Release Stabilization on Linux and Chrome

Md Tajmilur Rahman and Peter C. Rigby, Concordia University

// An empirical study of release stabilization for Linux and Chrome found that few changes were reverted, small teams controlled the stabilization effort and did much of the rework, and despite using rapid release, some rushed changes still occurred before stabilization. //

LARGE SOFTWARE PROJECTS make thousands of changes between releases. During development, new features and other major changes are implemented. Because relatively few developers and end users have used the new changes, those changes can destabilize the overall software system. A release engineer selects and stabilizes the changes before the system is released to a large user base.

We quantified the time and effort expended in the release stabilization of two large successful projects: the Linux kernel and Google Chrome. You can use our measurement tools (https://github.com/tajmilur-rahman/measurements) to compare your projects with Linux and Chrome regarding the questions covered in this article. To use the tools, all you need is a Git repository that contains tags indicating the start of release stabilization and the final release. We’re willing to help you make these measurements and hope that such grounded empirical findings will help transform software development into an engineering discipline.

How Rapid Are Your Releases?

In Linux development’s early days, releases sometimes occurred more than once a day, prompting Eric Raymond’s mantra of “Release early. Release often.” This trend has continued, with many projects adopting increasingly shorter release intervals. For example, Google Plus can release new changes in 36 hours, and Facebook.com releases twice a day on weekdays. Firefox and Chrome operate on six-week release cycles.

To quantify the time and effort for release stabilization, we used the Linux and Chrome process documents’ definitions of the development and stabilization branches. Figures 1 and 2 show these branches. The stabilization and release tags in Git let us traverse the Git directed acyclic graph and identify on which branch a change was made. We extracted churn—the number of lines of code added and removed per commit—from the Git version history. We used the Git author field, not the committer field, to credit work to developers. For more details on our extraction process, see our previous paper.

Figure 1 represents the Linux release process. Linux uses a flexible time-based release schedule, which consists of a merge window and stabilization period. The merge window opens to let developers merge changes into the stabilization mainline. The window is open for only two weeks, with a standard deviation of two 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. New release candidates will be created as regressions are found and fixed.
We found that, on average, there were six release candidates before the final public release. The time period for stabilizing a release continued until no important regressions were outstanding. Stabilization took on average 62 days (represented by the horizontal line in Figure 3), with a standard deviation of 10 days, a minimum of 45 days, and a maximum of 93 days. Since release 2.6.31, release stabilization has become more regular. Figure 3 shows the variations in the Linux release cycle.

Chrome’s release process consists of three channels: development, beta, and stable. At six-week intervals, the code transitions to the subsequent channel. For example, Figure 2 shows that when development began on release 4, release 3 moved to the beta channel, release 2 moved to the stable channel, and release 1 was published as a final production release. In our data extraction scripts, we can identify on which channel a commit was made, on the basis of its version number. In this article, we don’t differentiate between the beta and stable channels because both are related to release stabilization.

Release stabilization took an average of 91 days, with a standard deviation of 11 days, a minimum...
of 56 days, and a maximum of 149 days. Figure 3 shows the variations in Chrome’s release cycle; the horizontal line shows the ideal 12-week stabilization period. Immediately after Chrome adopted a rapid release cycle, release times varied significantly, with some releases taking substantially longer than 12 weeks. Recent releases have become much more regular.

**How Much Effort Do You Expend in Stabilizing a Release?**

To get a sense of the effort involved in developing and releasing Linux and Chrome, we used three basic measures: the number of commits, the churn, and the number of people working on the development versus stabilization branches.

For Linux, of the 381K commits made to kernel source files between 2005 and 2013, 77 percent were during development and 23 percent were during stabilization. As Figure 4 shows, the median development churn per release was 834 KLOC, compared to the stabilization churn of 83 KLOC. A Wilcoxon test showed that this difference was statistically significant, with \( p << 0.001 \). In the median case, 91 percent of the lines changed for a release were during development, with a ratio of 105 lines churned per commit, whereas 9 percent of the lines changed during stabilization, with 41 lines churned per commit. Linux release engineers tended to make small changes during stabilization.

For Chrome, of the 164K commits made to source files between 2008 and 2014, 85 percent were during development and 15 percent were during stabilization. The median development churn per release was 808K lines, compared to the
stabilization churn of 51 KLOC (see Figure 4). A Wilcoxon test showed that this difference was statistically significant, with \( p \ll 0.001 \). In the median case, 93 percent of the lines changed for a release were during development, with a ratio of 11 lines changed per commit, whereas 7 percent of the lines were changed during stabilization, with 165 lines churned per commit. Chrome release engineers tended to make large changes during stabilization.

