Release Stabilization on Linux and Chrome

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

An empirical study of the time and effort involved in release stabilization on Linux and Chrome finds that a small teams control the stabilization effort, few changes are reverted, and much of the re-work is not done by the original developer. Despite using regular rapid release cycles, there is a rush period before release stabilization begins and the length of a stabilization period can vary by 10 days.

LARGE software projects make thousands of changes between releases. During development, new features and other major changes are implemented. Since new changes have been used by relatively few developers and end users, they can have a destabilizing effect on the overall software system. The job of a release engineer is to select and stabilize the changes to a system before it is released to a large user base. This article quantifies the time and effort involved in the release stabilization of two large successful projects – the Linux kernel and the Google Chrome browser. We provide practitioners with our measurement tools,[1] so that they can compare their own projects with Linux and Chrome on the questions listed below. To use the tool, all you need is a Git repository that contains tags indicating the start of release stabilization and the final release. We are willing to help practitioners make these measurements and hope that this kind of grounded empirical findings will help transform software development into an engineering discipline.

How quickly do you release changes?
The longer it takes a change 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, there are still slips in the release schedule.

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 to fix their own code during stabilization instead of placing this effort on integrators. We find that much of the stabilization effort is still on the shoulders of release engineers.

Do you rush changes into a release to avoid waiting for the next release?
Nobody wants to wait for the next release, especially if there is only one release per year. As a release date approaches, developers may feel pressured to release features that are not yet stable and well integrated. Chrome switched to a shorter release cycle to avoid pressuring developers into rushing unstable code into a release. Even with these short release cycles, both Chrome and Linux see an increase of development work right before release stabilization begins. Some degree of rush seems unavoidable.

[1]Measurement tools `https://github.com/tajmilur-rahman/measurements`

Release Cycle

How rapid are your releases?

Although Linux and Chrome use regular rapid release schedules, release stabilization varies by approximately 10 days. Over time, both projects have become better at releasing on schedule.

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" [9]. This trend has continued with many projects adopting increasingly shorter release intervals [2]. For example, Google Plus can release new changes in 36 hours [6] and `Facebook.com` releases twice a day on weekdays [10]. Firefox and Chrome operate on six week release cycles [2, 4].

To quantify the time and effort involved in release stabilization, we use the definitions of the development and stabilization branches as defined by the Chrome and Linux process documents [5, 4]. These branches can be seen in Diagrams 1 and 2. The stabilization and release tags in Git allow us to traverse the Git DAG and identify on which branch a change was made. We extract Churn, *i.e.,* the number of lines added and removed per commit, from the Git version history. We use the Git author field and not the committer field to credit work to developers. For more details on our extraction process see our preliminary work [8].

The **Linux release process** is represented in Figure 1. According to the release process documents, Linux uses a flexible time-based release schedule [5], which consists of a merge window and stabilization period. 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

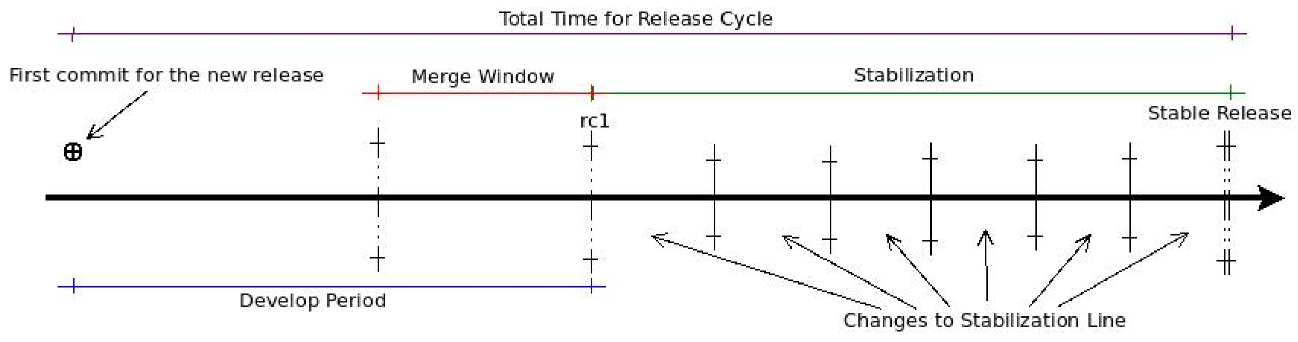
Total Time for Release Cycle
First commit for the new release
Merge Window
Stabilization
rc1
Stable Release
Develop Period
Changes to Stabilization Line

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.

