Study of File System Evolution

Swaminathan Sundararaman, Sriram Subramanian
Department of Computer Science
University of Wisconsin
{swami, srirams} @cs.wisc.edu

Abstract

File systems have traditionally been a major area of research and development. This is evident from the existence of over 50 file systems of varying popularity in the current version of the Linux kernel. They represent a complex subsystem of the kernel, with each file system employing different strategies for tackling various issues. Although there are many file systems in Linux, there has been no prior work (to the best of our knowledge) on understanding how file systems evolve. We believe that such information would be useful to the file system community allowing developers to learn from previous experiences.

This paper looks at six file systems (Ext2, Ext3, Ext4, JFS, ReiserFS, and XFS) from a historical perspective (between kernel versions 1.0 to 2.6) to get an insight on the file system development process. We study the source code of these file systems over years to observe changes in complexity of the system and relate these metrics to the features and stability of file systems. We identified that only a few components in file systems are difficult to get right the first time. Also, file system developers do not learn from each others mistake. As a result they end up introducing the same bugs in file systems.

1. Introduction

In today's world everything revolves around data. Data is critical for day-to-day operation of any system. Data management is taken care of by file systems which reliably store data on disks. Users typically have varied requirements ranging from scalability, availability, fault-tolerance, performance guarantees in business environment to small memory footprints, security, reliability in desktop environments. This has driven the file system community to develop a variety of systems that cater to different user requirements.

An open source environment like Linux has around 50 different file systems that provide different guarantees, targeting varied media. In an open source environment, file systems are typically developed and maintained by several programmer across the globe. At any point in time, for a file system, there are three to six active developers, ten to fifteen patch contributors but a single maintainer. These people communicate through individual file system mailing lists [14, 16, 18] submitting proposals for new features, enhancements, reporting bugs, submitting and reviewing patches for known bugs. The problems with the open source development approach is that all communication is buried in the mailing list archives and aren't easily accessible to others. As a result when new file systems are developed they do not leverage past experience and could end up re-inventing the wheel. To make things worse, people could typically end up doing the same mistakes as done in other file systems.

We chose to look at the file systems in the Linux kernel for the following three reasons: (1) Linux is open source, implying that we can get access to the source code of the file systems, without which it is almost impossible to study their evolution. (2) As compared to other systems Linux support a large variety of file systems. Most importantly, some of these have existed for a long time (10-15 years). Hence, it is possible to look at the file systems during different stages of its lifetime and derive conclusions from them. (3) All communications (discussion, patches, bugs, etc.) between file system developers can be obtained from the mailing list and are publicly available in many websites [16]. The bugzilla database, CVS, and git repositories are also publicly accessible. This allowed us to get insight into the development process of file systems in an open source environment. It would definitely be interesting to look at file systems in other operating systems like Solaris, BSD. or Windows to compare and contrast it with file systems developed in Linux.

In this paper, we look at file system code in the Linux kernel (version numbers 1.0 to 2.6). This covers 14 years of kernel development (1994 to 2008). We also look at patches, bug reports and other communication

that is available in each of the file system mailing list. We have built a tool that automatically crawls the mailing archives and extracts higher level information from them. We did not use Bugzilla database [9] as it is restricted to Redhat releases and they are only a subset of the changes that are available in the mailing lists. We did not use the CVS repositories for the file systems as they didn't have complete information. The repositories are solely used by the file system maintainer to consolidate the patches that developers submit over time and eventually the kernel maintainer picks up the patches from each of the maintainers and integrates it to the linux kernel. We already have this information by looking at the file system code between kernel releases and the mailing list. As far as we know, a study of similar nature does not exist in the file system community.

The contributions of this paper are as follows:

- We have developed tools to systematically mine information available in mailing lists [14,15,16,18] and source code repositories [10, 11] so that developers can easily query to know file system properties or bugs.
- As part of our preliminary analysis, we provide insights into the development process, common problem areas, and stability of file systems.
- Identification of common bugs in file systems. This could serve as a reference document to file system developers to check if their file system are immune to the common bugs.

From our study, we found that out of 60 or so file systems only a handful of them have lots of features and most widely developed and used. Most of the file systems are designed well and do not undergo design level changes during their lifetime. In a purely open source file system, many developers do contribute to the development of the file system and in contrast file systems that have been started. We also found that developers across file system do not communicate effectively and as a result end up repeating the same mistakes in their file systems.

Rest of the paper is organized as follows. Sections 2 explains the difficulties in extracting higher level information in file system development in Linux. Section 3 gives a broad overview of file system in the latest Linux kernel and justifies our choice of file systems included in this study. Section 4 analyzes the trends in code change and complexity across file systems. Section 5 and Section 6 looks at the code contributions of developers in file systems and frequently changed components. Section 7 discusses common bugs in file systems code and looks at turn around time for bugs in XFS. We talk about the implications of this work and how open source community should adapt itself for more efficient functioning in Section 8. We comment on related work in Section 9 and conclude in Section 10.

2. Problems in the Open Source Model

To our knowledge, there has been no prior work on understanding the file system development process, especially in the Linux world. Previous work on software evolution looked at software repositories like CVS, etc[3,4]. We cannot apply the same kind of analysis because the development model in Linux is very different. There is to no easy way to get this information in the Linux world. We comment briefly about the file system development process in Linux to understand the difficulties involved in it.

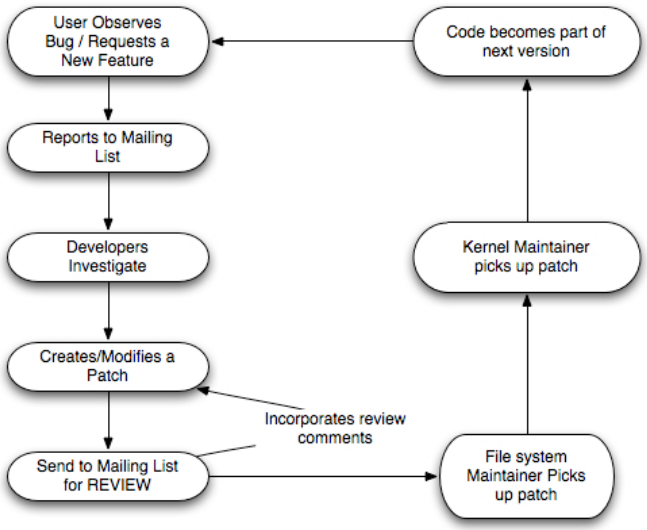


Figure 1: Linux kernel development process

Figure 1 shows the development process in linux. From the figure, one can see that most of the communication between the developers happens through the mailing lists. All the discussion, patches, and issues get buried in the mails and finally end up getting archived at some public website (e.g., [16]). Unlike the software development process employed in other open source projects, the Linux model is very different in the sense that there is no proper CVS repository where we can track every change in the file system. Everything has to