Figure 5 shows the distribution of developer contributions. For Linux, 10,000 developers contributed; however, 55 developers performed 80 percent of the development, and 23 developers performed 80 percent of the stabilization. For Chrome, 98 developers made 80 percent of the changes during development, and 10 developers made 80 percent of the changes during stabilization.

This result is similar to Audris Mockus and his colleagues’ finding that the Apache httpd server had a core group of 15 developers who wrote 80 percent of the code.6 Linux is a much larger project; 23 developers controlled stabilization. Mockus and his colleagues noted that as a system grows (for example, Mozilla), its management requires more complex mechanisms. The Mozilla project had 13 release engineers.9 To integrate the development effort from the larger group of 55 developers that accounted for 80 percent of the development effort, Mozilla used a chain of trust to pass changes from less trusted developers to the trusted stabilization mainline that Linus Torvalds controls and makes releases from.6 Stabilization occupies most of Torvalds’s time and clearly represents large contributions from other core developers.

Do You Stabilize Your Own Code during a Release?

DevOps combines operational work, including release engineering, with development work. One example of a DevOps combination is requiring developers, instead of integrators, to fix their own code during stabilization. We find that much of the stabilization effort is still on the release engineers’ shoulders.

The Linux Kernel has a policy that “the original developer should continue to take responsibility for the code [they contribute].”6 Chrome also has this expectation.5 We expect developers who modify files during development to fix any problems with those files that arise during stabilization.

Of the files that were modified in both periods, we measured the proportion of those that the original developer modified to those that other developers and integrators modified. In the median case per release,

- 161 Linux and 258 Chrome developers modified files during stabilization that they had changed during development,
- 480 Linux and 307 Chrome developers had their files modified by other developers during stabilization, and
- 171 Linux and 322 Chrome developers had changes requiring no modification during stabilization.

These sets of developers weren’t mutually exclusive. Figure 6 depicts this situation for Linux and Chrome. From these numbers, it would appear that many developers didn’t take on the responsibility to fix their bugs for a release. Because the number of developers making changes was much larger than the core group.
of developers, many of these changes were likely made by transient developers who didn’t remain to fix the bugs in their small code contribution. Instead, a small group of integrators (see Figure 5) handled integration and bug fixes of regressions during stabilization.

An alternative explanation, and a threat to our study’s validity, is that integrators were working in other areas of the file and weren’t modifying code lines related to the changes made during development. Although a fine-grained, line-level analysis is left to future research, it’s surprising that a different developer modified most files that needed modification during stabilization.

For Linux, the rework done by other developers fluctuated dramatically from 214 to 667 lines and didn’t show a clear trend. However, the number of developers whose files

![Figure 6](image-url)
didn’t need rework clearly increased, with an adjusted $R^2 = 0.97$ and $p << 0.001$. This trend likely showed a maturing in the selection of code from external contributors. For Chrome, all three categories increased, indicating an increase in the number of developers contributing to Chrome but obscuring other patterns.

Although release engineers modified a large number of files, few changes were reverted during stabilization. In the median case, there were 104 reverts per stabilization period for Linux and 3 for Chrome. This accounted for only 2.3 percent of the total stabilization commits for Linux and less than 1 percent for Chrome. For Linux, 55 percent of reverts were during stabilization; for Chrome, 76 percent were during development.

**How Quickly Do You Release Changes?**
The longer a change takes to transit from development through stabilization to release, the longer users will be waiting for bug fixes and features. In highly competitive environments, small differences in release date can be the difference between success and failure. We find that even with “fixed” rapid-release dates, slips in the release schedule still occur.

We wanted to understand how quickly bugs are fixed during stabilization and how quickly new development is incorporated into Linux and Chrome. *Transit time* is the number of days for a change to be integrated in the stabilization branch or included in a final release. Previous research measured the transit time for a change to be released but ignored the different purposes of changes and found large variations in transit times (three to six months). By differentiating between change types, we found that most of the variation can be explained by whether the change was a fix made during stabilization or a change made during normal development.

