A Study of Linux File System Evolution

LANYUE LU, ANDREA C. ARPACI-DUSSEAU, REMZI H. ARPACI-DUSSEAU, and SHAN LU, University of Wisconsin, Madison

We conduct a comprehensive study of file-system code evolution. By analyzing eight years of Linux file-system changes across 5079 patches, we derive numerous new (and sometimes surprising) insights into the file-system development process; our results should be useful for both the development of file systems themselves as well as the improvement of bug-finding tools.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management

General Terms: Design, Algorithms, Performance, Reliability

Additional Key Words and Phrases: File systems, patch, bug, failure, performance, reliability

ACM Reference Format:
Lu, L., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Lu, S. 2014. A study of Linux file system evolution. *ACM Trans. Storage* 10, 1, Article 3 (January 2014), 32 pages.
DOI:http://dx.doi.org/10.1145/2560012

1. INTRODUCTION

Open-source local file systems, such as Linux Ext4 [Mathur et al. 2007], XFS [Sweeney et al. 1996], and Btrfs [Mason 2007; Rodeh et al. 2012], remain a critical component in the world of modern storage. For example, many recent distributed file systems, such as Google GFS [Ghemawat et al. 2003] and Hadoop DFS [Shvachko et al. 2010], replicate data objects (and associated metadata) across local file systems. On smart phones, most user data is managed by a local file system; for example, Google Android phones use Ext4 [Kim et al. 2012; Simons 2011] and Apple's iOS devices use HFSX [Morrissey 2010]. Finally, many desktop users still do not backup their data regularly [Jobs et al. 2006; Marshall 2008]; 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 [Raymond 1999]. Major new file systems are introduced every few years [Bonwick and Moore 2007; Mason 2007; McKusick et al. 1984; Rosenblum and Ousterhout

An earlier version of this article appeared in *Proceedings of the 11th File and Storage Technologies Conference (FAST)*, USENIX Association, Berkeley, CA.
This material is based on work supported by the National Science Foundation under the following grants: CNS-1218405, CCF-0937959, CSR-1017518, CCF-1016924, CNS-1319405, as well as generous donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Samsung, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.
Authors' address: L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, Department of Computer Sciences, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706-1685; email: {ll, dusseau, remzi, shanlu}@cs.wisc.edu.

DOI:http://dx.doi.org/10.1145/2560012

1992; Sweeney et al. 1996]; with recent technology changes (e.g., Flash [Boboila and Desnoyers 2010; Grupp et al. 2009]), 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 to answer for different communities: for developers, so that they can improve current designs and implementations 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).

One way to garner insight into these questions is to study the artifacts themselves. Compared with proprietary software, open source projects provide a rich resource for source code and patch analysis. 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.

In this article, we perform the first comprehensive study of the evolution of Linux file systems, focusing on six major and important ones: Ext3 [Tweedie 1998], Ext4 [Mathur et al. 2007], XFS [Sweeney et al. 1996], Btrfs [Mason 2007; Rodeh et al. 2012], ReiserFS [Buchholz 2006], and JFS [Best 2000]. 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 (Section 3). 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%, about 1800 bugs), showing how much effort is required to slowly inch towards a "correct" implementation; perhaps this hard labor explains why some have found that the quality of open source projects is better than the proprietary software average [Coverity 2011]. Interestingly, the number of bugs does not die down over time (even for stable file systems), rather ebbing and flowing over time.

Breaking down the bug category further (Section 4), we find that semantic bugs, which require an understanding of file-system semantics to find or fix, 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 [Bessey et al. 2010; Padioleau et al. 2008]; more complex model checking [Yang et al. 2004] or formal specification [Klein et al. 2009] may be needed. Concurrency bugs are the next most common (about 20% of bugs), more prevalent than in user-level software [Li et al. 2006; Sahoo et al. 2010; Sullivan and Chillarege 1992]. Within this group, atomicity violations and deadlocks dominate. Kernel deadlocks are common (many caused by incorrectly using blocking kernel functions), hinting that recent research [Jula et al. 2008; Wang et al. 2008] might be needed in-kernel. The remaining bugs are split relatively evenly across memory bugs and improper error-code handling. In the memory bug category, memory leaks and null-pointer dereferences are common; in the error-code category, most bugs simply drop errors completely [Gunawi et al. 2008].

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; this result holds across semantic, concurrency, memory, and error code bugs. 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. File systems, when trying to react to a failed memory allocation, I/O error, or some other unexpected condition, are highly likely to make further mistakes, such as incorrect state updates and missing resource releases. These mistakes can lead to corruption, crashes, deadlocks and leaks. 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 (Section 5.1). 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 nonuniformly). Inclusion of a broader set of reliability techniques could harden all file systems.

Beyond these results, another outcome of our work is an annotated dataset of file-system patches, which we make publicly available for further study (at this URL: http://research.cs.wisc.edu/wind/Traces/fs-patch) by file-system developers, systems-language designers, and bug-finding tool builders. We show the utility of *PatchDB* by performing a case study (Section 6); specifically, we search the dataset to find bugs, performance fixes, and reliability techniques that are unusually common across all file systems. This example brings out one theme of our study, which is that there is a deep underlying similarity in Linux local file systems, even though these file systems are significantly different in nature (e.g., designs, features, and groups of developers). The commonalities we do find are good news: by studying past bug, performance, and reliability patches, and learning what issues and challenges lie therein, we can greatly improve the next generation of file systems and tools used to build them.