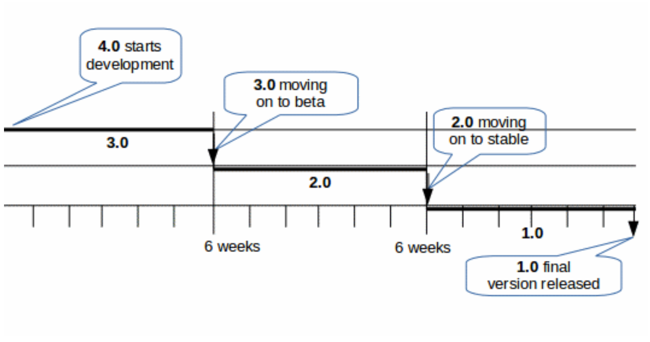

Figure 2: Chrome release process: Development occurs in parallel with two stabilization branches – beta and stable. At six week intervals, each branch is moved onto the subsequent stage, for example, development moves to beta.

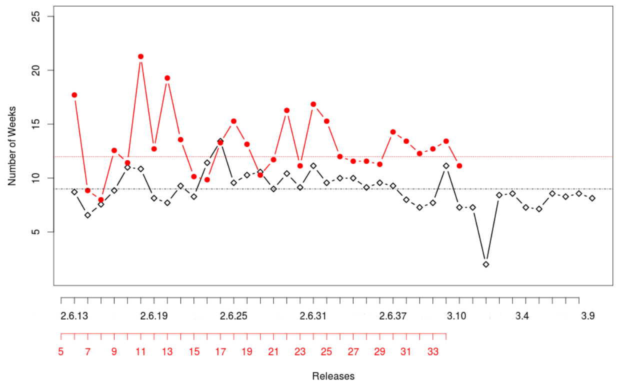

Figure 3: *Length of stabilization and development periods for Chrome, upper line, and Linux, lower line. Horizontal lines are expected amount of time for release stabilization.*

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. New release candidates will be created as regressions are found and fixed. We find that on average there will be six release candidates before the final public release. The time period for stabilizing a release continues until no important regressions are outstanding. Stabilization takes on average 62 days (represented by the horizontal line in Figure 3) with a standard deviation, minimum, and maximum of 10, 45, and 93 days, respectively. 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 into the subsequent channel [4]. For example, in Figure 2 we see that when development work begins on release 4, release 3 will be moved into the beta channel, release 2 will be moved to the stable channel, and release 1 will be published as a final production release. In our data extraction scripts we are able to identify which channel a commit was made on based on its version number. In this work, we do not differentiate between beta and stable as both channels are related to release stabilization.

We find that release stabilization takes an average of 91 days with a standard deviation, minimum, and maximum of 11, 56, and 149, respectively. Figure 3 shows the variations in Chrome's release cycle and the horizontal line shows the ideal 12 week stabilization period. Immediately after adopting a rapid release cycle, there was significant variance in release times with some releases taking substantially longer than 12 weeks. We can see that recent releases have become much more regular.

Effort

How much effort do you expend in stabilizing a release?

A very small group of developers control the stabilization of a release – 23 and 10 developers for Linux and Chrome. The majority of changes are made during development with 9% and 7% of lines changed during stabilization, respectively.