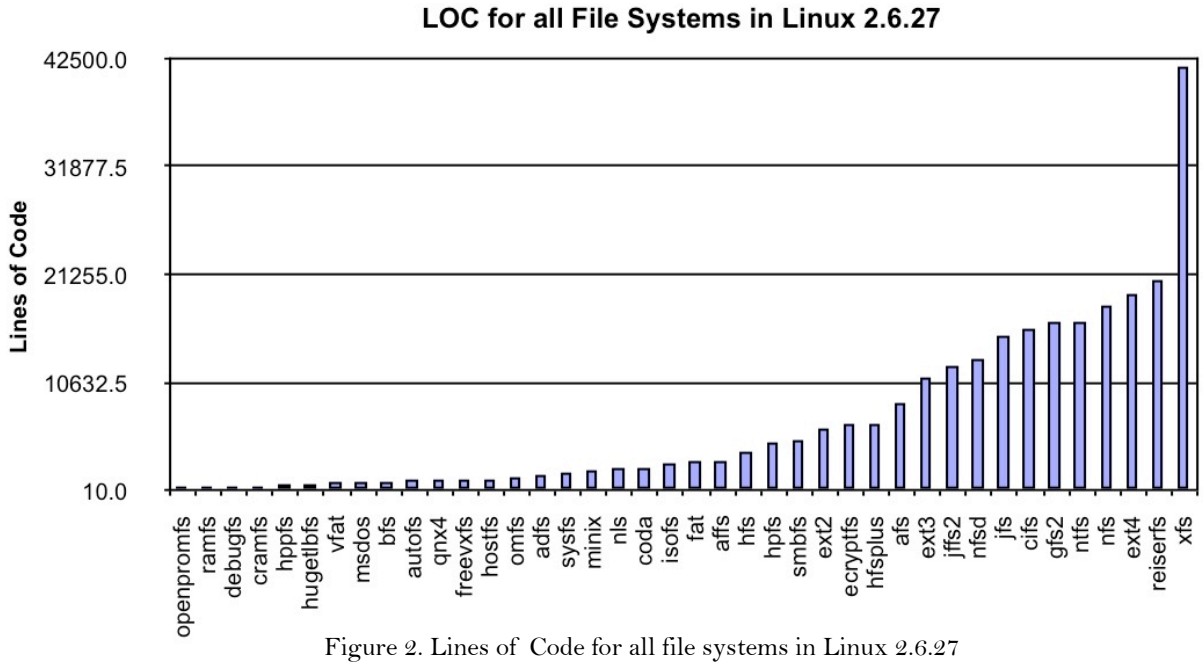


Figure 2. Lines of Code for all file systems in Linux 2.6.27

be inferred from the mails in the mailing lists. Although there are search engines that help extract information quickly from these archives, there are not tools that can infer trends and features from these mails. To our knowledge, existing tools to understand the open source project [7] etc. do not work well for extracting information from the file systems in Linux. To summarize, file system developers typically communicate within their community (file system mailing list) about the issues and fixes and there is no easy way to get this information to other file system developers. As a result, we believe that file system developers do not learn from each other's mistakes and end up repeating the same mistakes (Refer Section 6 for more details).

3 Choosing File systems

To obtain insight into how file systems have evolved, it is important to observe file systems over a period of time. For all the file systems that we looked at, we ensured that they existed for at least two years (this does not include the initial time spent on the file system before it was accepted in the Linux kernel).

We also wanted to look at file systems at various stages of their development along with how they were conceived. To be precise, we wanted diversity in the file systems origin in order to derive useful inferences. We chose Ext2, Ext3, Ext4, Reiserfs, XFS, and JFS in our study. The Ext2 file system was chosen because it is a mature file system that was derived from the Ext file system and has been around for more than 15 years. Also, Ext2 was the default file system for Linux for over 10 years which indicates its stability. We chose the Ext3 and Ext4 because they were derived from Ext2. Ext3 is a mature file system (it is now the *default* file system in Linux) and the development in Ext4 started recently (in 1995). The Ext2/3/4 file system were started by the open source community. ReiserFS was chosen as it was initially conceived by a single person (Hans Reiser) and was slowly adopted by the open source community. We selected the XFS and JFS file systems as they were initially developed by the industry (SGI [21] and IBM [22] respectively) and then were adopted by the open source community. By choosing file systems that have evolved from different sources we believe we can get a broad perspective on the file system development process. For completeness, we take a brief look at all file system in the current linux version.

3.1 File System Sizes

Even though we looked only at 6 file systems, we wanted to get a feel of all the file system that exists in the current Linux kernel. Figure 2 shows the lines of code(LOC) in file systems in Linux. For calculating LOC we do not include blank lines and comments.

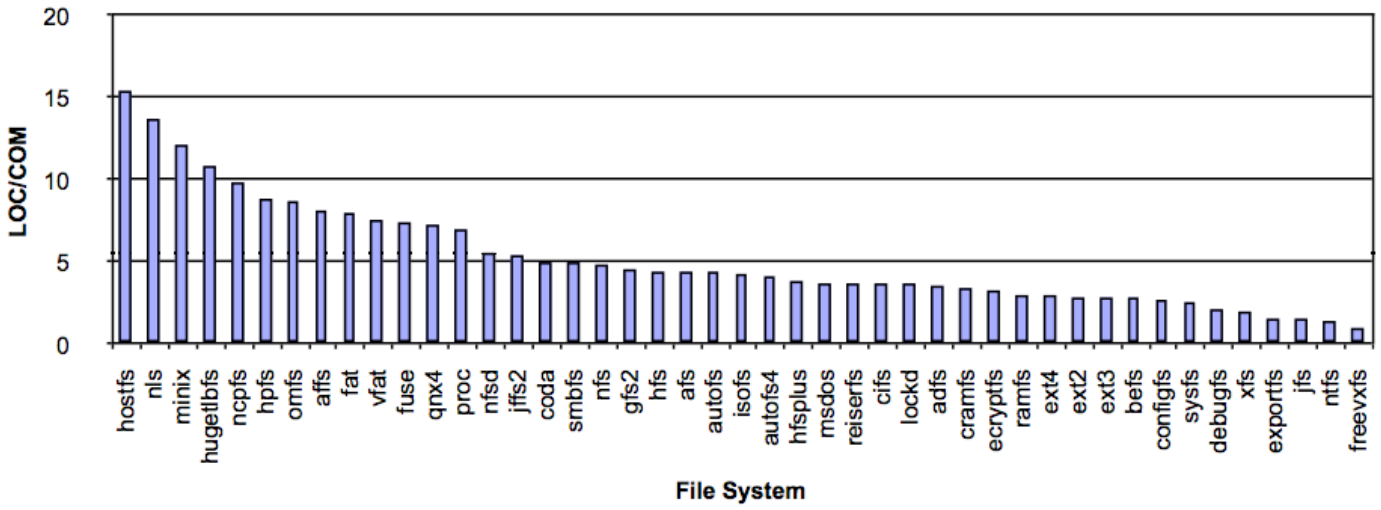


Figure 3. Lines of Comments for all file systems in Linux 2.6.27

File systems can be classified into three categories based on the lines of code. In the first category we have file systems that have less than 1000 lines of code. 20 of the 60 file systems studied fall into this category. These file systems are customized for a specific purpose or environment (for example, ramfs is customized for keeping all data in memory with no consistency guarantees). The interesting thing about these file systems is that they heavily use the library file system (libfs) code that is available in the Linux kernel, which also reduces the number of errors in these file systems.