This article is an expansion of our earlier work [Lu et al. 2013]. We have made three major additions to our study: a new graph on bug patch size distribution (Section 3.3), a section on file-system code evolution (Section 4.1), and many new code examples and analyses for file-system bugs (Section 4.6), failure paths (Section 4.7), performance patches (Section 5.1) and reliability patches (Section 5.2).

2. 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. 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 [Tweedie

1998], Ext4 with JBD2 [Mathur et al. 2007], XFS [Sweeney et al. 1996], Btrfs [Mason 2007; Rodeh et al. 2012], ReiserFS [Buchholz 2006], and JFS [Best 2000]. Ext3, JFS, ReiserFS and XFS were all stable and in production use before the Linux 2.6 kernel. Ext4 was introduced in Linux 2.6.19 and marked stable in Linux 2.6.28. Btrfs was added into Linux 2.6.29 and is still under active development.

2.2. 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 [Bugzilla 2012] and mailing lists [FSDEVEL 2012; LKML 2012] 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 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 cause of the null-pointer dereference and the location within the code. The patch also indicates that the bug was detected with Coverity [Bessey et al. 2010].

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

```
[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;
```

Fig. 1. An example patch. This figure shows an real Ext3 patch in Linux 2.6.7.

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 [McKusick et al. 1986].

3. PATCH OVERVIEW

File systems evolve through patches. A large number of patches are discussed and submitted to mailing lists, bug-report websites, and other 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?

3.1. Patch Type

We classify patches into five categories (Table I): 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(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.

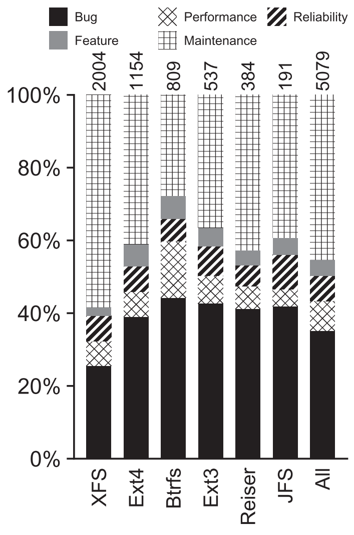
Maintenance patches are the largest group across all file systems (except Btrfs, a recent and not-yet-stable file system). These patches include changes to improve readability, simplify structure, and utilize cleaner abstractions; in general, these patches represent the necessary costs of keeping a complex open-source system well-maintained. Because maintenance patches are relatively uninteresting, we do not examine them further.

Bug patches have a significant presence, comprising nearly 40% of patches. 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 7). Because

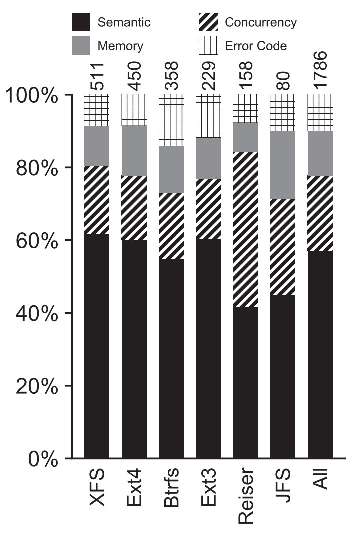
Table I. Patch Type

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)

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*).



(a) Patch Type



(b) Bug Pattern

Fig. 2. 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.

this class of patch is critical for developers and tool builders, we characterize them in detail later (Section 4).

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 (Section 5.1).

Finally, *feature* patches account for a small percentage of total patches; 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 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.

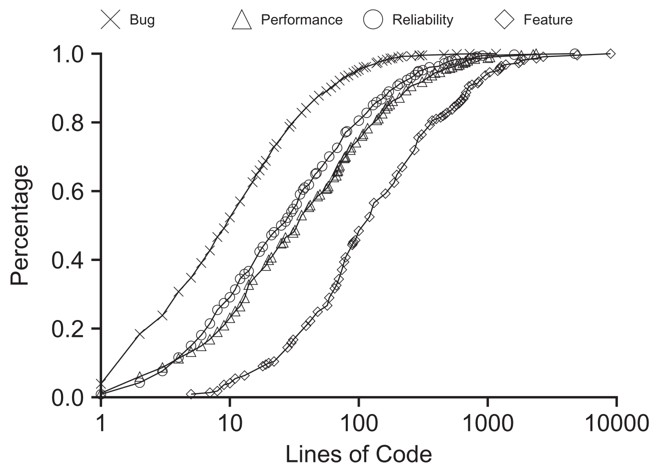


Fig. 3. Patch size. This figure shows the size distribution for different patch types, in terms of lines of modifications. The x-axis shows the lines of code in log scale; the y-axis shows the percentage of patches.

3.2. 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, 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 Section 4).

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

3.3. 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 3 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. Figure 4 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.

