit or failure paths (e.g., putting resource releases at the end of functions with `goto` statements). C4 of Table 2.15 shows a missing-unlock bug. `ext3_group_add()` locks the super block (line 1) but forgets to unlock on an error (line 3).

The remaining two categories account for a small percentage. A *double unlock* example is shown in C5 of Table 2.15. The double unlock (line 1 and line 3) of XFS inode results in a reproduced deadlock on platforms which do not handle double unlock nicely. Using *wrong locks* also happens. As shown in C6 of Table 2.15, Ext3 should lock the group descriptor lock instead of the block bitmap's lock when updating the group descriptor.

Summary: Concurrency bugs are much more common in file systems than in user-level software. Atomicity and deadlock bugs represent a significant majority of concurrency bugs; many deadlock bugs are caused by wrong kernel memory-allocation flags; most missing unlocks happen on exit or failure paths.

Memory Bugs

Memory-related bugs are common in many source bases, and not surprisingly have been the focus of many bug detection tools [29, 158]. We classify memory bugs into six categories, as shown in Table 2.9: *resource leaks*, *null pointer dereferences*, *dangling pointers*, *uninitialized reads*, *double* *btrfs/inode.c, 2.6.30* **Resource Leak (M1)**

```
    inode = new_inode(...);
    ret = btrfs_set_inode_index(...);
-   if (ret)    return ERR_PTR(ret);
+   if (ret) {
+     iput(inode);
+     return ERR_PTR(ret);
+   }
```

ext3/super.c, 2.6.7 **Null Pointer (M2)**

```
    if (!journal) {
+     return NULL;
    }
     journal->j_private = sb;
```

btrfs/volumes.c, 2.6.34 **Dangling Pointer (M3)**

```
-   submit_bio(cur->bi_rw, cur);
    if (bio_rw_flagged(cur, BIO_RW_SYNCIO))
      num_sync_run++;
+   submit_bio(cur->bi_rw, cur);
```

ext4/ext4.h, 2.6.38 **Uninit Read (M4)**

```
  #define EXT4_EINODE_GET_XTIME (...){
    if (EXT4_FITS_IN_INODE(...))
      ext4_decode_extra_time(...);
+   else
+     (inode)->xtime.tv_nsec = 0;
```

ext4/xattr.c, 2.6.33 **Double Free (M5)**

```
    while (...){
      if (error)
        goto cleanup;
      kfree(b_entry_name);
+     b_entry_name = NULL;
    }
  cleanup:
    kfree(b_entry_name);
```

Table 2.16: **Memory Bug Code Examples.** *This table shows five code examples of memory bugs. One example for each category:* resource leaks, null pointer dereferences, dangling pointers, uninitialized reads, *and* double frees.

frees, and *buffer overflows*.

Resource leaks are the most dominant, over 40% in aggregate; in contrast, studies of user-level programs show notably lower percentages [120, 181, 201]. We find that roughly 70% of resource leaks happen on exit or failure paths; we investigate this further later (§2.3.7). An example of resource leaks (M1 of Table 2.16) is found in `btrfs_new_inode()` which allocates an inode but forgets to free it upon failure.

As we see in Figure 2.13(c), *null-pointer* dereferences are also common in both mature and young file systems. An example is shown in M2 of Table 2.16; a return statement is missing, leading to a null-pointer dereference at line 4 following the check.

Dangling pointers have a sizeable percentage across file systems. A Btrfs example is shown in M3 of Table 2.16. After callling `submit_bio` at line 1, the `bio` structure can be released at any time. Checking the `bio` flags at line 2 may give a wrong answer or lead to a crash.

Uninitialized reads are popular in most file systems, accounting for about 16% of all memory bugs. An Ext4 example is shown in M4 of Table 2.16. Ext4 should should zero out inode's `tv_nsec`, otherwise `stat()` will return garbage in the nanosecond component of timestamps.

Double free and *buffer overflow* only account for a small percentage. M5 of Table 2.16 shows a double free bug in Ext4. The while loop contains a `goto` statement (line 3) to `cleanup`, which also frees `b_entry_name` at line 8. To eliminate the potential double free on this error path, Ext4 sets `b_entry_name` to `NULL` at line 5.

Summary: Resource leaks are the largest category of memory bug, significantly higher than that in user-level applications; null-pointer dereferences are also common; failure paths contribute strongly to these bugs; many of these bugs have simple fixes.

Error Code Bugs

File systems need to handle a wide range of errors, including memory-allocation failures, disk-block allocation failures, I/O failures [24, 25], and silent data corruption [165]. Handling such faults, and passing error codes through a complex code base, has proven challenging [84, 178]. Here, we further break down error-code errors.

We partition the error code bugs into *missing error codes* and *wrong error codes* as described in Table 2.9. Figure 2.13(d) shows the breakdown of error code bugs. Missing errors are generally twice as prevalent as wrong errors (except for JFS, which has few of these bugs overall).

A *missing error* code example is shown in E1 of Table 2.17. The routine `posix_acl_from_disk()` could return an error code (line 2). However, without error checking, `acl` is accessed at line 3 and thus the kernel crashes.

An example of a *wrong error* code is shown in E2 of Table 2.17. The return value should be `-EIO`. However, in line 3, the original code returns the close (but wrong) error code `EIO`; callers thus fail to detect the error.

Summary: Error handling bugs occur in two flavors, missing error handling or incorrect error handling; the bugs are relatively simple in nature compared with other types of bugs.

2.3.7 The Failure Path

Many bugs we found arose not in common-case code paths but rather in more unusual fault-handling cases [84, 180, 227]. This type of error handling (i.e., reacting to disk or memory failures) is critical to robustness, since bugs on failure paths can lead to serious consequences. We now quantify bug occurrences on failure paths; Tables 2.18 and Tables 2.19 present our accumulated results.

As we can see from the first table, roughly a third of bugs are introduced on failure paths across all file systems. Even mature file systems such as *reiserfs/xattr_acl.c, 2.6.16* **Miss Error (E1)**

```
    reiserfs_get_acl(...){
      acl = posix_acl_from_disk(...);
-     *p_acl = posix_acl_dup(acl);
+     if (!IS_ERR(acl))
+       *p_acl = posix_acl_dup(acl);
```

jfs/jfs_imap.c, 2.6.27 **Wrong Error (E2)**

```
    diAlloc(...){
      jfs_error(...);
-     return EIO;
+     return -EIO;
```

Table 2.17: **Error Code Bug Code Examples.** *This table shows two code examples of error code bugs. One example for each category: missing error codes and wrong error codes.*

XFS	Ext4	Btrfs	Ext3	ReiserFS	JFS
200 (39.1%)	149 (33.1%)	144 (40.2%)	88 (38.4%)	63 (39.9%)	28 (35%)

Table 2.18: **Failure Related Bugs By File System.** *This table shows the number and percentage of the bugs related to failures, partitioned by file systems.*

Semantic	Concurrency	Memory	Error Code
283 (27.7%)	93 (25.4%)	117 (53.4%)	179 (100%)

Table 2.19: **Failure Related Bugs By Bug Pattern.** *This table shows the number and percentage of the bugs related to failures, partitioned by bug patterns.*

ext4/resize.c, 2.6.25 **Semantic (F1)**

```
     ext4_group_extend(...)  {
       ext4_warning(sb, "multiple resizers are running");
       unlock_super(sb);
  +    ext4_journal_stop(handle);
       err = -EBUSY;
       goto exit_put;
```

ext4/mballoc.c, 2.6.27 **Concurrency (F2)**

```
     mb_free_blocks(...)  {
  +    ext4_unlock_group(sb, e4b->bd_group);
       ext4_error(sb, ...  "double-free of inode");
  +    ext4_lock_group(sb, e4b->bd_group);
```

btrfs/inode.c, 2.6.39 **Memory (F3)**

```
     btrfs_new_inode(...)  {
       path = btrfs_alloc_path();
       inode = new_inode(root->fs_info->sb);
  -    if (!inode)
  +    if (!inode) {
  +      btrfs_free_path(path);
         return ERR_PTR(-ENOMEM);
  +    }
```

Table 2.20: **Code Examples of Bugs on Failure Paths.** *This table shows three bug code examples of bugs on failure paths. One example for each category: semantic, concurrency, and memory.*

Ext3 and XFS make a significant number of mistakes on these error paths.

When broken down by bug type in the second table, we see that roughly a quarter of semantic bugs occur on failure paths, usually in the previously-defined *state* and *logic* categories. Once a failure happens (e.g., an I/O fails), the file system needs to free allocated disk resources and update related metadata properly; however, it is easy to forget these updates, or perform them incorrectly, leading to many *state* bugs. In addition, wrong algorithms (*logic* bugs) are common; for example, when block allocation fails, most file systems return `ENOSPC` immediately instead of retrying after committing buffered transactions.

An example of a *semantic* bug on failure paths is shown in F1 of Table 2.20. When Ext4 detects that multiple resizers are running on the file system at the same time, it forgets to stop the journal to prevent further data corruption.

A quarter of *concurrency* bugs arise on failure paths. Sometimes, file systems forget to unlock locks, resulting in deadlock. Moreover, when file systems output errors to users, they sometimes forget to unlock before calling blocking error-output functions (deadlock). These types of mistakes rarely arise in user-level code [131]. Such an example is shown in F2 of Table 2.20. `ext4_error()` is a blocking function, which cannot be called with a spinlock held. A correct fix is to unlock (line 2) before the blocking function and lock again (line 4) after that.

For *memory* bugs, most resource-leak bugs stem from forgetting to release allocated resources when I/O or other failures happen. There are also numerous null-pointer dereference bugs which incorrectly assume certain pointers are still valid after a failure. Finally (and obviously), all error code bugs occur on failure paths (by definition). A typical example of memory leak bugs on failure path is shown in F3 of Table 2.20. Btrfs forgets to release allocated memory for `path` (line 2) when inode memory allocation fails. Thus, Btrfs should free `path` (line 6) before returning an error code.

It is difficult to fully test failure-handling paths to find all types of bugs. Most previous work has focused on memory resource leaks [180, 227], missing unlock [180, 227] and error codes [84, 178]; however, existing work can only detect a small portion of failure-handling errors, especially omitting a large amount of semantic bugs on failure paths. Our results provide strong motivation for improving the quality of failure-handling code in file systems.

Summary: A high fraction of bugs occur due to improper behavior in the presence of failures or errors across all file systems; memory-related errors are particularly common along these rarely-executed code paths; a quarter of semantic bugs are found on failure paths.

2.4 Performance and Reliability

A small but important set of patches improve performance and reliability of file systems, which are quantitatively different than bug patches (as shown in Figure 2.4). Performance and reliability patches account for 8% and 7% of patches respectively.

2.4.1 Performance Patches

Performance is critical for all file systems. Performance patches are proposed to improve existing designs or implementations. We partition these patches into six categories as shown in Table 2.21, including synchronization (*sync*), access optimization (*access*), scheduling (*sched*), scalability (*scale*), locality (*locality*), and *other*. Figure 2.22(a) shows the breakdown.

Synchronization-based performance improvements account for over a quarter of all performance patches across file systems. Typical solutions used include removing a pair of unnecessary locks, using finer-grained locking, and replacing write locks with read/write locks. A *sync* patch

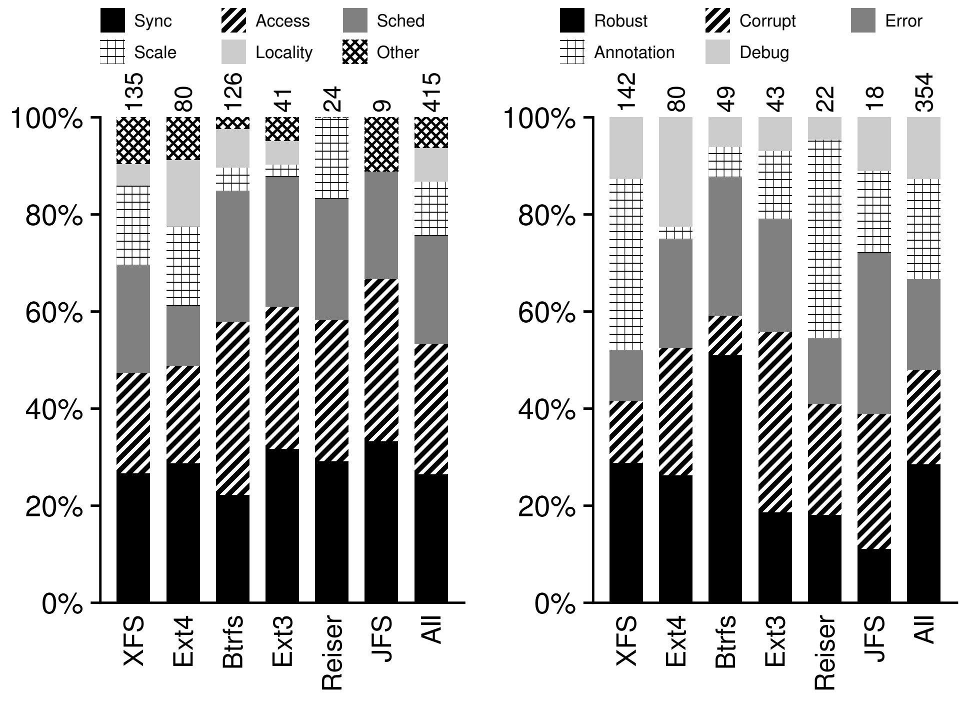
Type	Description
Synchronization	Inefficient usage of synchronization methods (e.g., removing unnecessary locks, using smaller locks, using read/write locks)
Access Optimization	Apply smarter access strategies (e.g., caching metadata and statistics, avoiding unnecessary I/O and computing)
Schedule	Improve I/O operations scheduling (e.g., batching writes, opportunistic readahead)
Scalability	Scale on-disk and in-memory data structures (e.g., using trees or hash tables, per block group structures, reducing memory usage of inodes)
Locality	Overcome sub-optimal data block allocations (e.g., reducing file fragmentation, clustered I/Os)
Other	Other performance improvement techniques (e.g., reducing function stack usage)

Table 2.21: **Performance Patch Type.** *This table shows the classification and definition of performance patches. There are six categories: synchronization (sync), access optimization (access), scheduling (sched), scalability (scale), locality (locality), and other.*

is shown in P1 of Table 2.23; `ext4_fiemap()` uses write instead of read semaphores, limiting concurrency.

Access patches use smarter strategies to optimize performance, including caching and work avoidance. For example, Ext3 caches metadata stats in memory, avoiding I/O. Figure 2.22(a) shows *access* patches are popular. An example Btrfs *access* patch is shown in P2 of Table 2.23; before searching for free blocks, it first checks whether there is enough free space, avoiding unnecessary work.

Sched patches improve I/O scheduling for better performance, such as batching writes, opportunistic readahead, and avoiding unnecessary synchrony in I/O. As can be seen, *sched* has a similar percentage compared to *sync* and *access*. An interesting example of *sched* patch is shown in P3 of Table 2.23. A little mistake is making all transactions synchronous, which



(a) Performance (b) Reliability

Figure 2.22: **Performance and Reliability Patch Patterns.** *This figure shows the performance and reliability patch patterns. The total number of patches is shown on top of each bar.*

reduces Ext3 performance to comical levels.

Scale patches utilize scalable on-disk and in-memory data structures, such as hash tables, trees, and per block-group structures. XFS has a large number of *scale* patches, as scalability was always its priority. An example Ext4 *scale* patch is shown in P4 of Table 2.23. Ext4 uses `ext4_lblk_t` (32 bits) instead of `sector_t` (64 bits) for logical blocks, saving unnecessary wasted memory in `ext4_inode_info` structure.

Locality patches overcome sub-optimal data locality on disk, such as reducing file fragmentation, improving the data block allocation algorithm. A *locality* patch of Btrfs is shown in P5 of Table 2.23. Btrfs uses larger meta-

ext4/extents.c, 2.6.31 **Synchronization (P1)**

```
   ext4_fiemap(...){
-    down_write(&EXT4_I(inode)->i_data_sem);
+    down_read(&EXT4_I(inode)->i_data_sem);
     error = ext4_ext_walk_space(...);
-    up_write(&EXT4_I(inode)->i_data_sem);
+    up_read(&EXT4_I(inode)->i_data_sem);
```

btrfs/free-space-cache.c, 2.6.39 **Access Optimization (P2)**

```
+    if (bg->free_space < min_bytes){
+      spin_unlock(&bg->tree_lock);
+      return -ENOSPC;
+    }
      /* start to search for blocks */
```

ext3/xattr.c, 2.6.21 **Schedule (P3)**

```
     error = ext3_journal_dirty_metadata(handle, bh);
-    handle->h_sync = 1;
+    if (IS_SYNC(inode))
+      handle->h_sync = 1;
```

ext4/ext4.h, 2.6.38 **Scalability (P4)**

```
   struct ext4_inode_info {
-    sector_t i_da_metadata_calc_last_lblock;
+    ext4_lblk_t i_da_metadata_calc_last_lblock;
```

btrfs/extent-tree.c, 2.6.29 **Locality (P5)**

```
     int empty_cluster = 2 * 1024 * 1024;
     if (data & BTRFS_BLOCK_GROUP_METADATA) {
-      empty_cluster = 64 * 1024;
+      if (!btrfs_test_opt(root, SSD))
+        empty_cluster = 64 * 1024;
```

xfs/xfs_dir2_leaf.c, 2.6.17 **Other (P6)**

```
-    xfs_dir2_put_args_t p;
+    xfs_dir2_put_args_t *p;
```

Table 2.23: **Performance Patch Code Examples.** *This table shows the code examples of performance patches. One example for each category: sync, access,sched, scale, locality, and other.*

data clusters for SSD devices to improve write performance (overwrite the whole SSD block to avoid expensive erase overhead). For spinning disks, Btrfs uses smaller metadata clusters to get better fsck performance.

There are also other performance improvement techniques, such as reducing function stack usage and using slab memory allocator. As shown in P6 of Table 2.23, XFS dynamically allocates a `xfs_dir2_put_args_t` structure to reduce stack pressure in function `xfs_dir2_leaf_getdents`.

Summary: Performance patches exist in all file systems; *sync*, *access*, and *sched* each account for a quarter of the total; many of the techniques used are fairly standard (e.g., removing locks); while studying new synchronization primitives, we should not forget about performance.

2.4.2 Reliability Patches

Type	Description
Robust	Enhance file-system robustness (e.g., boundary limits and access permission checking, additional internal assertions)
Corruption Defense	Improve file systems' ability to handle various possible corruptions
Error Enhancement	Improve original error handling (e.g., gracefully handling failures, more detailed error codes)
Annotation	Add endianness, user/kernel space pointer and lock annotations for early bug detection
Debug	Add more internal debugging or tracing support

Table 2.24: **Reliability Patch Type.** *This table shows the classification and definition of reliability patches. There are five categories: functional robustness (robust), corruption defense (corruption), error enhancement (error), annotation (annotation), and debugging (debug.)*

Finally we study our last class of patch, those that aim to improve file-system reliability. Different from bug-fix patches, reliability patches are not utilized for correctness. Rather, for example, such a patch may check whether the super block is corrupted before mounting the file system; further, a reliability patch might enhance error propagation [84] or add more debugging information. Table 2.24 presents the classification of these *reliability* patches, including adding assertions and other functional robustness (*robust*), corruption defense (*corruption*), error enhancement (*error*), annotation (*annotation*), and debugging (*debug*). Figure 2.22(b) displays the distributions.

Robust patches check permissions, enforce file-system limits, and handle extreme cases in a more friendly manner. Btrfs has the largest percentage of these patches, likely due to its early stage of development. A *robust* patch is shown in R1 of Table 2.25. JFS forbids users to change file flags on quota files, avoiding unnecessary problems caused by users.

Corruption defense patches validate the integrity of metadata when reading from disk. For example, a patch to Ext4 checks that a directory entry is valid before traversing that directory. In general, many *corruption* patches are found at the I/O boundary, when reading from disk. An example of a corruption defense patch of JBD2 (used by Ext4) is shown in R2 of Table 2.25. Since the journal may be too short due to disk corruption, JBD2 refuses to load a journal if its length is not valid.

Error enhancement patches improve error handling in a variety of ways, such as more detail in error codes, removing unnecessary error messages, and improving availability, for example by remounting read-only or returning error codes instead of crashing. This last class is common in all file systems, which each slowly replaced unnecessary `BUG()` and assertion statements with more graceful error handling. A typical example in Btrfs is shown R3 of Table 2.25. When the memory allocation at line 2 fails, instead of calling `BUG_ON(1)` to crash, Btrfs releases all allocated memory pages and returns an appropriate error code.

Annotation patches label variables with additional type information (e.g., endianness) and locking rules to enable better static checking. Reis-

jfs/ioctl.c, 2.6.24 **Robust (R1)**

```
   jfs_ioctl(...)  {
+    if (IS_NOQUOTA(inode))
+      return -EPERM;
     fs_get_inode_flags(jfs_inode);
```

jbd2/journal.c, 2.6.32 **Corruption Defense (R2)**

```
   journal_reset(...){
     first = be32_to_cpu(sb->s_first);
     last = be32_to_cpu(sb->s_maxlen);
+    if (first + JBD2_MIN_JOURNAL_BLOCKS > last + 1){
+      printk(KERN_ERR "JBD: Journal too short");
+      journal_fail_superblock(journal);
+      return -EINVAL;
+    }
```

btrfs/file.c, 2.6.38 **Error Enhancement (R3)**

```
   prepare_pages(...)  {
     pages[i] = grab_cache_page(...);
     if (!pages[i]) {
-      BUG_ON(1);
+      for (c = i - 1; c >= 0; c--) {
+        unlock_page(pages[c]);
+        page_cache_release(pages[c]);
+      }
+      return -ENOMEM;
     }
```

Table 2.25: **Reliability Patch Code Examples.** *This table shows the code examples of reliability patches. One example for categories: robust, corruption, and error.*

erFS uses lock annotations to help prevent deadlock, whereas XFS uses endianness annotations for numerous variable types. *Debug* patches simply add more diagnostic information at failure-handling points within the file system.

Interestingly, reliability patches appear more ad hoc than bug patches. For bug patches, most file systems have similar pattern breakdowns. In contrast, file systems make different choices for reliability, and do so in a generally non-uniform manner. For example, Btrfs focuses more on *Robust* patches, while Ext3 and Ext4 add more *Corruption* defense patches.

Summary: We find that *reliability* patches are added to file systems over time as part of hardening; most add simple checks, defend against corruption upon reading from disk, or improve availability by returning errors instead of crashing; annotations help find problems at compile time; debug patches add diagnostic information; reliability patch usage, across all file systems, seems ad hoc.

2.5 Case Study Using PatchDB

The patch dataset constructed from our analysis of 5079 patches contains fine-grained information, including characterization of bug patterns (e.g., which semantic bugs forget to synchronize data), detailed bug consequences (e.g., crashes caused by assertion failures or null-pointer dereferences), incorrect bug fixes (e.g., patches that are reverted after being accepted), performance techniques (e.g., how many performance patches remove unnecessary locks), and reliability enhancements (e.g., the location of metadata integrity checks). These details enable further study to improve file-system designs, propose new system language constructs, build custom bug-detection tools, and perform realistic fault injection.

In this section, we show the utility of *PatchDB* by examining which patches are common across all file systems. Due to space concerns, we

Patch Type	Typical Cases	XFS	Ext4	Btrfs	Ext3	Reiser	JFS
Semantic	forget sync	17	11	6	11	5	1
	forget config	43	43	23	16	8	1
	early enospc	5	9	14	7		
	wrong log credit	6	8	1	1	1	
Concurrency	lock inode update	6	5	2	4	4	2
	lock sleep	8	8	1	1	8	
	wrong kmalloc flag	20	3	3	2		1
	miss unlock	10	7	4	2	2	4
Memory	leak on failure	14	21	16	11	1	3
	leak on exit	1		1	4		1
Error Code	miss I/O error	10	11	8	15	4	1
	miss mem error	4	2	13	1	1	
	bad error access		3	8			2
Performance	remove lock	17	14	14	8	5	1
	avoid redun write	6	4	5	4		2
	check before work	8	5	15	2	1	
	save struct mem	3	9		1	3	
Reliability	metadata validation	12	9	1	7	2	1
	graceful handle	8	6	5	5	1	4

Table 2.26: **Common File-System Patches.** *This table shows the classification and count of common patches across all file systems.*

only highlight a few interesting cases. A summary is found in Table 2.26.

We first discuss specific common bugs. Within semantic bugs is *forget sync*, in which a file system forgets to force data or metadata to disk. Most *forget sync* bugs relate to *fsync*. Even for stable file systems, there are a noticeable number of these bugs, leading to data loss or corruption under power failures. Another common mistake is *forget config*, in which mount options, feature sets, or hardware support are overlooked. File systems also return the `ENOSPC` error code despite the presence of free blocks (*early enospc*); Btrfs has the largest number of these bugs, and even refers to the

Ext3 fix strategy in its patches. Even though semantic bugs are dominant in file systems, few tools can detect semantic bugs due to the difficulty of specifying correct behavior [70, 119, 121]. Fortunately, we find that many semantic bugs appear across file systems, which can be leveraged to improve bug detection.

For concurrency bugs, forgetting to lock an inode when updating it is common; perhaps a form of monitors [92] would help. Calling a blocking function when holding a spin lock (*lock sleep*) occurs frequently (also in drivers [54, 159]). As we saw earlier (§2.3.6), using the wrong kernel memory allocation flag is a major source of deadlock (particularly XFS). All file systems miss unlocks frequently, in contrast to user applications [131].

For memory bugs, leaks happen on failure or exit paths frequently. For error code bugs, there are a large number of *missed I/O error* bugs. For example, Ext3, JFS, ReiserFS and XFS all ignore write I/O errors on fsync before Linux 2.6.9 [165]; as a result, data could be lost even when fsync returned successfully. Memory allocation errors are also often ignored (especially in Btrfs). Three file systems mistakenly dereference error codes.

For performance patches, removing locks (without sacrificing correctness) is common. File systems also tend to write redundant data (e.g., *fdatasync* unnecessarily flushes metadata). Another common performance improvement case is *check before work*, in which missing specific condition checking costs unnecessary I/O or CPU overhead.

Finally, for reliability patches, metadata validation (i.e., inode, super block, directory and journal) is popular. Most of these patches occur in similar places (e.g., when mounting the file system, recovering from the journal, or reading an inode). Also common is replacing `BUG()` and `Assert()` calls with more graceful error handling.

Summary: Despite their diversity, file systems share many similarities across implementations; some examples occur quite frequently; PatchDB affords new opportunities to study such phenomena in great detail.

2.6 Conclusions

In this chapter, we perform a comprehensive study of 5079 patches across six Linux file systems; our analysis includes one of the largest studies of bugs to date (nearly 1800 bugs). We make the following high-level observations. A large number of patches (nearly 50%) are maintenance patches, reflecting the constant refactoring work needed to keep code simple and maintainable. The remaining dominant category is bugs (just under 40%), showing how much effort is required to slowly inch towards a "correct" implementation. Interestingly, the number of bugs does not die down over time (even for stable file systems), rather ebbing and flowing.

Breaking down the bug category further, we find that semantic bugs are the dominant bug category (over 50% of all bugs). These types of bugs are vexing, as most of them are hard to detect via generic bug detection tools; Concurrency bugs are the next most common (about 20% of bugs), more prevalent than in user-level software. The remaining bugs are split evenly across memory bugs and error handling.

We also categorize bugs along other axes to gain further insight. For example, when broken down by consequence, we find that most of the bugs we studied lead to crashes or corruption, and hence are quite serious. When categorized by data structure, we find that B-trees, present in many file systems for scalability, have relatively few bugs per line of code. When classified by whether bugs occur on normal or failure-handling paths, we make the following important discovery: nearly 40% of all bugs occur on failure-handling paths. Future system designs need better tool or language support to make these rarely-executed failure paths correct.