In the second category, the file system sizes range between 1000 and 5000. There are 25 file systems that fall into this category. These file systems do not support many features and have their development stopped within a few years of its acceptance in the kernel and hence are not so interesting to observe.

Finally we have 15 file systems in the third category in which lines of code are greater than 5000. These file systems are more interesting to study for the following reasons. (a) They have significant number of features, (b) Development of these file system have continued for more than two years, (c) They have at least 10 main contributors in their life time, and (d) these file systems are widely used.

3.2 Documentation in File systems

A major complaint with open source software is that many of them are not well documented. We wanted to verify if it holds true for file systems in Linux by computing the ratio of lines of code to lines of comments (LOC/COM) for each of the file systems. Figure 3 shows LOC/COM ratio for all file systems. It is interesting to see that file systems that have more lines of code (or more popular) are the ones that are well documented. One thing that stood out was XFS which has a very low LOC/COM ratio (it had a line of comment for every two lines of code). We believe there are two reasons for such good documentation. The primary reason is that it was developed in a company and secondary reason being that it has around 250 files and it is almost impossible to read code or track dependency without proper documentation.

3.3 Features in File Systems

We wanted to study the contribution of features in a file system towards its popularity and compare it against file system size. Table 1 shows popular features that are supported in Ext2, Ext3, Ext4, JFS, ReiserFS and XFS. We did not include the other file systems because they are not as feature heavy as the ones shown in Table 1. For instance, most of the other file systems did not provide any feature other than hard and soft links (just like the Ext2 file system). Some of these file system did not support ACL and extended attributes (e.g., cramfs and ramfs). It is obvious that only file systems that have large number of lines have more features in them. These are the file systems that are more popular amongst users [23].

In summary, although many file systems come along with the linux kernel, only a handful of those are popular. We observed that all popular file systems

	Hard / Soft Links	Journalling (All / Meta data)	Online Resizing (shrink / grow)	Offline Resizing (shrink / grow)	Clustering	Access Control Lists / Extended Attributes	XIP
Ext2	Yes	No	No	No	No	Yes	Yes
Ext3	Yes	Yes (Both)	Yes (only grow)	Yes (both)	Partial	Yes	Yes
Ext4	Yes	Yes (Both)	yes	Yes (both)	Partial	Yes	Yes
ReiserFS	Yes	Yes (Both)	Yes (only grow)	Yes (both)	Partial	Yes	No
JFS	Yes	Yes (Both)	Yes (only grow)	No	Partial	Yes	No
XFS	Yes	Yes (only Metadata)	Yes (only grow)	No	yes	Yes	No

Table 1: File System features

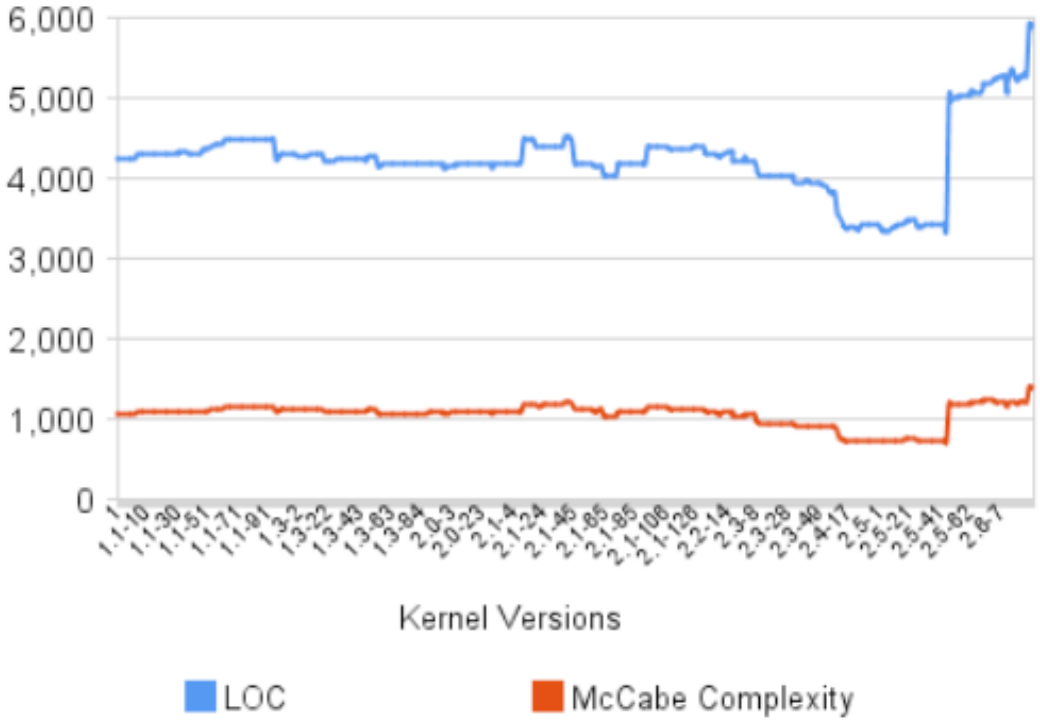


Figure 4: Ext2 Code Evolution

provided stronger data reliability guarantee through journalling. Journalling has become a de-facto feature in today's file systems. Other features critical to a file system success are online and offline resizing of file systems. It was interesting to observe that extended attributes were first introduced in the Ext2 file system and was adopted by all popular file systems as it provided more flexibility for the users to describe her stored data.

4. Analysis on the life span of File Systems

In order to obtain a better understanding file system development over time, we looked at changes made to a file system code over time and the code complexity as a result of these changes. To measure code complexity, we used McCabe cyclomatic complexity metric [24]. Cyclomatic complexity measures the number of linearly independent paths through a program's source code. It is the one of the most popular metrics to measure the complexity of any code.

We wanted to verify whether some of the popular wisdom in software engineering hold true. For example, more stable system do not change frequently or file systems in the initial stages of development tend to change quite often. We looked at each of the six file systems and tried to understand the nature of code changes across kernel versions.

Figure 4 shows Ext2 changes between versions 1.0 to 2.6.27. It was surprising to observe that although Ext2 had been establish well before v1.0, there are changes to the code till the latest version. On closer inspection of the *diffs* between the two versions where there was significant code change, we saw that a majority of the changes where due to interface changes in the linux kernel. There were a large number of modifications between versions 2.2 and 2.3, which has a significant drop in lines of code. On inspection, this was because the developers were trying to fix the file truncation logic and after a few attempts they threw away the existing code and rewrote it from scratch as the race conditions were difficult to handle. The huge spike near the 2.5.45 indicates the 1500 lines that was added to the Ext2 file system (Access Control Lists (ACL) and extended attributes (XATTR) were introduced in this version). Another interesting trend that we observed in that some of the changes in Ext3 file system that improved its performance were back ported back to the Ext2 file system so that users who still use Ext2 could benefit from it. For example, the last spike near 2.6.26 is due to the back porting of smarter resource management code using Red black trees used in Ext3 file system.

