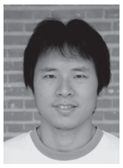
A Study of Linux File System Evolution

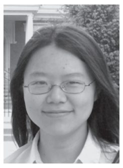
LANYUE LU, ANDREA C. ARPACI-DUSSEAU, REMZI H. ARPACI-DUSSEAU, AND SHAN LU



Lanyue Lu is a PhD student in Computer Sciences at the University of Wisconsin—Madison. His research interests include file systems, storage systems, and cloud computing. ll@cs.wisc.edu

Andrea Arpaci-Dusseau is a Professor and Associate Chair of Computer Sciences at the University of Wisconsin-Madison. Andrea co-leads a research group with her husband, Remzi Arpaci-Dusseau, and has advised 13 students through their PhD dissertations. She is currently a UW-Madison Vilas Associate and received the Carolyn Rosner "Excellent Educator" award; she has served on the NSF CISE Advisory Committee and as faculty Co-director of the Women in Science and Engineering (WISE) Residential Learning Community. dusseau@cs.wisc.edu

Remzi Arpaci-Dusseau is a Professor in the Computer Sciences Department at the University of Wisconsin—Madison. He received his BS in Computer Engineering summa cum laude from the University of Michigan, Ann Arbor, and MS and PhD in Computer Science from the University of California, Berkeley, under advisor David Patterson. Remzi co-leads a research group with his wife, Andrea Arpaci-Dusseau. Remzi also cares deeply about education and has won the SACM Student Choice Professor of the Year award four times and the Carolyn Rosner "Excellent Educator" award once for his efforts in teaching operating systems. remzi@cs.wisc.edu



Shan Lu is an Assistant Professor in the Computer Sciences Department at the University of Wisconsin—Madison. Her research interests include software reliability and computer systems. shanlu@cs.wisc.edu

We conducted a comprehensive study of Linux file system evolution by analyzing eight years of changes across 5,079 patches, deriving numerous new (and sometimes surprising) insights into the file-system development process. Our observations should be useful to file-system developers, systems researchers, and tool builders. Careful study of these results should bring about a new generation of more robust, reliable, and performant file systems.

A file system is not a static entity. Its code base constantly evolves through the addition of new features, repair of uncovered bugs, and improvement of performance and reliability. For young file systems, code sizes increase significantly over time. For example, ext4 nearly doubled its code size from Linux 2.6.19 (when it was introduced) to Linux 2.6.39. Even for ext3 (a stable file system), size increased more than 30% within eight years in Linux 2.6.

Patches describe how one version transforms to the next version and, thus, precisely represent the evolution of a file system code base. For open source file systems, every patch is available online, enabling us carefully to analyze in great detail how file systems change over time. A new type of "system archeology" is thus possible.

A comprehensive study of file system evolution can quantitatively answer many important questions. For example, where does the complexity of such systems lie? What types of bugs are dominant? Which performance techniques are utilized? Which reliability features exist? Is there any similarity across different file systems?

Such a study is valuable for different communities. For file system developers, they can learn from previous bug patterns to avoid repeating mistakes. They can improve existing designs by borrowing popular performance and reliability techniques. For system researchers, this study can help them identify real problems that plague existing systems, and match their research to reality. For tool builders, our study provides thousands of bug patterns, bug consequences, and performance and reliability techniques. These large-scale statistics can be leveraged to build various useful tools.

We studied six major file systems of Linux, including XFS, ext4, Btrfs, ext3, ReiserFS, and JFS. Readers may wonder why we only studied local file systems when distributed file systems are becoming increasingly important. We note that local file systems remain a critical component in modern storage, given that many recent distributed file systems, such as Google GFS and Hadoop DFS, all replicate data objects (and associated metadata) across local file systems. On smartphones and personal computers, most user data is also managed by a local file system; for example, Google Android phones use ext4 and Apple's iOS devices use HFSX.

Our study is based on manual patch inspection. We analyzed all patches of six file systems in Linux 2.6 multiple times. We have turned our manual analysis into an annotated data set, which enables us quantitatively to evaluate and study file systems in various aspects. We easily can analyze what types of patches exist, what the most common bug patterns are, how file systems reduce synchronization overhead, how file systems check for metadata corruption, and other interesting properties.

We make the following major observations:

- Bugs are prevalent in both young and mature file systems.
- Among these bugs, semantic bugs dominate.
- Over time, the number of bugs does not diminish, but rather remains a constant in a file system's lifetime.
- Data corruption and system crashes are the most common bug consequences.
- Metadata management has high bug density.
- Failure paths are particularly error-prone.
- The same performance techniques are used across file systems, whereas reliability techniques are included in a more ad hoc manner.