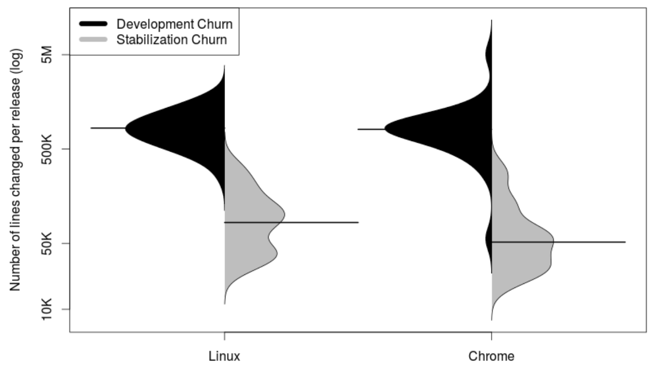

Figure 4: *Churned lines of code per release for development and stabilization*

We use three basic measures to get a sense of effort involved in developing and releasing Linux and Chrome. We measure the number of commits, churn (number of lines that changed), and the number of people working on the stabilization vs. development branches.

For **Linux** we find that, of the total 381k commits made to kernel source files between 2005 and 2013, 77% of commits are made during development and 23% are made as part of stabilization. In Figure 4, the median development churn per release is 834k lines compared to the stabilization churn of 83k lines. A Wilcoxon test shows that this difference is statistically significant with $p \ll 0.001$. 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 lines churned per commit. Linux tends to make smaller changes during release stabilization.

For **Chrome** we find that, of the total 164k commits made to source files between 2008 and 2014, 85% of the commits are made during development and 15% are made as part of stabilization. The median development churn per release is 808K lines compared to the stabilization churn of 51K lines (see in Figure 4). A Wilcoxon test shows that this difference is statistically significant with $p \ll 0.001$. In the median case 93% of the lines changed for a release are in development with a ratio of 11 lines changed per commit, while 7% of lines are changed during stabilization with 165 lines churned per commit. In contrast to Linux, it is interesting that Chrome release engineers tend to make very large changes during release stabilization.

For the **Chrome team** there are 10 developers that make 80% of the changes during stabilization and 98 developers change 80% of the lines changed during development. For **Linux**, 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 5).

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,

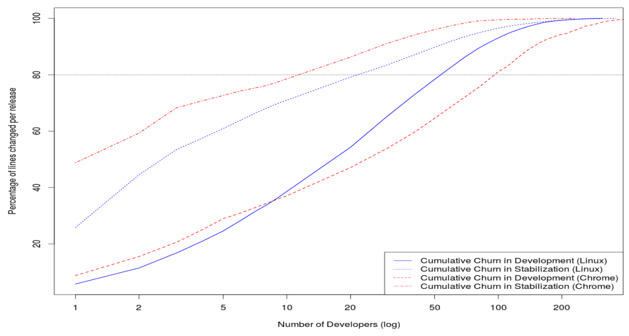

Figure 5: *Cumulative distribution of developer contributions*

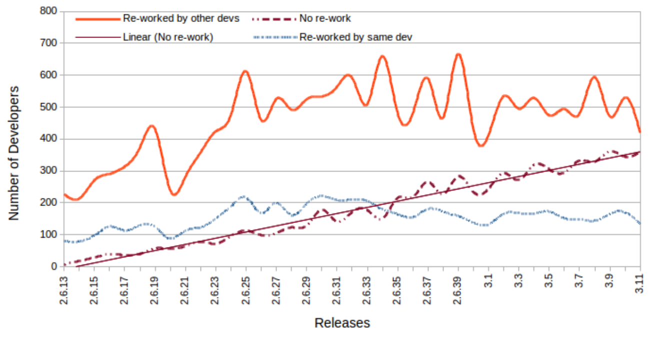

Figure 6: *Linux developers re-working files during stabilization*

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.

Ownership

Do you stabilize your own code during a release?

Many developers have their files modified by another developer during stabilization. Few commits are reverted during stabilization.

The Linux Kernel has a policy that 'the original developer should continue to take responsibility for the code [they contribute]' [5]. Chrome also has this expectation [4]. We expect developers who modify files during development to fix any problems with those files that arise during stabilization. Of the files that are modified in both periods, we measure the proportion that are done by the original developer vs. those that are modified by other developers and integrators.