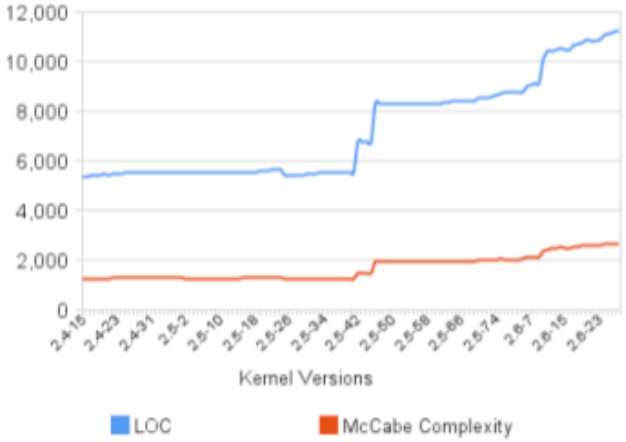


Figure 5a: Ext3 Code Evolution

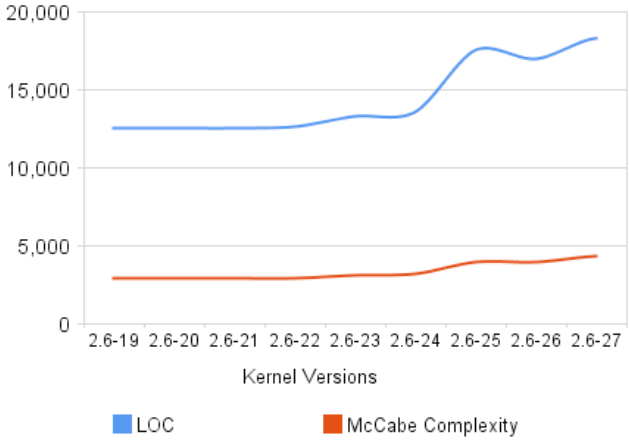


Figure 5b: Ext4 Code Evolution

The first two significant increases in the code was the introduction of a better directory look up mechanism using red black trees for improving performance and introduction of ACLs and XATTR respectively. The other increase in lines of code is due to the introduction of a new resource allocation policy using red black trees. Though people have vigorously started working on the Ext4 file system we still see a continuous development in the Ext3 file system. In Ext4 file system we see a significant amount of code churn as compared to Ext2 and Ext3 as it is still in the early stages of its development.

Figure 5c and Figure 5d shows the trend in JFS and ReiserFS. Figure 5e shows the code change in XFS

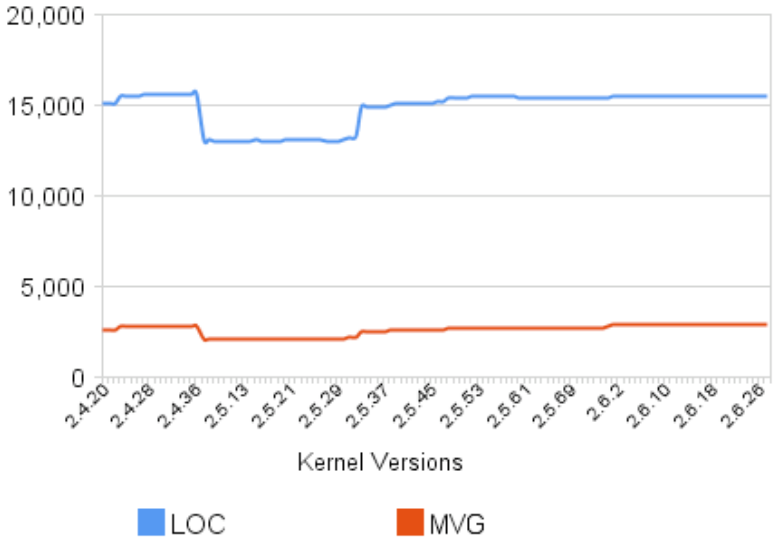


Figure 5c: JFS Code Evolution

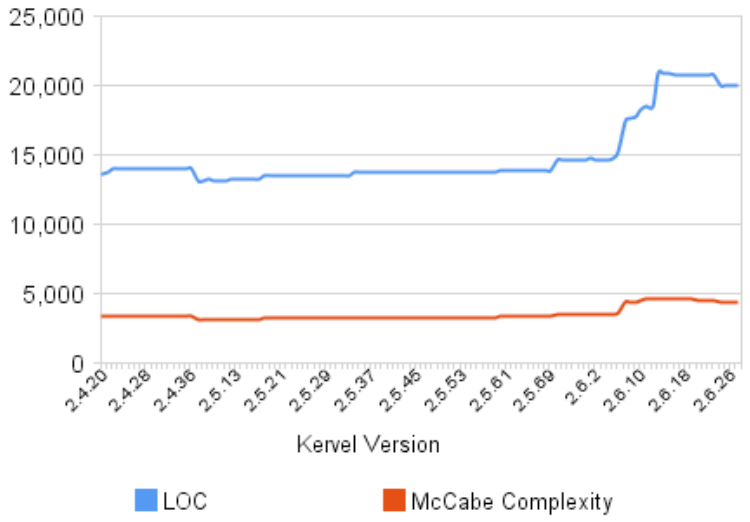


Figure 5d: ReiserFS Code Evolution

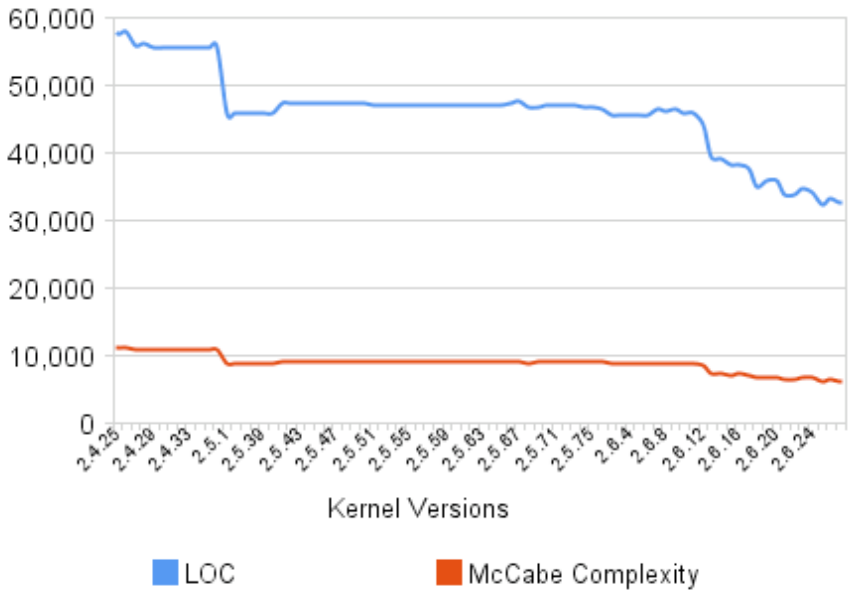


Figure 5e: ReiserFS Code Evolution

across kernel versions. The significant drop in the lines of code between 2.4.36 and 2.5 kernel is that buffer management and locking were inefficiently implemented before. They threw away almost all of the code and rewrote it from scratch. The other significant drop in the lines of code near 2.6.13 is due to simplification of the resize logic that they borrow from ext3 file system.