4. 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

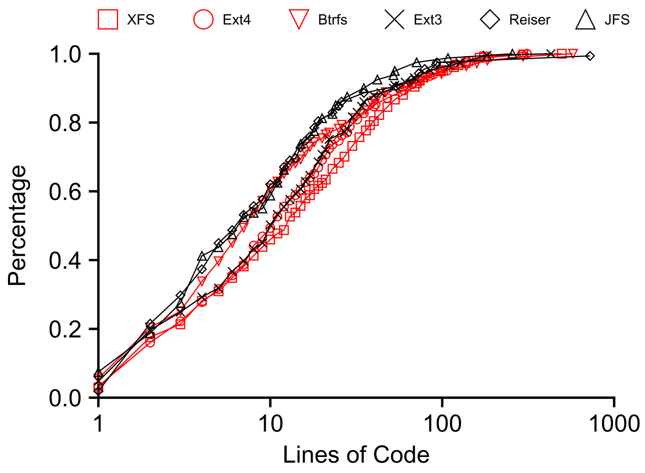


Fig. 4. Bug patch size. This figure shows the size distribution of bug patches for different file systems, in terms of lines of modifications. The x-axis shows the lines of code in log scale; the y-axis shows the percentage of patches.

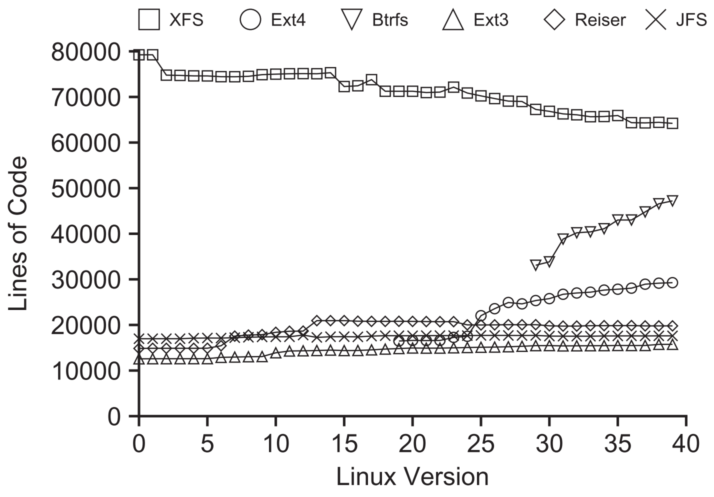


Fig. 5. File-system code evolution. This figure shows the lines of code for each file system in Linux 2.6 series. The x-axis shows 40 versions of Linux 2.6; the y-axis shows lines of code (LOC). Note that LOC of Ext3 includes the code of JBD and LOC of Ext4 includes the code of JBD2.

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.

4.1. Code Evolution

FFS had only 1200 lines of code [McKusick et al. 1984]; 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 5 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

Table II. Logical Components

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)

This table shows the classification and definition of file-system logical components. There are nine components for all file systems.

time, from nearly 80K LOC to 64K LOC. As Figure 2(a) in Section 3.1 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. 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 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 across versions while small changes arise due to major features or structure changes.

4.2. 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 II).

Figure 6 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 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

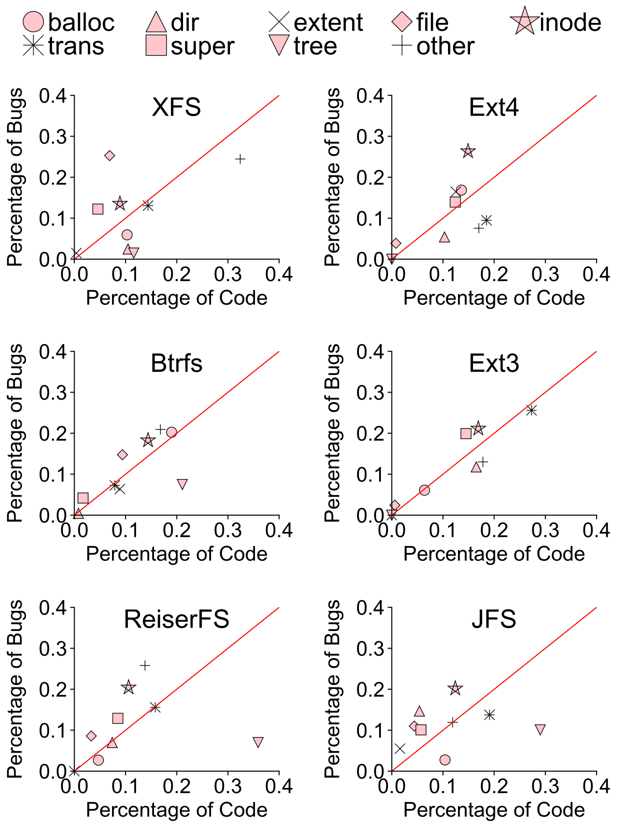


Fig. 6. File-system code and bug correlation. This figure shows the correlation between code and bugs. The x-axis shows the average percent of code of each component (over all versions); the y-axis shows the percent of bugs of each component (over all versions).

buffer cache and I/O manager for scalability [Sweeney et al. 1996]. 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.

Table III. Bug Pattern Classification

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

This table shows the classification and definition of file-system bugs. There are four big categories based on root causes and further subcategories for detailed analysis.