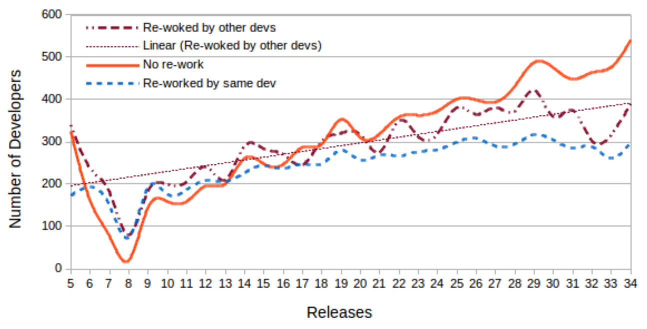

Figure 7: *Chrome developers re-working files during stabilization*

For Linux and Chrome, respectively, we find in the median case per release there are 161 and 258 developers who modified the same files they changed during development, 480 and 307 developers who had their files modified by other developers during stabilization, and 171 and 322 developers whose changes did not require any modification during stabilization. These sets of developers are not mutually exclusive. Figures 6 and 7 depicts this situation for Linux and Chrome, respectively. From these numbers, it would appear that many developers do not take on the responsibility to fix their bugs for a release. Since the number of developers making changes is much larger than the core group of developers, it is likely that many of these changes are made by transient developers who do not remain to fix bugs in their small code contribution. Instead a small group of integrators (See Figure 5) is responsible for integration and bug fixes of regressions during stabilization.

An alternative explanation, and threat to validity, is that integrators are working in other areas of the file and are not modifying code lines related to the changes made during development. While a fine-grained, line level analysis is left to future work, it is surprising that the majority of files that need modification during stabilization were modified by a different developer.

For Linux, the amount of re-work done by other developers fluctuates dramatically from 214 to 667 and does not show a clear trend. However, the number of developers who's files do not need to be re-worked shows a clear increasing trend line with an adjusted $R^2 = .97$ and $p \ll 0.001$. This trend likely shows a maturing in the selection of code from external contributors. For Chrome, all three categories are increasing, indicating an increase in the number of developers contributing to Chrome, but obscuring other patterns.

Although a large number of files are modified by release engineers, few changes are reverted during stabilization. For Linux and Chrome, in the median case, their are 104 and 3 reverts per stabilization period. This accounts for only 2.3% and less than 1% of total stabilization commits for Linux and Chrome. For Linux 55% of reverts are made during stabilization, while for Chrome, the majority of reverts, 76%, are made during development.

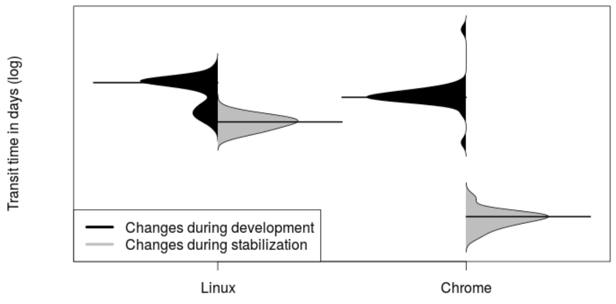

Figure 8: *Transit time for the commits to stabilization branch and release*

Speed

How quickly do you release changes?

Changes made during stabilization (*e.g.,* fixes to regressions) are integrated and released much more quickly than development changes.

We want to understand how quickly bugs are fixed during stabilization and new development incorporated into Linux and Chrome. We define transit time as the number of days it takes for a change to be (1) integrated in the stabilization branch or (2) included in a final release. Previous work measured the transit time for a change to be released [1], but ignored the different purposes of changes and found large variations in transit times (3 to 6 months). By differentiating between change types we find 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.

In Figure 8, for Linux and Chrome, we see that stabilization changes (fixes to regressions) take a median of only 8 days and less than 1 day to be included in a stabilization branch. In contrast, development changes take 35 and 21 days to reach the stabilization branch for Linux and Chrome, respectively. A Wilcoxon test shows that these differences are statistically significant at $p \ll 0.001$.

For Linux and Chrome, the transit time for a stabilization change to be released is 47 and 78 days, while a development change takes 97 to 109 days, respectively. Although Chrome starts release stabilization every six weeks and produces a new release every six weeks, changes to Chrome take longer to reach the user than Linux changes because Chrome stabilizes two releases at a time while Linux stabilizes only one release.