5. Code Contributors

The most important factor for the success or failure of an open source project are the people who contribute to it. Without constant contributions that fixes bugs, adds new features and most importantly reviews changes submitted by fellow contributors, an open source project cannot progress. The file systems under study represent fairly successful systems coming from different backgrounds. In this section we look at the various code contributors to each of these systems and try to understand how their development processes differ. Here, we present the amount of code contributed by the developers to these file systems. We measure code contribution by parsing emails that contain patches and figuring out what files and how many lines of code were affected. We consider a developer as a code contributor only when his overall contribution exceeds at least 50 lines. We don't have any idea about the initial contributors and the percentages that they contributed. This is an artifact of the kernel development process (the file system maintainers and the kernel maintainer have separate source code trees and they accumulate and eventually apply patches sent by other developers to their repositories). Due to this approach, it is difficult to find the original contributors of the code.

5.1 Methodology

One of the data sources for this project is the mailing list archives found at marc.info[15]. To extract all the mails sent to a particular mailing list we have to crawl all the pages of that list and download the emails. OSSMole[7] is an open source tool that can download mailing lists from sourceforge.net and populate a MySQL database. The problem with this tool is that it is highly customized to work with sourceforge and did not work well with marc.info. So we developed a simple crawler, that would download all the emails given the list name. The crawler has to keep track of links traversed once to avoid duplication. The next step is to preprocess the emails to get rid of the HTML tags. We then parse the emails to get relevant information out of them (sender, date, subject, content) and populate the MySQL database. We also segregate mails on contents, into bugs, patches and general discussion. We maintain separate databases for each mailing list and we can use these tables to run queries that could give us useful information. We illustrate the process in Figure 6.

5.2 Results

5.2.1 Ext2, Ext3, Ext4

Figure 7.a, 7.b. 7.c shows the percentage of code contributed by individual developers over the complete history of Ext2, Ext3 and Ext4. There are few interesting things that stand out from this pie chart.

- The overall contributions from major contributors (>5% code) is ~ 50% in each of the case, with the exception of Ext4 which is still under development and so fewer people contribute. The other side of the same picture is that almost 50% of the code comes from the community. This represents a significant contribution considering the fact that these are really complex systems, without the best documentation and development methodologies and standards.
- We find a lot of common developers (Andreas Dilger, Ming Ming Cao, Ted Tso, Andrew Morton etc) in these file systems clearly indicating the origin of the system (Ext3 evolved from Ext2 and Ext4 from Ext3).

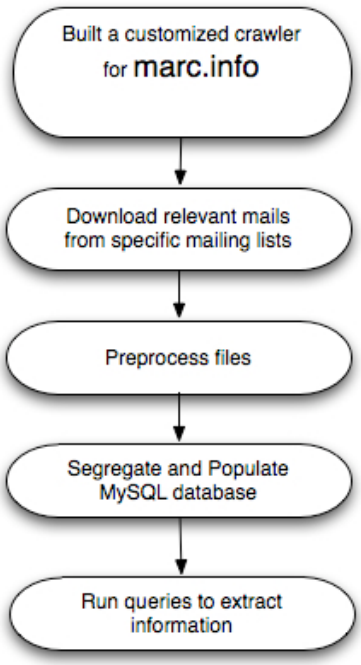
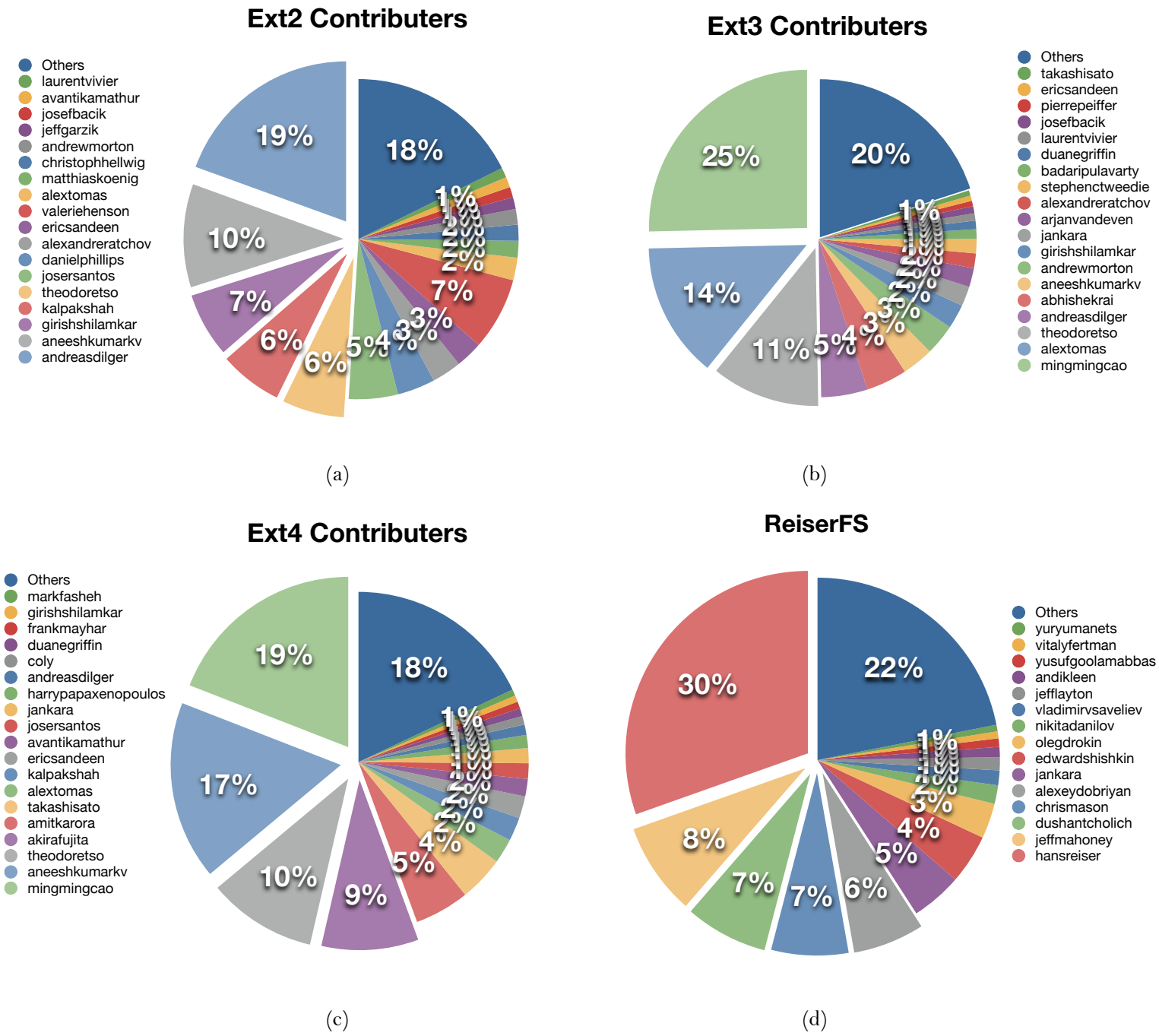