Finally, while bug patches comprise most of our study, performance and reliability patches are also prevalent. The performance techniques used are relatively common and widespread (e.g., removing an unnecessary I/O, or downgrading a write lock to a read lock). About a quarter of performance patches reduce synchronization overheads. In contrast to performance techniques, reliability techniques seem to be added in a rather ad hoc fashion (e.g., most file systems apply sanity checks non-uniformly).

Beyond numerous interesting results, another outcome of our work is an annotated dataset of file-system patches, which is publicly available for further study (`http://research.cs.wisc.edu/wind/Traces/fs-patch`). We hope that this dataset should be of utility to file-system developers, systems-language designers, and tool makers; the careful study of these results should result in a new generation of more robust, reliable, and performant file systems.

3 Physical Disentanglement in IceFS

From the previous chapter, we know that many bugs exist in file systems, and most of bugs can lead to serious consequences (e.g., data corruption or system crashes). One natural following question is that is there enough isolation within a file system to cope with corruption, crashes and performance problems. Unfortunately, we find that one aspect of current file-system design has remained devoid of isolation: the physical on-disk structures of file systems. Entanglement in a file system can cause shared failures and bundled performance problems for all workloads. Isolation, an important property of systems, is not preserved in file systems.

In this chapter, we first demonstrate the root problems caused by physical entanglement in current file systems. We then propose a new file system we call *IceFS*. IceFS provides users with a new basic abstraction in which to co-locate logically similar information; we call these containers *cubes*. IceFS then works to ensure that files and directories within cubes are physically distinct from files and directories in other cubes; thus data and I/O within each cube is *disentangled* from data and I/O outside of it.

Primary benefits of cube disentanglement are *localization* and *specialization*. First, we show how cubes enable localized *micro-failures*; crashes and read-only remounts that normally affect the entire system are now constrained to the faulted cube. Second, we show how cubes permit localized *micro-recovery*; instead of an expensive file-system wide repair, the disentanglement found at the core of cubes enables IceFS to fully (and quickly) repair a subset of the file system. Third, we illustrate how transaction splitting allows the file system to commit transactions from different cubes in parallel, greatly increasing performance (by a factor of 2x–5x) for some workloads. Because cubes are independent, it is natural for the file system to tailor the behavior of each. We realize the benefits of specialization by allowing users to choose different journaling modes per cube; doing so creates a performance/consistency knob that can be set as appropriate for a particular workload, enabling higher performance.

The rest of this chapter is organized as follows. We first show in Section 3.1 that the aforementioned problems exist through experiments. Then we introduce the three principles for building a disentangled file system in Section 3.2, describe our prototype IceFS and its benefits in Section 3.3, and evaluate IceFS in Section 3.4. Finally, we conclude in Section 3.5.

3.1 Motivation

Logical entities, such as directories, provided by the file system are an illusion; the underlying physical entanglement in data structures and transactional mechanisms does not provide true isolation. We describe three problems that this entanglement causes: global failure, slow recovery, and bundled performance. After discussing how current approaches fail to address them, we describe the negative impact on modern systems.

3.1.1 Entanglement Problems

Global Failure

Ideally, in a robust system, a fault involving one file or directory should not affect other files or directories, the remainder of the OS, or other users. However, in current file systems, a single fault often leads to a *global failure*.

Global Failures	Ext3	Ext4	Btrfs
Crash	129	341	703
Read-only	64	161	89

Table 3.1: **Global Failures in File Systems.** *This table shows the average number of crash and read-only failures in Ext3, Ext4, and Btrfs source code across 14 versions of Linux (3.0 to 3.13).*

A common approach for handling faults in current file systems is to either *crash* the entire system (e.g., by calling `BUG_ON`, `panic`, or `assert`) or to mark the whole file system *read-only*. Crashes and read-only behavior are not constrained to only the faulty part of the file system; instead, a global reaction is enforced for the whole system. For example, Btrfs crashes the entire OS when it finds an invariant is violated in its extent tree; Ext3 marks the whole file system as read-only when it detects a corruption in a single inode bitmap. To illustrate the prevalence of these coarse reactions, we analyzed the source code and counted the average number of such global failure instances in Ext3 with JBD, Ext4 with JBD2, and Btrfs from Linux 3.0 to 3.13. As shown in Table 3.1, each file system has hundreds of invocations to these poor global reactions.

Current file systems trigger global failures to react to a wide range of system faults. Table 3.2 shows there are many root causes: metadata failures and corruptions, pointer faults, memory allocation faults, and invariant faults. These types of faults exist in real systems [24, 25, 54, 126, 159, 200, 201], and they are used for fault injection experiments in many research projects [52, 165, 167, 203, 205, 233]. Responding to these various faults in a non-global manner is non-trivial; the table shows that a high percentage (89% in Ext3, 65% in Ext4) of these faults are caused by entangled data structures (e.g., bitmaps and transactions).

Fault Type	Ext3	Ext4
Metadata read failure	70 (66)	95 (90)
Metadata write failure	57 (55)	71 (69)
Metadata corruption	25 (11)	62 (28)
Pointer fault	76 (76)	123 (85)
Interface fault	8 (1)	63 (8)
Memory allocation	56 (56)	69 (68)
Synchronization fault	17 (14)	32 (27)
Logic fault	6 (0)	17 (0)
Unexpected states	42 (40)	127 (54)

Table 3.2: **Failure Causes in File Systems.** *This table shows the number of different failure causes for Ext3 and Ext4 in Linux 3.5, including those caused by entangled data structures (in parentheses). Note that a single failure instance may have multiple causes.*

Slow Recovery

After a failure occurs, file systems often rely on an offline file-system checker to recover [143]. The checker scans the whole file system to verify the consistency of metadata and repair any observed problems. Unfortunately, current file system checkers are *not scalable*: with increasing disk capacities and file system sizes, the time to run the checker is unacceptably long, decreasing availability. For example, Figure 3.3 shows that the time to run a checker [208] on an Ext3 file system grows linearly with the size of the file system, requiring about 1000 seconds to check an 800GB file system with 50% utilization. Ext4 has better checking performance due to its layout optimization [140], but the checking performance is similar to Ext3 after aging and fragmentation [133].

Despite efforts to make checking faster [31, 133, 161], check time is still constrained by file system size and disk bandwidth. The root problem is that current checkers are *pessimistic*: even though there is only a small piece of corrupt metadata, the entire file system is checked. The main reason

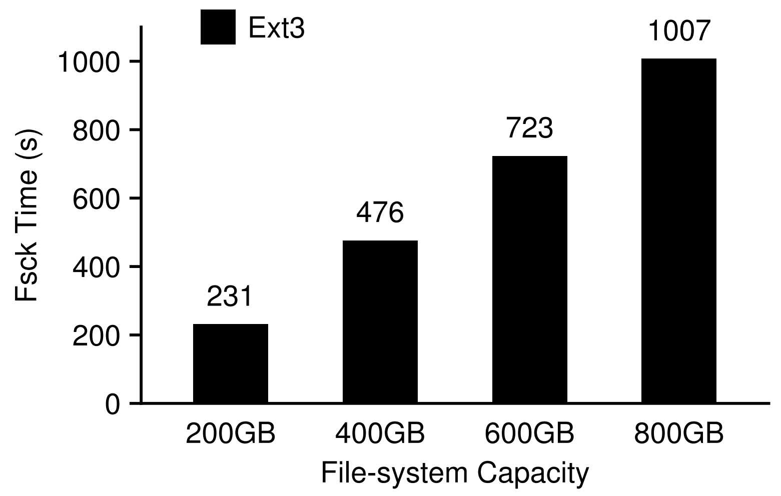


Figure 3.3: **Scalability of E2fsck on Ext3.** *This figure shows the fsck time on Ext3 with different file-system capacity. We create the initial file-system image on partitions of different capacity (x-axis). We make 20 directories in the root directory and write the same set of files to every directory. As the capacity changes, we keep the file system at 50% utilization by varying the amount of data in the file set.*

is that due to entangled data structures, it is hard or even impossible to determine which part of the file system needs checking.

Bundled Performance and Transactions

The previous two problems occur because file systems fail to isolate metadata structures; additional problems occur because the file system journal is a shared, global data structure. For example, Ext3 uses a generic journaling module, JBD, to manage updates to the file system. To achieve better throughput, instead of creating a separate transaction for every file system update, JBD groups all updates within a short time interval (e.g., 5s) into a single global transaction; this transaction is then committed periodically or when an application calls `fsync()`.

Unfortunately, these bundled transactions cause the performance of independent processes to be bundled. Ideally, calling `fsync()` on a file should flush only the dirty data belonging to that particular file to disk; unfortunately, in the current implementation, calling `fsync()` causes unrelated data in the same transaction to be flushed as well. Therefore, the performance of write workloads may suffer when multiple applications are writing at the same time.

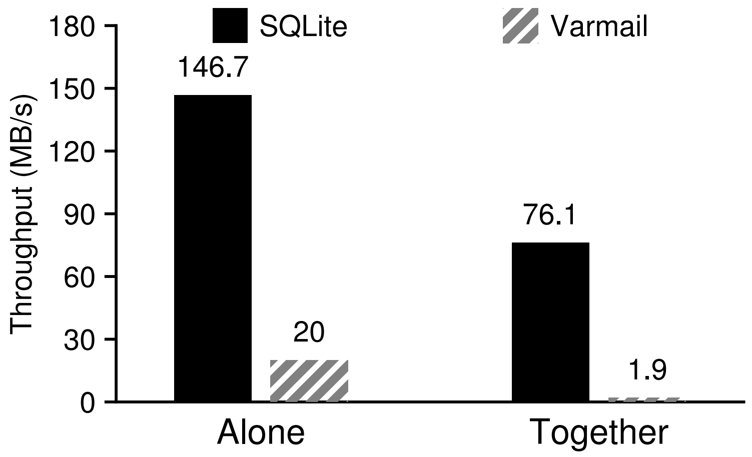


Figure 3.4: **Bundled Performance on Ext3.** *This figure shows the performance of running SQLite and Varmail on Ext3 in ordered mode. The SQLite workload, configured with write-ahead logging, asynchronously writes 40KB values in sequential key order. The Varmail workload involves 16 threads, each of which performs a series of create-append-sync and read-append-sync operations.*

Figure 3.4 illustrates this problem by running a database application SQLite [9] and an email server workload Varmail [3] on Ext3. SQLite sequentially writes large key/value pairs asynchronously, while Varmail frequently calls `fsync()` after small random writes. As we can see, when these two applications run together, both applications' performance degrades significantly compared with running alone, especially for Varmail. The main reason is that both applications share the same journaling layer and each workload affects the other. The `fsync()` calls issued by Varmail must wait for a large amount of data written by SQLite to be flushed together in the same transaction. Thus, the single shared journal causes performance entanglement for independent applications in the same file system. Note that we use an SSD to back the file system, so device performance is not a bottleneck in this experiment.

3.1.2 Limitations of Current Solutions

One popular approach for providing isolation in file systems is through the namespace. A namespace defines a subset of files and directories that are made visible to an application. Namespace isolation is widely used for better security in a shared environment to constrain different applications and users. Examples include virtual machines [40, 69], Linux containers [2, 7], `chroot`, BSD `jail` [105], and Solaris Zones [156].

However, these abstractions fail to address the problems mentioned above. Even though a namespace can restrict application access to a subset of the file system, files from different namespaces still share metadata, system states, and even transactional machinery. As a result, a fault in any shared structure can lead to a global failure; a file-system checker still must scan the whole file system; updates from different namespaces are bundled together in a single transaction.

Another widely-used method for providing isolation is through static disk partitions. Users can create multiple file systems on separate partitions. Partitions are effective at isolating corrupted data or metadata such that read-only failure can be limited to one partition, but a single `panic()` or `BUG_ON()` within one file system may crash the whole OS, affecting all partitions. In addition, partitions are not flexible in many ways and the number of partitions is usually limited. Furthermore, storage space may not be effectively utilized and disk performance may decrease due to the lack of a global block allocation. Finally, it can be challenging to manage a large number of partitions across different file systems and applications.

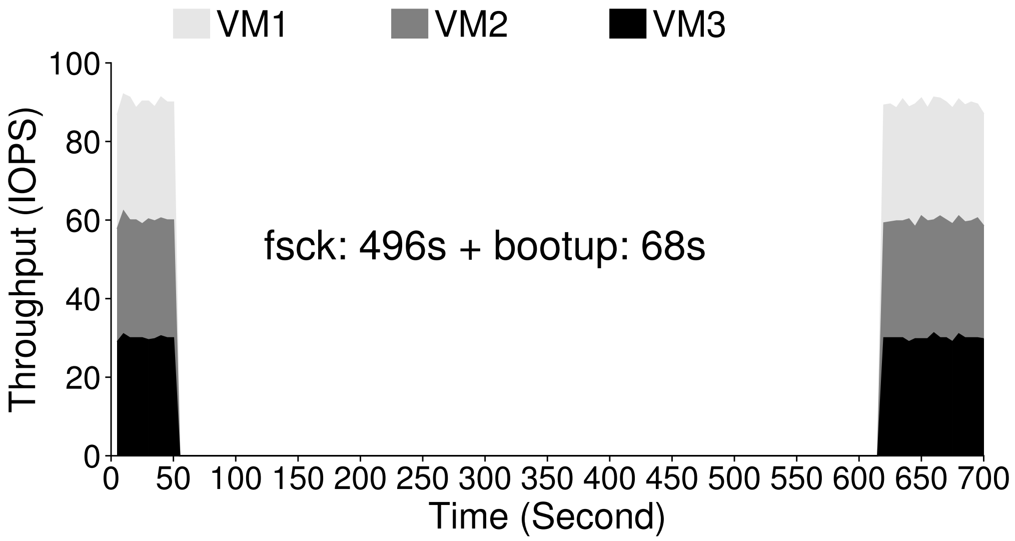


Figure 3.5: **Global Failure for Virtual Machines.** *This figure shows how a fault in Ext3 affects all three virtual machines (VMs). Each VM runs a workload that writes 4KB blocks randomly to a 1GB file and calls* `fsync()` *after every 10 writes. We inject a fault at 50s, run e2fsck after the failure, and reboot all three VMs.*

3.1.3 Usage Scenarios

Entanglement in the local file system can cause significant problems to higher-level services like virtual machines and distributed file systems. We now demonstrate these problems via two important cases: a virtualized storage environment and a distributed file system.

Virtual Machines

Fault isolation within the local file system is of paramount importance to server virtualization environments. In production deployments, to increase machine utilization, reduce costs, centralize management, and make migration efficient [55, 185, 210], tens of virtual machines (VMs) are often consolidated on a single host. The virtual disk image for each VM is usually stored as a single or a few files within the host file system. If a single fault triggered by one of the virtual disks causes the host file system to become read-only (e.g., metadata corruption) or to crash (e.g., assertion failures), then all the VMs suffer. Furthermore, recovering the file system using fsck and redeploying all VMs require considerable downtime.

Figure 3.5 shows how VMware Workstation 9 [213] running with an Ext3 host file system reacts to a read-only failure caused by one virtual disk image. When a read-only fault is triggered in Ext3, all three VMs receive an error from the host file system and are immediately shut down. There are 10 VMs in the shared file system; each VM has a preallocated 20GB virtual disk image. Although only one VM image has a fault, the entire host file system is scanned by e2fsck, which takes more than eight minutes. This experiment demonstrates that a single fault can affect multiple unrelated VMs; isolation across different VMs is not preserved.

Distributed File Systems

Physical entanglement within the local file system also negatively impacts distributed file systems, especially in multi-tenant settings. Global failures in local file systems manifest themselves as machine failures, which are handled by crash recovery mechanisms. Although data is not lost, fault isolation is still hard to achieve due to long timeouts for crash detection and the layered architecture. We demonstrate this challenge in HDFS [191], a popular distributed file system used by many applications.

Although HDFS provides fault-tolerant machinery such as replication and failover, it does not provide fault isolation for applications. Thus, applications (*e.g.*, , HBase [1, 88]) can only rely on HDFS to prevent data loss and must provide fault isolation themselves. For instance, in HBase multi-tenant deployments, HBase servers can manage tables owned by various clients. To isolate different clients, each HBase server serves a certain number of tables [6]. However, this approach does not provide

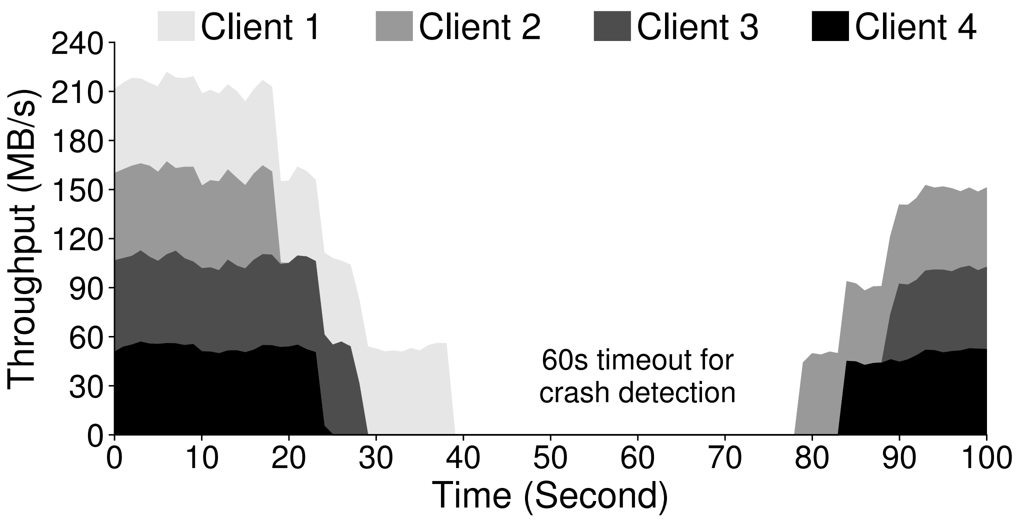


Figure 3.6: **Impact of Machine Crashes in HDFS.** *This figure shows the negative impact of physical entanglement within local file systems on HDFS. A kernel panic of a local file system leads to a machine failure, which negatively affects the throughput of multiple clients.*

complete isolation: although HBase servers are grouped based on tables, their tables are stored in HDFS nodes, which are not aware of the data they store. Thus, an HDFS server failure will affect multiple HBase servers and clients. Although indirection (e.g., HBase on HDFS) simplifies system management, it makes isolation in distributed systems challenging.

Figure 3.6 illustrates such a situation: four clients concurrently read different files stored in HDFS when a machine crashes; the crashed machine stores data blocks for all four clients. In this experiment, only the first client is fortunate enough to not reference this crashed node and thus finishes early. The other three lose throughput for 60 seconds before failing over to other nodes. Although data loss does not occur as data is replicated on multiple nodes in HDFS, this behavior may not be acceptable for latency-sensitive applications.

3.2 File System Disentanglement

To avoid the problems described in the previous section, file systems need to be redesigned to avoid artificial coupling between logical entities and physical realization. In this section, we first discuss a key abstraction that enables such disentanglement: the file system cube. We then discuss three key principles that realizes disentanglement: no shared physical resources, no access dependencies, and no bundled transactions.

3.2.1 The Cube Abstraction

We propose a new file system abstraction, the *cube*, that enables applications to specify which files and directories are logically related. The file system can safely combine the performance and reliability properties of groups of files and their metadata that belong to the same cube; each cube is physically isolated from others and is thus completely independent at the file system level.

The cube abstraction is easy to use, with the following operations:

Create a cube: A cube can be created on demand. A default global cube is created when a new file system is created with the `mkfs` utility.

Set cube attributes: Applications can specify customized attributes for each cube. Supported attributes include: failure policy (e.g., read-only or crash), recovery policy (e.g., online or offline checking) and journaling mode (e.g., high or low consistency requirement).

Add files to a cube: Users can create or move files or directories into a cube. By default, files and directories inherit the cube of their parent directory.

Delete files from a cube: Files and directories can be removed from the cube via `unlink`, `rmdir`, and `rename`.

Remove a cube: Applications can delete a cube completely along with all files within it. The released disk space can then be used by other cubes.

The cube abstraction has a number of attractive properties. First, each cube is *isolated* from other cubes both logically and physically; at the file system level, each cube is independent for failure, recovery, and journaling. Second, the use of cubes can be *transparent* to applications; once a cube is created, applications can interact with the file system without modification. Third, cubes are *flexible*; cubes can be created and destroyed on demand, similar to working with directories. Fourth, cubes are *elastic* in storage space usage; unlike partitions, no storage over-provision or reservation is needed for a cube. Fifth, cubes can be *customized* for diverse requirements; for example, an important cube may be set with high consistency and immediate recovery attributes. Finally, cubes are *lightweight*; a cube does not require extensive memory or disk resources.

3.2.2 Disentangled Data Structures

To support the cube abstraction, key data structures within modern file systems must be disentangled. We discuss three principles of disentangled data structures: no shared physical resources, no access dependencies, and no shared transactions.

No Shared Physical Resources

For cubes to have independent performance and reliability, multiple cubes must not share the same physical resources within the file system (e.g., blocks on disk or pages in memory). Unfortunately, current file systems freely co-locate metadata from multiple files and directories into the same unit of physical storage.

In classic Ext-style file systems, storage space is divided into fixed-size *block groups*, in which each block group has its own metadata (i.e., a group descriptor, an inode bitmap, a block bitmap, and inode tables). Files and directories are allocated to particular block groups using heuristics to improve locality and to balance space. Thus, even though the disk is partitioned into multiple block groups, any block group and its corresponding metadata blocks can be shared across any set of files. For example, in Ext3, Ext4 and Btrfs, a single block is likely to contain inodes for multiple unrelated files and directories; if I/O fails for one inode block, then all the files with inodes in that block will not be accessible. As another example, to save space, Ext3 and Ext4 store many group descriptors in one disk block, even though these group descriptors describe unrelated block groups.

This false sharing percolates from on-disk blocks up to in-memory data structures at runtime. Shared resources directly lead to global failures, since a single corruption or I/O failure affects multiple logically-independent files. Therefore, to isolate cubes, a disentangled file system must partition its various data structures into smaller independent ones.

No Access Dependency

To support independent cubes, a disentangled file system must also ensure that one cube does not contain references to or need to access other cubes. Current file systems often contain a number of data structures that violate this principle. Specifically, *linked lists* and *trees* encode dependencies across entries by design. For example, Ext3 and Ext4 maintain an orphan inode list in the super block to record files to be deleted; Btrfs and XFS use Btrees extensively for high performance. Unfortunately, one failed entry in a list or tree affects all entries following or below it.

The most egregious example of access dependencies in file systems is commonly found in the implementation of the *hierarchical directory structure*. In Ext-based systems, the path for reaching a particular file in the directory structure is implicitly encoded in the physical layout of those files and directories on disk. Thus, to read a file, all directories up to the root must be accessible. If a single directory along this path is corrupted or unavailable, a file will be inaccessible.

No Bundled Transactions

The final data structure and mechanism that must be disentangled to provide isolation to cubes are transactions. To guarantee the consistency of metadata and data, existing file systems typically use journaling (e.g., Ext3 and Ext4) or copy-on-write (e.g., Btrfs and ZFS) with transactions. A transaction contains temporal updates from many files within a short period of time (e.g., 5s in Ext3 and Ext4). A shared transaction batches multiple updates and is flushed to disk as a single atomic unit in which either all or none of the updates are successful.

Unfortunately, transaction batching artificially tangles together logically independent operations in several ways. First, if the shared transaction fails, updates to all of the files in this transaction will fail as well. Second, in physical journaling file systems (*e.g.,* , Ext3), a `fsync()` call on one file will force data from other files in the same transaction to be flushed as well; this falsely couples performance across independent files and workloads.

3.3 The Ice File System

We now present IceFS, a file system that provides cubes as its basic new abstraction. We begin by discussing the important internal mechanisms of IceFS, including novel directory independence and transaction splitting mechanisms. Disentangling data structures and mechanisms enables the file system to provide behaviors that are localized and specialized to each container. We describe three major benefits of a disentangled file system (localized reactions to failures, localized recovery, and specialized journaling performance) and how such benefits are realized in IceFS.

3.3.1 IceFS

We implement a prototype of a disentangled file system, IceFS, as a set of modifications to Ext3, a standard and mature journaling file system in many Linux distributions. We disentangle Ext3 as a proof of concept; we believe our general design can be applied to other file systems as well.

Realizing the Cube Abstraction

The cube abstraction does not require radical changes to the existing POSIX interface. In IceFS, a cube is implemented as a special directory; all files and sub-directories within the cube directory belong to the same cube.

To create a cube, users pass a cube flag when they call `mkdir()`. IceFS creates the directory and records that this directory is a cube. When creating a cube, customized cube attributes are also supported, such as a specific journaling mode for different cubes. To delete a cube, only `rmdir()` is needed.

IceFS provides a simple mechanism for filesystem isolation so that users have the freedom to define their own policies. For example, an NFS server can automatically create a cube for the home directory of each user, while a VM server can isolate each virtual machine in its own cube. An application can use a cube as a data container, which isolates its own data from other applications.

Physical Resource Isolation

A straightforward approach for supporting cubes is to leverage the existing concept of a block group in many existing file systems. To disentangle shared resources and isolate different cubes, IceFS dictates that a block group can be assigned to only one cube at any time, as shown in Figure 3.7; in this way, all metadata associated with a block group (e.g., bitmaps and inode tables) belongs to only one cube. A block group freed by one

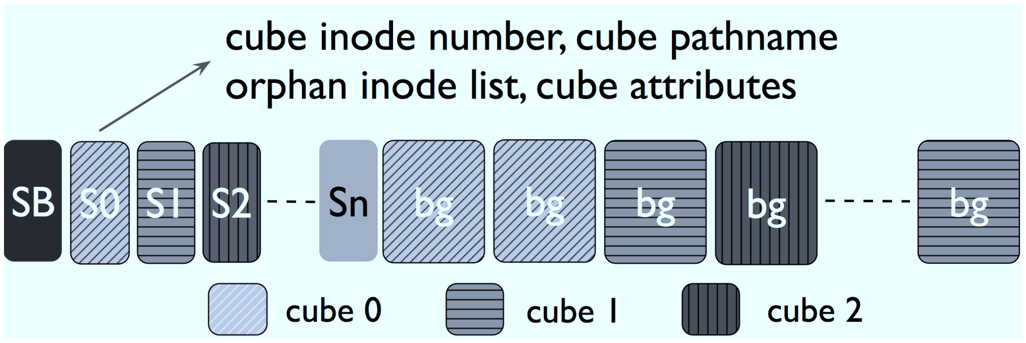

cube inode number, cube pathname
orphan inode list, cube attributes
SB
S0
S1
S2
Sn
bg
bg
bg
bg
bg
cube 0
cube 1
cube 2

Figure 3.7: **Disk Layout of IceFS.** *This figure shows the disk layout of IceFS. Each cube has a sub-super block, stored after the global super block. Each cube also has its own separated block groups. Si: sub-super block for cube i; bg: a block group.*

cube can be allocated to any other cube. Compared with partitions, the allocation unit of cubes is only one block group, much smaller than the size of a typical multiple GB partition.

When allocating a new data block or an inode for a cube, the target block group is chosen to be either an empty block group or a block group already belonging to the cube. Enforcing the requirement that a block group is devoted to a single cube requires changing the file and directory allocation algorithms such that they are cube-aware without losing locality.

To identify the cube of a block group, IceFS stores a cube ID in the group descriptor. To get the cube ID for a file, IceFS simply leverages the static mapping of inode numbers to block groups as in the base Ext3 file system; after mapping the inode of the file to the block group, IceFS obtains the cube ID from the corresponding group descriptor. Since all group descriptors are loaded into memory during the mount process, no extra I/O is required to determine the cube of a file.