Rush

Do you rush changes into a release to avoid waiting for the next release?

Two weeks before stabilization begins, the daily churn rate increases by 20% and 21% for Linux and Chrome, respectively.

As a release date approaches, developers may feel pressured to release features that are not yet be stable and well integrated. This pressure increases with long release cycles as developers may rush changes into a release to avoid waiting for the next release. Our goal is to empirically test whether developers rush changes in right before release stabilization and feature freeze. To test this, we define the churn rate as the number of lines changed per day. We define the rush period as two weeks before release stabilization begins. The rush period corresponds to the Linux merge window and is one third of the Chrome development period. The normal development period is defined as the period between releases before the rush period begins. On Linux, this is the two months before the merge window opens and on Chrome it is the first four weeks of the development cycle. Since we are interested in development and not integration, we excluded all merge commits. We also used the author date instead of the committer date so that 'cherrypicked' changes will be counted during the development period not when they are picked. We hypothesize that the churn rate in the rush period will be higher than the churn rate during normal development.

To test this hypothesis, we use the non-parametric Wilcoxon test to compare the churn rate of the two distributions. For Linux and Chrome, we find a statistically significant difference in daily churn rate between normal development and the two weeks before stabilization begins ($p = 0.007$ and $p = 0.0008$, respectively). In Figure 9, we see that in the median case, Linux developers change 5k and 6k lines per day for normal and rush period respectively. The values for Chrome are 14k and 17k. These differences represent a 20% and 21% increase in median daily churn during the rush period for Linux and Chrome, respectively. Despite have a rapid release cycle, there is still some degree of rush before release stabilization begins.

CROSSCUTTING the time and effort measure we have examined are the three factors that Chuck Rossi, the lead release engineer at Facebook, considers when creating a release: the schedule, the quality, and the feature set [10]. All three cannot be optimized at the same time, so Chuck sacrifices the feature set, but releases stable features on schedule and drops any feature that would reduce quality.

Quality is paramount to Linus Torvalds who's main job is integration and release stabilization. He ranks first in terms of number of integration merges and 52nd in terms of the number of changes made to Linux.

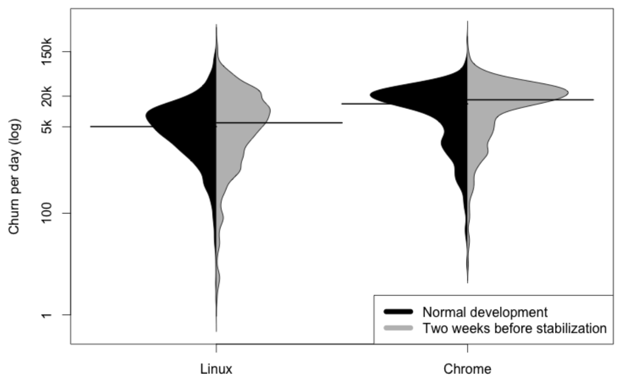

Figure 9: *Distribution of daily churn during normal development (left) and two weeks before stabilization (right)*

"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."
–Torvalds, [11]

Likewise, in an article titled "release early, release often," Anthony Laforge, who introduced rapid release to Chrome development, states:

"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."
–Laforge, [3]

Chrome and Linux value quality over schedule and schedule over features.

1. REFERENCES

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

[2] 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.

[3] A. Laforge. Release Early, Release Often. `http://blog.chromium.org/2010/07/release-early-release-often.html`, July 2010.

[4] A. Laforge. Chrome release cycle. bit.ly/1qz4ATj, January 2011.

[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] M. T. Rahman and P. C. Rigby. Contrasting Development and Release Stabilization Work on the Linux Kernel. In *International Workshop on Release Engineering 2014*, 2014.

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

[10] C. Rossi. Native mobile app releases. https://www.youtube.com/watch?v=Nffzkkdq7GM, April 2014.

[11] L. Torvalds. "Re: IRQF_DISABLED problem [maintaining status quo unless change is obviously better]". Linux Kernel Mailing List `http://kerneltrap.org/mailarchive/linux-kernel/2007/7/26/122293`, 2007.