Figure 6: Information Extraction Methodology



(a) (b)

(c) (d)

Figure 7: Contributions to file system code by individual developers

5.2.2. ReiserFS

As mentioned ReiserFS is an unique case - it was sponsored by Namesys (and supported by Novell) and initially developed only by Hans Reiser. Figure 7d shows the contributors to ReiserFS. Apart from Hans Reiser (~30%), other major contributors like Jeff Mahoney (SUSE/Novell, ~8%), Chris Mason (SUSE, ~7%) etc account for over 50% of the code.

5.2.3. XFS

XFS, as mentioned before, was originally developed by SGI Corp for the IRIX operating system and later made open source. To this day, SGI continues to spearhead the development of XFS and so most of the major code contributions come from people directly employed by SGI. This is clear from Figure 7e, which shows that over 70% of the code was written by SGI employees (Chris Hellwig, Barry Naujok, Dave Chinner etc). The fewer developer count can also be attributed to the fact that XFS code base is generally considered to be complex with over 200 files. Even with well documented code, new developers find it hard to understand this massive code base.

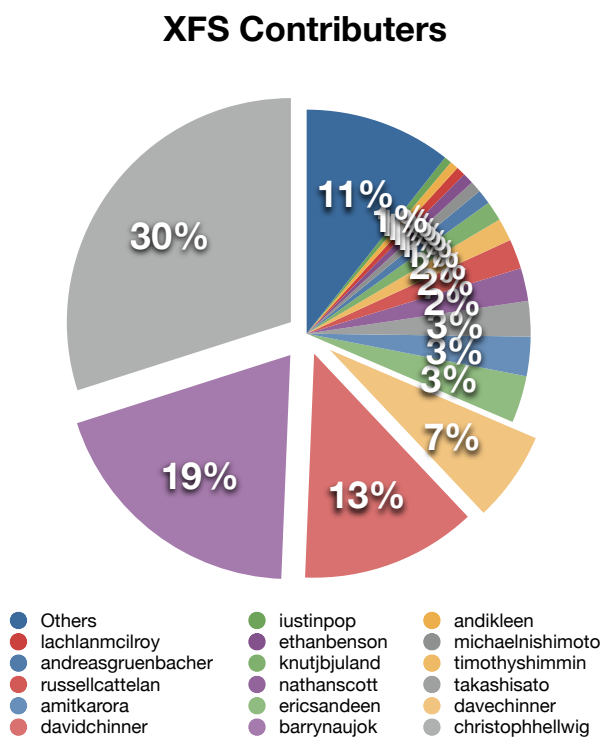


Figure 7e: XFS Contributers

Summary

As we can observe, purely open source projects tend to have a lot more developers contributing whereas projects driven by corporations have fewer developers. Another thing to note is simpler file systems are easier to understand and so the community can contribute to its development. We notice that a lot of people are continuously involved in the development process of Ext2, Ext3, Ext4 file system and it is this constant influx of contributions that keep these projects successful. Ext2, Ext3 are relatively mature systems and the code is well understood, where as Ext4 is a system under development and so people are still in the process of understanding how it works. More over, one wouldn't expect a new system to have good documentation. Once Ext4 stabilizes, we expect the same level of involvement from the community.

6. Frequently Changed Components

In this section we find what components were modified frequently in each of the file systems. Frequently changed components not only represent hot spots for changes, but also could relate to components that are more bug prone and are difficult to get right at the first attempt. To evaluate this metric, we create a list of file names in each email that was downloaded. This is under the assumption that when a developer submits a patch, the file names and the lines modified are included in the email. By searching for specific patterns like "*fs/reiserfs/*.c or *.h*", we can generate a list of files that were modified in a patch. To avoid double counting of files, we group emails by similarity of their subjects and each file mentioned in the concatenated contents is counted exactly once. We have done this study at a file level. It is possible to drill down to the granularity of functions, but that would require correlating diffs to function names.

In Figure 8a, the slices represent the frequency at which a file was modified. In case of Ext2, we see that the superblock (super.c), inode (inode.c), inode allocation (ialloc.c) and bitmap allocation (balloc.c) represents over 60% of all modification done to ext2. This includes frequent changes in kernel interfaces as well as mount options. We see a similar trend in ext3, ext4 and ReiserFS as well (Figure 8b, 8c, 8d respectively). This goes to show that in these popular file systems, there are few components that are hard to get right. Most others are relatively stable and dont often change.