IceFS trades disk and memory space for the independence of cubes. To save memory and reduce disk I/O, Ext3 typically places multiple contiguous group descriptors into a single disk block. IceFS modifies this policy so that only group descriptors from the same cube can be placed

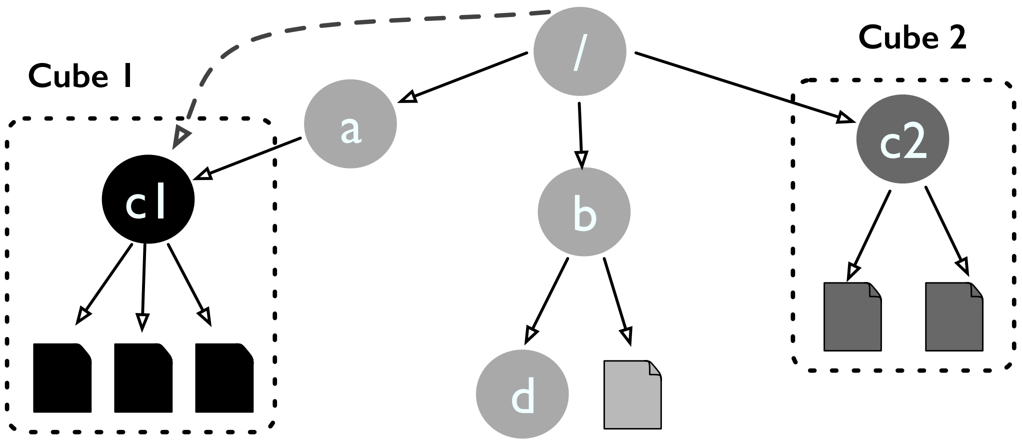


Figure 3.8: **An Example of Cubes and Directory Indirection.** *This figure shows how the cubes are organized in a directory tree, and how the directory indirection for a cube is achieved.*

in the same block. This approach is similar to the meta-group of Ext4 for combining several block groups into a larger block group [139].

Access Independence

To disentangle cubes, no cube can reference another cube. Thus, IceFS partitions each global list that Ext3 maintains into per-cube lists. Specifically, Ext3 stores the head of the global orphan inode list in the super block. To isolate this shared list and the shared super block, IceFS uses one *sub-super* block for each cube; these sub-super blocks are stored on disk after the super block and each references its own orphan inode list as shown in Figure 3.7. IceFS preallocates a fixed number of sub-super blocks following the super block. The maximum number of sub-super blocks is configurable at `mkfs` time. These sub-super blocks can be replicated within the disk similar to the super block to avoid catastrophic damage of sub-super blocks.

In contrast to a traditional file system, if IceFS detects a reference from one cube to a block in another cube, then it knows that reference is incorrect. For example, no data block should be located in a different cube than the inode of the file to which it belongs.

To disentangle the file namespace from its physical representation on disk and to remove the naming dependencies across cubes, IceFS uses *directory indirection*, as shown in Figure 3.8. With directory indirection, each cube records its top directory; when the file system performs a pathname lookup, it first finds a longest prefix match of the pathname among the cubes' top directory paths; if it does, then only the remaining pathname within the cube is traversed. For example, if the user wishes to access `/home/bob/research/paper.tex` and `/home/bob/research/` designates the top of a cube, then IceFS will skip directly to parsing `paper.tex` within the cube. As a result, any failure outside of this cube, or to the `home` or `bob` directories, will not affect accessing `paper.tex`.

In IceFS, the path lookup performed by the VFS layer is modified to provide directory indirection for cubes. The inode number and the pathname of the top directory of a cube are stored in its sub-super block; when the file system is mounted, IceFS pins in memory this information along with the cube's dentry, inode, and pathname. Later, when a pathname lookup is performed, VFS passes the pathname to IceFS so that IceFS can check whether the pathname is within any cube. If there is no match, then VFS performs the lookup as usual; otherwise, VFS uses the matched cube's dentry as a shortcut to resolve the remaining part of the pathname.

Transaction Splitting

To disentangle transactions belonging to different cubes, we introduce *transaction splitting*, as shown in Figure 3.9. With transaction splitting, each cube has its own running transaction to buffer writes. Transactions from different cubes are committed to disk in parallel without any waiting or dependencies across cubes. With this approach, any failure along the

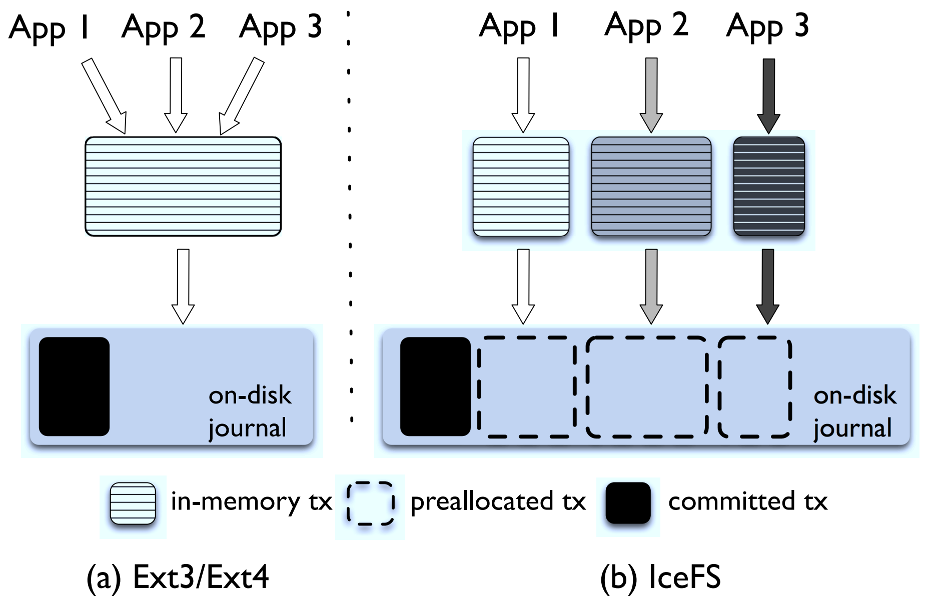


Figure 3.9: **Transaction Split Architecture.** *This figure shows the different transaction architectures in Ext3/4 and IceFS. In IceFS, different colors represent different cubes' transactions.*

transaction I/O path can be attributed to the source cube, and the related recovery action can be triggered only for the faulty cube, while other healthy cubes still function normally.

IceFS leverages the existing generic journaling module of Ext3, JBD. To provide specialized journaling for different cubes, each cube has a virtual journal managed by JBD with a potentially customized journaling mode. When IceFS starts an atomic operation for a file or directory, it passes the related cube ID to JBD. Since each cube has a separate virtual journal, a commit of a running transaction will only be triggered by its own `fsync()` or timeout without any entanglement with other cubes.

Different virtual journals share the physical journal space on disk. At the beginning of a commit, IceFS will first reserve journal space for the transaction of the cube; a separate committing thread will flush the transaction to the journal. Since transactions from different cubes write to different places on the journal, IceFS can perform multiple commits in parallel. Note that, the original JBD uses a shared lock to synchronize various structures in the journaling layer, while IceFS needs only a single shared lock to allocate transaction space; the rest of the transaction operations can now be performed independently without limiting concurrency.

3.3.2 Localized Reactions to Failures

As shown in Section 3.1, current file systems handle serious errors by crashing the whole system or marking the entire file system as read-only. Once a disentangled file system is partitioned into multiple independent cubes, the failure of one cube can be detected and controlled with a more precise boundary. Therefore, failure isolation can be achieved by transforming a global failure to a local per-cube failure.

Fault Detection

Our goal is to provide a new fault-handling primitive, which can localize global failure behaviors to an isolated cube. This primitive is largely orthogonal to the issue of detecting the original faults. We currently leverage existing detection mechanism within file systems to identify various faults.

For example, file systems tend to detect metadata corruption at the I/O boundary by using their own semantics to verify the correctness of file system structures; file systems check error conditions when interacting with other subsystems (e.g., failed disk read/writes or memory allocations); file systems also check assertions and invariants that might fail due to concurrency problems.

IceFS modifies the existing detection techniques to make them cube-aware. For example, Ext3 calls `ext3_error()` to mark the file system as read-only on an inode bitmap read I/O fault. IceFS instruments the fault-handling and crash-triggering functions (e.g., `BUG_ON()`) to include the ID of the responsible cube; pinpointing the faulty cube is straightforward as all metadata is isolated. Thus, IceFS has cube-aware fault detectors.

One can argue that the incentive for detecting problems in current file systems is relatively low because many of the existing recovery techniques (e.g., calling `panic()`) are highly pessimistic and intrusive, making the entire system unusable. A disentangled file system can contain faults within a single cube and thus provides incentive to add more checks.

Localized Read-Only

As a recovery technique, IceFS enables a single cube to be read-only. In IceFS, only files within a faulty cube are made read-only, and other cubes remain available for both reads and writes, improving the overall availability of the file system. IceFS performs this per-cube reaction by adapting the existing mechanisms within Ext3 for making all files read-only.

To guarantee read-only for all files in Ext3, two steps are needed. First, the transaction engine is immediately shut down. Existing running transactions are aborted, and attempting to create a new transaction or join an existing transaction results in an error code. Second, the generic VFS super block is marked as read-only; as a result, future writes are rejected.

To localize read-only failures, a disentangled file system can execute two similar steps. First, with the transaction split framework, IceFS individually aborts the transaction for a single cube; thus, no more transactions are allowed for the faulty cube. Second, the faulty cube alone is marked as read-only, instead of the whole file system. When any operation is performed, IceFS now checks this per-cube state whenever it would usually check the super block read-only state. As a result, any write to a read-only cube receives an error code, as desired.

Localized Crashes

Similarly, IceFS is able to localize a crash for a failed cube, such that the crash does not impact the entire operating system or operations of other cubes. Again, IceFS leverages the existing mechanisms in the Linux kernel for dealing with crashes caused by `panic()`, `BUG()`, and `BUG_ON()`. IceFS performs the following steps:

- *Fail the crash-triggering thread:* When a thread fires an assertion failure, IceFS identifies the cube being accessed and marks that cube as crashed. The failed thread is directed to the failure path, during which the failed thread will free its allocated resources (e.g., locks and memory). IceFS adds this error path if it does not exist in the original code.

- *Prevent new threads:* A crashed cube should reject any new file-system request. IceFS identifies whether a request is related to a crashed cube as early as possible and return appropriate error codes to terminate the related system call. Preventing new accesses consists of blocking the entry point functions and the directory indirection functions. For example, the state of a cube is checked at all the callbacks provided by Ext3, such as super block operations (e.g., `ext3_write_inode()`), directory operations (e.g., `ext3_readdir()`), and file operations (e.g., `ext3_sync_file()`). One complication is that many system calls use either a pathname or a file descriptor as an input; VFS usually translates the pathname or file descriptor into an inode. However, directory indirection in IceFS can be used to quickly prevent a new thread from entering the crashed cube. When VFS conducts the directory indirection, IceFS will see that the pathname belongs to a crashed cube and VFS will return an appropriate error to the application.

- *Evacuate running threads:* Besides the crash-triggering thread, other threads may be accessing the same cube when the crash happens. IceFS waits for these threads to leave the crashed cube, so they will free their kernel and file-system resources. Since the cube is marked as crashed, these running threads cannot read or write to the cube and will exit with error codes. To track the presence of on-going threads, IceFS maintains a simple counter for each cube; the counter is incremented when a system call is entered and decremented when a system call returns, similar to the system-call gate [142].

- *Clean up the cube:* Once all the running threads are evacuated, IceFS cleans up the memory states of the crashed cube similar to the unmount process. Specifically, dirty file pages and metadata buffers belonging to the crashed are dropped without being flushed to disk; clean states, such as cached dentries and inodes, are freed.

3.3.3 Localized Recovery

As shown in Section 3.1, current file system checkers do not scale well to large file systems. With the cube abstraction, IceFS can solve this problem by enabling per-cube checking. Since each cube represents an independent fault domain with its own isolated metadata and no references to other cubes, a cube can be viewed as a basic checking unit instead of the whole file system.

Offline Checking

In a traditional file-system checker, the file system must be offline to avoid conflicts with a running workload. For simplicity, we first describe a per-cube offline checker.

Ext3 uses the utility e2fsck to check the file system in five phases [143]. IceFS changes e2fsck to make it cube-aware; we call the resulting checker ice-fsck. The main idea is that IceFS supports partial checking of a file system by examining only faulty cubes. In IceFS, when a corruption is detected at run time, the error identifying the faulty cube is recorded in fixed locations on disk. Thus, when ice-fsck is run, erroneous cubes can be easily identified, checked, and repaired, while ignoring the rest of the file system. Of course, ice-fsck can still perform a full file system check and repair, if desired.

Specifically, ice-fsck identifies faulty cubes and their corresponding block groups by reading the error codes recorded in the journal. Before loading the metadata from a block group, each of the five phases of ice-fsck first ensures that this block group belongs to a faulty cube. Because the metadata of a cube is guaranteed to be self-contained, metadata from other cubes not need to be checked. For example, because an inode in one cube cannot point to an indirect block stored in another cube (or block group), ice-fsck can focus on a subset of the block groups. Similarly, checking the directory hierarchy in ice-fsck is simplified; while e2fsck must verify that every file can be connected back to the root directory, ice-fsck only needs to verify that each file in a cube can be reached from the entry points of the cube.

Online Checking

Offline checking of a file system implies that the data will be unavailable to important workloads, which is not acceptable for many applications. A disentangled file system enables on-line checking of faulty cubes while other healthy cubes remain available to foreground traffic, which can greatly improve the availability of the whole service.

Online checking is challenging in existing file systems because metadata is shared loosely by multiple files; if a piece of metadata must be repaired, then all the related files should be frozen or repaired together. Coordinating concurrent updates between the checker and the file system is non-trivial. However, in a disentangled file system, the fine-grained isolation of cubes makes online checking feasible and efficient.

We note that online checking and repair is a powerful recovery mechanism compared to simply crashing or marking a cube read-only. Now, when a fault or corruption is identified at runtime with existing detection techniques, IceFS can unmount the cube so it is no longer visible, and then launch ice-fsck on the corrupted cube while the rest of the file system functions normally. In our implementation, the on-line ice-fsck is a user-space program that is woken up by IceFS informed of the ID of the faulty cubes.

3.3.4 Specialized Journaling

As described previously, disentangling journal transactions for different cubes enables write operations in different cubes to proceed without impacting others. Disentangling journal transactions (in conjunction with disentangling all other metadata) also enables different cubes to have different consistency guarantees.

Journaling protects files in case of system crashes, providing certain consistency guarantees, such as metadata or data consistency. Modern journaling file systems support different modes; for example, Ext3 and Ext4 support, from lowest to highest consistency: *writeback*, *ordered*, and *data*. However, the journaling mode is enforced for the entire file system, even though users and applications may desire differentiated consistency guarantees for their data. Transaction splitting enables a specialized journaling protocol to be provided for each cube.

A disentangled file system is free to choose customized consistency modes for each cube, since there are no dependencies across them; even if the metadata of one cube is updated inconsistently and a crash occurs, other cubes will not be affected. IceFS supports five consistency modes, from lowest to highest: *no fsync*, *no journal*, *writeback journal*, *ordered journal* and *data journal*. In general, there is an incentive to choose modes with lower consistency to achieve higher performance, and an incentive to choose modes with higher consistency to protect data in the presence of system crashes.

For example, a cube that stores important configuration files for the system may use data journaling to ensure both data and metadata consistency. Another cube with temporary files may be configured to use *no journal* (*i.e.*, , behave similarly to Ext2) to achieve the highest performance, given that applications can recreate the files if a crash occurs. Going one step further, if users do not care about the durability of data of a particular application, the *no fsync* mode can be used to ignore `fsync()` calls from applications. Thus, IceFS gives more control to applications and users, allowing them to adopt a customized consistency mode for their data.

IceFS uses the existing implementations within JBD to achieve the three journaling modes of writeback, ordered, and data. Specifically, when there is an update for a cube, IceFS uses the specified journaling mode to handle the update. For *no journal*, IceFS behaves like a non-journaled file system, such as Ext2, and does not use the JBD layer at all. Finally, for *no fsync*, IceFS ignores `fsync()` system calls from applications and directly returns without flushing any related data or metadata.

3.3.5 Implementation Complexity

We added and modified around 6500 LOC to Ext3/JBD in Linux 3.5 for the data structures and journaling isolation, 970 LOC to VFS for directory indirection and crash localization, and 740 LOC to e2fsprogs 1.42.8 for file system creation and checking. The most challenging part of the implementation was to isolate various data structures and transactions for cubes. Once we carefully isolated each cube (both on disk and in memory), the localized reactions to failures and recovery was straightforward.

Workload	**Ext3 (MB/s)**	**IceFS (MB/s)**	**Difference**
Sequential write	98.9	98.8	0%
Sequential read	107.5	107.8	+0.3%
Random write	2.1	2.1	0%
Random read	0.7	0.7	0%
Fileserver	73.9	69.8	-5.5%
Varmail	2.2	2.3	+4.5%
Webserver	151.0	150.4	-0.4%

Table 3.10: **Micro and Macro Benchmarks on Ext3 and IceFS.** *This table compares the throughput of micro and macro benchmarks on Ext3 and IceFS. Sequential write/read are writing/reading a 1GB file in 4KB requests. Random write/read are writing/reading 128MB of a 1GB file in 4KB requests. Fileserver has 50 threads performing creates, deletes, appends, whole-file writes, and whole-file reads. Varmail emulates a multi-threaded mail server. Webserver is a multi-threaded read-intensive workload.*

3.4 Evaluation of IceFS

We present evaluation results for IceFS. We first evaluate the basic performance of IceFS through a series of micro and macro benchmarks. Then, we show that IceFS is able to localize many failures that were previously global. All the experiments are performed on machines with an Intel(R) Core(TM) i5-2500K CPU (3.30 GHz), 16GB memory, and a 1TB Hitachi Deskstar 7K1000.B hard drive, unless otherwise specified.

3.4.1 Overall Performance

We assess the performance of IceFS with micro and macro benchmarks. First, we mount both file systems in the default ordered journaling mode, and run several micro benchmarks (sequential read/write and random read/write) and three macro workloads from Filebench (Fileserver, Varmail, and Webserver). For IceFS, each workload uses one cube to store its data. Table 3.10 shows the throughput of all the benchmarks on Ext3 and IceFS. From the table, one can see that IceFS performs similarly to Ext3, indicating that our disentanglement techniques incur little overhead.

IceFS maintains extra structures for each cube on disk and in memory. For each cube IceFS creates, one sub-super block (4KB) is allocated on disk. Similar to the original super block, sub-super blocks are also cached in memory. In addition, each cube has its own journaling structures (278 B) and cached running states (104 B) in memory. In total, for each cube, its disk overhead is 4 KB and memory overhead is less than 4.5 KB.

3.4.2 Localize Failures

We show that IceFS converts many global failures into local, per-cube failures. We inject faults into core file-system structures where existing checks are capable of detecting the problem. These faults are selected from Table 3.2 and they cover all different fault types, including memory allocation failures, metadata corruption, I/O failures, `NULL` pointers, and unexpected states. To compare the behaviors, the faults are injected in the same locations for both Ext3 and IceFS. Overall, we injected nearly 200 faults. With Ext3, in every case, the faults led to global failures of some kind (such as an OS panic or crash). IceFS, in contrast, was able to localize the triggered faults in every case.

However, we found that there are also a small number of failures during the mount process, which are impossible to isolate. For example, if a memory allocation failure happens when initializing the super block during the mount process, then the mount process will exit with an error code. In such cases, both Ext3 and IceFS will not be able to handle it because the fault happens before the file system starts running.

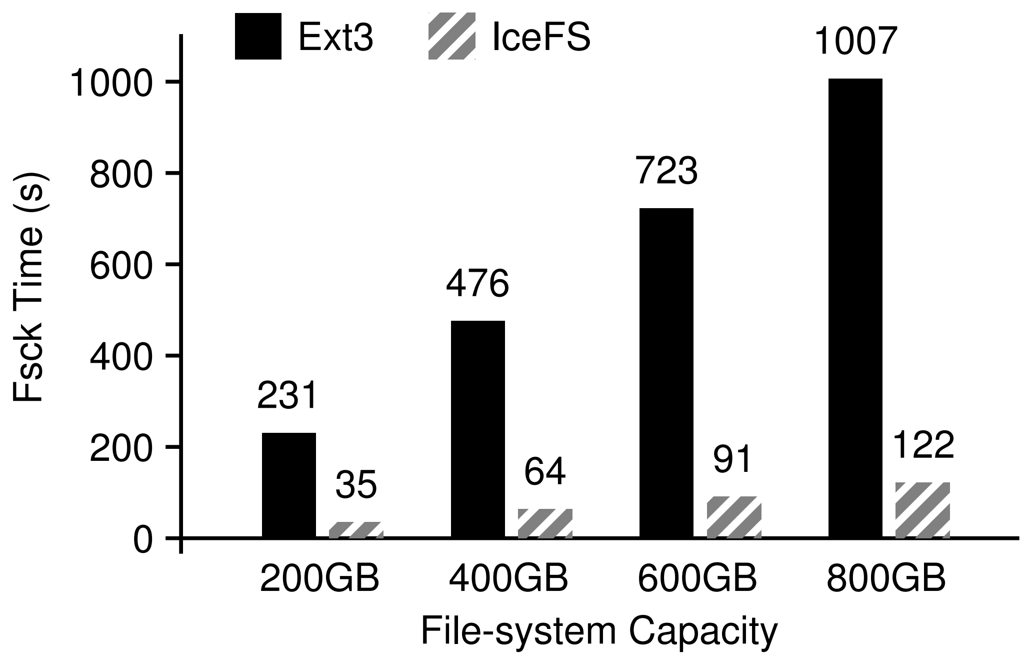


Figure 3.11: **Performance of IceFS Offline Fsck.** *This figure compares the running time of offline fsck on ext3 and on IceFS with different file-system size.*

3.4.3 Fast Recovery

With localized failure detection, IceFS is able to perform offline fsck only on the faulted cube. To measure fsck performance on IceFS, we first create file system images in the same way as described in Figure 3.3, except that we make 20 cubes instead of directories. We then fail one cube randomly and measure the fsck time. Figure 3.11 compares the offline fsck time between IceFS and Ext3. The fsck time of IceFS increases as the capacity of the cube grows along with the file system size; in all cases, fsck on IceFS takes much less time than Ext3 because it only needs to check the consistency of one cube.

3.4.4 Specialized Journaling

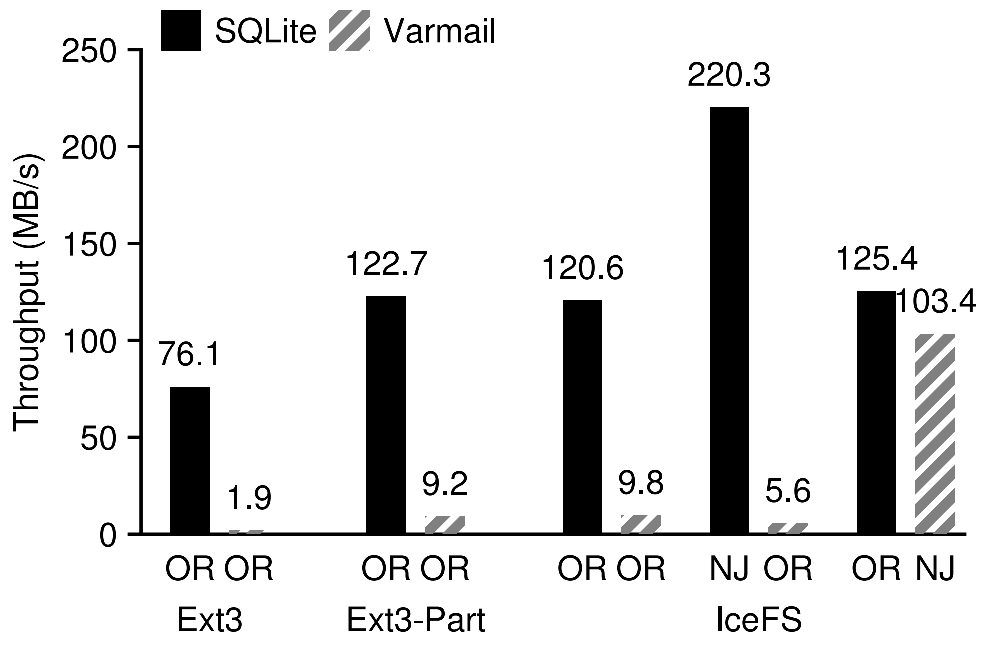


Figure 3.12: **Running Two Applications on IceFS with Different Journaling Mode.** *This figure compares the performance of simultaneously running SQLite and Varmail on Ext3, partitions and IceFS. In Ext3, both applications run in ordered mode (OR). In Ext3-Part, two separated Ext3 run in ordered mode (OR) on two partitions. In IceFS, two separate cubes used different journaling modes: ordered mode (OR) and no-journal mode (NJ).*

We now demonstrate that a disentangled journal enables different consistency modes to be used by different applications on a shared file system. For these experiments, we use a Samsung 840 EVO SSD (500GB) as the underlying storage device. Figure 3.12 shows the throughput of running two applications, SQLite and Varmail, in Ext3, two separated Ext3 on partitions (Ext3-Part) and IceFS. When running with Ext3 and ordered journaling (two leftmost bars), both applications achieve low performance because they share the same journaling layer and both workloads affect the other. When the applications run with IceFS on two different cubes, their performance increases significantly since `fsync()` calls to one cube do not force out dirty data to the other cube. Compared with Ext3-Part, we can find that IceFS achieves great isolation for cubes at the file system level, similar to running two different file systems on partitions.

We also demonstrate that different applications can benefit from different journaling modes; in particular, if an application can recover from inconsistent data after a crash, the no-journal mode can be used for much higher performance while other applications can continue to safely use ordered mode. As shown in Figure 3.12, when either SQLite or Varmail is run on a cube with no journaling, that application receives significantly better throughput than it did in ordered mode; at the same time, the competing application using ordered mode continues to perform better than with Ext3. We note that the ordered competing application may perform slightly worse than it did when both applications used ordered mode due to increased contention for resources outside of the file system (i.e., the I/O queue in the block layer for the SSD); this demonstrates that isolation must be provided at all layers for a complete solution. In summary, specialized journaling modes can provide great flexibility for applications to make trade-offs between their performance and consistency requirements.

3.4.5 Limitations

Although IceFS has many advantages as shown in previous sections, it may perform worse than Ext3 in certain extreme cases. The main limitation of our implementation is that IceFS uses a separate journal commit thread for every cube. The thread issues a device cache `flush` command at the end of every transaction commit to make sure the cached data is persistent on device; this cache flush is usually expensive [53]. Therefore, if many active cubes perform journal commits at the same time, the performance of IceFS may be worse than Ext3 that only uses one journal commit thread for all

Device	**Ext3 (MB/s)**	**Ext3-Part (MB/s)**	**IceFS (MB/s)**
SSD	40.8	30.6	35.4
Disk	2.8	2.6	2.7

Table 3.13: **Limitation of IceFS On Cache Flush.** *This table compares the aggregated throughput of four Varmail instances on Ext3 and IceFS. Each Varmail instance runs in a directory of Ext3, an Ext3 partition (Ext3-Part), or a cube of IceFS. We run the same experiment on both a SSD and hard disk.*

updates. The same problem exists in separated file systems on partitions.