Although bug patches also relate to feature patches, it is difficult to correlate them precisely. Code changes partly or totally overlap each other over time. A bug patch may involve 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.

4.3. 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 III. The four major categories are *semantic* [Li et al. 2006; Sullivan and Chillarege 1991], *concurrency* [Fonseca et al. 2010; Lu et al. 2008], *memory* [Chou et al. 2001; Li et al. 2006; Sullivan and Chillarege 1991], and *error code* bugs [Gunawi et al. 2008; Rubio-Gonzalez et al. 2009].

Figure 2(b) (page 6) 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 [Bessey et al. 2010]) 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

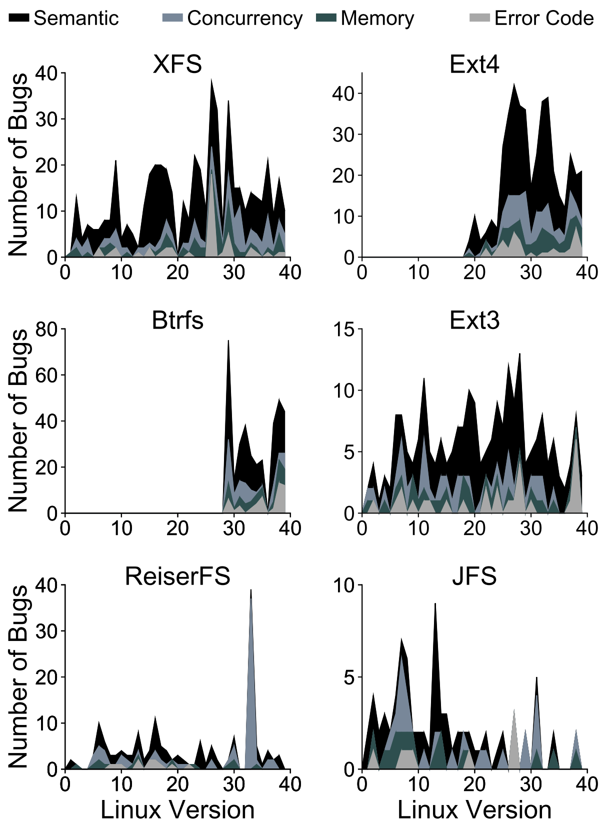


Fig. 7. Bug pattern evolution. This figure shows the bug pattern evolution for each file system over all versions.

software where fewer than 3% of bugs are concurrency-related [Li et al. 2006; Sahoo et al. 2010; Sullivan and Chillarege 1992]. 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 [Li et al. 2006; Sullivan and Chillarege 1992]. Many research and commercial tools have been developed to detect memory bugs [Bessey et al. 2010; Padioleau et al. 2008], and some of them 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.

4.4. 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?

Table IV. Bug Consequence Classification

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)

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.

Our results (Figure 7) 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 [Wikipedia 2012]. JFS and ReiserFS both have relatively small developer and user bases compared to the more active file systems XFS, Ext4 and Btrfs.

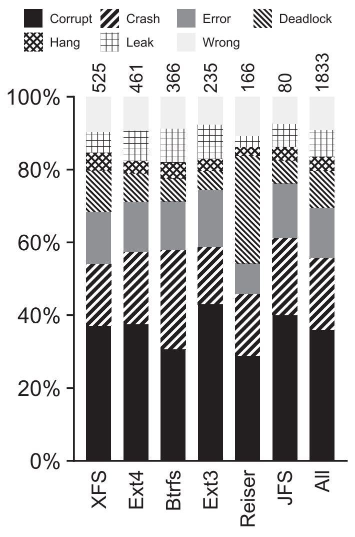
Summary. Bug patterns do not change significantly over time, increasing and decreasing cyclically; large deviations arise due to major structural changes.

4.5. Bug Consequences

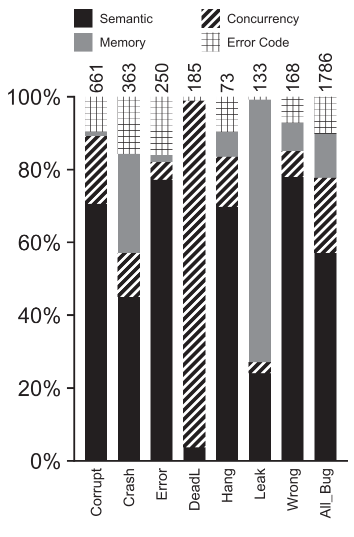
As shown in Figure 2(b) (on page 6), there are a significant number of bugs in file systems. But how serious are these file-system bugs? We now categorize each bug by impact; such *bug consequences* include severe ones (data corruption, system crashes, unexpected errors, deadlocks, system hangs and resource leaks), and other wrong behaviors. Table IV provides more detail on these categories.

Figure 8(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 [Li et al. 2006].

Given that file-system bugs are serious bugs, we were curious: do certain bug types (e.g., semantic, concurrency, memory, or error code) exhibit different levels of severity? Figure 8(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



(a) By File Systems



(b) By Bug Patterns

Fig. 8. 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(b).

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.

4.6. 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).

