Contrasting Development and Release Stabilization Work on the Linux Kernel

Md Tajmilur Rahman
Concordia University
Montreal, Quebec, Canada
mdt_rahm@encs.concordia.ca

Peter C. Rigby
Concordia University
Montreal, Quebec, Canada
peter.rigby@concordia.ca

ABSTRACT

Releasing software involves stabilizing recent development efforts. In this paper we contrast the development period, which involves implementing new features and other major changes, and the stabilization period, which allows for only bug fixes to regressions. In the context of the Linux Kernel we characterize development and stabilization in terms of process, developer effort, developer work areas, commit lag time to release, and the number of changes that are rushed into a release. We find the following. Linux has a relatively long stabilization period (median 2 months). A small group of 55 and 23 developers control the development and stabilization periods, respectively. Much of the re-work done during stabilization is not done by the original developer. The Linux project does not allow developers to rush changes into a release.

1. INTRODUCTION

Large software projects make thousands of changes between releases. During development, new features and other major changes are implemented. Since new changes have been exercised by relatively few developers and end users, they can have a destabilizing effect on the overall software project. To minimize the disruption on others, developers try to isolate functionally independent changes on separate branches or lines of development. Before a new version of the system can be created, these development branches must be integrated and stabilized. A release will be composed of all the features added to the stabilization branch. Once the features have been added, the stabilization branch will only accept bug fixes to regressions. We characterize and contrast development and stabilization work in the following research questions.

RQ1, Release Process: What release process does Linux use? An understanding of the release process provides context for our findings and guides our data mining methodology.

RQ2, Developer Effort: How much effort is expended during development vs stabilization? We want to contrast the effort expended on normal development activities with the effort involved in integration and stabilization of the software for release.

RQ3, Developer Work Area: Do developers work in the same set of files during development and stabilization? During stabilization, no new features are allowed and Linux developers are expected to fix any regressions in the code that they wrote during development. We compare the files and work areas of developers on development and stabilization branches.

RQ4, Lag Time: How long does it take for development vs stabilization changes to be released? Previous work has measured the time amount of time for a change to be released [3], but ignored the different purposes of changes. We measure how quickly bugs are fixed (stabilization) and new features incorporated (development) into the kernel.

RQ 5, Rush-to-Release: Is there a rush-to-release period? The longer the release cycle, the more likely developers will rush in changes so as to avoid waiting for the next release. We measure the amount of churn that occurs before the release stabilization begins.

2. BACKGROUND AND PROCESS

In the early days of Linux development releases were sometimes made more than once per day, prompting Raymond's mantra of "release early, release often" [8]. This trend has continued with many projects adopting increasingly shorter release intervals [4]. For example, Google Plus can release new changes in 36 hours [6] and Firefox and Chromium operate on six week release cycles [4]. In contrast, Linux has adopted comparatively long release cycles.

RQ1, Release Process: What release process does Linux use? Linux using a flexible time-based release schedule [5], which consists of a merge window and stabilization period (See Figure 1). The merge window opens to allow developers to merge changes into the stabilization mainline. The window is open for only two weeks with a standard deviation of 2 days. After the window closes, the first release candidate (rc1) will indicate the start of release stabilization. During stabilization only fixes to regressions and isolated changes, such as device drivers, are merged into the mainline. The time period for stabilizing a release continues until no important regressions are outstanding. Stabilization takes an average of 62 days with a standard deviation, minimum, and maximum of 10, 45, and 93 days, respectively. Figure 2 shows the variations in the Linux release cycle.

Linux clearly has a longer release cycle than Firefox and Google projects. For Firefox, the rapid release cycle has lead to an increasing reliance on automated tests as the community is unable to manually test such frequent releases [4]. In contrast, Linux has few automated tests and relies on developers to test their changes before inclusion in a maintainer's repository. The community uses and tests the mainline for regressions during release stabilization. It is possible that this lack of a large automated test suite increases the length of the release cycle.