To show this effect, we choose Varmail as our testing workload. Varmail utilizes multiple threads; each of these threads repeatedly issues small writes and calls `fsync()` after each write. We run multiple instances of Varmail in different directories, partitions or cubes to generate a large number of transaction commits, stressing the file system.

Table 3.13 shows the performance of running four Varmail instances on our quad-core machine. When running on an SSD, IceFS performs worse than Ext3, but a little better than Ext3 partitions (Ext3-Part). When running on a hard drive, all three setups perform similarly. The reason is that the cache flush time accounts for a large percentage of the total I/O time on an SSD, while the seeking time dominates the total I/O time on a hard disk. Since IceFS and Ext3-Part issue more cache flushes than Ext3, the performance penalty is amplified on the SSD.

Note that this style of workload is an extreme case for both IceFS and partitions. However, compared with separated file systems on partitions, IceFS is still a single file system that can utilize all the related semantic information of cubes for further optimization. For example, IceFS can pass per-cube hints to the block layer, which can optimize the cache flush cost and provide other performance isolation for cubes.

3.4.6 Usage Scenarios

We demonstrate that IceFS improves overall system behavior in the two motivational scenarios initially introduced in Section 3.1.3: virtualized environments and distributed file systems.

Virtual Machines

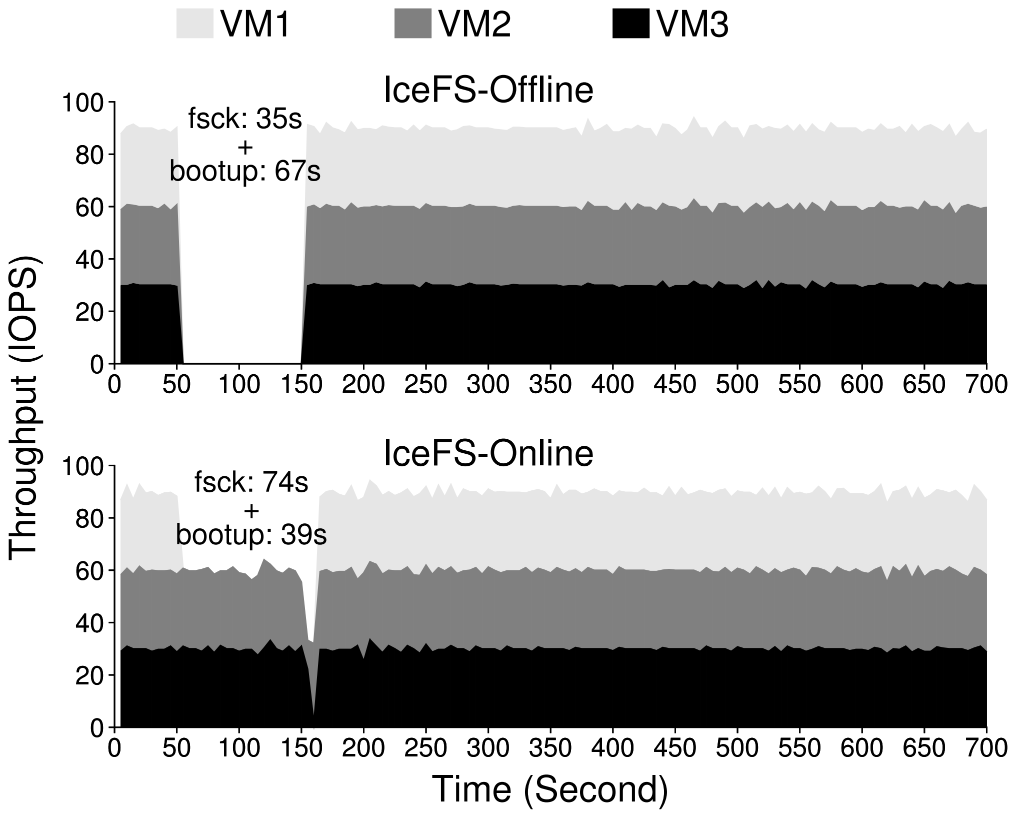


Figure 3.14: **Failure Handling for Virtual Machines.** *This figure shows how IceFS handles failures in a shared file system which supports multiple virtual machines.*

To show that IceFS enables virtualized environments to isolate failures within a particular VM, we configure each VM to use a separate cube in

IceFS. Each cube stores a 20GB virtual disk image, and the file system contains 10 such cubes for 10 VMs. Then, we inject a fault to one VM image that causes the host file system to be read-only after 50 seconds.

Figure 3.14 shows that IceFS greatly improves the availability of the VMs compared to that in Figure 3.5 using Ext3. The top graph illustrates IceFS with offline recovery. Here, only one cube is read-only and crashes; the other two VMs are shut down properly so the offline cube-aware check can be performed. The offline check of the single faulty cube requires only 35 seconds and booting the three VMs takes about 67 seconds; thus, after only 150 seconds, the three virtual machines are running normally again.

The bottom graph illustrates IceFS with online recovery. In this case, after the fault occurs in VM1 (at roughly 50 seconds) and VM1 crashes, VM2 and VM3 are able to continue. At this point, the online fsck of IceFS starts to recover the disk image file of VM1 in the host file system. Since fsck competes for disk bandwidth with the two running VMs, checking takes longer (about 74 seconds). Booting the single failed VM requires only 39 seconds, but the disk activity that arises as a result of booting competes with the I/O requests of VM2 and VM3, so the throughput of VM2 and VM3 drops for that short time period. In summary, these two experiments demonstrate that IceFS can isolate file system failures in a virtualized environment and significantly reduce system recovery time.

Distributed File System

We illustrate the benefits of using IceFS to provide flexible fault isolation in HDFS. Obtaining fault isolation in HDFS is challenging, especially in multi-tenant settings, primarily because HDFS servers are not aware of the data they store, as shown in Section 3.1.3. IceFS provides a natural solution for this problem. We use separate cubes to store different applications' data on HDFS servers. Each cube isolates the data from one application to another; thus, a cube failure will not affect multiple applications. In

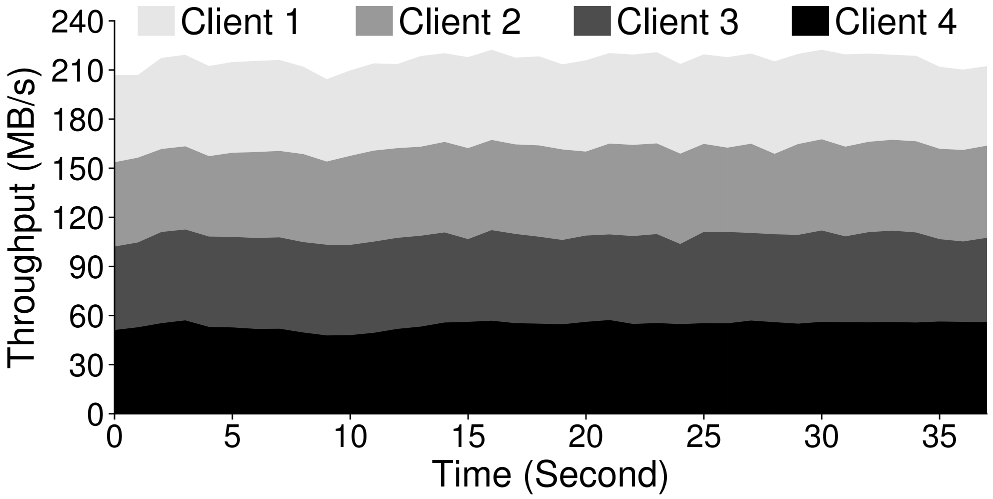


Figure 3.15: **Impact of Cube Failures in HDFS.** *This figure shows the throughput of 4 different clients when a cube failure happens at time 10 second. Impact of the failure to the clients' throughput is negligible.*

this manner, IceFS provides end-to-end isolation for applications in HDFS. We added 161 lines to storage node code to make HDFS IceFS-compatible and aware of application data. We do not change any recovery code of HDFS. Instead, IceFS turns global failures (*e.g.*, , kernel panic) into partial failures (*i.e.*, , cube failure) and leverages HDFS recovery code to handle them. This facilitates and simplifies our implementation.

Figure 3.15 shows the benefits of IceFS-enabled application-level fault isolation. Here, four clients concurrently access different files stored in HDFS when a cube that stores data for Client 2 fails and becomes inaccessible. Other clients are completely isolated from the cube failure. Furthermore, the failure negligibly impacts the throughput of the client as it does not manifest as machine failure. Instead, it results in a soft error to HDFS, which then immediately isolates the faulty cube and returns an

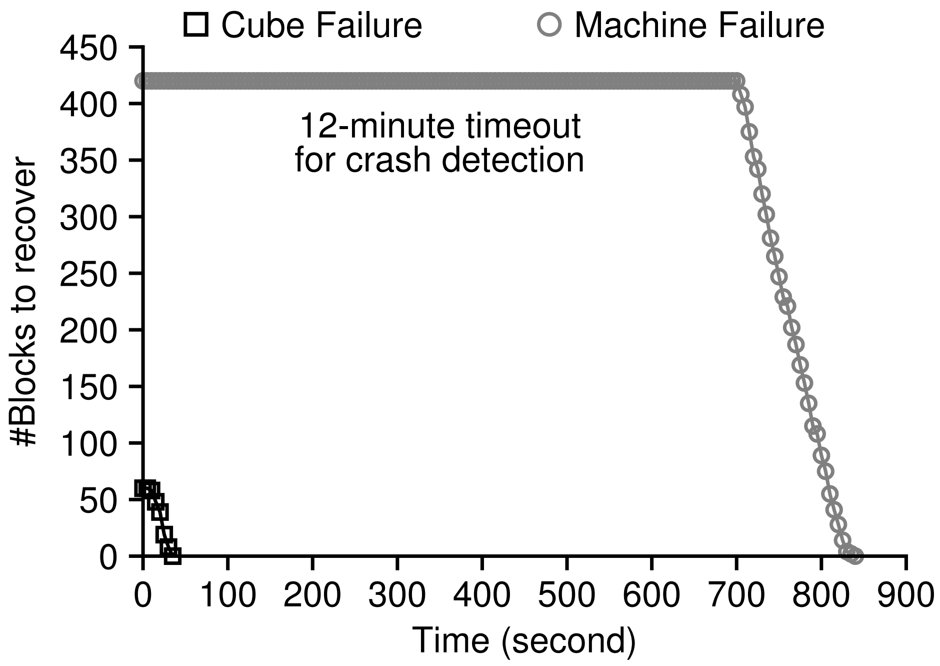


Figure 3.16: **Data Block Recovery in HDFS.** *The figure shows the number of lost blocks to be regenerated over time in two failure scenarios: cube and whole machine failure. Cube failure results into less blocks to recover in less time.*

error code the client. The client then quickly fails over to other healthy copies. The overall throughput is stable for the entire workload, as opposed to 60-second period of losing throughput as in the case of whole machine failure described in Section 3.1.3.

In addition to end-to-end isolation, IceFS provides scalable recovery as shown in Figure 3.16. In particular, IceFS helps reduce network traffic required to regenerate lost blocks, a major bandwidth consumption factor in large clusters [170]. When a cube fails, IceFS again returns an error code to the host server, which then immediately triggers a block scan to find out data blocks that are under-replicated and regenerates them. The number of blocks to recover is proportional to the cube size. Without IceFS, a kernel panic in local file system manifests as whole machine failure, causing a 12-minute timeout for crash detection and making the number of blocks lost and to be regenerated during recovery much larger. In summary, IceFS helps improve not only flexibility in fault isolation but also efficiency in failure recovery.

3.5 Conclusion

Despite isolation of many components in existing systems, the file system still lacks physical isolation. In this chapter, we have designed and implemented IceFS, a file system that achieves physical disentanglement through a new abstraction called cubes. IceFS uses cubes to group logically related files and directories, and ensures that data and metadata in each cube are isolated. There are no shared physical resources, no access dependencies, and no bundled transactions among cubes.

Through experiments, we demonstrate that IceFS is able to localize failures that were previously global, and recover quickly using localized online or offline fsck. IceFS can also provide specialized journaling to meet diverse application requirements for performance and consistency. Furthermore, we conduct two cases studies where IceFS is used to host multiple virtual machines and is deployed as the local file system for HDFS data nodes. IceFS achieves fault isolation and fast recovery in both scenarios, proving its usefulness in modern storage environments.

The source code for IceFS and its user-level utilities (mkfs and fsck) can be obtained at: `http://cs.wisc.edu/adsl/Software/icefs`. We hope that this sharing will inspire further disentanglement techniques in other file systems for better reliability and performance.

4 Key-Value Separation in WiscKey

From the previous chapter, we demonstrate that physical separation of file-system structures can provide reliability isolation and even performance improvement. Since persistent key-value stores play a critical role in a variety of modern data-intensive applications, we are curious to explore similar techniques for this new type of storage systems.

For write-intensive workloads, key-value stores based on Log Structured Merge Trees (LSM-trees) [157] have become the state of the art. To deliver high write performance, LSM-trees batch key-value pairs and write them out sequentially. Subsequently, to enable efficient lookups (for both individual keys as well as range queries), LSM-trees continuously read, sort, and write out key-value pairs in the background, thus maintaining keys and values in sorted order. As a result, the same data is read and written multiple times throughout its lifetime; this I/O amplification in typical LSM-trees can reach a factor of 50x or higher.

In this chapter, we present WiscKey, a persistent LSM-tree-based key-value store with a novel performance-oriented data layout that separates keys and values to minimize I/O amplification. The data layout and I/O patterns of WiscKey are highly optimized for SSD devices. We solve a number of reliability and performance challenges introduced by the new key-value separation architecture. We propose a parallel range query design to leverage the SSD's internal parallelism for better range query performance on unordered datasets. We also introduce an online and light-weight garbage collector for WiscKey to reclaim the invalid key-value pairs without affecting the foreground workloads much. We demonstrate the advantages of WiscKey with both microbenchmarks and YCSB workloads. Microbenchmark results show that WiscKey is 2.5× - 111× faster than LevelDB for loading a database and 1.6× - 14× faster for random lookups. WiscKey is faster than both LevelDB and RocksDB in all six YCSB workloads, and follows a trend similar to the microbenchmarks.

The rest of this chapter is organized as follows. We first describe the background and motivation in Section 4.1. Section 4.2 explains the design of WiscKey, and Section 4.3 analyzes its performance. Finally, we conclude in Section 4.4.

4.1 Background and Motivation

In this section, we first describe the concept of a Log-Structured Merge-tree (LSM-tree). Then, we explain the design of LevelDB, a popular key-value store based on LSM-tree technology. We investigate read and write amplification in LevelDB. Finally, we describe the characteristics of modern storage hardware.

4.1.1 Log-Structured Merge-Tree

An LSM-tree is a persistent structure that provides efficient indexing for a key-value store with a high rate of inserts and deletes [157]. It defers and batches data writes into large chunks to use the high sequential bandwidth of hard drives. Since random writes are nearly two orders of magnitude slower than sequential writes on hard drives, LSM-trees provide better write performance than traditional B-trees, which require random accesses.

An LSM-tree consists of a number of components of exponentially increasing sizes, C_0 to C_k, as shown in Figure 4.1. The C_0 component is a

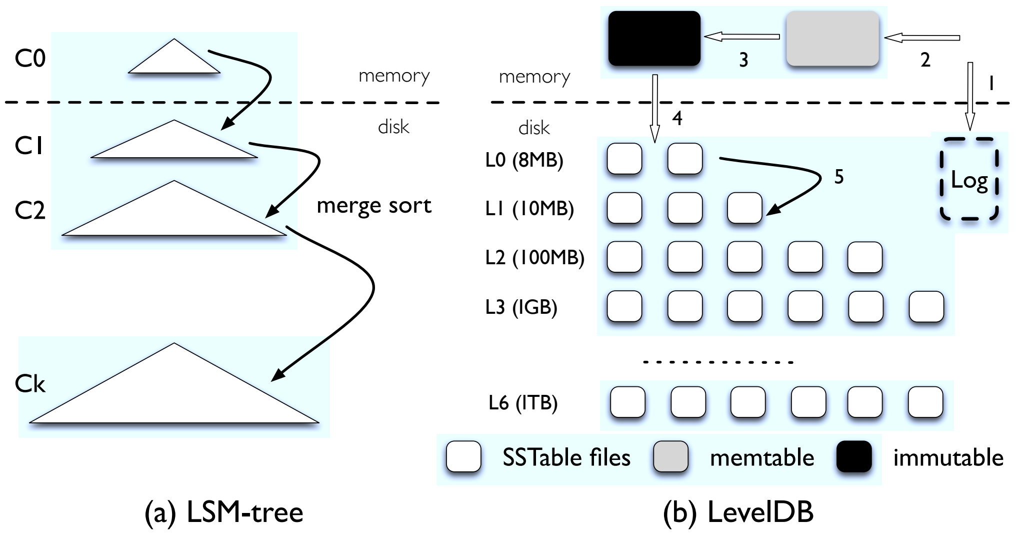


Figure 4.1: **LSM-tree and LevelDB Architecture.** *This figure shows the standard LSM-tree and LevelDB architecture. For LevelDB, inserting a key-value pair goes through many steps: (1) the log file; (2) the memtable; (3) the immutable memtable; (4) a SSTable in L0; (5) compacted to further levels.*

memory-resident update-in-place sorted tree, while the other components C_1 to C_k are disk-resident append-only B-trees.

During an insert in an LSM-tree, the inserted key-value pair is appended to an on-disk sequential log file, so as to enable recovery in case of a crash. Then, the key-value pair is added to the in-memory C_0, which is sorted by keys; C_0 allows efficient lookups and scans on recently inserted key-value pairs. Once C_0 reaches its size limit, it will be merged with the on-disk C_1 in an approach similar to merge sort; this process is known as *compaction*. The newly merged tree will be written to disk sequentially, replacing the old version of C_1. Compaction (i.e., merge sorting) also happens for on-disk components, when each C_i reaches its size limit. Note that compactions are only performed between adjacent levels (C_i and C_{i+1}), and they can be executed asynchronously in the background.

To serve a lookup operation, LSM-trees may need to search multiple components. Note that C_0 contains the most fresh data, followed by C_1, and so on. Therefore, to retrieve a key-value pair, the LSM-tree searches components starting from C_0 in a cascading fashion until it locates the desired data in the smallest component C_i. Compared with B-trees, LSM-trees may need multiple reads for a point lookup. Hence, LSM-trees are most useful when inserts are more common than lookups [157].

4.1.2 LevelDB

LevelDB is a widely used key-value store based on LSM-trees that is inspired by BigTable [48, 184]. LevelDB supports range queries, snapshots, and other features, which are useful in modern applications. In this section, we briefly describe the core design of LevelDB.

The overall architecture of LevelDB is shown in Figure 4.1. The main data structures in LevelDB are an on-disk log file, two in-memory sorted skiplists (*memtable* and *immutable memtable*), and seven levels (L_0 to L_6) of on-disk Sorted String Table (*SSTable*) files. LevelDB initially stores inserted key-value pairs in a log file and the in-memory *memtable*. Once the memtable is full, LevelDB switches to a new memtable and log file to handle further inserts from the user. In the background, the previous memtable is converted into an immutable memtable, and a compaction thread then flushes it to the disk, generating a new SSTable file (about 2 MB usually) at level 0 (L_0); the previous log file is discarded.

The size of all files in each level is limited, and increases by a factor of ten with the level number. For example, the size limit of all files at L_1 is 10 MB, while the limit of L_2 is 100 MB. To maintain the size limit, once the total size of a level L_i exceeds its limit, the compaction thread will choose one file from L_i, merge sort with all the overlapped files of L_{i+1}, and generate new L_{i+1} SSTable files. The compaction thread continues until all levels are within their size limits. Also, during compaction, LevelDB ensures that all files in a particular level, except L_0, do not overlap in their key-ranges; keys in files of L_0 can overlap with each other since they are directly flushed from memtable.

To serve a lookup operation, LevelDB searches the memtable first, immutable memtable next, and then files L_0 to L_6 in order. The number of file searches required to locate a random key is bounded by the maximum number of levels, since keys do not overlap between files within a single level, except in L_0. Since files in L_0 can contain overlapping keys, a lookup may search multiple files at L_0. To avoid a large lookup latency, LevelDB slows down the foreground write traffic if the number of files at L_0 is bigger than 8, in order to wait for the compaction thread to compact some files from L_0 to L_1.

4.1.3 Write and Read Amplification

Write and read amplification are major problems in LSM-trees such as LevelDB. Write (read) amplification is defined as the ratio between the amount of data written to (read from) the underlying storage device and the amount of data requested by the user. In this section, we analyze the write and read amplification in LevelDB.

To achieve mostly-sequential disk access, LevelDB writes more data than necessary (although still sequentially), i.e., LevelDB has high write amplification. Since the size limit of L_i is 10 times that of L_{i-1}, when merging a file from L_{i-1} to L_i during compaction, LevelDB may read up to 10 files from L_i in the worst case, and write back these files to L_i after sorting. Therefore, the write amplification of moving a file across two levels can be up to 10. For a large dataset, since any newly generated table file can eventually migrate from L_0 to L_6 through a series of compaction, write amplification can be over 50 (10 for each gap between L_1 to L_6).

Read amplification has been a major problem for LSM-trees due to trade-offs made in the design. There are two sources of read amplification in LevelDB. First, to lookup a key-value pair, LevelDB may need to check multiple levels. In the worst case, LevelDB needs to check eight files in

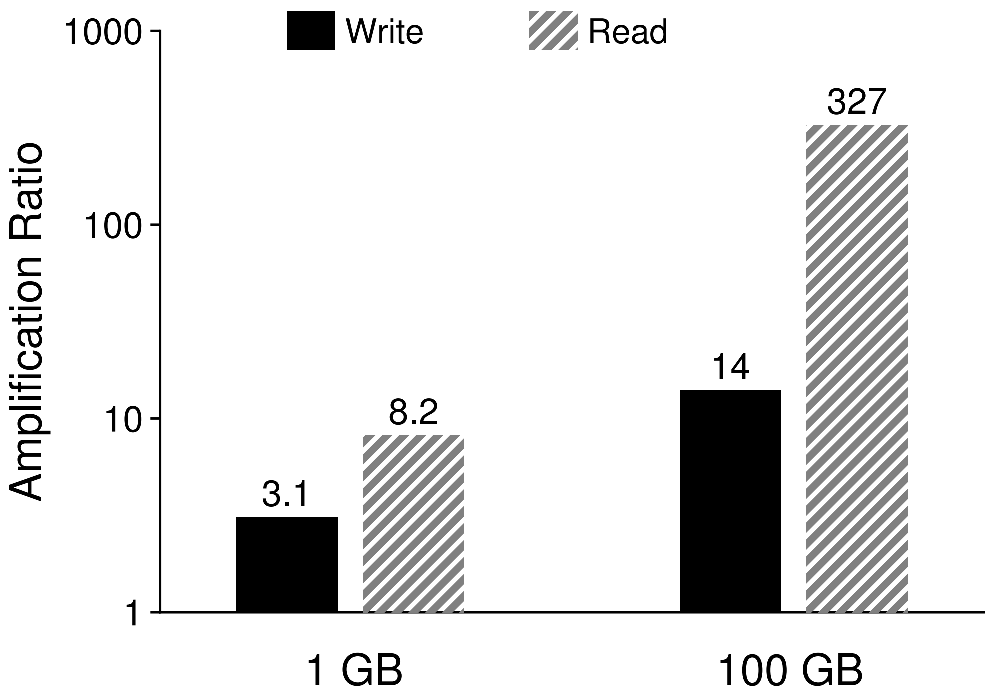


Figure 4.2: **Write and Read Amplification.** *This figure shows the write amplification and read amplification of LevelDB for two different database sizes, 1 GB and 100 GB. Key size is 16 B and value size is 1 KB.*

L_0, and one file for each of the remaining six levels: a total of 14 files. Second, to find a key-value pair within a SSTable file, LevelDB needs to read multiple metadata blocks within the file. Specifically, the amount of data actually read is given by `(index block + bloom-filter blocks + data block)`. For example, to lookup a 1 KB key-value pair, LevelDB needs to read a 16 KB index block, a 4 KB bloom-filter block, and a 4 KB data block; in total, 24 KB. Therefore, considering the 14 SSTable files in the worst case, the read amplification of LevelDB is 24 * 14 = 336. Smaller key-value pairs will lead to an even higher read amplification.

To measure the amount of amplification seen in practice with LevelDB, we perform the following experiment. We first load a database with 1 KB key-value pairs, and then lookup 100,000 entries from the database; we use two different database sizes for the initial load, and choose keys randomly from a uniform distribution. Figure 4.2 shows write amplification during the load phase and read amplification during the lookup phase. For a 1 GB database, write amplification is 3.1, while for a 100 GB database, write amplification increases to 14. Read amplification follows the same trend: 8.2 for the 1 GB database and 327 for the 100 GB database. The reason write amplification increases with database size is straightforward. With more data inserted into a database, the key-value pairs will more likely travel further along the levels; in other words, LevelDB will write data many times when compacting from low levels to high levels. However, write amplification does not reach the worst-case predicted previously, since the average number of files merged between levels is usually smaller than the worst case of 10. Read amplification also increases with the dataset size, since for a small database, all the index blocks and bloom filters in SSTable files can be cached in memory. However, for a large database, each lookup may touch a different SSTable file, paying the cost of reading index blocks and bloom filters each time.

It should be noted that the high write and read amplifications are a justified tradeoff for hard drives. As an example, for a given hard drive with a 10 ms seek latency and a 100 MB/s throughput, the approximate time required to access a random 1K of data is 10 ms, while that for the next sequential block is about 10 us – the ratio between random and sequential latency is 1000:1. Hence, compared to alternative data structures such as B-Trees that require random write accesses, a sequential-write-only scheme with write amplification less than 1000 will be faster on a hard-drive [157, 187]. On the other hand, the read amplification for LSM-trees is still comparable to B-Trees. For example, considering a B-Tree with a height of 5 and a block size of 4 KB, a random lookup for a 1 KB key-value pair would access 6 blocks, resulting in a read amplification of 24.

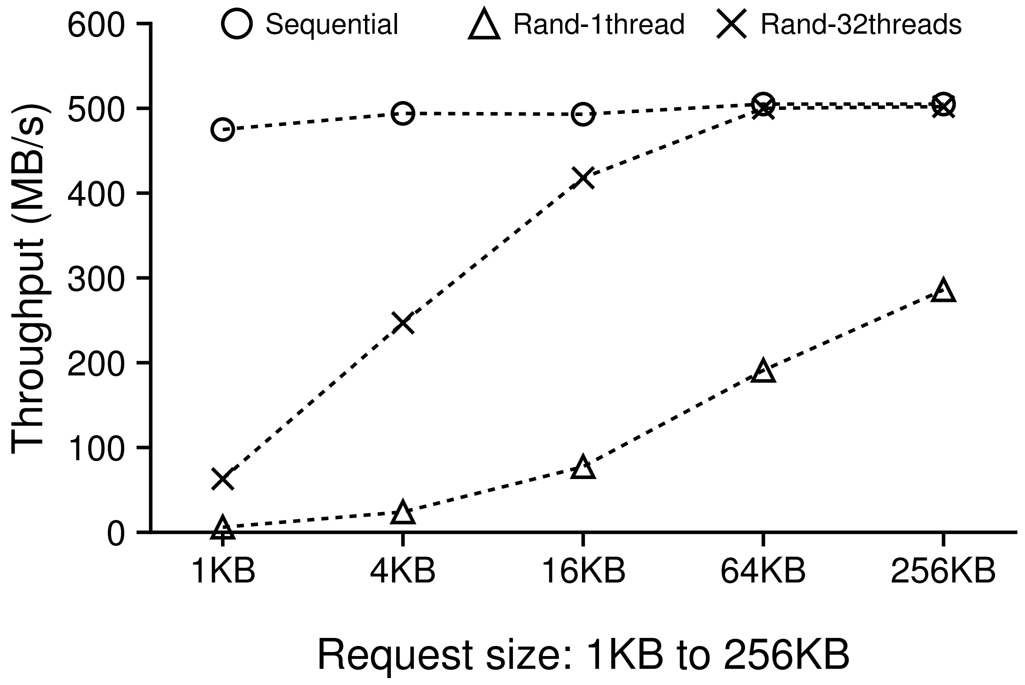


Figure 4.3: **Sequential and Random Reads on SSD.** *This figure shows the sequential and random read performance for various request sizes on a modern SSD device. All requests are issued to a 100 GB file on ext4.*

4.1.4 Fast Storage Hardware

Many modern servers adopt SSD devices to achieve high performance. Similar to hard drives, random writes are considered harmful also in SSDs [110, 116, 147] due to their unique erase-write cycle and expensive garbage collection. Although initial random-write performance for SSD devices is good, the performance can significantly drop after the reserved blocks are utilized. The LSM-tree characteristic of avoiding random writes is hence a natural fit for SSDs; many SSD-optimized key-value stores are based on LSM-trees [71, 190, 215, 224].