More results and analysis are discussed in our FAST '13 paper [3]. Another outcome of our work is an annotated data set of file-system patches, which we make publicly available for further study (at http://www.cs.wisc.edu/adsl/Traces/fs-patch).

Methodology

We chose a diverse set of file systems: XFS, ext4, Btrfs, ext3, ReiserFS, and JFS. These file systems are developed by different groups of people, use various techniques, and even represent a range of maturity. For each file system, we conducted a comprehensive study of its evolution by examining all patches from Linux 2.6.0 (Dec '03) to 2.6.39 (May '11). We manually analyzed each patch to understand its purpose and functionality, examining 5,079 patches in total.

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.

Listing 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 [1].

```
[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;
```

Listing 1: An ext3 patch

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).

Limitations: Our study is limited by the file systems we chose, which may not reflect the characteristics of other 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.

Major Results

In this section, we present our major study results of bug and performance patches. Our results are illustrated around several key questions in the following sections.

What Do Patches Do?

We classified patches into five categories: 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 1(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.

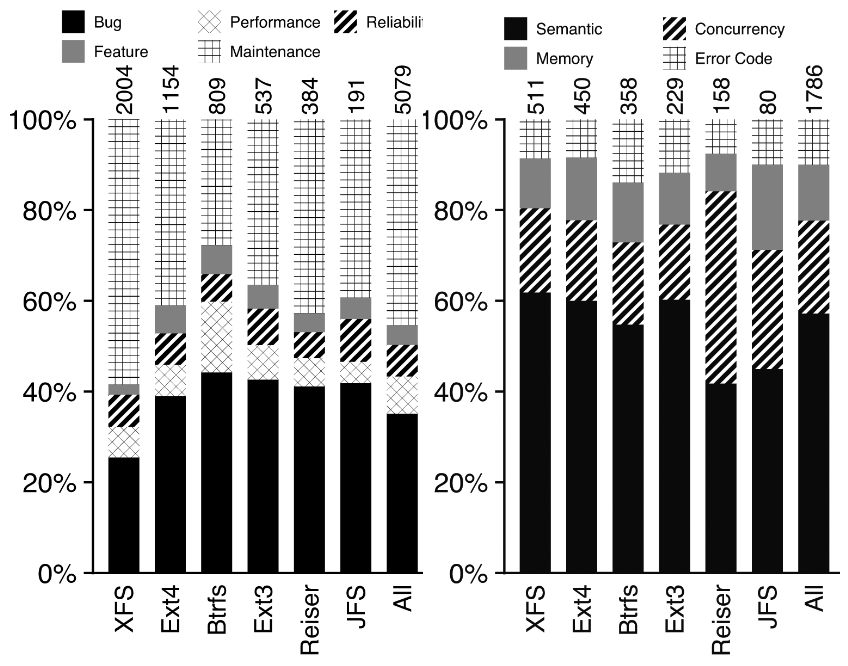


Figure 1: This figure shows the distribution of patch types and bug patterns. The total number of patches is on top of each bar.

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 1: Bug Pattern Classification. This table shows the classification and definition of file-system bugs.

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 across file systems. Not surprisingly, 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 (see "Do Bugs Diminish Over Time?" below).

Both *performance* and *reliability* patches occur as well, although with less frequency than maintenance and bug patches. They reveal a variety of the same techniques used by different file systems. Finally, feature patches account for a small percentage of total patches; but, most of feature patches contain more lines of code than other patches.

What Do Bugs Look Like?

We partitioned file-system bugs into four categories based on their root causes. The four major categories are incorrect design or implementation (*semantic*), incorrect concurrent behaviors (*concurrency*), incorrect handling of memory objects (*memory*), and missing or wrong error code handling (*error code*). The detailed classification is shown in Table 1. Figure 1(b) shows the total number and percentage of each type of bug across file systems. There are about 1,800 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 [1]) may have a hard time finding these bugs. An example of a *logic* bug is shown in S1 of Table 2: `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.