Another interesting difference is that Google uses few branches

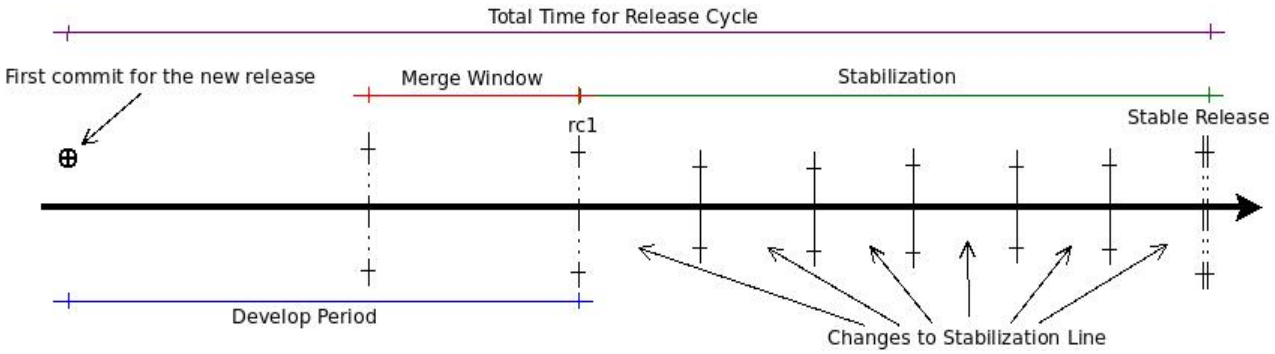

Figure 1: Linux Release Process. Development of subsequent releases occurs in parallel with the stabilization of a release. The two stages join during the merge window where new development is moved onto the stabilization mainline.

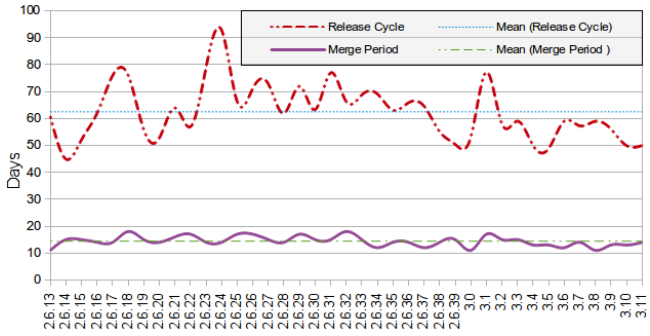

Figure 2: *Time periods for stabilization and merge window*

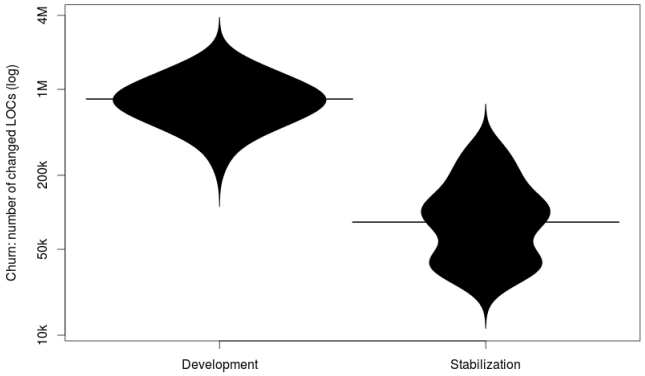

Figure 3: *Churned lines of code per release for development and stabilization*

compared with Linux [6]. The Linux maintainers file contains over 60 different repositories (*i.e.* branches) covering more than 600 topics that crosscut kernel development [9]. This distribution allows functionally independent development efforts to co-exist without conflict and for stabilization and development to proceed in parallel.

3. METHODOLOGY

Development and stabilization work occurs in parallel on different branches, so one cannot differentiate between them based on the time a change was authored or committed (*e.g.,* a development change may be made during the stabilization period). Instead one must look at whether the change was made on a development branch or the stabilization branch.