However, unlike hard-drives, the relative performance of random reads (compared to sequential reads) is significantly better on SSDs; furthermore, when random reads are issued concurrently in an SSD, the aggregate throughput can match sequential throughput for some workloads [49]. As an example, Figure 4.3 shows the sequential and random read performance of a 500 GB Samsung 840 EVO SSD, for various request sizes. For random reads by a single thread, the throughput increases with the request size, reaching half the sequential throughput for 256 KB. With concurrent random reads by 32 threads, the aggregate throughput matches sequential throughput when the size is larger than 16 KB. For more high-end SSDs, the gap between concurrent random reads and sequential reads is much smaller [13, 136].

As we showed in this section, LSM-trees suffer from a high write and read amplification, which is acceptable for hard drives. Using LSM-trees on a high-performance SSD may waste a large percentage of device resources unnecessarily with excessive writing and reading. In this project, our goal is to improve the performance of LSM-trees on SSD devices to efficiently exploit device bandwidth.

4.2 WiscKey

The previous section explained how LSM-trees maintain sequential I/O access by increasing I/O amplification. While this trade-off between sequential I/O access and I/O amplification is justified for traditional hard disks, they are not optimal for modern hardware utilizing SSDs. In this section, we present the design of WiscKey, a key-value store that minimizes I/O amplification on SSDs.

To realize an SSD-optimized key-value store, WiscKey includes four critical ideas. First, WiscKey separates keys from values, keeping only keys in the LSM-tree and the values in a separate log file. Second, to deal with unsorted values (which necessitate random access during range queries), WiscKey uses the parallel random-read characteristic of SSD devices. Third, WiscKey utilizes unique crash-consistency and garbage-collection techniques to efficiently manage the value log. Finally, WiscKey optimizes performance by removing the LSM-tree log without sacrificing consistency, thus reducing system-call overhead from small writes.

4.2.1 Design Goals

WiscKey is a single-machine persistent key-value store, derived from LevelDB. It can be deployed as the storage engine for a relational database (e.g., MySQL) or a distributed key-value store (e.g., MongoDB). It provides the same API as LevelDB, including `Put(key, value)`, `Get(key)`, `Delete(key)` and `Scan(start, end)`. The design of WiscKey follows these main goals.

Low write amplification. Write amplification introduces extra unnecessary writes to the device. Even though SSD devices have higher bandwidth compared to hard drives, large write amplification can consume most of the write bandwidth (over 90% is not uncommon) and decrease the SSD's lifetime due to limited erase cycles. Therefore, it is important for WiscKey to minimize write amplification, so as to improve workload performance and SSD lifetime.

Low read amplification. Large read amplification causes two problems. First, the throughput of lookups is significantly reduced by issuing multiple reads for each lookup. Second, the large amount of data loaded into memory decreases the efficiency of the cache. WiscKey targets a small read amplification to speedup lookups.

SSD optimized. WiscKey is optimized for SSD devices by matching its I/O patterns with the performance characteristics of SSD devices. Specifically, sequential writes and parallel random reads are effectively utilized in WiscKey so that applications can fully utilize the device's bandwidth.

Feature-rich API. WiscKey aims to support modern features that have made LSM-trees popular, such as range queries and snapshots. Range

queries allow scanning a contiguous sequence of key-value pairs. Snapshots allow capturing the state of the database at a particular time and then performing lookups on the state.
Realistic key-value sizes. Keys are usually small in modern workloads (e.g., 16 B) [20, 21, 23, 64, 122], though value sizes can vary widely (e.g., 100 B to larger than 4 KB) [17, 23, 64, 80, 114, 187]. WiscKey aims to provide high performance for this realistic set of key-value sizes.

4.2.2 Key-Value Separation

The major performance cost of LSM-trees is the compaction process, which constantly sorts SSTable files. During compaction, multiple files are read into memory, sorted, and written back, which could significantly affect the performance of foreground workloads. However, sorting is required for efficient retrieval; with sorting, range queries (i.e., scan) will result mostly in sequential access to multiple files, while point queries would require accessing at most one file at each level.

WiscKey is motivated by a simple revelation. Compaction only needs to sort keys, while values can be managed separately [153]. Since keys are usually smaller than values, compacting only keys could significantly reduce the amount of data needed during the sorting. In WiscKey, only the location of the value is stored in the LSM-tree with the key, while the actual values are stored elsewhere in an SSD-friendly fashion. With this design, for a database with a given size, the size of the LSM-tree of WiscKey is much smaller than that of LevelDB. The smaller LSM-tree can remarkably reduce the write amplification for modern workloads that have a moderately large value size. For example, assuming a 16 B key, a 1 KB value, and a write amplification of 10 for keys (in the LSM-tree) and 1 for values, the effective write amplification of WiscKey is only (10 *16 + 1024) / (16 + 1024) = **1.14**. In addition to improving the write performance of applications, the reduced write amplification also improves an SSD's

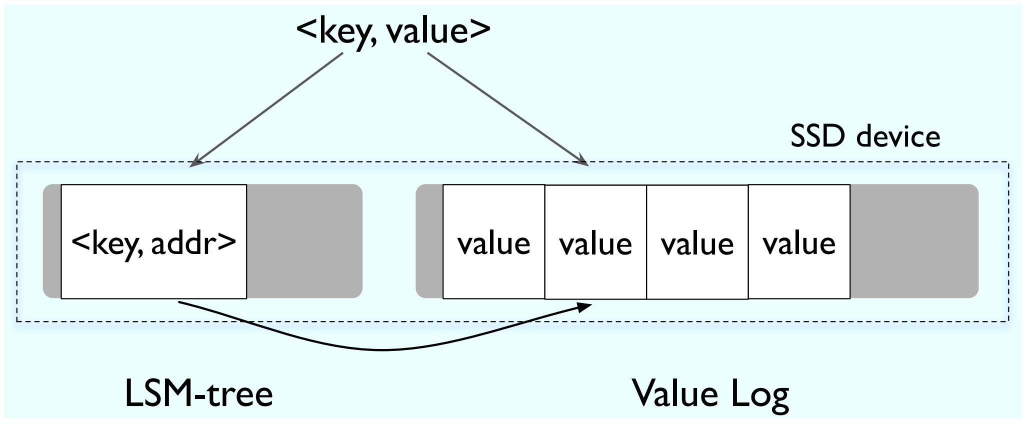


Figure 4.4: **WiscKey Data Layout on SSD.** *This figure shows the data layout of WiscKey on a single SSD device. Keys and value's locations are stored in LSM-tree while values are appended to a separate value log file.*

lifetime by requiring fewer erase cycles.

WiscKey's smaller read amplification improves lookup performance. During lookup, WiscKey first searches the LSM-tree for the key and the value's location; once found, another read is issued to retrieve the value. Readers might assume that WiscKey will be slower than LevelDB for lookups, due to its extra I/O to retrieve the value. However, since the LSM-tree of WiscKey is much smaller than LevelDB (for the same database size), a significant portion of the LSM-tree can be easily cached in memory. Hence, each lookup only requires a single random read (for retrieving the value) and thus achieves a lookup performance better than LevelDB. For example, assuming 16 B keys and 1 KB values, if the size of the entire key-value dataset is 100 GB, then the size of the LSM-tree is only around 2 GB (assuming a 12 B cost for a value's location and size), which can be easily cached in modern servers which have over 100 GB of memory.

WiscKey's architecture is shown in Figure 4.4. Keys are stored in a LSM-tree while values are stored in a separate value-log file, *vLog*. The artificial value stored along with the key in the LSM-tree is the address of the actual value in the vLog.

When the user inserts a key-value pair in WiscKey, the value is first appended to the vLog, and the key is then inserted into the LSM tree along with the value's address (`<vLog-offset, value-size>`). Deleting a key simply deletes it from the LSM tree, without touching the vLog. All valid values in the vLog have corresponding keys in the LSM-tree; the other values in the vLog are invalid and will be garbage collected later (§ 4.2.3).

When the user queries for a key, the key is first searched in the LSM-tree, and if found, the corresponding value's address is retrieved. Then, WiscKey reads the value from the vLog. Note that this process is applied to both point queries and range queries.

Although the idea behind key-value separation is simple, it leads to many challenges and optimization opportunities described in the following subsections.

4.2.3 Challenges

The separation of keys and values makes range queries require random I/O. Furthermore, the separation makes both garbage collection and crash consistency challenging. We now explain how we solve these challenges.

Parallel Range Query

Range queries are an important feature of modern key-value stores, allowing users to scan a range of key-value pairs. Relational databases [72], local file systems [100, 172, 190], and even distributed file systems [134] use key-value stores as their storage engines, and range queries are a core API requested in these environments.

For range queries, LevelDB provides the user with an iterator-based interface with `Seek(key)`, `Next()`, `Prev()`, `Key()` and `Value()` operations. To scan a range of key-value pairs, users can first `Seek()` to the starting key, then call `Next()` or `Prev()` to iterate key-value pairs one by one. To retrieve the key or the value of the current iterator position, users call `Key()` or `Value()`, respectively.

In LevelDB, since keys and values are stored together and sorted, a range query can sequentially read key-value pairs from SSTable files. However, since keys and values are stored separately in WiscKey, range queries require random reads, and are hence not efficient. As we see in Figure 4.3, the random read performance of a single thread on SSD cannot match the sequential read performance. However, parallel random reads with a fairly large request size can fully utilize the device's internal parallelism, getting performance similar to sequential reads.

To make range queries efficient, WiscKey leverages the parallel I/O characteristic of SSD devices to prefetch values from the vLog during range queries. The underlying idea is that, with SSDs, *only* keys require special attention for efficient retrieval. So long as keys are retrieved efficiently, range queries can use parallel random reads for efficiently retrieving values.

The prefetching framework can easily fit with the current range query interface. In the current interface, if the user requests a range query, an iterator is returned to the user. For each `Next()` or `Prev()` requested on the iterator, WiscKey tracks the access pattern of the range query. Once a contiguous sequence of key-value pairs is requested, WiscKey starts reading a number of following keys from the LSM-tree sequentially. The corresponding value addresses retrieved from the LSM-tree are inserted into a queue; multiple threads will fetch these addresses from the vLog concurrently in the background.

Garbage Collection

Key-value stores based on standard LSM-trees do not immediately reclaim free space when a key-value pair is deleted or overwritten. Rather, during compaction, if data relating to a deleted or overwritten key-value pair is found, the data is discarded and space is reclaimed. In WiscKey, only invalid keys are reclaimed by the LSM-tree compaction. Since WiscKey does not compact values, it needs a special garbage collector to reclaim free space in the vLog.

Since we only store the values in the vLog file (§ 4.2.2), a naive way to reclaim free space from the vLog is to first scan the LSM-tree to get all the valid value addresses; then, all the values in the vLog without any valid reference from the LSM-tree can be viewed as invalid and reclaimed. However, this method is too heavyweight and is only usable for offline garbage collection.

WiscKey targets a lightweight and online garbage collector. To make this possible, we introduce a small change to WiscKey's basic data layout: while storing values in the vLog, we also store the corresponding key along with the value. The new data layout is shown in Figure 4.5: the tuple `(key size, value size, key, value)` is stored in the vLog.

WiscKey's garbage collection aims to keep valid values (that do not correspond to deleted keys) in a contiguous range of the vLog, as shown in Figure 4.5. One end of this range, the *head*, always corresponds to the end of the vLog where new values will be appended. The other end of this range, known as the *tail*, is where garbage collection starts freeing space whenever it is triggered. Only the part of the vLog between the head and the tail contains valid values and will be searched during lookups.

During garbage collection, WiscKey first reads a chunk of key-value pairs (e.g., several MBs) from the tail of the vLog, then finds which of those values are valid (not yet overwritten or deleted) by querying the LSM-tree. WiscKey then appends valid values back to the head of the vLog. Finally, WiscKey frees the space occupied previously by the chunk, and updates the tail pointer accordingly.

To avoid losing any data if a crash happens during garbage collection, WiscKey has to make sure that the newly appended valid values and the new tail are persistent on the device before actually freeing space. WiscKey

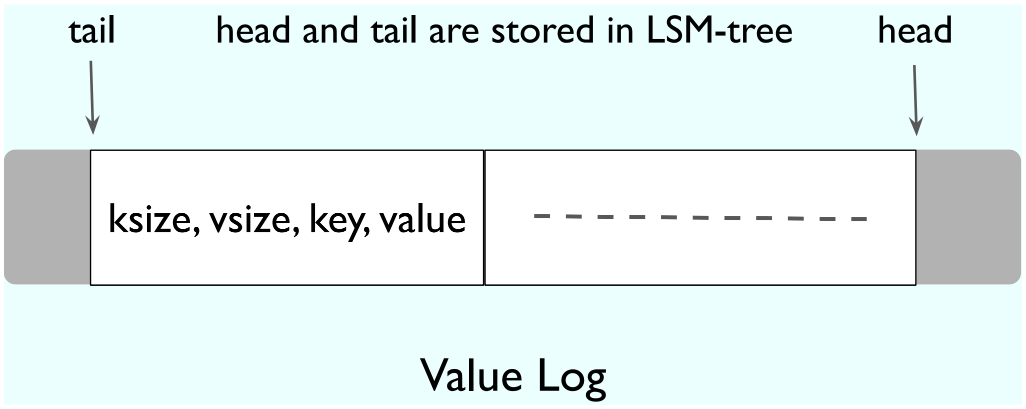


Figure 4.5: **WiscKey New Data Layout for Garbage Collection.** *This figure shows the new data layout of WiscKey to support an efficient garbage collection. A head and tail pointer are maintained in memory and stored persistently in the LSM-tree. Only the garbage collection thread changes the tail, while all writes to the vLog are append to the head.*

achieves this using the following steps. After appending the valid values to the vLog, the garbage collection calls a `fsync()` on the vLog. Then, it adds these new value's addresses and current tail to the LSM-tree in a synchronous manner; the tail is stored in the LSM-tree as `<‘‘tail’’, tail-vLog-offset>`. Finally, the free space in the vLog is reclaimed.

WiscKey can be configured to initiate and continue garbage collection periodically or until a particular threshold is reached. The garbage collection can also run in offline mode for maintenance. Garbage collection can be triggered rarely for workloads with few deletes and for environments with overprovisioned storage space.

Crash Consistency

On a system crash, LSM-tree implementations usually guarantee atomicity of inserted key-value pairs and in-order recovery of inserted pairs. Since WiscKey's architecture stores values separately from the LSM-tree, obtaining the same crash guarantees can appear complicated. However, WiscKey provides the same crash guarantees by using an interesting property of modern file systems (such as ext4, btrfs, and xfs). Consider a file that contains the sequence of bytes $\langle b_1 b_2 b_3 ... b_n \rangle$, and the user appends the sequence $\langle b_{n+1} b_{n+2} b_{n+3} ... b_{n+m} \rangle$ to it. If a crash happens, after file-system recovery in modern file systems, the file will be observed to contain the sequence of bytes $\langle b_1 b_2 b_3 ... b_n b_{n+1} b_{n+2} b_{n+3} ... b_{n+x} \rangle \ \exists \ x < m$, i.e., only some prefix of the appended bytes will be added to the end of the file during file-system recovery [163]. It is not possible for random bytes or a non-prefix subset of the appended bytes to be added to the file. Since values are appended sequentially to the end of the vLog file, the aforementioned property conveniently translates as follows: if a value X in the vLog is lost in a crash, all future values (inserted after X) are lost too.

When the user queries a key-value pair, if WiscKey cannot find the key in the LSM-tree because the key had been lost during a system crash, WiscKey behaves exactly like traditional LSM-trees: even if the value had been written in vLog before the crash, it will be garbage collected later. If the key could be found in the LSM tree, however, an additional step is required to maintain consistency. In this case, WiscKey first verifies whether the value address retrieved from the LSM-tree falls within the current valid range of the vLog, and then whether the value found corresponds to the queried key. If the verifications fail, WiscKey assumes that the value was lost during a system crash, deletes the key from the LSM-tree, and informs the user that the key was not found. Since each value added to the vLog has a header including the corresponding key, verifying whether the key and the value match is straightforward; if necessary, a magic number or checksum can be easily added to the header.

LSM-tree implementations also guarantee the user durability of key value pairs after a system crash if the user requests synchronous inserts specifically. WiscKey implements synchronous inserts by flushing the vLog before performing a synchronous insert into its LSM-tree.

4.2.4 Optimizations

Separating keys and values in WiscKey provides an opportunity to rethink how the value log is updated and the necessity of the LSM-tree log. We now describe how these opportunities can lead to improved performance.

Value-Log Write Buffer

For each `Put()`, WiscKey needs to append the value to the vLog by using a `write()` system call. However, for an insert-intensive workload, issuing a large number of small writes to a file system can introduce a noticeable overhead, especially on a fast storage device [45, 162]. Figure 4.6 shows the total time to sequentially write and call `fsync()` on a 10 GB file in ext4 (Linux 3.14). When small writes are used, the overhead of each system call aggregates significantly, leading to a long run time. With large writes (larger than 4 KB), the device throughput is fully utilized.

To reduce overhead, WiscKey buffers values in a userspace buffer, and flushes the buffer only when the buffer size exceeds a threshold or when the user requests a synchronous insertion. Thus, WiscKey only issues large writes and reduces the number of `write()` system calls. For a lookup, WiscKey needs to first search the vLog buffer, and if not found in the buffer, actually reads from the vLog. Obviously, this mechanism might result in some data (that is buffered) to be lost during a crash; the crash-consistency guarantee obtained is similar to LevelDB.

Optimizing LSM-tree Log

As shown in Figure 4.1, a log file is usually used in LSM-trees. The LSM-tree tracks inserted key-value pairs in the log file so that, if the user requests synchronous inserts and there is a crash, the log can be scanned after reboot and the inserted key-value pairs recovered.

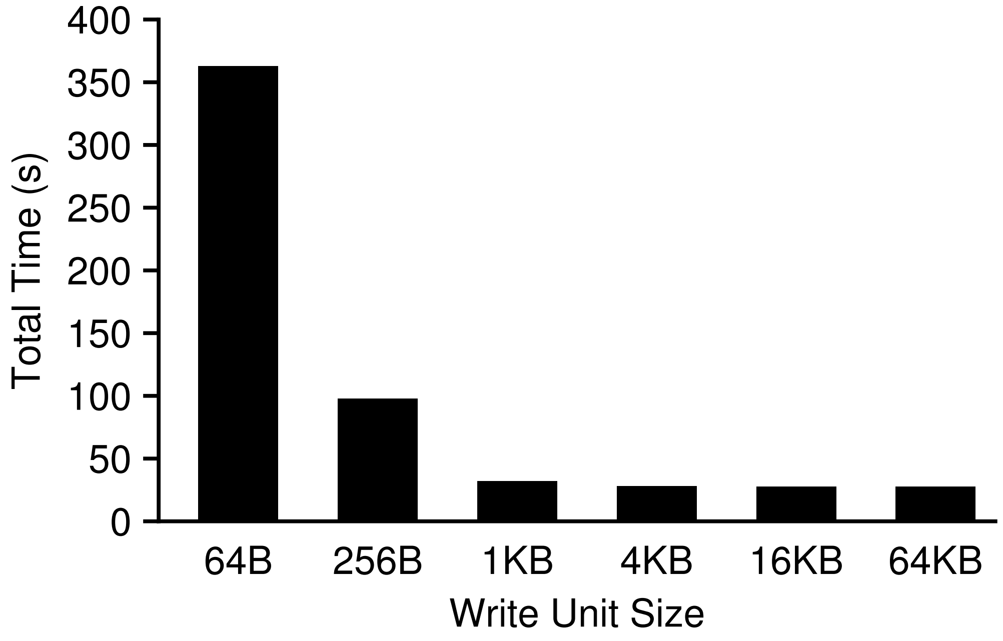


Figure 4.6: **Impact of Write Unit Size.** *This figure shows the total time to write a 10 GB file to an ext4 file system on an SSD device, followed by a* `fsync()` *at the end. We vary the size of each* `write()` *system call.*

In WiscKey, the LSM-tree is only used for keys and value addresses. Moreover, the vLog also records inserted keys to support garbage collection as described in the previous section. Hence, writes to the LSM-tree log file can be avoided without affecting correctness.

If a crash happens before the keys are persistent in the LSM-tree, they can be recovered by scanning the vLog. However, a naive algorithm would require scanning the entire vLog for recovery. So as to require scanning only a small portion of the vLog, WiscKey records the head of the vLog periodically in the LSM-tree, as a key-value pair `<‘‘head’’, head-vLog-offset>`. When a database is opened, WiscKey starts the vLog scan from the most recent head position stored in the LSM-tree, and continues scanning until the end of the vLog. Since the head is stored in the LSM-tree, and the LSM-tree inherently guarantees that keys inserted into the LSM-tree will be recovered in the inserted order, this optimization is crash consistent. Therefore, removing the LSM-tree log of WiscKey is a safe optimization, and improves performance especially when there are many small insertions.

4.2.5 Implementation

WiscKey is based on LevelDB 1.18. WiscKey creates a vLog when creating a new database, and manages the keys and value addresses in the LSM-tree. The vLog is internally accessed by multiple components with different access patterns. For example, a lookup is served by randomly reading the vLog, while the garbage collector sequentially reads from the tail and appends to the head of the vLog file. We use `posix_fadvise()` to predeclare access patterns for the vLog under different situations.

For range queries, WiscKey maintains a background thread pool with 32 threads. These threads sleep on a thread-safe queue, waiting for new value addresses to arrive. When prefetching is triggered, WiscKey inserts a fixed number of value addresses to the worker queue, and then wakes up all the sleeping threads. These threads will start reading values in parallel, caching them in the buffer cache automatically.

To efficiently garbage collect the free space of the vLog, we use the hole-punching functionality of modern file systems (`fallocate()`). Punching a hole in a file can free the physical space allocated, and allows WiscKey to elastically use the storage space. In general, the maximal individual file size on modern file systems is big enough for WiscKey to run a long time without wrapping back to the beginning of the file; for example, the maximal individual file size is 64 TB on ext4, 8 EB on XFS and 16 EB on btrfs. The vLog can be trivially adapted into a circular log if necessary.

4.3 Evaluation

In this section, we present evaluation results that demonstrate the benefits of the design choices made in WiscKey. Specifically, we seek to answer the following fundamental performance questions about WiscKey:

1. Does key-value separation result in lower write and read amplification, and how does it impact the performance and endurance of the SSD devices?
2. Does the parallel range query in WiscKey work efficiently with modern SSDs?
3. How does WiscKey perform on real workloads YCSB, compared to other popular peers?
4. What is the effect of garbage collection on WiscKey's performance?
5. Does WiscKey maintain crash consistency, and how long does it take to recover after a crash?

4.3.1 Experimental Setup

All experiments are run on a testing machine with two Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz processors and 64 GB of memory. The operating system is 64-bit Linux 3.14, and the file system used is ext4. The storage device used is a 500 GB Samsung 840 EVO SSD, which has 500 MB/s sequential-read and 400 MB/s sequential-write maximal performance. Random read performance of the device is shown in Figure 4.3.

4.3.2 Microbenchmarks

We use `db_bench`, the default set of microbenchmarks included in LevelDB, to evaluate LevelDB and WiscKey. It contains a series of basic key-value store benchmarks, such as loading a database and random lookups. We

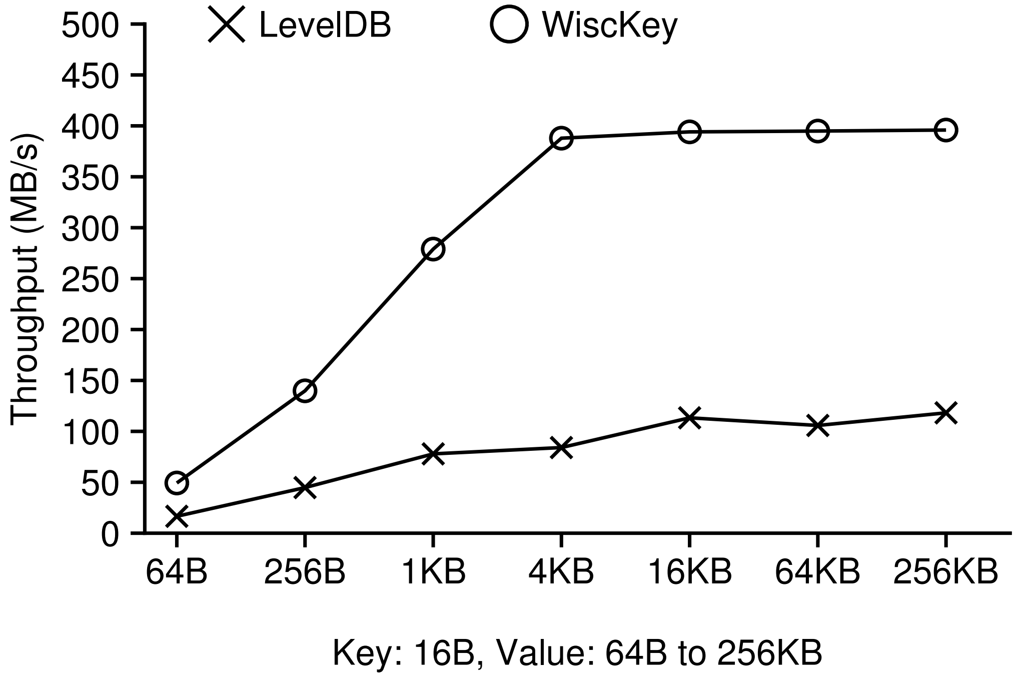


Figure 4.7: **Sequential-load Performance.** *This figure shows the sequential-load throughput of LevelDB and WiscKey for different value sizes for a 100 GB dataset. Key size is 16 B.*

always use a key size of 16 B, but perform experiments for different value sizes. We disable data compression in both key-value stores for easier understanding and analysis of their performance.

Load Performance

We now describe the results for the sequential-load and random-load microbenchmarks. The former benchmark constructs a 100 GB database by inserting keys in a sequential order, while the latter inserts keys in a uniformly distributed random order. Note that the sequential-load benchmark does not cause compaction in either LevelDB or WiscKey, while the random-load does.