Concurrency bugs account for about 20% of bugs on average across file systems (except for ReiserFS), providing a stark contrast to user-level software in

ext3/ialloc.c, 2.6.4 **Semantic (S1)**	*ext4/extents.c, 2.6.30* **Concurrency (C1)**
`1     find_group_other(...){`	`1     ext4_ext_put_in_cache(...){`
`2 -     group = parent_group + 1;`	`2 +     spin_lock(i_block_reservation_lock);`
`3 -     for (i = 2; i < ngroups; i++) {`	`3       cex = &EXT4_I(inode)->i_cached_extent;`
`4 +     group = parent_group;`	`4...6   cex->ec_FOO = FOO; // elided for brevity`
`5 +     for (i = 0; i < ngroups; i++) {`	`7 +     spin_unlock(i_block_reservation_lock);`
ext3/super.c, 2.6.7 **Memory (M1)**	*reiserfs/xattr_acl.c, 2.6.16* **Error Code (E1)**
`1     ext3_get_journal(...){`	`1     reiserfs_get_acl(...){`
`2       if (!journal) {`	`2       acl = posix_acl_from_disk(...);`
`3         ...   ...`	`3 -     *p_acl = posix_acl_dup(acl);`
`4 +       return NULL;`	`4 +     if (!IS_ERR(acl))`
`5       }`	`5 +       *p_acl = posix_acl_dup(acl);`
`6       journal->j_private = sb;`	
ext4/resize.c, 2.6.25 **Failure Path (F1)**	*ext4/inode.c, 2.6.22* **Failure Path (F2)**
`1     ext4_group_extend(...){`	`1     ext4_read_inode(...){`
`2       if (count != ext4_blocks_count(es)){`	`2     ...   ...`
`3         ext4_warning(...);`	`3     if (inode_is_bad){`
`4 +       ext4_journal_stop(handle);`	`4 +     brelse(bh);`
`5         error = -EBUSY;`	`5       goto bad_inode;`
`6         goto exit_put;`	`6     }`
ext4/extents.c, 2.6.31 **Performance (P1)**	*btrfs/free-space-cache.c, 2.6.39* **Performance (P2)**
`1     ext4_fiemap(...){`	`1     btrfs_find_space_cluster(...){`
`2 -     down_write(&EXT4_I(inode)->i_data_sem);`	`2 +     if (bg->free_space < min_bytes){`
`3 +     down_read(&EXT4_I(inode)->i_data_sem);`	`3 +       spin_unlock(&bg->tree_lock);`
`4       error = ext4_ext_walk_space(...);`	`4 +       return -ENOSPC;`
`5 -     up_write(&EXT4_I(inode)->i_data_sem);`	`5 +     }`
`6 +     up_read(&EXT4_I(inode)->i_data_sem);`	`6        /* start to search for blocks */`

Table 2: Code Examples. This table shows the code examples of bug and performance patches.

which fewer than 3% of bugs are concurrency-related [2]. 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. An example of an atomicity violation bug in ext4 is shown in C1 of Table 2. 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 also a fair number of memory-related bugs in all file systems; their percentages are lower than that reported in user-level software [2]. Many research and commercial tools have been developed to detect memory bugs [1, 5], and some of them are used to detect file-system bugs. An example of a null-pointer dereference bug is shown in M1 of Table 2; a return statement is missing, leading to a null-pointer dereference.

Error code bugs account for only 10% of total bugs. A missing error code example is shown in E1 of Table 2. 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).

Do Bugs Diminish Over Time?

File systems mature from the initial development stage to the stable stage over time, by applying bug-fixing and 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, if so, in what way?

Overall, our results (Figure 2) show that the number of bugs does not die down over time (even for stable file systems), but rather ebbs and flows. A great example is XFS, which under constant

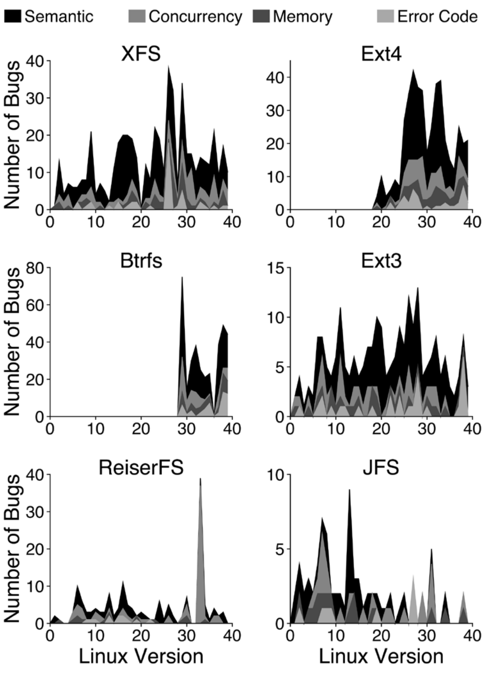


Figure 2: Bug Pattern Evolution. This figure shows the bug pattern evolution for each file system over all versions.