To determine the type of branch a change was made on, one must examine what Linux developers refer to as the mainline [5]. Branches and, by extension, commits that are merged into the mainline will become part of the next stable release. However, as described above, the mainline goes through two distinct phases: the merge window and the stabilization period. The tags on this mainline dictate the type of work (stabilization or development) that occurred on the branches that merge with the mainline. The merge window is open between a stable release tag (*i.e.* a tag that doesn't contain rc, such as 2.6.14) and the rc1 tag. Any change that is merged onto the mainline between these tags is a development change. In contrast, any change that is merged on the mainline between the rc1 and a stable release is a stabilization change. By walking the git directed cyclic graph we are able to determine which branch a change is made on. Details of the algorithm used can be found in a technical report.[1]

4. RQ2 – DEVELOPER EFFORT

How much effort is expended during development vs stabilization?

In order to quantify the effort involved, we measure the number of commits, churn (number of lines that changed in '.c' and '.h' source files), and the number of people working on the stabilization vs development branches. These basic measures give us a sense of effort involved in developing and releasing Linux.

We find that, of the total 381k commits made to the kernel between 2005 and 2013, 77% of commits are made during development and 23% are made as part of stabilization. In Figure 3, the median development churn per release is 834k lines compared to the stabilization churn of 83k lines. In the median case 91% of the lines changed for a release are made in development with a ratio of 105 lines churned per commit, while 9% of lines changed are during stabilization with 41

[1]Algorithm for traversing the git DAG `http://tajmilur-rahman.com/git_dag.pdf`

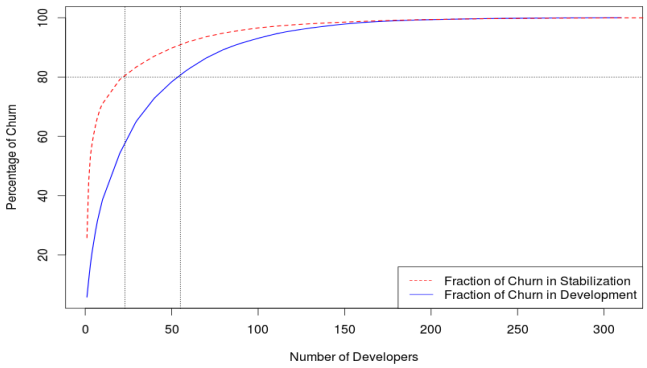

Figure 4: *Cumulative distribution of developer contributions*

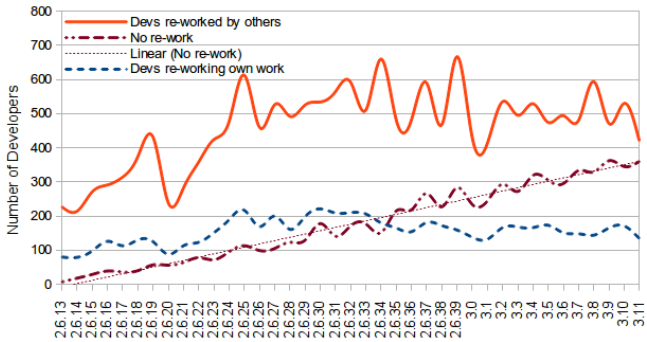

Figure 5: *Developers re-working files during stabilization*

lines churned per commit.

In terms of developers, 10K developers have contributed to Linux, however, 55 developers have done 80% of the development work, while 23 developers have done 80% of the stabilization work (See Figure 4). This result is similar to Mockus *et al.*'s [7] finding that the Apache httpd server had a core group of 15 developers who wrote 80% of the code. Linux is a much larger project, we see that during stabilization 23 developers control the stabilization process. Mockus *et al.* noted that as a system grows, *e.g.,* Mozilla, more complex mechanisms must be used to manage it. In order to integrate the development effort from the larger group of 55 developers that account for 80% of the development effort a chain-of-trust is used to pass changes from less trusted developers up to the trusted stabilization mainline that Torvalds controls and makes releases from [5]. Stabilization work occupies the majority of Torvalds's time and clearly represents large contributions from other core developers. Future work is necessary to compare the level of development and stabilization effort on other projects.