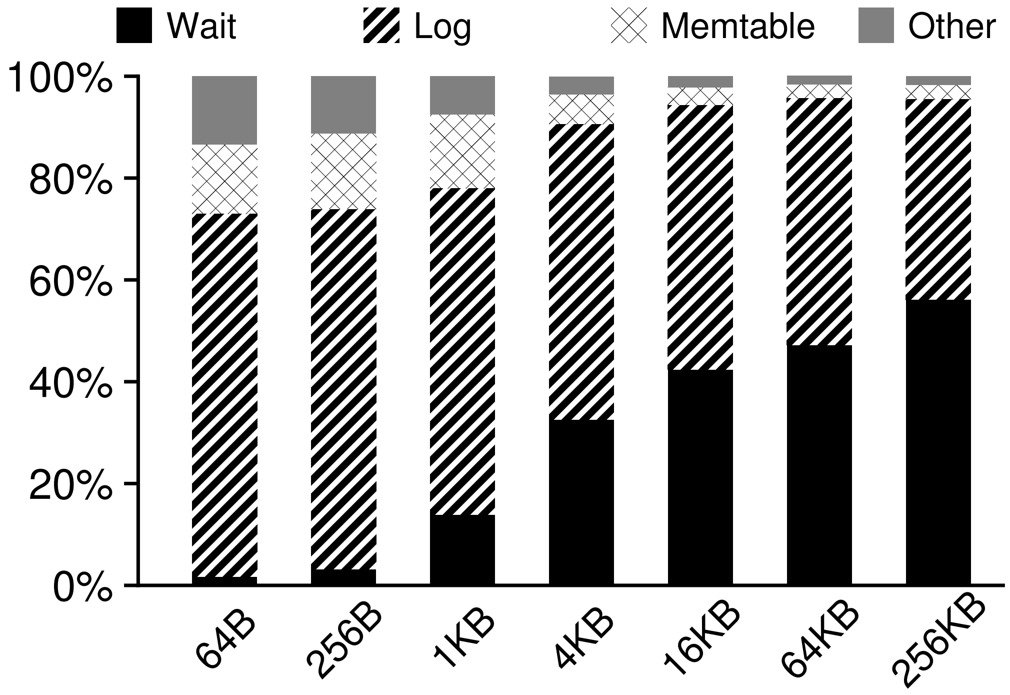


Figure 4.8: **Sequential-load Time Breakup of LevelDB.** *This figure shows the percentage of time incurred in different components during sequential load in LevelDB.*

Figure 4.7 shows the sequential-load throughput of LevelDB and WiscKey for a wide range of value sizes: the throughput of both stores increases with the value size. But, even for the largest value size considered (256 KB), LevelDB's throughput is far from the device bandwidth. To analyze this further, Figure 4.8 shows the distribution of the time spent in different components during each run of the benchmark, for LevelDB; time is spent in three major components: writing to the log file, inserting to the memtable, and waiting for the memtable to be flushed to the device. For small key-value pairs, writing to the log file accounts for the most significant percentage of the total time, for the reasons explained in Figure 4.6. For larger pairs, log writing and the memtable sorting are more efficient, while memtable flushes are the bottleneck. Unlike LevelDB,

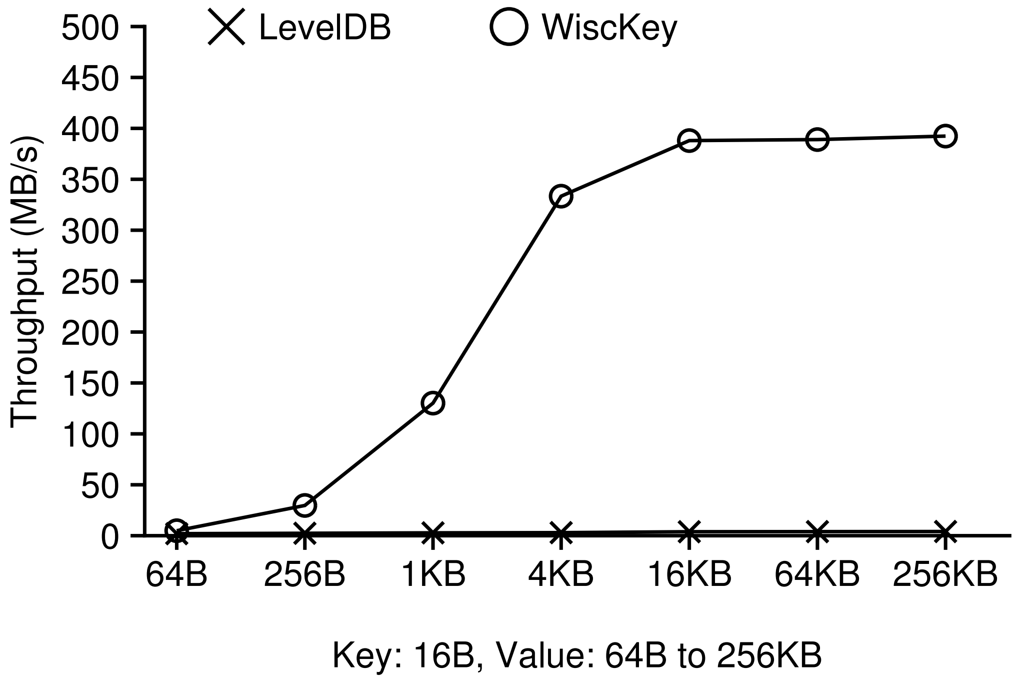


Figure 4.9: **Random-load Performance.** *This figure shows the random-load throughput of LevelDB and WiscKey for different value sizes for a 100 GB dataset. Key size is 16 B.*

WiscKey reaches the full device bandwidth for value sizes more than 4 KB. Since WiscKey does not write to the LSM-tree log and buffers appends to the vLog, it is 3× faster than LevelDB even for small values.

Figure 4.9 shows the random-load throughput of LevelDB and WiscKey for different value sizes. LevelDB's throughput ranges from only 2 MB/s (64 B value size) to 4.1 MB/s (256 KB value size), while WiscKey's throughput increases with the value size, reaching the peak device write throughput after the value size is bigger than 4 KB. WiscKey's throughput is 46× and 111× of LevelDB for the 1 KB and 4 KB value size respectively. LevelDB has low throughput because compaction both consumes a large percentage of the device bandwidth and also slows down foreground

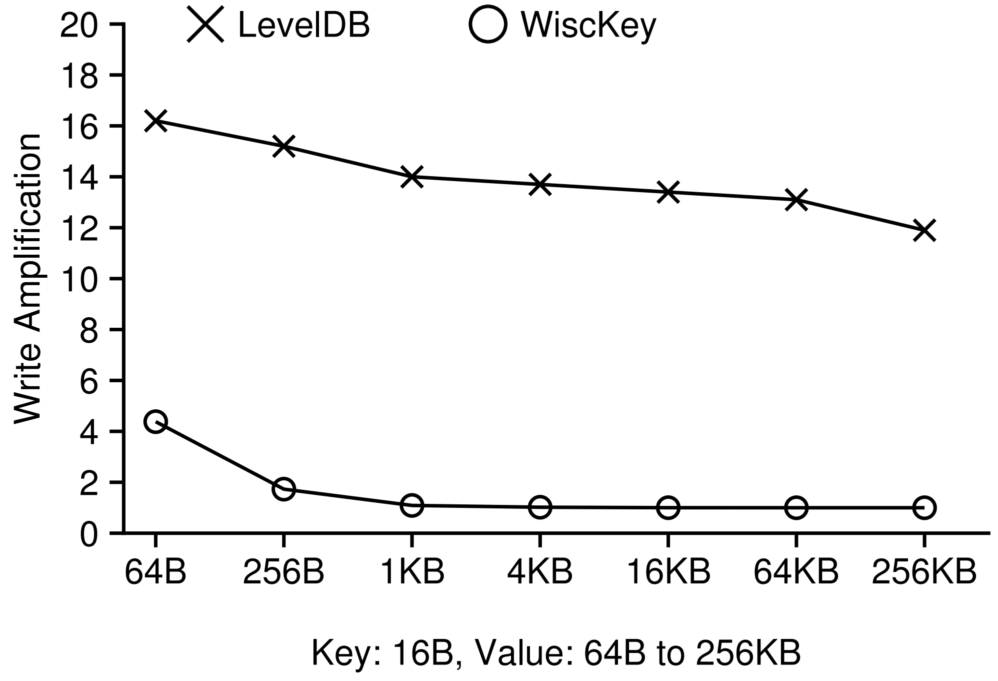


Figure 4.10: **Write Amplification of Random Load.** *This figure shows the write amplification of LevelDB and WiscKey for randomly loading a 100 GB database.*

writes (to avoid overloading the L_0 of the LSM-tree, as described in Section 4.1.2). In WiscKey, compaction only introduces a small overhead, leading to the full device bandwidth being effectively utilized. To analyze this further, Figure 4.10 shows the write amplification of LevelDB and WiscKey for the random-load experiment. The write amplification of LevelDB is always more than 12, while that of WiscKey decreases quickly to nearly 1 when the value size reaches 1 KB, because the LSM-tree of WiscKey is significantly smaller.

Query Performance

We now compare the random lookup (point query) and range query performance of LevelDB and WiscKey. Figure 4.11 presents the random lookup

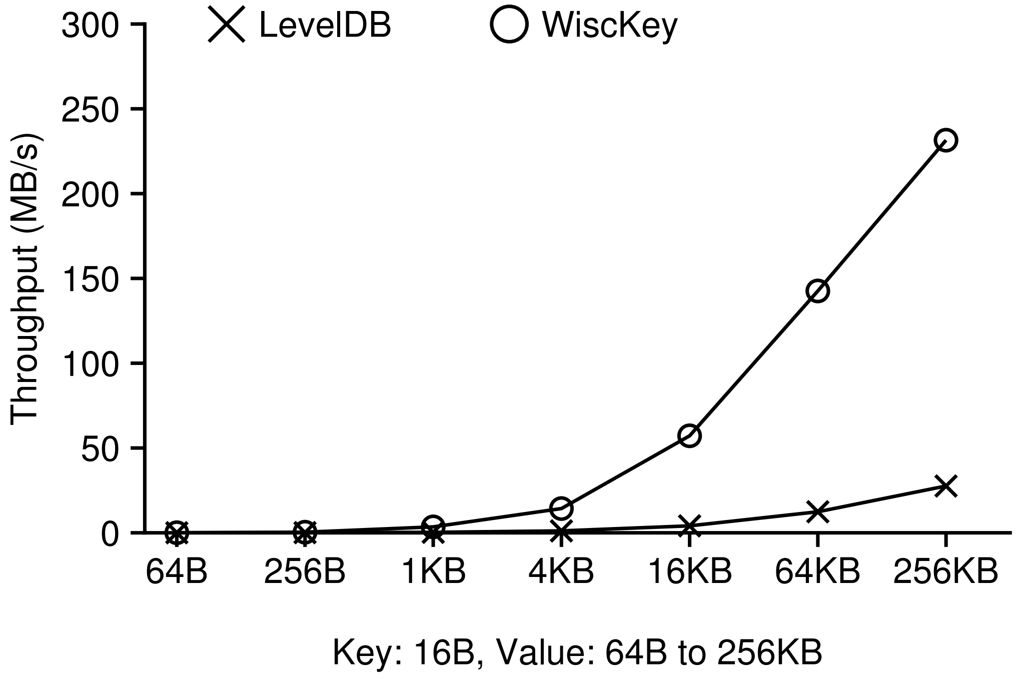


Figure 4.11: **Random Lookup Performance.** *This figure shows the random lookup performance for 100,000 operations on a 100 GB database that is randomly loaded.*

results of 100,000 operations on a 100 GB random-loaded database. Even though a random lookup in WiscKey needs to check both the LSM-tree and the vLog, the throughput of WiscKey is still much better than LevelDB: for 1 KB value size, WiscKey's throughput is 12× of that of LevelDB. For large value sizes, the throughput of WiscKey is only limited by the random read throughput of the device, as shown in Figure 4.3. LevelDB has low throughput because of the high read amplification mentioned in Section 4.1.3. WiscKey performs significantly better because the read amplification is lower due to a smaller LSM-tree (which causes the LSM-tree to have a higher cache hit rate). Another reason for WiscKey's better performance is that the compaction process in WiscKey is less intense than LevelDB, thus avoiding many background reads and writes.

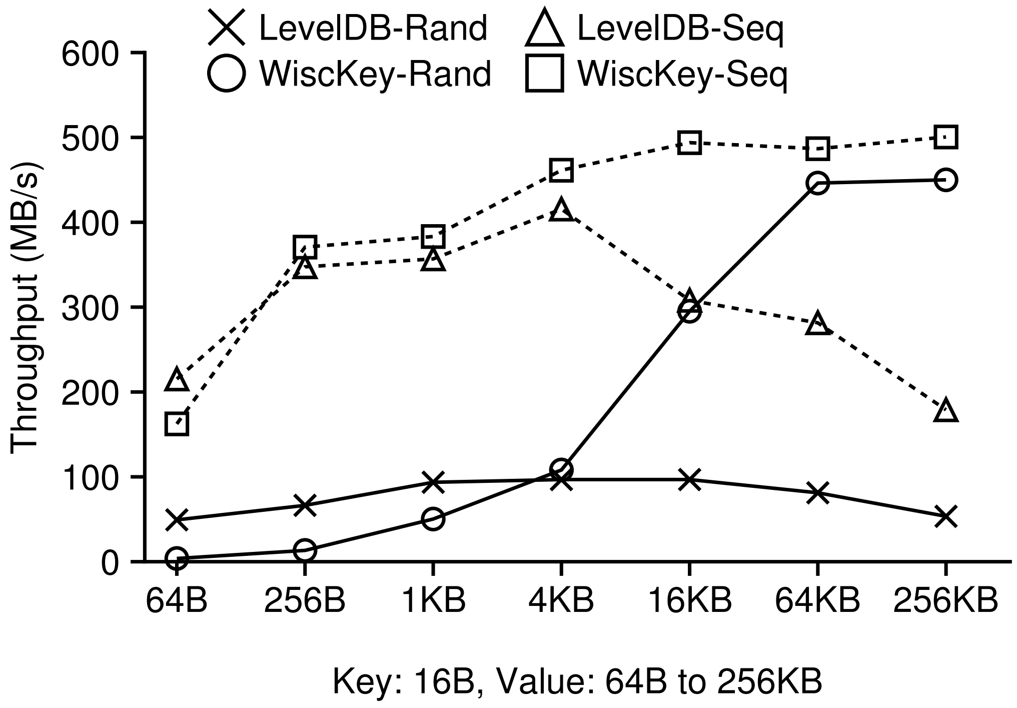


Figure 4.12: **Range Query Performance.** *This figure shows range query performance. 4 GB of data is queried from a 100 GB database that is randomly (Rand) and sequentially (Seq) loaded.*

Figure 4.12 shows the range query (scan) performance of LevelDB and WiscKey. For a randomly-loaded database, LevelDB reads multiple files from different levels, while WiscKey requires random accesses to the vLog (but WiscKey leverages parallel random reads). As can be seen from Figure 4.12, the throughput of LevelDB initially increases with the value size for both databases. However, beyond a value size of 4 KB, since an SSTable file can store only a small number of key-value pairs, the overhead is dominated by opening many SSTable files and reading the index blocks and bloom filters in each file. For larger key-value pairs, WiscKey can deliver the device's sequential bandwidth, up to 8.4× of LevelDB. However, WiscKey performs 12× worse than LevelDB for 64 B key-value pairs due to the device's limited parallel random-read throughput for small request sizes; WiscKey's relative performance is better on high-end SSDs with higher parallel random-read throughput [13]. Furthermore, this workload represents a worst-case where the database is randomly-filled and the data is unsorted in the vLog.

Figure 4.12 also shows the performance of range queries when the data is sorted, which corresponds to a sequentially-loaded database; in this case, both LevelDB and WiscKey can sequentially scan through data. Performance for sequentially-loaded databases follows the same trend as randomly-loaded databases; for 64 B pairs, WiscKey is 25% slower because WiscKey reads both the keys and the values from the vLog (thus wasting bandwidth), but WiscKey is 2.8× faster for large key-value pairs. Thus, with small key-value pairs, log reorganization (sorting) for a random-loaded database can make WiscKey's range-query performance comparable to LevelDB's performance.

Garbage Collection

We now investigate WiscKey's performance while garbage collection is performed in the background. The performance can potentially vary depending on the percentage of free space found during garbage collection, since this affects the amount of data written and the amount of space freed by the garbage collection thread. We use random-load (the workload that is most affected by garbage collection) as the foreground workload, and study its performance for various percentages of free space. Our experiment specifically involves three steps: we first create a database using random-load, then delete the required percentage of key-value pairs, and finally, we run the random-load workload and measure its throughput while garbage collection happens in the background. We use a key-value size of 4 KB and vary the percentage of free space from 25% to 100%.

Figure 4.13 shows the results: if 100% of data read by the garbage

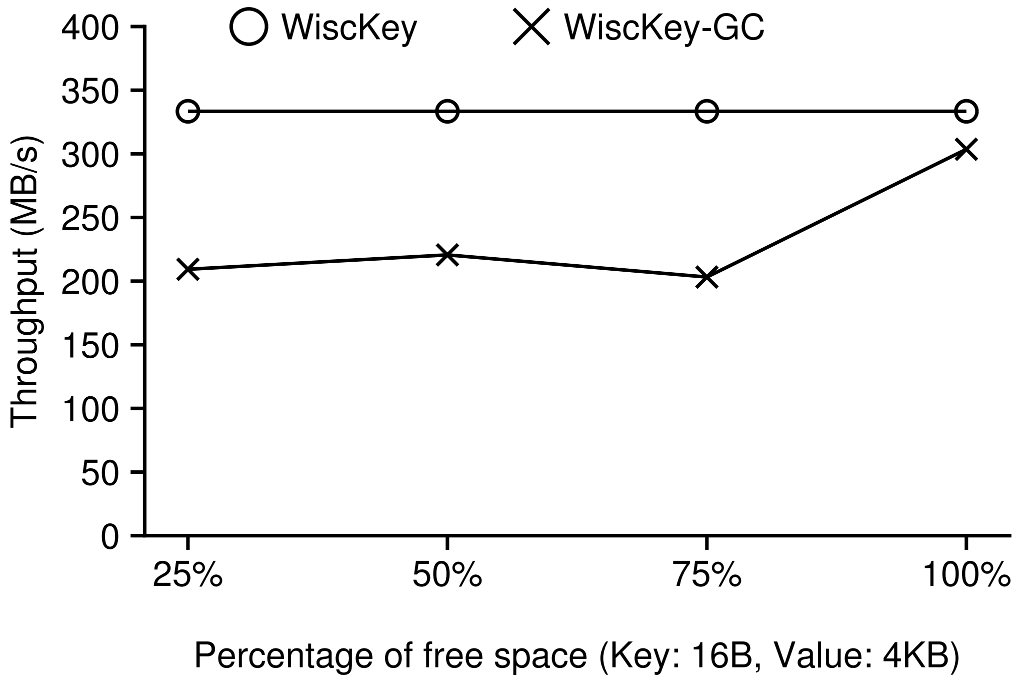


Figure 4.13: **Garbage Collection.** *This figure shows performance under garbage collection for various free-space ratios.*

collector is invalid, the throughput is only 10% lower. This is because garbage collection reads from the tail of the vLog and writes only valid key-value pairs to the head; if the data read is entirely invalid, no key-value pairs needs to be written. For other percentages of free space, throughput drops about 35% since the garbage collection thread performs additional writes. Note that, in all cases, while garbage collection is happening, WiscKey is at least 70× faster than LevelDB.

Crash Consistency

Separating keys and values necessitates additional mechanisms to maintain crash consistency. We verify WiscKey's crash consistency mechanisms by using the ALICE tool [163]; the tool chooses and simulates a comprehensive set of system crashes that have a high probability of exposing inconsistency. We use a testcase which invokes a few asynchronous and synchronous `Put()` calls. When configured to run tests for ext4, xfs, and btrfs, ALICE checks more than 3000 selectively-chosen system crashes for the given testcase, and does not report any consistency vulnerability introduced by WiscKey.

The new consistency mechanism also affects WiscKey's recovery time after a crash, and we design an experiment to measure the worst-case recovery time of WiscKey and LevelDB. LevelDB's recovery time is proportional to the size of its log file after the crash; the log file exists at its maximum size just before the memtable is written to disk. WiscKey, during recovery, first retrieves the head pointer from the LSM-tree, and then scans the vLog file from the head pointer till the end of the file. Since the updated head pointer is persisted on disk when the memtable is written, WiscKey's worst-case recovery time also corresponds to a crash happening just before then. We measured the worst-case recovery time induced by the situation described so far; for 1 KB values, LevelDB takes 0.7 seconds to recover the database after the crash, while WiscKey takes 2.6 seconds. Note that WiscKey can be configured to persist the head pointer more frequently if necessary.

Space Amplification

When evaluating a key-value store, most previous work focused only on read and write amplification. However, space amplification is important for flash devices because of their expensive price-per-GB compared with hard drives. Space amplification is the ratio of the actual size of the database on disk to the logical size of the the database [15]. For example, if a 1-KB key-value pair takes 4 KB of space on disk, then the space amplification is 4. Compression decreases space amplification while extra data (garbage, fragmentation, or metadata) increases space amplification.

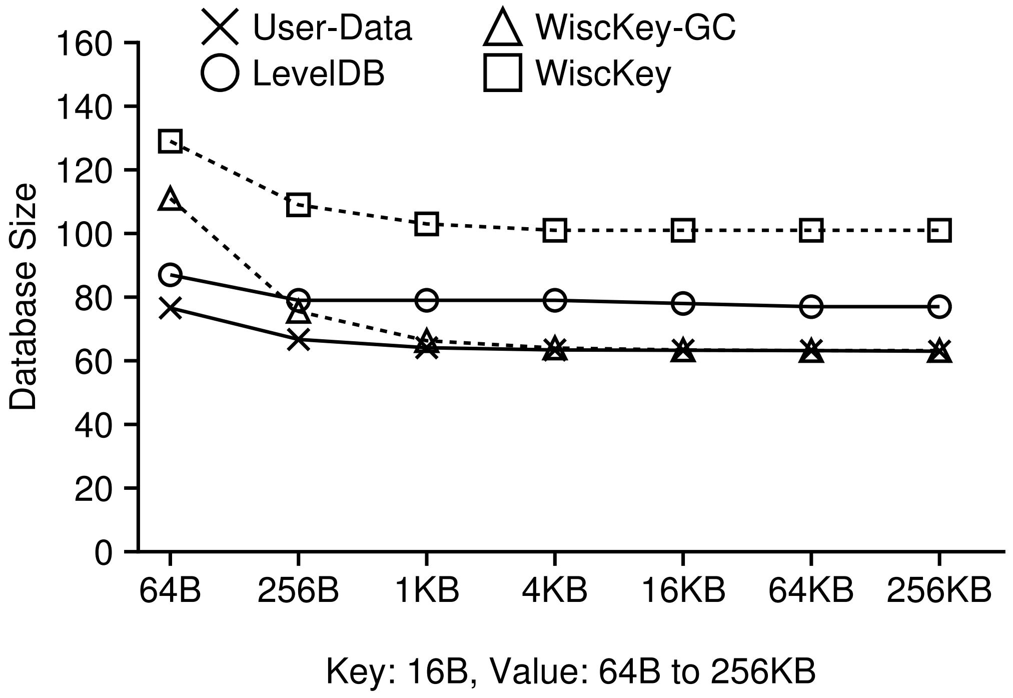


Figure 4.14: **Space Amplification.** *This figure shows the actual database size of LevelDB and WiscKey for a random-load workload of a 100-GB dataset. User-Data represents the logical database size.*

Compression is disabled to make the discussion simple.

For a sequential-load workload, the space amplification can be near one, given that the extra metadata in LSM-trees is minimal. For a random-load or overwrite workload, space amplification is usually more than one when invalid pairs are not garbage collected fast enough.

Figure 4.14 shows the database size of LevelDB and WiscKey after randomly loading a 100-GB dataset (the same workload as Figure 4.9). The space overhead of LevelDB arises due to invalid key-value pairs that are not garbage collected when the workload is finished. The space overhead of WiscKey includes the invalid key-value pairs and the extra metadata (pointers in the LSM-tree and the tuple in the vLog as shown in Figure 4.5).

After garbage collection, the database size of WiscKey is close to the logical database size when the extra metadata is small compared to the value size.

No key-value store can minimize read amplification, write amplification, and space amplification at the same time. Tradeoffs among these three factors are balanced differently in various systems. In LevelDB, the sorting and garbage collection are coupled together. LevelDB trades higher write amplification for lower space amplification; however, the workload performance can be significantly affected. WiscKey consumes more space to minimize I/O amplification when the workload is running; because sorting and garbage collection are decoupled in WiscKey, garbage collection can be done later, thus minimizing its impact on foreground performance.

CPU Usage

We now investigate the CPU usage of LevelDB and WiscKey for various workloads shown in previous sections. The CPU usage shown here includes both the application and operating system usage.

As shown in Table 4.15, LevelDB has higher CPU usage for sequential-load workload. As we explained in Figure 4.8, LevelDB spends a large amount of time writing key-value pairs to the log file. Writing to the log file involves encoding each key-value pair, which has high CPU cost. Since WiscKey removes the log file as an optimization, WiscKey has lower CPU usage than LevelDB. For the range query workload, WiscKey uses 32 background threads to do the prefetch; therefore, the CPU usage of WiscKey is much higher than LevelDB.

We find that CPU is not a bottleneck for both LevelDB and WiscKey in our setup. The architecture of LevelDB is based on single writer protocol. The background compaction also only uses one thread. Better concurrency design for multiple cores is explored in RocksDB [71].

Workloads	Seq Load	Rand Load	Rand Lookup	Range Query
LevelDB	10.6%	6.3%	7.9%	11.2%
WiscKey	8.2%	8.9%	11.3%	30.1%

Table 4.15: **CPU Usage of LevelDB and WiscKey.** *This table compares the CPU usage of four workloads on LevelDB and WiscKey. Key size is 16 B and value size is 1 KB. Seq-Load and Rand-Load sequentially and randomly load a 100-GB database respectively. Given a 100-GB random-filled database, Rand-Lookup issues 100 K random lookups, while Range-Query sequentially scans 4-GB data.*

4.3.3 YCSB Benchmarks

The YCSB benchmark [60] provides a framework and a standard set of six workloads for evaluating the performance of key-value stores. We use YCSB to compare LevelDB, RocksDB [71], and WiscKey, on a 100 GB database. In addition to measuring the usual-case performance of WiscKey, we also run WiscKey with garbage collection always happening in the background so as to measure its worst-case performance. RocksDB [71] is a SSD-optimized version of LevelDB with many optimizations, including multiple memtables and background threads for compaction. We use RocksDB with the default configuration parameters. We evaluated the key-value stores with two different value sizes, 1 KB and 16 KB (data compression is disabled).

WiscKey performs significantly better than LevelDB and RocksDB, as shown in Figure 4.16. For example, during load, for 1 KB values, WiscKey performs at least 50× faster than the other databases in the usual case, and at least 45× faster in the worst case (with garbage collection); with 16 KB values, WiscKey performs 104× better, even under the worst case.