4.6.1. Semantic Bugs. Semantic bugs are dominant in file systems, as shown in Figure 2(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 III, including *state*, *logic*, *config*, *I/O timing* and *generic*. Figure 9(a) shows the percentage breakdown and total number of semantic bugs; each is explained in detail in the following text.

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 or access may lead to serious consequences. As shown in Figure 9(a), these *state* bugs contribute to roughly 40% of semantic bugs. An example of a *state* bug is shown in S1 of Table V, which misses an inode-field update. Specifically, the buggy

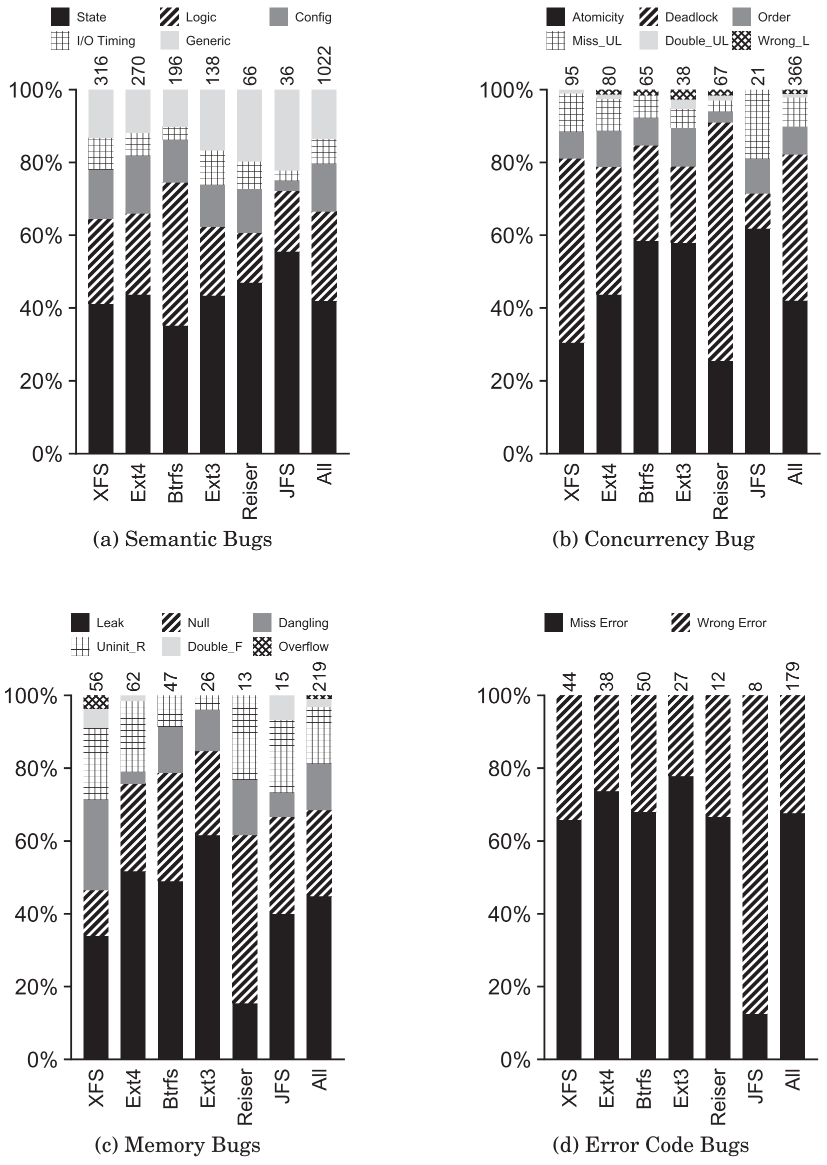


Fig. 9. Detailed bug patterns. The detailed classification for each bug pattern; total number of bugs is shown on top of each bar.

version of `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 V: `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

Table V. Semantic Bug Code Examples

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

This table shows five code examples of semantic bugs. One example for each category: *state*, *logic*, *config*, *I/O timing*, and *generic*.

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 V; when Ext4 loads the journal from disk, it forgets to check if the device is read-only before updating the on-disk 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 V. 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 [Wang et al. 2012]), and may be fixed without much file-system knowledge. A generic semantic bug is shown in S5 of Table V. In Ext3, sbi->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 semantic 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.

4.6.2. Concurrency Bugs. Concurrency bugs have attracted a fair amount of attention in the research community as of late [Fonseca et al. 2010; Jula et al. 2008; Lu et al. 2008; Wang et al. 2008; Xiong et al. 2010]. To better understand file-system concurrency bugs, we classify them into six types as shown in Table III (on page 11): *atomicity violations*, *deadlocks*, *order violations*, *missed unlocks*, *double unlocks*, and *wrong locks*.

Figure 9(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 VI. 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 VI. Btrfs uses extent_readpages() to read free space information; however, it 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 VI. 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 VI shows a missing-unlock bug. ext3_group_add() locks the super block (line 1) but forgets to unlock on an error (line 4).