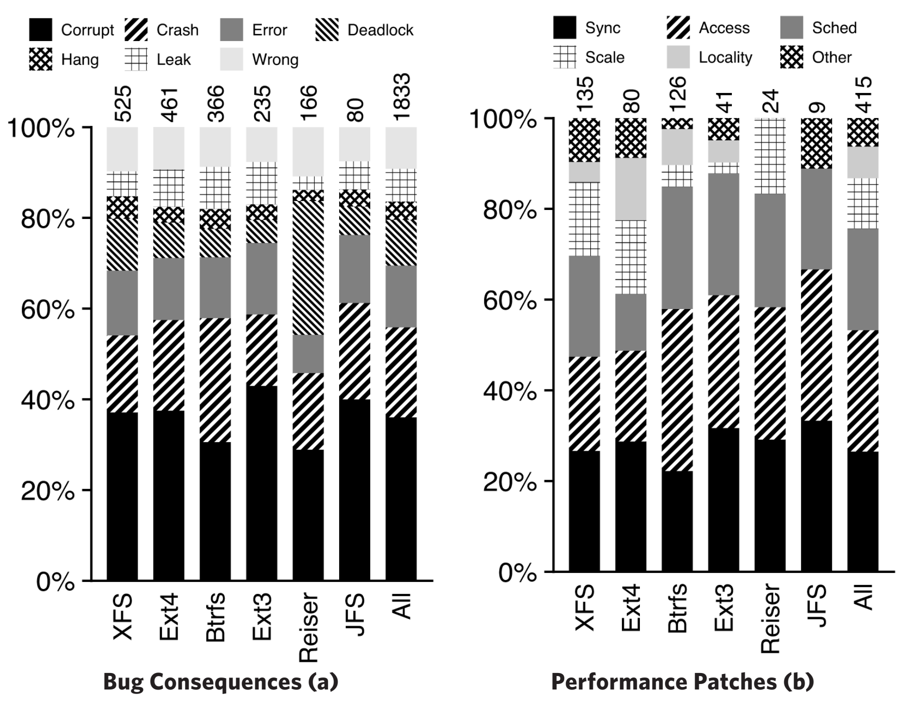


Figure 3: This figure displays the breakdown of bug consequences and performance patches. The total number of consequences and patches is shown on top of each bar. A single bug may cause multiple consequences; thus, the number of consequences instances is slightly higher than that of bugs in Figure 1(b).

development goes through various cycles of higher and lower numbers of bugs. A similar trend applies to ext3. For ext3, a new block reservation algorithm was added at Linux 2.6.10, and extended attributes in inodes were added at Linux 2.6.11. Therefore, a surge of bug patches are related to these new features. Similar things happened at 2.6.17, where a new multiple block allocation algorithm was introduced. Then, many related bug patches followed. At Linux 2.6.38, the spike is because ext3 fixed multiple error-handling bugs.

New file systems, such as ext4 and Btrfs, have a high number of bugs at the beginning of their lifetime. JFS and ReiserFS both have relatively small developer and user bases compared to the more active file systems XFS, ext4, and Btrfs. JFS does experience a decline in bug patches.

Within bugs, the relative percentage of semantic, concurrency, memory, and error code bugs varies over time but does not converge. Interesting exceptions occasionally arise (e.g., the BKL removal from ReiserFS led to a large increase in concurrency bugs in 2.6.33).

What Consequences Do Bugs Have?

As shown in Figure 1(b), 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.

Figure 3(a) shows the per-system breakdowns. If the patch mentions that the crash also causes corruption, then we classify this bug with multiple consequences. 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. Unexpected errors and deadlocks occur quite frequently (just below 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 [2].

Where Does Complexity Lie?

The code complexity of file systems is growing. The original FFS had only 1,200 lines of code; modern systems are notably larger, including ext4 (29K LOC), Btrfs (47K LOC), and XFS (64K LOC). Several fundamental questions are germane: How are the code and bugs distributed? 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: data block allocation (*balloc*), directory management (*dir*), extent mapping (*extent*), file read and write operations (*file*), inode metadata (*inode*), transactional support (*trans*), superblock metadata (*super*), generic tree procedures (e.g., insert an entry) (*tree*) and other supporting components (*other*).