For reads, the Zipf distribution used in most workloads allows popular items to be cached and retrieved without incurring disk access, thus reducing WiscKey's advantage over LevelDB and RocksDB. Hence, Wis-

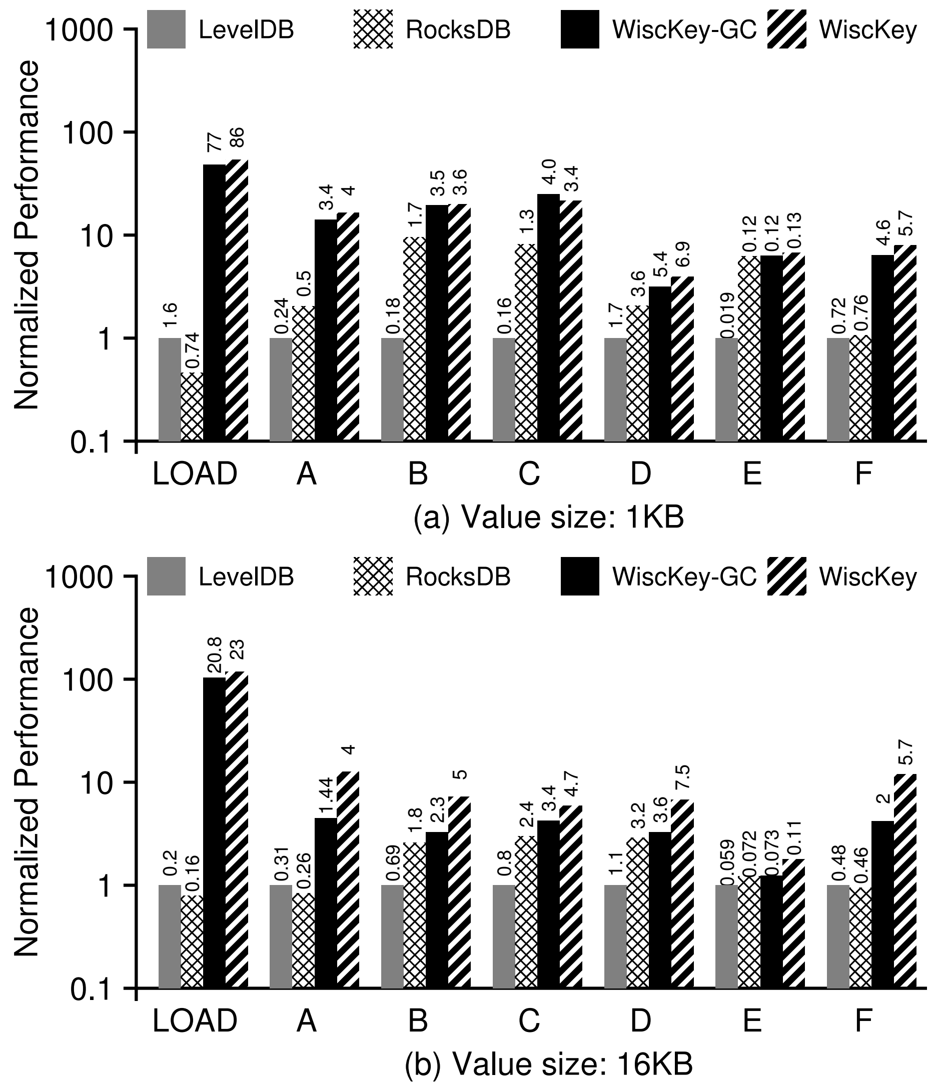


Figure 4.16: **YCSB Macrobenchmark Performance.** *This figure shows the performance of LevelDB, RocksDB, and WiscKey for various YCSB workloads. The X-axis corresponds to different workloads, and the Y-axis shows the performance normalized to LevelDB's performance. The number on top of each bar shows the actual throughput achieved (K ops/s). (a) shows performance under 1 KB values and (b) shows performance under 16 KB values. The load workload corresponds to constructing a 100 GB database and is similar to the random-load microbenchmark. Workload-A has 50% reads and 50% updates, Workload-B has 95% reads and 5% updates, and Workload-C has 100% reads; keys are chosen from a Zipf, and the updates operate on already-existing keys. Workload-D involves 95% reads and 5% inserting new keys (temporally weighted distribution). Workload-E involves 95% range queries and 5% inserting new keys (Zipf), while Workload-F has 50% reads and 50% read-modify-writes (Zipf).*

cKey's relative performance (compared to the LevelDB and RocksDB) is better in Workload-A (50% reads) than in Workload-B (95% reads) and Workload-C (100% reads). However, RocksDB and LevelDB still do not match WiscKey's performance in any of these workloads.

The worst-case performance of WiscKey (with the garbage collection turned on always, even for read-only workloads) is better than LevelDB and RocksDB. However, the impact of garbage collection on performance is markedly different for 1 KB and 16 KB values. Garbage collection repeatedly selects and cleans a 4 MB chunk of the vLog; with small values, the chunk will include many key-value pairs, and thus garbage collection spends more time accessing the LSM-tree to verify the validity of each pair. For large values, garbage collection spends less time on the verification, and hence aggressively writes out the cleaned chunk, affecting foreground throughput more. Note that, if necessary, garbage collection can be throttled to reduce its foreground impact.

Unlike the microbenchmark considered previously, Workload-E has multiple small range queries, with each query retrieving between 1 and 100 key-value pairs. Since the workload involves multiple range queries, accessing the first key in each range resolves to a random lookup – a situation favorable for WiscKey. Hence, WiscKey performs better than RocksDB and LevelDB even for 1 KB values.

4.4 Conclusions

Key-value stores have become a fundamental building block in data-intensive applications. In this paper, we propose WiscKey, a novel LSM-tree-based key-value store that separates keys and values to minimize write and read amplification. The data layout and I/O patterns of WiscKey are highly optimized for SSD devices.

We compare the performance of WiscKey with LevelDB and RocksDB, two popular LSM-tree key-value stores. For most workloads, WiscKey performs significantly better. With LevelDB's own microbenchmark, WiscKey is 2.5×–111× faster than LevelDB for loading a database, depending on the size of the key-value pairs; for random lookups, WiscKey is 1.6×–14× faster than LevelDB. Under YCSB macrobenchmarks that reflect real-world use cases, WiscKey is faster than both LevelDB and RocksDB in all six YCSB workloads, and follows a trend similar to the load and random lookup microbenchmarks.

Our hope is that key-value separation and various optimization techniques in WiscKey will inspire the future generation of high-performance key-value stores.

5 Related Work

In this chapter, we discuss various research efforts and systems that are related to this dissertation. First, we describe previous system bug related studies over the years. Then, we describe work related to provide reliability and performance isolation in file systems. Finally, we discuss various key-value stores based on different structures and performance optimizations.

5.1 System Bug Study

Faults in Linux have been systematically studied in last decade [54, 159]. Static analysis tools are used to find potential bugs in Linux 1.0 to 2.4.1 [54] and Linux 2.6.0 to 2.6.33 [159]. Most detected faults are generic memory and concurrency bugs. Both studies find that device drivers contain the most faults, while Palix et al. [159] also show that file-system errors are rising. Yin et al. [228] analyze incorrect bug-fixes in several operating systems, and find that about 20% of fixes are incorrect. We also observe similar results in file system bugs. Our work embellishes these studies, focusing on all file-system bugs found and fixed over eight years and providing more detail on which bugs plague file systems.

Various aspects of modern user-level open source software bugs have also been studied, including bug patterns, bug impacts, reproducibility, and bug fixes [74, 120, 131, 181, 225]. As our findings show, file-systems bugs display different characteristics compared with user-level software bugs, both in their patterns and consequences (e.g., file-system bugs have more serious consequences than user-level bugs; concurrency bugs are much more common). One other major difference is scale; the number of bugs (about 1800) and the time span (about 8 years of development) we study is much larger than previous efforts [74, 120, 131, 181, 225].

In addition to system bug studies, several research projects have been proposed to detect and analyze file-system bugs. For example, Yang et al. [226, 227] use model checking to detect file-system errors; Gunawi et al. [84] use static analysis techniques to determine how error codes are propagated in file systems; Rubio-Gonzalez et al. [178] utilize static analysis to detect similar problems; Prabhakaran et al. [165] study how file systems handle injected failures and corruptions. Our work complements this work with insights on bug patterns and root causes. Further, our public bug dataset provides useful hints and patterns to aid in the development of new file-system bug-detection tools.

There are several interesting related research projects after our paper is published. Similar to our study methodology and analysis categories, software bug studies are also conducted for modern cloud systems [83, 95]. Both projects analyze development and deployment issues found in popular distributed systems, such as Hadoop, HDFS, and Cassandra. We have two important findings in our study: semantic bugs dominate and failure-handling paths are error-prone. Min et al. [146] propose a tool that automatically infers file-system semantics directly from source code. The tool compares and contrasts multiple existing file-system implementations to detect the semantic bugs. Yuan et al. [229] find that majority of catastrophic failures in distributed systems could easily be prevented by performing simple testing on error handling code. They develop a simple static checker to locate these error-handling bugs. In addition to bug-finding tools, another way to overcome file system bugs is to design verifiable file systems. BilbyFS [109] uses layered domain-specific languages to generate code and proofs, while FSCQ [50] extends Hoare logic with crash predicates, recovery procedures and logical address spaces for a complete file system.

5.2 File System Isolation

Isolation is an important property in modern systems. Various isolation techniques are proposed for different parts of the system. Typical examples include virtual machines [40, 69], Linux Containers [7], isolation kernel [218], BSD `jail` [105] and Solaris zones [156]. They are used to create an independent environment for the target applications or even operating systems to run in a shared platform. For the file system component, these frameworks only provide namespace isolation for different clients by constraining clients to only a subset of the shared file system. However, as shown in this thesis, the failure, recovery and journaling performance are not isolated in a shared file system. Our work is to provide a file system level container which can be used together with the above techniques to provide full isolation for applications.

IceFS has derived inspiration from a number of projects for improving file system recovery and repair, and for tolerating system crashes.

Many existing systems have improved the reliability of file systems with better recovery techniques. Fast checking of the Solaris UFS [161] has been proposed by only checking the working-set portion of the file system when failure happens. Changing the I/O pattern of the file system checker to reduce random requests has been suggested [31, 133]. A background fsck in BSD [142] checks a file system snapshot to avoid conflicts with the foreground workload. WAFL [91] employs Wafliron [151], an online file system checker, to perform online checking on a volume but the volume being checked cannot be accessed by users. Our recovery idea is based on the cube abstraction which provides isolated failure, recovery and journaling. Under this model, we only check the faulty part of the file system without scanning the whole file system. The above techniques can be utilized in one cube to further speedup the recovery process.

Several repair-driven file systems also exist. Chunkfs [89] does a partial check of Ext2 by partitioning the file system into multiple chunks; however, files and directory can still span multiple chunks, reducing the independence of chunks. Windows ReFS [193] can automatically recover corrupted data from mirrored storage devices when it detects checksum mismatch. Our earlier work [125] proposes a high-level design to isolate file system structures for fault and recovery isolation. Here, we extend that work by addressing both reliability and performance issues with a real prototype and demonstrations for various applications. Compared with these file systems, IceFS disentangles the file system into both logically and physically isolated parts. In this systematical manner, the fault detection and recovery can be more independent and localized.

Many ideas for tolerating system crashes have been introduced at different levels. Microrebooting [43] partitions a large application into rebootable and stateless components; to recover a failed component, the data state of each component is persistent in a separate store outside of the application. Nooks [205] isolates failures of device drivers from the rest of the kernel with separated address spaces for each target driver. Membrane [203] handles file system crashes transparently by tracking resource usage and the requests at runtime; after a crash, the file system is restarted by releasing the in-use resources and replaying the failed requests. The Rio file cache [52] protects the memory state of the file system across a system crash, and conducts a warm reboot to recover lost updates. Inspired by these ideas, IceFS localizes a file system crash by microisolating the file system structures and microrebooting a cube with a simple and light-weight design. Address space isolation technique could be used in cubes for better memory fault isolation.

In addition to reliability isolation, performance isolation within file systems and SSDs also have been proposed. When running multiple workloads on a machine with fast storage devices and many cores, the contention of in-memory locks from different threads may introduce a large overhead. Similar to cubes in IceFS, Multi-lane [107] and SpanFS [108] propose to isolate I/O stacks (both in-memory and on-disk structures) for different domains; in this manner, domains will not compete for shared locks, avoiding lock scalability bottlenecks when running multiple workloads in a many core machine. Similar to metadata entanglement in file systems, data with different lifetime could be stored together in the same flash page in modern SSDs, leading to excessive garbage collection traffic. Multi-streamed SSDs [106] propose streams for applications, mapping data with different lifetimes to SSD streams. The multi-streamed SSD ensures that the data in a stream are not only written together to a physically related flash space, but also separated from data in other streams; thus, the SSD throughput and latency QoS can be significantly improved. Mapping cubes to different streams may improve performance of IceFS when running on such multi-streamed SSDs.

5.3 Key-Value Store Optimization

In this section, we discuss related work in both persistent and in-memory key-value stores. We focus on different data structures and optimization techniques used.

Various key-value stores based on hash tables have been proposed for SSD devices. FAWN [21] keeps key-value pairs in a append-only log on the SSD, and uses an in-memory hash table index for fast lookups. FlashStore [64] and SkimpyStash [65] follow the same design, but optimize the in-memory hash table; FlashStore uses cuckoo hashing and compact key signatures, while SkimpyStash moves a part of the table to the SSD using linear chaining. BufferHash [20] uses multiple in-memory hash tables, with bloom filters to choose which hash table to use for a lookup. SILT [122] is highly optimized for memory, and uses a combination of log-structure, hash-table, and sorted-table layouts; the lookup is served with a hash table and a trie in memory. WiscKey shares the log-structure data layout with these key-value stores. However, these stores use hash tables for indexing, and thus do not support modern features that have been built atop LSM-tree stores, such as range queries or snapshots. WiscKey instead targets a feature-rich key-value store which can be used by various modern applications and environments.

Much work has gone into optimizing the original LSM-tree key-value store [157]. bLSM [187] presents a new merge scheduler to bound write latency, thus maintaining a steady write throughput, and also uses bloom filters to improve performance. VT-tree [190] avoids sorting any previously sorted key-value pairs during compaction, by using a layer of indirection. However, the performance of VT-trees depends on the key ranges of the SSTable files. WiscKey instead directly separates values from keys, significantly reducing write amplification regardless of the key distribution in the workload. LOCS [215] exposes internal flash channels to the LSM-tree key-value store, which can exploit the abundant parallelism for a more efficient compaction. Atlas [114] is a distributed key-value store based on ARM processors and erasure coding, and stores keys and values on different hard drives. WiscKey is a standalone key-value store, where the separation between keys and values is highly optimized for SSD devices to achieve over 100$\times$ performance gains. LSM-trie [224] uses a trie structure to organize keys, and proposes a more efficient compaction based on the trie; however, this design sacrifices LSM-tree features such as efficient support for range query. RocksDB, described previously, still exhibits high write amplification due to its design being fundamentally similar to LevelDB; RocksDB's optimizations are orthogonal to WiscKey's design.

Walnut [51] is a hybrid object store which stores small objects in a

LSM-tree and writes large objects directly to the file system. IndexFS [173] stores its metadata in a LSM-tree with the column-style schema to speed up the throughput of insertion. Purity [57] also separates its index from data tuples by only sorting the index and storing tuples in time order. All three systems use similar techniques as WiscKey. However, we solve this problem in a more generic and complete manner, and optimize both load and lookup performance for SSD devices across a wide range of workloads.

Key-value stores based on data structures other than the LSM-tree and hash table have also been proposed. TokuDB [28, 39] is based on fractal-tree indexes, which buffer updates in internal nodes; the keys are not sorted, and a large index has to be maintained in memory for good performance. BetrFS [100] is a file system built on top of TokuDB in the kernel to get high performance for small updates. ForestDB [17] uses a HB+-trie to efficiently index long keys, improving the performance and reducing the space overhead of internal nodes. NVMKV [136] is a FTL-aware key-value store which uses native FTL capabilities, such as sparse addressing, and transactional supports. Since these key-value stores are based on different data structures, they each have different trade-offs relating to performance; instead, WiscKey proposes improving the widely used LSM-tree structure.

Many proposed techniques seek to overcome the scalability bottlenecks of in-memory key-value stores, such as Mastree [135], MemC3 [73], Memcache [152], MICA [123] and cLSM [80]. Mastree [135] is a in-memory key-value database designed for SMP machines. Its main data structure is a trie-like concatenation of B+-trees, each of which handles a fixed-length slice of a variable-length key. MemC3 [73] optimizes for read-mostly workloads. It proposes a new hashing scheme, optimistic cuckoo hashing, that achieves high space occupancy and concurrency. cLSM [80] tries to improve the scalability of LSM-tree based key-value stores on multicore servers. It presents an algorithm for scalable concurrency in LSM-trees, which exploits multiprocessor-friendly data structures and non-blocking synchronization. WiscKey focuses on improving the performance of LSM-trees by optimizing the data layout and I/O patterns. Once WiscKey can fully utilize the storage device bandwidth, the next step would be the scalability of in-memory structures. These scalable techniques used in in-memory key-value stores may be adapted for WiscKey to further improve its performance in future.

6 Future Work and Conclusions

Storage has becoming increasingly important to our modern world. Data is generated in an unprecedented speed, and new storage hardware is also on the rise. Under this situation, various storage systems and applications have been developed to manage data more reliably and efficiently. In this dissertation, we presented study results that help us to understand the real problems in file systems and proposed solutions that offer better reliability and performance than existing file systems and key-value stores.

We started by analyzing a large number of file system patches to understand what are the real problems in modern file systems (§2). We focused on file system bugs, performance and reliability patches. We found that there are many bugs in both new and mature file systems, and most of bugs may lead to serious consequences (data corruption or system crashes). However, there is not enough isolation within a file system to cope with corruption, crashes and performance problems. Therefore, we presented our solution IceFS, a new file system that separates physical structures of the file system for better reliability and performance isolation (§3). In addition to file systems, we also found that physical entanglement of keys and values in LSM-trees can lead to large I/O amplification and disappointed performance in fast SSDs. We proposed WiscKey, a novel LSM-tree based key-value store with a performance-oriented data layout that separates keys and values to minimize I/O amplification(§4).

In this chapter, we first summarize our contribution of this dissertation (§6.1). We then describe a set of general lessons we have learned in the course of this dissertation work (§6.2). Finally, we discuss several possible future research directions (§6.3).

6.1 Summary

This dissertation is mainly comprised of two parts. In the first part, we analyzed and studied file-system patches to understand the real problems of modern file systems. In the second part, we built a file system and a key-value store with physical separation techniques for better reliability and performance. We summarize each part in turn.

6.1.1 File System Study

In the first part of the dissertation, we performed the first comprehensive study of the evolution of Linux file systems. First, we investigate the overview of file-system patches. We found that nearly half of total patches are for code maintenance and documentation. The remaining dominant category is bugs, existing in both new and mature file systems. Interestingly, file-system bugs do not diminish despite the stability. We also found that bug patches are generally small while feature patches are significantly larger.

Second, we further analyzed bugs in detail. We found that semantic bugs are the dominant bug category (over 50% of all bugs), which are hard to detect via generic bug detection tools. Concurrency bugs are the next most common (about 20% of bugs), more prevalent than in user-level software. The remaining bugs are split relatively evenly across memory bugs and improper error-code handling. Unfortunately, most of the bugs we studied lead to crashes or corruption, and hence are quite serious. We also made an important discovery that nearly 40% of all bugs occur on failure-handling paths.

Finally, we also studied performance and reliability patches. The performance techniques used were relatively common and widespread. About a quarter of performance patches reduced synchronization overheads. Reliability techniques seemed to be added in a rather ad hoc fashion. Inclusion of a broader set of reliability techniques could harden all file systems.

6.1.2 Physical Separation in Storage Systems

In the second part of this dissertation, we presented our new physical separation techniques in two important types of storage systems: file systems and key-value stores. By building IceFS and WiscKey, we demonstrated that physical separation is a useful and practical technique, which can lead to significantly better reliability and performance for various workloads and environments.

First, we proposed IceFS, a novel file system that separates physical structures of the file system for better isolation. A new abstraction, the cube, was provided to enable the grouping of files and directories inside a physically isolated container. To realize disentanglement, IceFS was built upon three core principles: no shared physical resources, no access dependencies, and no bundled transactions among cubes. IceFS ensured that files and directories within cubes are physically distinct from files and directories in other cubes; thus data and I/O within each cube is disentangled from data and I/O outside of it.

We showed three major benefits of cubes within IceFS: localized reaction to faults, fast recovery, and concurrent file-system updates. We showed how cubes enable localized micro-failures; crashes and read-only remounts that normally affect the entire system are now constrained to the faulted cube. We also showed how cubes permit localized micro-recovery; instead of an expensive file-system wide repair, the disentanglement found at the core of cubes enables IceFS to fully (and quickly) repair a subset of the file system (and even do so online). In addition, we illustrated how transaction splitting allows the file system to commit transactions from different cubes in parallel, greatly increasing performance (by a factor of 2× - 5×) for some workloads. Furthermore, we conducted two cases studies where IceFS is used to host multiple virtual machines and is deployed as the local file system for HDFS data nodes. IceFS achieved fault isolation and fast recovery in both scenarios, proving its usefulness in modern storage environments.

Second, we presented WiscKey, an SSD-conscious persistent key-value store derived from the popular LSM-tree implementation, LevelDB. The central idea behind WiscKey was the separation of keys and values; only keys were kept sorted in the LSM-tree, while values were stored separately in a log. This simple technique can significantly reduce write amplification by avoiding the unnecessary movement of values while sorting. Furthermore, the size of the LSM-tree was also noticeably decreased, leading to better caching and fewer device reads during lookups. WiscKey retained the benefits of LSM-tree technology, including excellent insert and lookup performance, but without excessive I/O amplification.

We solved a number of reliability and performance challenges introduced by the new key-value separation architecture. First, range query (scan) performance may be affected because values are not stored in sorted order anymore. We proposed a parallel range query design to leverage the SSD's internal parallelism for better range query performance on unordered datasets. Second, WiscKey needed garbage collection to reclaim the free space used by invalid values. We introduced an online and lightweight garbage collector for WiscKey to reclaim the invalid key-value pairs without affecting the foreground workloads much. We demonstrated the advantages of WiscKey with both microbenchmarks and YCSB workloads. Microbenchmark results showed that WiscKey is 2.5× - 111× faster than LevelDB for loading a database and 1.6× - 14× faster for random lookups; similar results hold for YCSB workloads.

6.2 Lessons Learned

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

Large-scale studies are valuable. In general, studies drive system designs. Researchers and practitioners conducted numerous studies in the past to help understand system behaviors, workload patterns, and various design tradeoffs. These detailed studies can provide practical motivation and useful design guidelines for next generation systems.

For the file system study project, we found many interesting and important details after studying a large number of patches across six file systems. These details can inspire new research opportunities. For example, once we understand how file systems leak their resources, we can build a specialized tool to detect them more efficiently. Once we know how file systems crash, we can improve current systems to tolerate them more effectively. These vivid examples in the study really teach us what are the important problems in file systems.

The scale of the study is also very important. More patches we studied, more interesting patterns we found. More importantly, only after a large number of cases are studied, we can make some observations which may be statistically significant and insightful.

Conquer a large-scale study with small steps. Conducting a large-scale study is definitely time consuming. The total number of patches we studied comprehensively is about 2000 (bugs, performance and reliability patches). If we knew in advance that we need to analyze 2000 hard patches, we may feel overwhelmed and give up this project early on. The way we handled this study is starting from small beginnings.

Initially, we were just curious about Ext3's patches, since Ext3 was a stable and popular file system. We studied 5 versions of Ext3 patches, and classified them into different categories. We found the results very interesting and surprising. Then, we thought that we need to study more patches of Ext3 to better understand the broader patterns. Once we finished all 40 versions of Ext3 in Linux 2.6 series, we wondered that whether Ext4 has similar bugs and performance techniques as Ext3. After Ext4, we continued to ask that what about other types of file systems, such as Btrfs (copy-on-write) and XFS (logical journaling). In this incremental manner, we grew our study base from one file system to six popular file systems, and from less than 100 patches to over 5000 patches in total finally. It took us one and half year to finish the study. Thus, a large-scale study is still feasible and manageable. Starting from small beginnings and making consistent progress keep us interested and mentally sane in this long journey.

Research should match reality. Most previous work of bug studies and bug-finding tools focused on generic bugs (such as memory bugs, concurrency bugs and error handling bugs). However, in our study, we found that a majority of file system bugs are semantic bugs, which require file-system domain knowledge to understand and fix. Furthermore, we found that none of these semantic bugs was detected by existing tools, such as Coverity. This striking gap between research and reality demonstrates the importance of finding and solving real problems.

In our study, we advocated that new tools are highly desired for semantic bugs in file systems. We also suggested that more attention may be required to make failure paths more correct. Fortunately, following research papers [146, 229] proposed solutions for these two real problems.

History repeats itself. We observed that similar mistakes happened again and again, within a single file system and across different file systems. Developers of new file systems even borrowed bug-fixing solutions from patches of old file systems. We also found that similar performance and reliability techniques were used across file systems in different time when performance bottlenecks were detected or data corruption occurred.

System researchers and developers should not only focus on innovative designs for future systems, but also respect the history of existing systems widely deployed and researched. Learning from these rich history will never be wasting of time; instead, it will provide new insights and experiences of what worked and what not. More importantly, the same mistakes can be avoided at the first place. We should pay more attention to system histories, learn from them, and build a correct, high-performance and robust next generation systems from the beginning.

Inspiration can come from a different area. After we finished the file system study, we were looking for what to work next. During that period, we occasionally read a security paper from our colleague [211], which solved the problem of performance interference among virtual machines co-located in the same physical machine in the cloud. Since there was little isolation for VMs from different users, it is possible to slow down other VMs co-located in the same machine by running a carefully design workload from the attacker VM.

Even though this is a security paper, it immediately inspired us to think about isolation for VMs or users on the same machine in the context of file systems. Since we just finished the file system study, we knew that file systems have many bugs, and bugs cause data corruption and system crashes, which will affect all VMs or users relying on the same file system. An interesting research question for us is that how can we isolate reliability interference in the file system layer. This is how we started the IceFS project. Later, we also extended IceFS to handle the performance interference in the file system by isolating transactions.

Ideas from a different area may help you think about your research in a new perspective. Viewing problems in a different angle is hard without any new source of inputs. These small and random kicks from other research areas may inspire you in unexpected ways and play an important role for new research ideas.

Don’t settle for existing abstraction. Files and directories are basic and long-standing abstraction provided by file systems. However, these logical entities are an illusion; the underlying physical entanglement in file system data structures and transactional mechanisms does not provide true isolation. To provide true isolation within a file system, we proposed a new abstraction, called cube. This new abstraction connects the logical abstraction provided to users and the underlying physical structures on disk. Based on this new abstraction, it is straightforward for us to design an isolation file system which can provide both reliability and transaction performance isolation.

New abstraction fosters new research. If a research problem cannot be solved by existing abstraction, a new abstraction may be required. New abstraction should be simple and easy to use.

Isolation should be a fundamental design goal. When we analyzed the shared failures and bundled performance in file systems, we found the root cause is the entanglement of on-disk structures and in-memory transactions for different files. In other words, file systems were not designed to provide isolation at the first place. To provide better data locality, metadata from multiple files is stored together in the same disk block. To provide better I/O performance, updates from different files are batched in the same transaction. Data locality and I/O performance were the main goals when designing file systems. However, isolation was omitted as a fundamental design goal.

Isolation is becoming more important in new environments. As the workloads of the world move to the cloud, as the computing moves to virtual machines and containers, as the multi-tenant world becomes the only world we will live in, isolation is the key property to give us the illusion that we have our own machines. We should rethink systems underneath our applications at the very basic levels, both in terms of data layouts and I/O patterns. We should design systems with strong isolation for well-defined boundaries from the beginning.

Don't put old software in new hardware. Originally, LSM-trees were designed for machines with hard drives and a small number of cores. As long as the write and read amplification are smaller than 1000, LSM-trees are good enough for a wide range of workloads. With the rise of SSDs on modern servers, while replacing an HDD with an SSD underneath an LSM-tree does improve performance, the SSD's true potential goes largely unrealized with the old LSM-trees as we demonstrated in Chapter **??**.