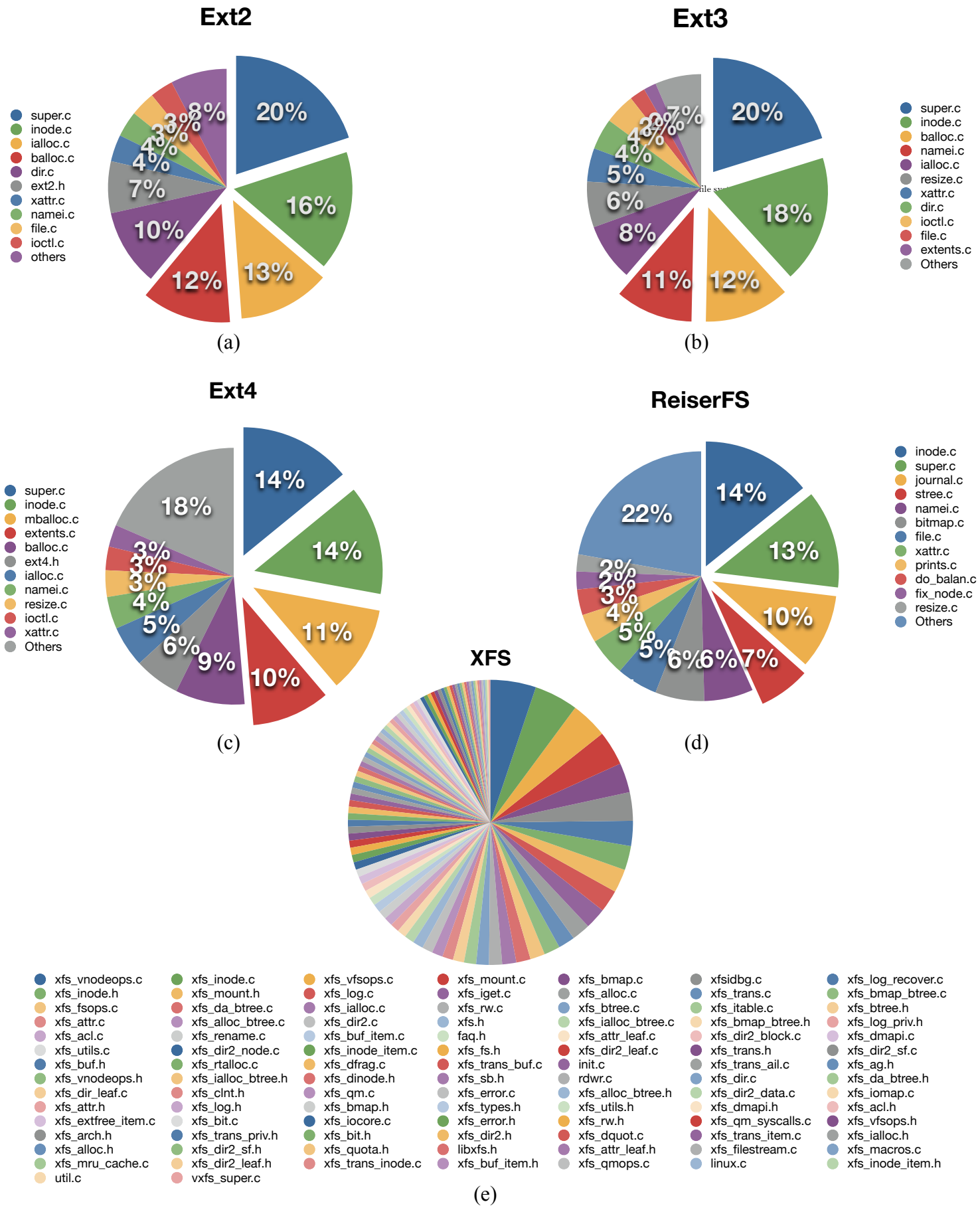


Figure 8: Contributions to file system code by individuals

On the other hand, Figure 8e shows the pie chart for frequently changed components in XFS. All of the 250 files in XFS were modified sometime or the other and no single file contributed to a major modification which is clearly evident from the figure. We understand from this graph is that we find a lot of dependancy between components in XFS and functionality distributed over multiple files. This also relates to the fact that XFS code is a lot more complex than Ext or ReiserFS.

In order to understand the nature of changes in frequently changed files we looked at super.c and balloc.c in all the Ext series file system. super.c and balloc.c was the among the top three frequently changed files. The changes in super.c where mainly related to the interface changes in the kernel across versions. As super.c is typically the interfaces between the kernel and the file system it needs to be changed whenever the kernel interface changes. The other changes in super.c were related to parsing the mount option and super block flags. On the other hand, in the balloc.c file, several corner cases were ignored.

7 Bugs in file system code

We wanted to systematically study file system bugs as they are the primary cause of file system instability. This holds true for most of the software systems. We attempted to provide a deep insight into the origin and likely cause of these bugs. Bugs in file systems need not only result from logical errors, but may also represent fundamental design issues that restrict the file system from functioning correctly. Our end goal was to come up with a bug classification taxonomy would provide the right level of abstraction to allow developers to communicate and reason about bugs. We believe that it could give further insight into density of each class of bugs and how this varies with the various releases as well as within each release.

We were really disappointed in the way bugs were reported and tracked in the Linux kernel. Though there is a bugzilla database for the linux kernel [8] most of the bugs aren't reported here. Even though there is a standard format for submitting bugs to the mailing list [25] most of the reports do not follow this format. To make things worse, the developers do not include proper explanation about how they fixed these bugs. They consolidate their other changes along with the bug fixes to the mailing list. They also use different subjects to discuss bugs that are reported in the mailing lists which made it even worse to track bugs in these file systems. In file systems like ReiserFS, they acknowledge that it is a bug and reply saying the fix will be available in the next version. Almost in all the file systems, the patches do not mention about the bugs that are being fixed. As a result, we had a tough time in consolidating bug reports to read the bug report and track patches. We ended up manually looking at most of the mails that had subject like 'bugs', 'oops', 'panic' or 'crash' and body containing the above mentioned key words. We only present our preliminary analysis of our findings here. We plan to continue working on it and try to come up with a taxonomy which will be part of the future work.

Common File System Mistakes

Though there are many file systems present today, the higher level operations on each of them are very similar. Interestingly, simple checks in one file system are also applicable to other file systems. For example, file size can never be greater than the file system size. We randomly selected 10% of the useful bug reports from each of the file systems and looked at the report to understand the type of bugs. We define useful bug report as one that has initial bug report along with a discussion on the bug (whose thread count is greater than 1) that has the same subject. We were happy to identify bugs that were reported in more than one file system. The common bugs in the file system code can broadly classified in terms of following four category. It is important to note that this is no way complete and should be considered as a preliminary finding. More through study is required to quantify the results.

Trivial Errors. These were very simple errors usually caused by programmers negligence or misunderstanding of the file system API's. Few of the errors that we observed in this category were as follows: (a) Missed update: this is the case when the programmer does not update part of the data structure. For example, in ReiserFS file system we saw that developer forgot to update the atime field in the Inode structure during stat operations. (b) Forgetting to lock shared data structures (c) Revert back patches by mistake. In all file system we observed that patches were reverted back by mistakes and as a result previously fixed bugs were reintroduced. (d) Changed the negation condition. Negation of the condition could exercise the wrong code which results in a bug. The following code patch fixes a bug in the ReiserFS code.

```
-} else if (!mutex_trylock(&journal->j_flush_mutex)) {
+} else if (mutex_trylock(&journal->j_flush_mutex)) {
        BUG();
```

Miss Dependency between two components. These errors usually arise because the developer is not very familiar with the code or more importantly due to insufficient documentation in the code. We observed bugs of this nature in Ext2, ReiserFS and also in XFS.

Not handling corner cases. As mentioned earlier. Some of the sanity checks are common across file system. Typically, file system developers miss these corner cases. For example, we saw that Ext2, Ext3 and ReiserFS didnt not check if the file size is greater than the file system size.

Failure to handle partial disk failures. Most of the file systems in our study neglected the errors reported by the disks in some I/O paths. As a result it caused them to panic the kernel. We observed bug reports in Ext3, ReiserFS and XFS about this behavior.

8 Discussion

During the course of this project, we faced a lot of difficulty in extracting information required to understand the file system development process. This made us wonder if the current practices are good enough. In this section, we would like to discuss some of the pitfalls in file system development and possible solutions that could make file system development in a open source development faster and enjoyable. File systems are one of the critical component of the operating system as it is responsible for storing data persistently in the storage media. Hence, more attention should be paid to the development process in order to avoid pitfalls that could lead to data loss.