5. RQ3 – DEVELOPER WORK AREA

Do developers work in the same set of files during development and stabilization?

The Linux Kernel has a policy that 'the original developer should continue to take responsibility for the code [they contribute]' [5]. We expect to see developers who modify files during development to fix any problems with those files that arise during stabilization. Of the files that receive rework, we measure the proportion that are done by the original developer vs those that are modified by other developers.

We found that, in the median case per release, there are 161 developers who re-work the same files they modified during development, 480 developers who had their files re-worked by other developers during stabilization, and 171 developers whose changes did not require any re-work during stabilization. These sets of developers are not mutually exclusive. Figure 5 depicts this situation. From these numbers, it would appear that many developers do not take on the responsibility to fix their bugs for a release. Instead a small group of maintainers (See Figure 4) is responsible for integration and bug fixes of regressions during stabilization. We also note a possible trend in Figure 5 that the number of developers who made changes that do not need re-work is increasing.

An alternative explanation, and threat to construct validity, is that integrators are not fixing other developers' bugs, but are re-working the same files to integrate other sets of changes. While a finegrained, line level analysis is left to future work, it is surprising that the majority of files that need re-work were modified by a different developer.

6. RQ4 – LAG TIME

How long does it take for development vs stabilization changes to be released?

We define lag time as the number of days it takes for a change to be integrated into the mainline or a final release. Jiang *et al.* [3] examined the factors that influence lag time on the Linux Kernel. They found large variation in lag times (3 to 6 months), with experienced developers having drastically shorter times. In this work, we are interested in differentiating between development and stabilization work because we want to understand how quickly bugs are fixed (stabilization) and new development incorporated into the kernel.

In Figure 6, we see that stabilization changes take a median of only 8 days to be included in the mainline, while development changes take 35 days to reach the mainline. The median lag time to a release is 47 and 97 days for stabilization and development changes, respectively. Since a small group of expert developers make the majority of stabilization changes (See Figure 4), it would be interesting to add the type of change to Jiang *et al.*'s [3] statistical model of lag times.

7. RQ5 – RUSH-TO-RELEASE

Is there a rush-to-release period?

According to Fogel [1], the greater the time between releases, the more developers will rush changes into the current release in order to avoid waiting for the next release cycle. To empirically test Fogel's statement, we examine the peak weekly churns. We define the peak churn as the top 10% of weekly churned lines. We hypothesize that the peaks in the merge window will be higher than the peaks in the normal development period. A Kolmogorov-Smirnov test of the two distributions gives a p-value of 0.18 indicating that the peak weekly churn in the merge window cannot be statistically differentiated from the peaks in normal development. Figure 7 visually demonstrates the absence of a rush