We need to evolve old software for new hardware. There are lots of progresses on building storage systems in last several decades. Virtually, all of those intelligence was based on hard drives. Recently, we really transition our storage system to flash based devices. We should re-evaluate systems designed before, leverage things worked well, and optimize them further to leverage the new hardware. In WiscKey, we leverage the good parts of LSM-trees (such as sequential I/O patterns and rich features), and further optimize it in new ways for SSDs to get the best of two worlds.

Work on systems extremely slow or unreliable. At the conference Usenix FAST 2009, Marshall Kirk McKusick said that the reason he worked on FFS (Fast File System) [141] was that the default UNIX file system that time only utilized 2% of the device bandwidth. There was a huge room to improve it. Therefore, FFS was proposed and it can reach 47% of the device bandwidth, more than 20$\times$ of the baseline.

WiscKey is also such an example. When our experiments showed that LevelDB has high I/O amplification, and it can only utilize about 1% of the SSD device bandwidth, we felt that it is a great opportunity to make LevelDB significantly faster. After an array of new designs and optimization, WiscKey can be over 100$\times$ faster than LevelDB.

We believe that when choosing what to work on, try to choose an existing system which is extremely slow or unreliable. More opportunities lie at these corners.

6.3 Future Work

Non-volatile memory devices are at the rise. NAND-flash based solid state disks (SSD) [46, 82], phase-change memory (PCM) [45, 58] and memristor [198] provide microsecond level latency and high internal data parallelism, which can greatly boost application performance. This revolution of storage technologies is transforming the state-of-art of the storage hardware and software.

However, the entire storage stack and many key-value stores are designed for an ancient technology: the classic (and slow) hard drive [44, 97, 111, 141, 188, 223]. Various designs and architectures are based on assumptions of slow I/O bottleneck in the system. Simply replacing hard disks with fast SSDs will probably not achieve the full performance benefits with current system software.

In addition, commodity servers contain an increasing number of computing cores. Servers with tens to hundreds of cores are available already [11]. Trends indicate that the number of cores within a single machine will continue to increase in future [35]. The cache and memory capacity also increases with the number of cores for balanced performance; it is not uncommon that a single server machine contains over 100 GB of DRAM for high performance [56, 63].

We believe these two hardware trends (fast storage devices and many cores) will continue in foreseen future. Our vision is to build highly scalable storage stack and applications. In this section, we discuss several directions for such vision.

6.3.1 Scalable Virtual File System (VFS)

VFS is the entrance of all system calls. All the generic file system structures are maintained in VFS, such as inode, super block, and dentry. VFS also maintains metadata and file data caches for fast accesses: inode cache, dentry cache and page cache. When reading or writing a file, the related cached structures also need to be updated. We are interested in exploring several potential scalability bottlenecks in VFS.

First, VFS uses spin locks to protect the global inode hash table and dentry LRU list. Frequent insertion and deletion of inodes or directory entries may trigger these synchronization bottlenecks. To solve this challenge, we propose to decompose the global shared structures into multiple smaller ones. A new partition domain (e.g., a similar abstraction as cube) can be introduced in VFS to isolate these generic structures; thus, the big lock can be avoided.

Second, directories in file systems are organized as a tree hierarchy both logically and physically. To access a file `/home/bob/research/paper/foo`, all the directory entries from the root to the target file must be parsed. During the directory traversal, if multiple threads access different files in deep directories, then all the dentries and paths along the directory path will be frequently locked and released. This style of directory hierarchy dependency may affect scalability across many cores. Furthermore, if each directory contains many entries, then the traversal process may require a fair number of I/Os. Even though efficient lookup structures, such as Btree, are helpful to reduce the overhead of lookup, the fundamental bottleneck still exist because of the directory hierarchy.

We should decouple the logical hierarchy and the physical layout of directories for better scalability. We propose to build a directory hierarchy over a object-based store. In this manner, given a pathname, the file system only needs to lookup the corresponding object on disk, instead of parsing all the entries along the path.

6.3.2 Scalable Local File Systems

Many popular Linux file systems, such as Ext3, Ext4 and XFS were designed decades ago. The scalability of local file systems on new hardware is also very interesting to explore.

First, file systems use a wide range of synchronization methods to protect their internal metadata, such as spin locks, read/write locks and mutexes. These lock primitives are also used extensively in kernel for shared data structures. Developers make constant efforts to optimize existing locks for better concurrency [126], such as removing unnecessary locks, using finer-grained locking instead of big locks, and replacing write locks with RCU.

However, these techniques do not consider the scalability issue on many cores. Typical locks used in Linux may not be scalable across cores due to the cache coherence protocol [63]. It may be the worth effort to adopt more complex and scalable locks in file systems [75]. Furthermore, we could change current locking granularity for better concurrency. For example, to update a file, the inode lock is required for exclusive accesses. Two threads may update different pieces of metadata or data of the same file concurrently. Thus, fine-grained locking may be more scalable for certain workloads.

Second, transaction management in file systems can cause significant performance degradation for certain workloads. For example, Ext3/4 maintain only one running transaction, and use one single thread to flush the buffered transaction to disk periodically or as triggered by `fsync()`. For a multiple-thread write intensive workload, the journaling layer can block the applications due to the serialization in the transaction layer.

We propose to parallelize the journaling layer for better scalability. First, we need to maintain multiple running transactions in memory to buffer independent updates. As a result, updates from applications should be classified in some way to be independent from each other. Second, during commit, we will use multiple threads to commit transactions in parallel. Third, a shared physical journal or multiple physical journals should both work, since the bottleneck is not supposed to be at the device level.

Third, storage device locality may be irrelevant, if there is little difference between random and sequential performance on fast devices. However, most existing allocation or scheduling algorithms are optimized for data locality, which could limit the concurrency of file systems. For example, to allocate data blocks for a file in Ext3 or Ext4, the block group of its parent directory is preferred. If multiple threads allocate blocks for files under the same parent directory, a shared bitmap will be updated from multiple cores, limiting concurrency.

We propose randomized algorithms to replace traditional locality based algorithms in file systems. A randomized allocation design can spread the metadata updates uniformly across the whole device. We will revisit all the locality-based algorithms in file systems, and replace them for better scalability if possible.

6.3.3 Scalable Block Layer

The block layer is below the local file system. It is responsible for scheduling the low level I/O requests to the storage device. We are interested in two scalability issues in the block layer.

First, I/O schedulers usually store many pending requests in queues before dispatching them, such as CFQ and Deadline. One potential bottleneck is the shared request queue, which is used to buffer all the incoming I/O requests. The shared queue lock is required when the block layer does request insertion, request merging, fairness scheduling, I/O accounting, and request deletion. This single point processing could be a scalability bottleneck for fast devices [32]. We are curious to further explore other structures and algorithms in the block layer which can slow down I/O request processing.

Second, the block layer also sends device cache flush requests from file systems to the underlying device drivers. For example, a `fsync()` request from an application could force a device cache flush for durability of its data. However, cache flush is expensive, and this could cause slowdown for applications. For a highly parallel application, there may be many cache flush requests from different threads. We are interested in investigating new techniques to smartly schedule these cache flush requests for both durability and scalability.

6.3.4 Scalable Key-Value Stores

LSM-trees were designed for machines with hard drives, and a small number of cores. In WiscKey, we optimize LSM-trees for SSDs by separating keys and values. There are many other aspects we can further improve WiscKey for SSDs, large memory and many cores.

In WiscKey, the garbage collection is done by a single background thread. The thread reads a chunk of key-value pairs from the tail of the vLog file; then for each key-value pair, it checks the LSM-tree for validity; finally, the valid key-value pairs are written back to the head of the vLog file. We can improve the garbage collection in two ways. First, lookups in the LSM-tree are slow since multiple random reads may be required. To speedup this process, we can use multiple threads to do the lookup concurrently for different key-value pairs. Second, we can make garbage collection more effective by maintaining a bitmap of invalid key-value pairs in the vLog file. When the garbage collection is triggered, it will first reclaim the chunk with the highest percentage of free space.

Another interesting direction to scale LevelDB or WiscKey is sharding the database. Many components of LevelDB are single-threaded due to a single shared database. As we discuss before, there is a single memtable to buffer writes in memory. When the memtable is full, the foreground writes will be stalled until the compaction thread flushes the memtable to disk. In LevelDB, only a single writer can be allowed to update the database. The database is protected by a global mutex. The background compaction thread also needs to grab this mutex when sorting the key-value pairs, competing with the foreground writes. For a multiple writer workload, this architecture can unnecessarily block concurrent writes. One solution is to partition the database and related memory structures into multiple smaller shards. Each shard's keys will not overlap with others. Under this design, writes to different key-value ranges can be done concurrently to different shards. A random lookup can also be distributed to one target shard, without searching all shards. This new design may make lookups faster because of a smaller dataset to search.

6.4 Closing Words

Digital data is universal and essential in many aspects of our lives and businesses. With the increasing amount of data generated and the rise of new hardware, accessing data both reliably and efficiently become critical for modern storage systems.

In this dissertation, we began our journey with a comprehensive study of Linux file systems evolution, to better understand the reliability and performance problems that plague existing systems for decades. Then, we demonstrated that physical separation of fundamental data structures in file systems and key-value stores can provide isolated reliability and significantly better performance. This is extremely important for our changing storage world, which becomes virtualized, multi-tenant and failure-prone. We hope that this dissertation can serve as a simple but detailed example for researchers and system builders to rethink the fundamental data layouts and I/O patterns of existing systems, leverage past valuable experiences, and embrace new hardware and optimization for a better next generation storage system.

Bibliography

[1] Apache HBase. `https://hbase.apache.org/`.

[2] Docker: The Linux Container Engine. `https://www.docker.io`.

[3] Filebench. `http://sourceforge.net/projects/filebench`.

[4] Firefox 3 Uses fsync Excessively. `https://bugzilla.mozilla.org/show_bug.cgi?id=421482`.

[5] Fsyncers and Curveballs. `http://shaver.off.net/diary/2008/05/25/fsyncers-and-curveballs/`.

[6] HBase User Mailing List. `http://hbase.apache.org/mail-lists.html`.

[7] Linux Containers. `https://linuxcontainers.org/`.

[8] Solving the Ext3 Latency Problem. `http://lwn.net/Articles/328363/`.

[9] SQLite. `https://sqlite.org`.

[10] 2013 Study on Data Center Outages. Research Report, Independently conducted by Ponemon Institute, 2013.

[11] SeaMicro: High Density, Low Power Fabric Compute. `http://www.seamicro.com`, May 2014.

[12] The Digital Universe Driving Data Growth in Healthcare. EMC White Paper, 2014.

[13] Fusion-IO ioDrive2. `http://www.fusionio.com/products/iodrive2`, 2015.

[14] Riak. `http://docs.basho.com/riak/`, 2015.

[15] RocksDB Blog. `http://rocksdb.org/blog/`, 2015.

[16] 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 *Proceedings of The 31st IEEE International Conference on Computer Communications (INFOCOM '12)*, Orlando, FL, March 2012.

[17] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *IEEE Transactions on Computers*, Preprint, May 2015.

[18] Amazon. Amazon Elastic Block Store (EBS). http://aws.amazon.com/ebs/, 2012.

[19] Amazon. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/, 2012.

[20] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and Large CAMs for High-performance Data-intensive Networked Systems. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI '10)*, San Jose, California, April 2010.

[21] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[22] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.

[23] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)*, Santa Clara, California, July 2015.

[24] 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.

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

[26] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[27] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav HarâŁ™El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[28] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan Fogel, Bradley Kuszmaul, and Jelani Nelson. Cache-Oblivious Streaming B-trees. In *Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '07)*, San Diego, California, June 2007.

[29] 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.

[30] Steve Best. JFS Overview. `http://jfs.sourceforge.net/project/pub/jfs.pdf`, 2000.

[31] Eric J. Bina and Perry A. Emrath. A Faster fsck for BSD Unix. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '89)*, San Diego, California, January 1989.

[32] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR '13)*, Haifa, Israel, June 2013.

[33] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

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

[35] Shekhar Borkar. Thousand Core Chips: A Technology Perspective. In *Proceedings of the 44th Annual Design Automation Conference (DAC '07)*, San Deigo, California, June 2007.

[36] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[37] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)*, Boston, Massachusetts, June 2010.

[38] Florian Buchholz. The structure of the Reiser file system. `http://homes.cerias.purdue.edu/~florian/reiser/reiserfs.php`, January 2006.

[39] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. On External Memory Graph Traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, San Francisco, California, January 2000.

[40] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[41] Bugzilla. Kernel Bug Tracker. http://bugzilla.kernel.org/, September 2012.

[42] Calton Pu and Tito Autrey and Andrew Black and Charles Consel and Crispin Cowan and Jon Inouye and Lakshmi Kethana and Jonathan Walpole and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

[43] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 31–44, San Francisco, California, December 2004.

[44] 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.

[45] 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.

[46] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 217–228, Washington, DC, March 2009.

[47] CDW. Silver Linings and Surprises: CDW's 2013 State of The Cloud Report. http://webobjects.cdw.com/webobjects/media/pdf/CDW-2013-State-Cloud-Report.pdf, 2013.

[48] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, Washington, November 2006.

[49] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA-11)*, San Antonio, Texas, February 2011.

[50] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, October 2015.

[51] Jianjun Chen, Chris Douglas, Michi Mutsuzaki, Patrick Quaid, Raghu Ramakrishnan, Sriram Rao, and Russell Sears. Walnut: A Unified Cloud Object Store. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, Scottsdale, Arizona, May 2012.

[52] 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.

[53] 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.

[54] 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.

[55] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

[56] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[57] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, Melbourne, Victoria, June 2015.

[58] 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.

[59] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!âŁ™s Hosted Data Serving Platform. In *Proceedings of the VLDB Endowment (PVLDB 2008)*, Auckland, New Zealand, August 2008.

[60] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, Indiana, June 2010.

[61] Enrico Costanza, Sarvapali D. Ramchurn, and Nicholas R. Jennings. Understanding Domestic Energy Consumption Through Interactive Visualisation: A Field Study. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp '12)*, Pittsburgh, PA, September 2012.

[62] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, California, November 1994.

[63] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[64] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. In *Proceedings of the 36th International Conference on Very Large Databases (VLDB 2010)*, Singapore, September 2010.

[65] 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 (SIGMOD '11)*, Athens, Greece, June 2011.

[66] 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.

[67] 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.

[68] 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.

[69] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[70] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[71] Facebook. RocksDB. `http://rocksdb.org/`, 2013.

[72] Facebook. RocksDB 2015 H2 Roadmap. `http://rocksdb.org/blog/2015/rocksdb-2015-h2-roadmap/`, 2015.

[73] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, Illinois, April 2013.

[74] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A Study of the Internal and External Effects of Concurrency Bugs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '10)*, Chicago, USA, June 2010.

[75] Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. Mellor-Crummey, John M. and Scott, Michael L. *ACM Transactions of Computer Systems*, 9(1), February 1991.

[76] FSDEVEL. Linux Filesystem Development List. http://marc.info/?l=linux-fsdevel, September 2012.

[77] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

[78] Garth A. Gibson and R. Van Meter. Network Attached Storage Architecture. *Communications of the ACM*, 43(11), Nov 2000.

[79] Manfred Gilli, Evis Kellezi, and Hilda Hysi. A Data-driven Optimization Heuristic for Downside Risk Minimization. *The Journal of Risk*, 8(3), 2006.

[80] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the EuroSys Conference (EuroSys '15)*, Bordeaux, France, April 2015.

[81] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium (Sec '96)*, San Jose, California, 1996.

[82] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*, New York, New York, December 2009.

[83] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, November 2014.

[84] Haryadi S. Gunawi, Cindy Rubio-Gonzalez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 207–222, San Jose, California, February 2008.

[85] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006)*, Melbourne, Australia, Nov 2006.

[86] Donald J. Haderle and Robert D. Jackson. IBM Database 2 Overview. *IBM Systems Journal*, 23(2):112–125, 1984.

[87] Derrick Harris. http://gigaom.com/cloud/vmware-the-software-defined-data-center-is-coming, May 2012.

[88] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, California, February 2014.

[89] 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.

[90] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference*, Berkeley, CA, January 1994.

[91] 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.

[92] C.A.R. Hoare. Monitors: An Operating System Structuring Construct. *Communications of the ACM*, 17(10), October 1974.

[93] Micha Hofri. Disk scheduling: FCFS vs.SSTF revisited. *Communications of the ACM*, 23(11):645–653, 1980.

[94] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[95] Jian Huang, Xuechen Zhang, and Karsten Schwan. Understanding Issue Correlations: A Case Study of the Hadoop System. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '15)*, Kohala, Hawaii, August 2015.

[96] IBM. http://www.almaden.ibm.com/StorageSystems/file_systems/storage_tank/index.shtml, 2004.

[97] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 117–130, Banff, Canada, October 2001.

[98] Pankaj K. Jain. Financial Market Design and the Equity Premium: Electronic Versus Floor Trading. *The Journal of Finance*, 60(6):2955–2985, 2005.

[99] Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin Goel. Application-Level Isolation and Recovery with Solitude. In *Proceedings of the EuroSys Conference (EuroSys '08)*, Glasgow, Scotland UK, March 2008.

[100] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-ColtonâŁ , Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST '15)*, Santa Clara, California, February 2015.

[101] 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.

[102] Steve Jobs, Bertrand Serlet, and Scott Forstall. Keynote Address. Apple World-wide Developers Conference, 2006.

[103] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[104] 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.

[105] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Second International System Administration and Networking Conference (SANE '00)*, May 2000.

[106] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The Multi-streamed Solid-State Drive. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '14)*, Philadelphia, PA, June 2014.

[107] Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai. MultiLanes: Providing Virtualized Storage for OS-level Virtualization on Many Cores. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, California, February 2014.

[108] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*, Santa Clara, CA, July 2015.

[109] Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File Systems Deserve Verification Too! In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems (PLOS '13)*, Farmington, PA, November 2013.

[110] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[111] Steve R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '86)*, pages 238–247, Atlanta, Georgia, June 1986.

[112] J. Zico Kolter and Matthew J. Johnson. REDD: A Public Data Set for Energy Disaggregation Research. In *Proceedings of the SustKDD workshop on Data Mining Applications in Sustainability*, San Diego, CA, August 2011.

[113] 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 (WWW '10)*, Raleigh, NC, April 2010.

[114] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: BaiduâĽ™s Key-value Storage System for Cloud Data. In *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST '15)*, Santa Clara, California, May 2015.

[115] Avinash Lakshman and Prashant Malik. Cassandra – A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, Big Sky Resort, Montana, Oct 2009.

[116] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST '15)*, Santa Clara, California, February 2015.

[117] Min Kyung Lee, Daniel Kusbit, Evan Metsky, and Laura Dabbish. Working with Machines: The Impact of Algorithmic and Data-Driven Management on Human Workers. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*, Seoul, Korea, April 2015.

[118] Xin Li, Michael C. Huang, , and Kai Shen. An Empirical Study of Memory Hardware Errors in A Server Farm. In *The 3rd Workshop on Hot Topics in System Dependability (HotDep '07)*, Edinburgh, UK, June 2007.

[119] 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.

[120] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have Things Changed Now? – An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Workshop on Architectural and System Support for Improving Software Dependability (ASID '06)*, San Jose, California, October 2006.

[121] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '05)*, Lisbon, Portugal, September 2005.

[122] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[123] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, Washington, April 2014.

[124] LKML. Linux Kernel Mailing List. http://lkml.org/, September 2012.

[125] Lanyue Lu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Fault Isolation And Quick Recovery in Isolation File Systems. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, CA, June 2013.

[126] 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.

[127] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. *;login: The USENIX Magazine*, 38(3), June 2013.

[128] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. *ACM Transactions on Storage*, 10(1), Feb 2014.

[129] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST '16)*, Santa Clara, California, February 2016.

[130] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical Disentanglement in a Container-Based File System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

[131] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[132] C. Lumb, J. Schindler, G.R. Ganger, D.F. Nagle, and E. Riedel. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 87–102, San Diego, California, October 2000.

[133] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The Fast File System Checker. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[134] Haohui Mai and Jing Zhao. Scaling HDFS to Manage Billions of Files with Key Value Stores. In *The 8th Annual Hadoop Summit*, San Jose, California, Jun 2015.

[135] Yandong Mao, Eddie Kohler, and Robert Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the EuroSys Conference (EuroSys '12)*, Bern, Switzerland, April 2012.

[136] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)*, Santa Clara, California, July 2015.

[137] Cathy Marshall. "It's like a fire. You just have to move on": Rethinking Personal Digital Archiving. Keynote at FAST 2008, February 2008.

[138] Chris Mason. The Btrfs Filesystem. `oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf`, September 2007.

[139] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Alex Tomas Andreas Dilge and, and Laurent Vivier. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[140] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S.A.S. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[141] 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.

[142] Marshall Kirk McKusick. Running 'fsck' in the Background. In *Proceedings of BSDCon 2002 (BSDCon '02)*, San Fransisco, California, February 2002.

[143] 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.

[144] R. McMillan. Amazon Blames Generators For Blackout That Crushed Netflix. `http://www.wired.com/wiredenterprise/2012/07/amazonexplains/`, 2012.

[145] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[146] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[147] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[148] C. Mohan, Bruce Lindsay, and Ron Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[149] Sean Morrissey. *iOS Forensic Analysis: for iPhone, iPad, and iPod Touch*. Apress, 2010.

[150] Kannan Muthukkaruppan. Storage Infrastructure Behind Facebook Messages. In *Proceedings of International Workshop on High Performance Transaction Systems (HPTS '11)*, Pacific Grove, California, October 2011.

[151] NetApp. Overview of WAFL_check. `http://uadmin.nl/init/?p=900`, Sep. 2011.

[152] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, Illinois, April 2013.

[153] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*, Minneapolis, Minnesota, May 1994.

[154] Aaron Oord, Sander Dieleman, and Benjamin Schrauwen. Deep Content-based Music Recommendation. In *Proceedings of Twenty-seventh Annual Conference on Neural Information Processing Systems (NIPS '13)*, Lake Tahoe, CA, December 2013.

[155] Oracle. Oracle Berkeley DB C Version. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`, 1994.

[156] Oracle Inc. Consolidating Applications with Oracle Solaris Containers. `http://www.oracle.com/technetwork/server-storage/solaris/documentation/consolidating-apps-163572.pdf`, Jul 2011.

[157] Patrick OâĽ™Neil, Edward Cheng, Dieter Gawlick, and Elizabeth OâĽ™Neil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[158] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the EuroSys Conference (EuroSys '08)*, Glasgow, Scotland UK, March 2008.

[159] Nicolas Palix, Gael Thomas, Suman Saha, Christophe Calves, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*, Newport Beach, California, March 2011.

[160] 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.

[161] 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.

[162] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, and Thomas Anderson. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

[163] 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.

[164] 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.

[165] 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.

[166] Feng Qin, Shan Lu, and Yuanyuan Zhou. Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, San Francisco, California, February 2005.

[167] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs As Allergies. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[168] Wullianallur Raghupathi and Viju Raghupathi. Big Data Analytics in Healthcare: Promise and Potential. *Health Information Science and Sysstems*, 2(3), 2014.

[169] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development*, 52(4):465–479, 2008.

[170] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, CA, June 2013.

[171] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, October 1999.

[172] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '13)*, San Jose, California, June 2013.

[173] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, New Orleans, Louisana, Nov 2014.

[174] Riverbed. Riverbed Whitewater Cloud Solutions. http://www.riverbed.com/us/products/cloud_products/whitewater.php, 2011.

[175] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree Filesystem. Technical Report RJ10501, IBM Research Report, July 2012.

[176] 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.

[177] Michael Rubin. File Systems in the Cloud. In *Linux Foundation Collaboration Summit (LFCS '11)*, San Francisco, CA, April 2011.

[178] Cindy Rubio-Gonzalez, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error Propagation Analysis for File Systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, Dublin, Ireland, June 2009.

[179] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[180] Suman Saha, Julia Lawall, and Gilles Muller. Finding Resource-Release Omission Faults in Linux. In *Workshop on Programming Languages and Operating Systems (PLOS '11)*, Cascais, Portugal, October 2011.

[181] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, Cape Town, South Africa, May 2010.

[182] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: building reliable enterprise storage systems on the cheap. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, Massachusetts, October 2004.

[183] Russel Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, pages 119–130, Berkeley, CA, June 1985.

[184] Sanjay Ghemawat and Jeff Dean. LevelDB. `http://code.google.com/p/leveldb`, 2011.

[185] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[186] 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.

[187] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, Scottsdale, Arizona, May 2012.

[188] 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.

[189] Kai Shen and Stan Park. FlashFQ: A Fair Queueing I/O Scheduler for Flash-based SSDs. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, San Jose, CA, June 2013.

[190] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[191] 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.

[192] Tony Simons. First Galaxy Nexus ROM Available, Features Ext4 Support. http://androidspin.com/2011/12/06/first-galaxy-nexus-rom-available-features-ext4-support/, December 2011.

[193] Steven Sinofsky. Building the Next Generation File System for Windows: ReFS. `http://blogs.msdn.com/b/b8/archive/2012/01/16/building-the-next-generation-file-system-for-windows-refs aspx`, Jan. 2012.

[194] Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Junwen Lai, Yilie Shao, Chi Zhang, Wlisha Ziskind, and Arvind Krishnamurthy. Segank: A Distributed Mobile Storage System. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 239–252, San Francisco, California, April 2004.

[195] 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.

[196] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. In *IEEE Transactions on Knowledge and Data Engineering*, pages 340–355, 1986.

[197] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The Missing Memristor Found. *Nature*, 453(7191):80–83, 2008.

[198] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 453:80–83, 2008.

[199] Mark Sullivan and Ram Chillarege. Software Defects and their Impact on System Availability – A Study of Field Failures in Operating Systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, Montreal, Canada, June 1991.

[200] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, Montreal, Canada, June 1991.

[201] Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *Proceedings of the 22st International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 475–484, Boston, USA, July 1992.

[202] Sun Microsystems. MySQL White Papers, 2008.

[203] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Membrane: Operating System Support for Restartable File Systems. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[204] 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.

[205] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[206] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 1–16, San Francisco, California, December 2004.

[207] Nisha Talagala. *Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks*. PhD thesis, University of California at Berkeley, September 1999.

[208] Theodore Ts'o. `http://e2fsprogs.sourceforge.net`, June 2001.

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

[210] Satyam B. Vaghani. Virtual Machine File System. *ACM SIGOPS Operating Systems Review*, 44(4):57–70, Dec 2010.

[211] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M. Swift. Resource-freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12)*, Raleigh, North Carolina, October 2012.

[212] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance Isolation: Sharing and Isolation in Shared-memory Multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 181–192, San Jose, California, October 1998.

[213] VMware Inc. VMware Workstation. `http://www.vmware.com/products/workstation`, Apr 2014.

[214] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[215] 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 EuroSys Conference (EuroSys '14)*, Amsterdam, Netherlands, April 2014.

[216] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving Integer Security for Systems. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, California, October 2012.

[217] Yin Wang, Terence Kelly, Manjunath Kudlur, StÃ©phane Lafortune, and Scott Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[218] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[219] Wikipedia. IBM Journaled File System. http://en.wikipedia.org/wiki/ JFS_(file_system), September 2012.

[220] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[221] 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.

[222] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[223] 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.

[224] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)*, Santa Clara, California, July 2015.

[225] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad Hoc Synchronization Considered Harmful. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[226] 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.

[227] 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.

[228] 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 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, Szeged, Hungary, September 2011.

[229] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

[230] 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.

[231] Georgios Zervas, Davide Proserpio, and John W. Byers. The Rise of the Sharing Economy: Estimating the Impact of Airbnb on the Hotel Industry. In *Proceedings of the Sixteenth ACM Conference on Economics and Computation (EC '15)*, Portland, OR, June 2015.

[232] Yiying Zhang, Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[233] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.