Figure 7 shows that the median time for stabilization changes (fixes to regressions) to be included in a stabilization branch was only eight days for Linux and less than one day for Chrome. In contrast, the time for development changes to reach the stabilization branch was 35 days for Linux and 21 days for Chrome. A Wilcoxon test showed that these differences were statistically significant at $p << 0.001$.

The transit time for a stabilization change to be released was 47 days for Linux and 78 days for Chrome, whereas a development change took 97 days for Linux and 109 days for Chrome. Although Chrome started release stabilization every six weeks and produced a new release every six weeks, changes to Chrome took longer to reach users than Linux changes. This is because Chrome stabilized two releases at a time, whereas Linux stabilized only one.

**Do You Rush Changes into a Release to Avoid Waiting for the Next One?**
Nobody wants to wait for the next release, especially if there’s only one release per year. As a release date approaches, developers might feel pressured to release features that aren’t yet stable and well integrated. This pressure increases with long release cycles because developers might rush changes into a release to avoid waiting for the next one. Chrome switched to a shorter release cycle
to avoid pressuring developers into rushing unstable code into a release.

We wanted to empirically test whether developers rushed changes in right before release stabilization and feature freeze. We defined the churn rate as the number of lines changed per day and defined the rush period as two weeks before release stabilization. The rush period corresponds to the Linux merge window and is one-third of the Chrome development period. The normal development period is the period between releases before the rush period begins. On Linux, this is the two months before the merge window opens; on Chrome, it’s the development cycle’s first four weeks. Because we were interested in development and not integration, we excluded all merge commits. We also used the author date instead of the committer date so that “cherry-picked” changes would be counted during development, not when they were picked. We hypothesized that the churn rate in the rush period would be higher than the churn rate during normal development.

To test this hypothesis, we used the nonparametric Wilcoxon test to compare the two distributions’ churn rate. We found a statistically significant difference in the churn rate between normal development and the two weeks before stabilization ($p = 0.007$ for Linux and $p = 0.0008$ for Chrome). Figure 8 shows that, in the median case, Linux developers changed 5 KLOC per day for the normal period and 6 KLOC per day for the rush period. The corresponding values for Chrome were 14 KLOC and 17 KLOC. These differences represented an increase in median daily churn during the rush period of 20 percent for Linux and 21 percent for Chrome. Despite a rapid release cycle, some rush still occurred before release stabilization. It would appear that some degree of rush is unavoidable.

Crosscutting the time and effort measures we examined are the three factors that Facebook’s lead release engineer Chuck Rossi considers when creating a release: the schedule, the quality, and the feature set. All three can’t be optimized at the same time, so Rossi sacrifices the feature set but releases stable features on schedule and drops any feature that would reduce quality.

Quality is also paramount to Linus Torvalds, whose main jobs are integration and release stabilization. He ranks first in the number of integration merges and 52nd in the number of changes made to Linux. He says,

I’m not claiming this [change …] is really any better/worse than the current behaviour from a theoretical standpoint, but at least the current behaviour is _tested_, which makes it better in practice. So if we want to change this, I think we want to change it to something that is _obviously_ better.11

Likewise, in the article “Release Early, Release Often,” Anthony Laforge, who introduced rapid release to Chrome development, states that

While pace is important to us, we are all committed to maintaining high quality releases—if a feature is not ready, it will not ship in a stable release.12

So, we conclude that Chrome and Linux value quality over schedule and schedule over features. ☞

---

**FIGURE 8.** The distribution of daily churn during normal development and two weeks before stabilization. The daily churn rate increased by approximately 20 percent during the rush period before release stabilization.
MD TAJMILUR RAHMAN is a graduate student in software engineering at Concordia University in Montreal. He’s studying release-engineering practices, including feature toggles and branching strategies. His research interests include identifying and mitigating the events that disrupt developers and lead to software failures. Rahman received a BSc in computer science and engineering from Northern University Bangladesh. Contact him at mdt_rahm@encs.concordia.ca.

PETER C. RIGBY is an assistant professor of software engineering at Concordia University in Montreal. His research interests focus on understanding how developers collaborate to produce successful software systems. Rigby received a PhD in computer science from the University of Victoria. Contact him at peter.rigby@concordia.ca.

REFERENCES

4. C. Rossi, “Native Mobile App Releases,” 2014; www.youtube.com/watch?v=EtZkkq7dGM.

ABOUT THE AUTHORS

Showcase Your Multimedia Content on Computing Now!


If you’re interested, contact us at cga@computer.org. All content will be reviewed for relevance and quality.