The remaining two categories account for a small percentage. A *double unlock* example is shown in C5 of Table VI. The double unlock (line 2 and line 5) of XFS inode ip results in a reproduced deadlock on platforms which do not handle double unlock nicely. Using *wrong locks* also happens. For example, as shown in C6 of Table VI, 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

Table VI. Concurrency Bug Code Examples

ext4/extents.c, 2.6.30 **Atomicity (C1)**

```
   ext4_ext_put_in_cache(...){
+    spin_lock(i_block_reservation_lock);
     cex = &EXT4_I(inode)->i_cached_extent;
     cex->ec_block = block;
     cex->ec_len = len;
     cex->ec_start = start;
+    spin_unlock(i_block_reservation_lock);
```

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

```
   extent_readpages(...){
     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)**

```
   exit_ext4_mballoc(...){
+    /*
+     * Wait for completion of call_rcu()'s on ext4_pspace_cachep
+     * before destroying the slab cache.
+     */
+    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);
     err = -EBUSY;
     goto exit_journal;
```

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

```
   xfs_swap_extents(...){
     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)**

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

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*.

concurrency bugs; many deadlock bugs are caused by wrong kernel memory-allocation flags; most missing unlocks happen on exit or failure paths.

4.6.3. Memory Bugs. Memory-related bugs are common in many source bases, and not surprisingly have been the focus of many bug detection tools [Bessey et al. 2010; Padioleau et al. 2008]. We classify memory bugs into six categories, as shown in Table III: *resource leaks*, *null pointer dereferences*, *dangling pointers*, *uninitialized reads*, *double 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 [Li et al. 2006; Sahoo et al. 2010; Sullivan and Chillarege 1992]. We find that roughly 70% of resource leaks happen on exit or failure paths; we investigate this further later (Section 4.7). An example of resource leaks (M1 of Table VII) is found in `btrfs_new_inode()` which allocates an inode but forgets to free it upon failure.

As we see in Figure 9(c), *null-pointer* dereferences are also common in both mature and young file systems. An example is shown in M2 of Table VII; a return statement is missing, leading to a null-pointer dereference.

Dangling pointers have a sizeable percentage across file systems. A Btrfs example is shown in M3 of Table VII. After callling `submit_bio` at line 2, the `bio` structure can be released at any time. Checking the `bio` flags at line 3 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 VII. 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 VII shows a double free bug in Ext4. The while loop contains a `goto` statement (line 4) to `cleanup`, which also frees `b_entry_name` at line 9. To eliminate the potential double free on this error path, Ext4 sets `b_entry_name` to `NULL` at line 6.

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.

4.6.4. Error Code Bugs. File systems need to handle a wide range of errors, including memory-allocation failures, disk-block allocation failures, I/O failures [Bairavasundaram et al. 2007, 2008], and silent data corruption [Prabhakaran et al. 2005]. Handling such faults, and passing error codes through a complex code base, has proven challenging [Gunawi et al. 2008; Rubio-Gonzalez et al. 2009]. 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 III. Figure 9(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 VIII. The routine `posix_acl_from_disk()` could return an error code (line 2). However, without error checking, `acl` is accessed and thus the kernel crashes (line 3).

An example of a *wrong error* code is shown in E2 of Table VIII. `diAlloc()`'s 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.

Table VII. Memory Bug Code Examples

btrfs/inode.c, 2.6.30 **Resource Leak (M1)**

```
   btrfs_new_inode(...){
     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)**

```
   ext3_get_journal(...){
     if (!journal) {
       ... ...
+      return NULL;
     }
      journal->j_private = sb;
```

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

```
   run_scheduled_bios(...){
-    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)**

```
   ext4_expand_extra_isize_ea(...){
     while (...){
       if (error)
         goto cleanup;
       kfree(b_entry_name);
+      b_entry_name = NULL;
     }
   cleanup:
     kfree(b_entry_name);
```

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*.

4.7. The Failure Path

Many bugs we found arose not in common-case code paths but rather in more unusual fault-handling cases [Gunawi et al. 2008; Saha et al. 2011; Yang et al. 2004]. This type of error handling (i.e., reacting to disk or memory failures) is critical to robustness,

Table VIII. Error Code Bug Code Examples

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;
```

This table shows two code examples of error code bugs. One example for each category: *missing error codes* and *wrong error codes*.

Table IX. Failure Related Bugs by File System

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

This table shows the number and percentage of the bugs related to failures, partitioned by file systems.

Table X. Failure Related Bugs by Bug Pattern

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

This table shows the number and percentage of the bugs related to failures, partitioned by bug patterns.

since bugs on failure paths can lead to serious consequences. We now quantify bug occurrences on failure paths; Tables IX and Tables X 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 Ext3 and XFS make a significant number of mistakes on these rarer code 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 XI. 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 [Lu et al. 2008].

Table XI. Code Examples of Bugs on Failure Paths

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

```
1     ext4_group_extend(...) {
2       ext4_warning(sb, "multiple resizers run on filesystem!");
3       unlock_super(sb);
4 +     ext4_journal_stop(handle);
5       err = -EBUSY;
6       goto exit_put;
```

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

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

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