Need for better information dissemination. File systems are typically implemented by a bunch of developers who get it to a working state before it gets included in the Linux kernel. Each file system has their own mailing lists and we saw that not much communication happens between developers across file systems. Documentation about critical decisions, trade offs, bugs are almost never documented and end up getting buried in the huge mailing list archives. Also people do not use bugzilla for reporting and tracking bugs. Currently, they are all tracked through mails which as we all know is very inefficient. Other open source systems (e.g., Mozilla, MySQL, etc.) have embraced Buzilla and have shown that it is very effective in tracking bugs in their file systems. It would be really useful if the file system maintainers enforce the use of bug tracking systems and document critical design decisions and changes and send it to a common list (linux-fsdevel) so as to inform other maintainers. Once a centralized bug tracking system is put in place, the patches should include the bug numbers to help reviewers easily understand the context.

Avoiding common bugs. Once a repository of common bugs is created, we can create complier extensions (as in [26]) that would automatically check for certain common errors. For bugs that cant be checked using the above tool (e.g., file size greater than file system size), we need to develop a regression suite that can be used by all file systems.

9. Related Work

Mining Software Repository provides a valuable resource for relevant work on extraction of information from software repositories [3, 5, 7]. Previous work on extracting information from CVS Repositories for various open source projects, Sourceforge mailing lists and also bugzilla database, correlating these sources to provide a global view of changes has also been explored [6, 4]. People have also looked at patch submission and acceptance in open source communities [8].

Quite a few researchers have looked at creating taxonomies for bugs. Lu et al.[27] have presented a comprehensive list of concurrency bugs in 4 open source projects and constructed a taxonomy for the same. Previous work on bug ontologies have an exhaustive list that include categories like requirements, design, implementation (structural, control flow, sequencing), data etc. [1].

In [28], Chou et al. looked at operating system bugs that were found mainly by a static analysis tool. They further looked at turnaround time and error rates. These dont include bugs that are reported through the mailing lists. To our knowledge, no prior work has been done in understanding the file system bugs in Linux kernel. We plan to leverage some of the above techniques to extract information from multiple mailing lists and understand file system evolution.

10 Conclusions

This paper attempts to trace the file system development process in the Linux kernel. To our best knowledge, no prior work exists on providing an insight in to file system development process. It is important to

note that we have just scratched the surface and more detailed analysis is required to quantify our results. The main contributions of this paper are three folds. First, a tool to automatically mine information from source code repositories and mailing lists. Second, insights into the development process, common problem areas, and stability of file systems. Third, identification of common errors that happen across file systems we studied.

In our study, we verified that like other software systems, lines of code, code complexity, and bugs are correlated in file systems. Even though many developers work on a file system, they do not affect the stability of the file system due to the strong code control practice employed by the Linux kernel community. From our bug analysis study, we found that file system developers do not learn from each other mistakes and end up repeating the same mistakes. Bug tracking mechanisms are almost nonexistent in many file systems. We believe adopting Bugzilla, documenting design decisions, common errors, implementation issues would help file system developers to learn from each others mistake.

Acknowledgements

We would like to thank Prof. Ben Liblit for his guidance and for his generous feedback at various stages of our project. We would also like to thank Prof. Remzi H. Arpaci-Dusseau for suggesting this interesting idea and for his valuable suggestion for improving the project. We thank all the file system developers in Linux, without them there would be no file systems for us to study. You guys are doing a wonderful job and do continue the good work. Finally, we would like to thank Sun Micro Systems for donating Sun Ultra-20 workstation to ADSL group for their research projects.

Reference

[1] B. Beizer and O. Vinter. Bug taxonomy and statistics. Software Engineering Mentor, New York, NY, USA, 2001.
[2] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops, page 26,Washington, DC, USA, 2007. IEEE Computer Society.
[3] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. Software Maintenance, IEEE International Conference on, 0:23, 2003.
[4] D. M. German. Mining CVS repositories, the softchange experience. In Proceedings of the First InternationalWorkshop on Mining Software Repositories, pages 17–21, Edinburg, Scotland, UK, 2004.
[5] M. H. Halstead. Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., New York, NY, USA, 1977.
[6] S. Henry and D. Kafura. Software structure metrics based on information flow. IEEE Trans. Softw. Eng., 7(5):510–518, 1981.
[7] J. Howison and K. Crowston. The perils and pitfalls of mining sourceforge. In In Proceedings of the International Workshop on Mining Software Repositories (MSR 2004, pages 7–11, 2004.
[8] H. Kagdi, S. Yusuf, and J. I. Maletic. Mining sequences of changed-files from version histories. In MSR '06: Proceedings of the 2006 international workshop on Mining software repositories, pages 47–53, New York, NY, USA, 2006. ACM.
[9] Kernel Bugzilla. http://bugzilla.kernel.org/.
[10] Kernel Git Repository. http://git.kernel.org/.
[11] Kernel Repository. http://pub.kernel.org/.
[12] S. Kim and J. E. James Whitehead. How long did it take to fix bugs? In MSR '06: Proceedings of the 2006 international workshop on Mining software repositories, pages 173–174, New York, NY, USA, 2006. ACM.
[13] File systems in Linux. http://www.linux.org/lessons/advanced/x1254.html.
[14] Linux kernel mailing list. http://lkml.org.
[15] Linux file system development mailing list. http://marc.info/?l=linux-fsdevel.
[16] Reiserfs mailing list. http://marc.info/?l=reiserfsdevel.
[17] A. SchrÃ¨uter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk... (short paper). In Proceedings of the 5th International Symposium on Empirical Software Engineering. Volume II: Short Papers and Posters, pages 18–20, September 2006.
[18] Ext2 development mailing list. https://lists.sourceforge.net/lists/listinfo/ext2-devel.
[19] Tim Littlefair. An investigation into the role of Software Metrics in software quality improvement. http://www.chs.ecu.edu.au/ tlittlef/.
[20] T. Zimmermann. Preprocessing CVS data for finegrained analysis. In Proceedings of the First, pages 2–6, 2004.
[21] SGI Corporation. http://sgi.com
[22] IBM Corporation. http://ibm.com

[23] File System Primer http://wiki.novell.com/index.php/File_System_Primer
[24] Thomas J McCabe. A Complexity Measure IEEE Transcations on Software Engineering, Vol SE-2, No 4. December 1976
[25] Reporting bugs for the Linux kernel. http://www.kernel.org/pub/linux/docs/lkml/reporting-bugs.html
[26] Benjamin Chelf , Dawson Engler , Seth Hallem, How to write system-specific, static checkers in metal, Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, p.51-60, November 18-19, 2002, Charleston, South Carolina, USA
[27] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes --- A Comprehensive Study on Real World Concurrency Bug Characteristics" The Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems, March 2008.
[28] Andy Chou , Junfeng Yang , Benjamin Chelf , Seth Hallem , Dawson Engler, An empirical study of operating systems errors, Proceedings of the eighteenth ACM symposium on Operating systems principles, October 21-24, 2001, Banff, Alberta, Canada