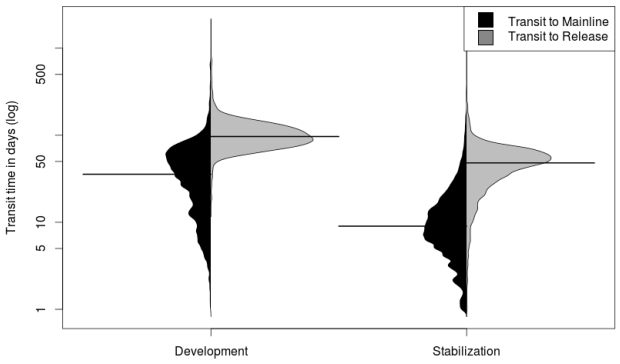

Figure 6: *Lag for the commits to mainline and release*

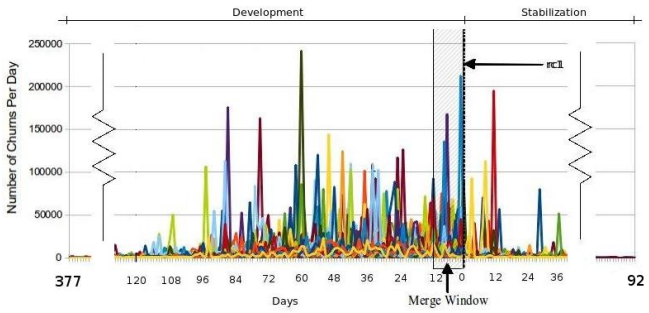

Figure 7: *Time series of daily churn for all releases, anchored at the start of the release stabilization period for each release (*i.e. *rc1)*

before stabilization begins. Although there are clear peaks in the figure, the peaks immediately before the release process begins appear no larger than earlier development peaks. It is also interesting that some rare commits take over a year to be committed into the mainline and released.

While Linux has a relatively long stabilization period, the Kernel is know for being conservative about new features [5], which likely reduces any rush period before release stabilization begins. Furthermore, the far right portion of Figure 7 shows a drastic reduction in churn right before the final stable release is made.

8. CONCLUSION

In this work we differentiated between normal development work and stabilization work. Previous studies of release engineering have looked at the time immediately before and after a release (*e.g.,* [2]) or combined development and stabilization work to examine the entire development cycle [3, 4]. In our study we found the following.

Process: Although Linux spends a median of two months stabilizing a release, development effort continues on other branches with some changes being over a year old (See Figures 2 and 7).

Effort: A large number of developers contribute to Linux, however, 80% of the changes are made by 55 and 23 developers during development and stabilization, respectively (See Figure 4). A small group of developers ensures that each release of Linux is stable.

Work area: The large majority of developers who make a change during development will not fix regressions in the files that they changed. While a fine-grained, line based analysis might produce a different result, it appears to violate the policy that 'the original developer should continue to take responsibility for the code [they contribute]' [5].

Lag time: Stabilization changes, such as bug fixes, take a median of only 8 days to be integrated into the mainline. Development changes take a median of 35 days to reach the mainline (See Figure 6).

Rush-to-Release: Despite a relatively long release cycle, Linux does not allow changes to be rushed into a release. The emphasis on a stable product by core developers likely curbs this impulse (See Figure 7).

This preliminary work has many avenues of future work. Currently our findings describe Linux development, but do not tie different styles of development to outcome measures. For example, we are in the process of examining the impact of the 'fix your own code policy' on the number of regressions seen in a file. We are also keen to replicate this study and extend this study on other software projects.

9. REFERENCES

[1] K. Fogel. *Producing Open Source Software*. O'Reilly, 2005.

[2] A. Hindle, M. W. Godfrey, and R. C. Holt. Release pattern discovery via partitioning: Methodology and case study. In *Proceedings of Mining Software Repositories*, MSR '07, 2007.

[3] Y. Jiang, B. Adams, and D. M. German. Will my patch make it? and how fast?: case study on the linux kernel. In *Proceedings Mining Software Repositories*, pages 101–110. IEEE Press, 2013.

[4] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do faster releases improve software quality? an empirical case study of mozilla firefox. In *Proceedings of Mining Software Repositories*, pages 179–188, June 2012.

[5] Linux. The linux kernel development process. `https://www.kernel.org/doc/Documentation/development-process/2.Process` Accessed February 2013.

[6] J. Micco. *Tools for Continuous Integration at Google Scale*. Google Tech Talk, Google Inc., 2012.

[7] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, 2002.

[8] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly and Associates, 1999.

[9] P. C. Rigby, E. T. Barr, C. Bird, P. Devanbu, and D. M. German. What effect does distributed version control have on oss project organization? In *International Workshop on Release Engineering 2013*, 2013.