```
1     btrfs_new_inode(...) {
2       path = btrfs_alloc_path();
3       inode = new_inode(root->fs_info->sb);
4 -     if (!inode)
5 +     if (!inode) {
6 +       btrfs_free_path(path);
7         return ERR_PTR(-ENOMEM);
8 +     }
```

This table shows three bug code examples of bugs on failure paths. One example for each category: *semantic*, *concurrency*, and *memory*.

Such an example is shown in F2 of Table XI. `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 XI. 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 [Saha et al. 2011; Yang et al. 2004], missing unlock [Saha et al. 2011; Yang et al. 2004] and error codes [Gunawi et al. 2008; Rubio-Gonzalez et al. 2009]; 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.

Table XII. Performance Patch Type

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 suboptimal data block allocations (e.g., reducing file fragmentation, clustered I/Os)
Other	Other performance improvement techniques (e.g., reducing function stack usage)

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*.

5. PERFORMANCE AND RELIABILITY

A small but important set of patches improve performance and reliability, which are quantitatively different than bug patches (Figure 3). Performance and reliability patches account for 8% and 7% of patches respectively.

5.1. Performance Patches

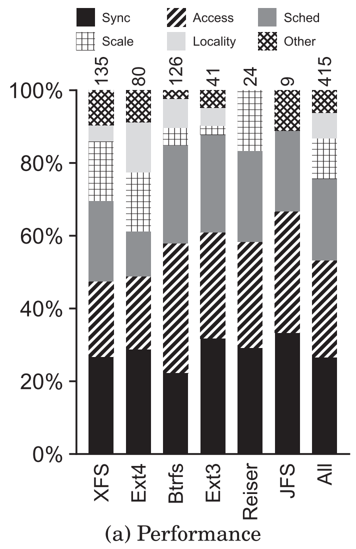
Performance is critical for all file systems. Performance patches are proposed to improve existing designs or implementations. We partition these patches into six categories as shown in Table XII, including synchronization (*sync*), access optimization (*access*), scheduling (*sched*), scalability (*scale*), locality (*locality*), and *other*. Figure 10(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 is shown in P1 of Table XIII; `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 10(a) shows *access* patches are popular. An example Btrfs *access* patch is shown in P2 of Table XIII; 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 of 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 XIII. A little mistake in a previous bug fixing patch is making all transactions synchronous, which 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



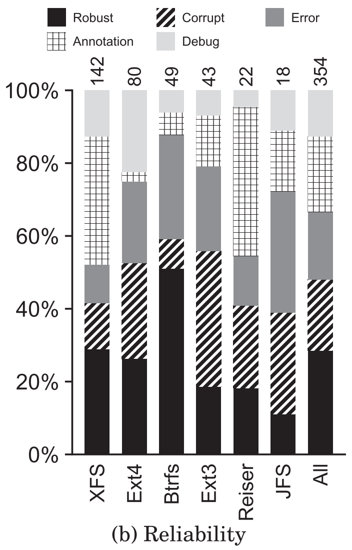


Fig. 10. Performance and reliability patches. This figure shows the *performance* and *reliability* patterns. The total number of patches is shown on top of each bar.

of Table XIII. 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 suboptimal 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 XIII. Btrfs uses larger metadata 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 XIII, 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.

5.2. Reliability Patches

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 [Gunawi et al. 2008] or add more debugging information. Table XIV 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 10(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,

Table XIII. Performance Patch Code Examples

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)**

```
   btrfs_find_space_cluster(...){
+    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)**

```
   ext3_xattr_release_block(...){
     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 {
     unsigned int i_allocated_meta_blocks;
-    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)**

```
   find_free_extent(...) {
     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_leaf_getdents(...) {
     int nmap;
-    xfs_dir2_put_args_t p;
+    xfs_dir2_put_args_t *p;
```

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

likely due to its early stage of development. A *robust* patch is shown in R1 of Table XV. 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

Table XIV. Reliability Patch Type

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

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*).

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 XV. 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 XV. 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. ReiserFS 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 nonuniform 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.

6. 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

Table XV. Reliability Patch Code Examples

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