Figure 4 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

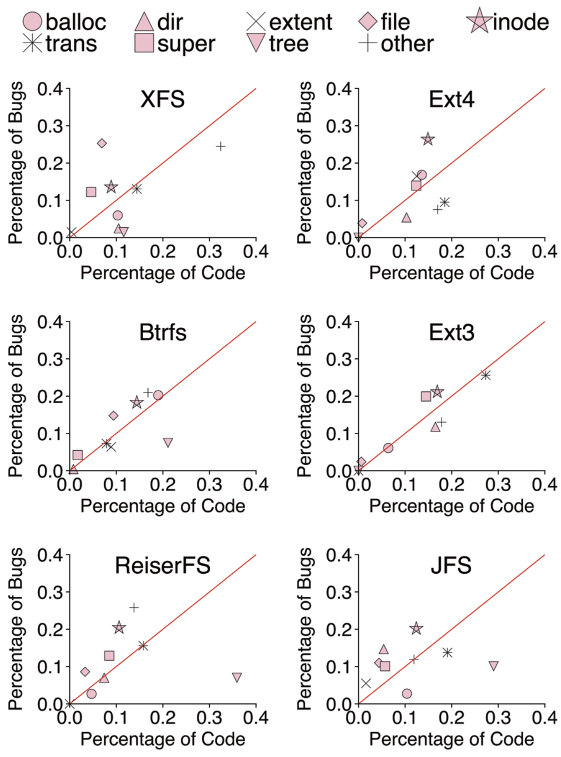


Figure 4: File System Code and Bug Correlation. This figure shows the correlation between code and bugs. The x-axis shows the average percentage of code of each component (over all versions); the y-axis shows the percentage of bugs of each component (over all versions).

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. 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, adding a transaction checksum to JBD) 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; whereas 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.

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

(a) By File System

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

(b) By Bug Pattern

Table 3: Failure Related Bugs. This table shows the number and percentage of the bugs related to failures in file systems.

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.

Do Bugs Occur on Failure Paths?

Many bugs we found arose not in common-case code paths but rather in more unusual fault-handling cases. File systems need to handle a wide range of failures, including resource allocation, I/O operations, silent data corruption, and even incorrect system states. These failure paths have a unique code style. *Goto* statements are frequently used. Error codes are also propagated to indicate various failures detected. We now quantify bug occurrences on failure paths; Table 3 presents our accumulated results.

As we can see from the Table 3a, 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 we break it down by bug type in Table 3b, we see that roughly a quarter of semantic bugs occur on failure paths. 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 to perform them incorrectly. An example of a semantic bug on failure path is shown in F1 of Table 2. When ext4 detects multiple resizers run at the same time, it forgets to stop the journal to prevent potential 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 [4].

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 that incorrectly assume certain pointers are still valid after a failure. An example of a memory bug on failure path is shown in F2 of Table 2. When ext4 detects a corrupted inode, it forgets to release the allocated buffer head. Finally (and obviously), all error code bugs occur on failure paths (by definition).

Testing failure-handling paths to find all types of bugs is difficult. Most previous work has focused on memory resource leaks and missing unlock and error codes; 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.

What Performance Techniques Are Used?

Performance is critical for all file systems. Performance patches are proposed to improve existing designs or implementations. We partition these patches into six categories: inefficient usage of synchronization methods (*sync*), smarter access strategies (*access*), I/O scheduling improvement (*sched*), scale on-disk and in-memory data structures (*scale*), data block allocation optimization (*locality*), and other performance techniques (*other*). Figure 3(b) shows the breakdown.

Synchronization-based performance improvements account for more than 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 2; `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 3(b) shows access patches are popular. An example Btrfs access patch is shown in P2 of Table 2; before searching for free blocks, the patch 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. 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.

Lessons Learned

Beyond the results, we also want to share several lessons we learned from this project. First, a large-scale study of file systems is feasible and valuable. Finishing this study took us one and half years. Even though the work is time-consuming, it is still manageable. A similar study may be interesting for other OS components, such as the virtual memory system.

Second, details matter. We found many interesting and important details, which may inspire new research opportunities. For example, once you know how file systems leak resources, you can build a specialized tool to detect leaks more efficiently. Once you know how file systems crash, you can improve current systems to tolerate crashes more effectively.

Third, research should match reality. New tools are highly desired for semantic bugs. More attention may be required to make failure paths correct.

Finally, history repeats itself. We observed that similar mistakes recur, both within a single file system and across different file systems. We also observed that similar (but old) performance and reliability techniques were utilized in new file systems. We should pay more attention to system history, learn from it, and use this knowledge to help build a correct, high-performance, and robust next-generation file system from the beginning.

Acknowledgments

This material is based upon work supported by the National Science Foundation under the following grants: CNS-1218405, CCF-0937959, CSR-1017518, CCF-1016924, as well as generous support from NetApp, EMC, and Google.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] 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.

[2] 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," Workshop on Architectural and System Support for Improving Software Dependability (ASID '06), San Jose, California, October 2006.

[3] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu, "A Study of Linux File System Evolution," Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13), San Jose, California, February 2013.

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

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