```
  jfs_ioctl(...) {
+   /* Is it quota file? Do not allow user to mess with it */
+   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;
    }
```

This table shows the code examples of reliability patches. One example for each category: *robust*, *corruption*, and *error*.

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

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 [Engler et al. 2001; Li and Zhou 2005; Li et al. 2004]. Fortunately, we find that many semantic bugs appear across file systems, which can be leveraged to improve bug detection.

Table XVI. Common File-System Patches

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

This table shows the classification and count of common patches across all file systems.

For concurrency bugs, forgetting to lock an inode when updating it is common; perhaps a form of monitors [Hoare 1974] would help. Calling a blocking function when holding a spin lock (*lock sleep*) occurs frequently (also in drivers [Chou et al. 2001; Palix et al. 2011]). As we saw earlier (Section 4.6.2), 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 [Lu et al. 2008].

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 [Prabhakaran et al. 2005]; 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-system patches share many similarities across implementations; some examples occur quite frequently; PatchDB affords new opportunities to study such phenomena in great detail.

7. RELATED WORK

Operating-System Bugs. Faults in Linux have been studied [Chou et al. 2001; Palix et al. 2011]. Static analysis tools are used to find potential bugs in Linux 1.0 to 2.4.1 [Chou et al. 2001] and Linux 2.6.0 to 2.6.33 [Palix et al. 2011]. Most detected faults are generic memory and concurrency bugs. Both studies find that device drivers contain the most faults, while Palix et al. [2011] also show that file-system errors are rising. Yin et al. [2011] analyze incorrect bug-fixes in several operating systems. 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.

User-Level Bugs. Various aspects of modern user-level open source software bugs have also been studied, including patterns, impacts, reproducibility, and fixes [Fonseca et al. 2010; Li et al. 2006; Lu et al. 2008; Sahoo et al. 2010; Xiong et al. 2010]. 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) we study is larger than previous efforts [Fonseca et al. 2010; Li et al. 2006; Lu et al. 2008; Sahoo et al. 2010; Xiong et al. 2010].

File-System Bugs. Several research projects have been proposed to detect and analyze file-system bugs. For example, Yang et al. [2004, 2006] use model checking to detect file-system errors; Gunawi et al. [2008] use static analysis techniques to determine how error codes are propagated in file systems; Rubio-Gonzalez et al. [2009] utilize static analysis to detect similar problems; Prabhakaran et al. [2005] 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.

8. CONCLUSIONS

We performed 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). Our observations, summarized in the introduction and throughout, 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.

REFERENCES

Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. 2007. An analysis of latent sector errors in disk drives. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07).*

Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. An analysis of data corruption in the storage stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST'08).*

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM.*

Steve Best. 2000. JFS Overview. http://jfs.sourceforge.net/project/pub/jfs.pdf.

Simona Boboila and Peter Desnoyers. 2010. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST'10).*

Jeff Bonwick and Bill Moore. 2007. ZFS: The last word in file systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

Florian Buchholz. 2006. The structure of the Reiser file system. http://homes.cerias.purdue.edu/~florian/reiser/reiserfs.php.

Bugzilla. 2012. Kernel bug tracker. http://bugzilla.kernel.org/.

Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. 73–88.

Coverity. 2011. Coverity Scan: 2011 Open source integrity report. http://www.coverity.com/library/pdf/coverity-scan-2011-open-source-integrity-report.pdf.

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. 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)*. 57–72.

Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. 2010. A study of the internal and external effects of concurrency bugs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'10)*.

FSDEVEL. 2012. Linux filesystem development list. http://marc.info/?l=linux-fsdevel.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 29–43.

L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*.

Haryadi S. Gunawi, Cindy Rubio-Gonzalez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. 2008. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST'08)*. 207–222.

C. A. R. Hoare. 1974. Monitors: An operating system structuring construct. *Commun. ACM 17*, 10.

Steve Jobs, Bertrand Serlet, and Scott Forstall. 2006. Keynote address. *Apple World-Wide Developers Conference*.

Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. 2008. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI'08)*.

Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST'12)*.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*.

Zhenmin Li and Yuanyuan Zhou. 2005. 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)*.

Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. 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)*.

Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now? – An empirical study of bug characteristics in modern open source software. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID'06)*.

LKML. 2012. Linux kernel mailing list. http://lkml.org/.

Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST'13)*.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. 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)*.

Cathy Marshall. 2008. "It's like a fire. You just have to move on": Rethinking personal digital archiving. In *Proceedings of FAST'08.*

Chris Mason. 2007. The Btrfs filesystem. oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf.

Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Alex Tomas Andreas Dilge, and Laurent Vivier. 2007. The New Ext4 filesystem: Current status and future plans. In *Proceedings of the Ottawa Linux Symposium (OLS'07)*.

Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. 1984. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2*, 3, 181–197.

Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1986. Fsck - The UNIX file system check program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version.

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

Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the EuroSys Conference (EuroSys'08)*.

Nicolas Palix, Gael Thomas, Suman Saha, Christophe Calves, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten years later. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*.

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. 206–220.

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

Ohad Rodeh, Josef Bacik, and Chris Mason. 2012. *BTRFS: The Linux B-tree Filesystem*. Tech. rep. RJ10501. IBM.

Mendel Rosenblum and John Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1, 26–52.

Cindy Rubio-Gonzalez, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error propagation analysis for file systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI'09)*.

Suman Saha, Julia Lawall, and Gilles Muller. 2011. Finding resource-release omission faults in Linux. In *Proceedings of the Workshop on Programming Languages and Operating Systems (PLOS'11)*.

Swarup Kumar Sahoo, John Criswell, and Vikram Adve. 2010. 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)*.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST'10)*.

Tony Simons. 2011. First Galaxy Nexus ROM available, features Ext4 support. http://androidspin.com/2011/12/06/first-galaxy-nexus-rom-available-features-ext4-support/.

Mark Sullivan and Ram Chillarege. 1991. 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)*.

Mark Sullivan and Ram Chillarege. 1992. 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)*. 475–484.

Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference (USENIX'96)*.

Stephen C. Tweedie. 1998. Journaling the Linux ext2fs file system. In *Proceedings of the 4th Annual Linux Expo*.

Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Improving integer security for systems. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI'12)*.

Yin Wang, Terence Kelly, Manjunath Kudlur, Stphane Lafortune, and Scott Mahlke. 2008. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI'08)*.

Wikipedia. 2012. IBM Journaled file system. http://en.wikipedia.org/wiki/ JFS_(file_system).

Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad hoc synchronization considered harmful. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI'10)*.

Junfeng Yang, Can Sar, and Dawson Engler. 2006. 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)*.

Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2004. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*.

Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. 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)*.

Received May 2013; accepted June 2013