Git-ing a Better Understanding of Commits

An Exploratory Study of Git Repository Commit Size Metrics

Drew Guarnera, Heather Michaud, Evan Purkhiser  
{dtg3,hmm34,emp36}@zips.uakron.edu

*Abstract*—Version control is an essential tool of modern software development. It provides the means to reverse detrimental changes with minimal effort, allows developers to work collaboratively on large scale applications, and can provide a detailed history of an application’s lifecycle. The historical archive of a system proves crucial in revealing valuable patterns, and the understanding of these evolutionary patterns can improve the discipline of software development. Many version control systems exist, however this work specifically focuses on the Git version control tool, a modern system that is rapidly gaining ground within the software development community. Since Git Write follows a ‘distributed version control’ model, any source code repository with a publicly accessible Git endpoint may be cloned by anyone using the Git tool. Changes made to each repository can live completely independent or may be pushed or pulled between clones. Due to this flexibility, Git truly sets itself apart from older and more established version control tools by offering users a truly customizable workflow.

This study is an attempt to improve upon existing work by examining the commit history for 13 Git repositories. This data is analyzed and compared to the foundational work to determine if use of the Git version control system changes the way developers make commits. Additionally, commit size metrics are calculated to investigate if oversized commits are prevalent during development, considering that more modern practices and version control tools such as Git emphasize on small commits.

Keywords—DVCS; Git, Source Control; Source Code Metrics

# Introduction

Throughout the course of a software’s life cycle, any changes made to the project are recorded in the project’s version control system. When that version control system is Git, modifications are made to the developer’s local repository by committing the changes, and these local modifications are put into the central repository by pushing the changes. By exploiting the qualities of a version control system, the entire history of a project can be obtained through commit records.

Modern software practices encourage small commits to gradually improve the system and decrease the likelihood of bugs [citation?]. Git specifically motivates these practices with the commit-push workflow. Smaller changes can be committed, and the larger aggregation of changes can be pushed.

We feel that by studying repositories using modern version control system such as Git, we can gain a better understanding of common software development practices in terms of commit sizes. In previous studies [commit size distribution] [typical commit], commit size metrics were obtained from repositories and it was found that less than 15% of commits were considered large. In [typical commit] specifically, only repositories that used SVN were studied. Our ultimate goal is to determine if large commits are a frequent and problematic occurrence in repositories using Git, whose workflow exemplifies modern practices and stresses small commits. To address this goal, we examined three size metrics utilizing a tool we developed for traversing repository commit histories:

1. Number of lines of code (LOC) added, deleted, or modified in a commit
2. Number of files added, deleted, or modified in a commit
3. Number of hunks added, deleted, or modified in a commit

All metrics were obtained using the pygit2 [pygit2] library, which exposes the core functionality of Git. The aggregate files that have been modified in a commit contain the number of lines of code that have been modified at the lowest measured level of granularity. In between LOC and files measurements lie the number of hunks modified. A hunk is a group of lines that have been modified together within a file.

The study presented here examines 13 open source software projects. After collecting data for each project’s entire revision history, each calculated metric for lines, files, and hunks are separated into categories of extra-small, small, medium, large, and extra-large. These categories allow us to calculate trends over system and study relationships between categories and metrics by computing correlation coefficients.

The paper is organized as follows. Section 2 discusses related work in commit size studies, including the original study for which this paper replicates. Section 3 presents a background of Git, discusses how it is separated from other distributed version control systems and common workflows associated with Git. Section 4 presents design and implementation details of our python tool created to analyze source code size metrics regarding multiple repositories’ commit histories. Section 5 demonstrates an evaluation of the results gathered. Section 6 explores threats to the validity of the results. Section 7 describes future work, and Section 8 concludes.

# Background

Git is a distributed version control system (DVCS) offering speed and flexibility not found in other well-known version control systems. Originally developed in 2005 by Linux Torvalds for the primary purpose of managing the source code of the Linux kernel, Git has recently taken the software development world by storm.. With the advent of social coding platforms such as GitHub, Bitbucket, and Gitorious, Git has seen widespread growth with over 6 million repositories hosted on GitHub alone.

## Distributed Version Control

Git sets itself apart from many other version control systems thanks to the fact that it offers a distributed approach to source control. While most well established version control systems follow a centralized model, such as SVN, where the source code exists in one central location and all changes to the source are made to that central repository, Git does not. Instead of “checking out” a local copy of a remote repositories working tree like with SVN, Git allows for a complete copy (known as a “clone” operation) of any given repository to be made. Each clone of a Git repository can be thought of as a completely independent repository that simply shares ancestral commits. When committing to a cloned repository the changes will only become available in that particular “clone” of the repository. Git does offers the ability to share these changes between repositories using the concepts of ‘pushing’ and ‘pulling’ changes between repositories that have previous commits in common (ancestral commits).

While most version control systems support the concept of branching Git makes this a first class feature of the tool and it becomes a core concept of the nature of Git. Git offers the ability to very quickly branch off of the current tip of a checked out branch, creating a new branch referencing that same commit. When making commits on a newly created branch these commits will only be available on the new branch while the original branch will be unchanged. By default, all repositories start with a ‘master’ branch.

Branching also plays a key part in the distributed nature of Git as any cloned repository may be thought of as a “remote branch” from which it was cloned from (known as the “origin”). It is also possible to have a remote branch of any clone of the repository, Git makes no distinction between these remote branches. Changes between any branch can then be pushed and pulled between local branches, remote and local branches, or even between two remote branches.

## DVCS Workflows

Because of the distributed nature of Git, the user has the ability to completely customize their workflow of both their usage of Git, as well as the workflow for the entire project team. These workflows can become relevant in the study of commit metrics thus the following workflows have been taken into consideration:

1. Incremental Commit

A common way for new users to work with Git is to use the very basic workflow of continuously making changes to the source code, and incrementally committing their changes to the projects master branch as they make these changes. If working in a team environment, there will often only be one central Git repository that committers push their changes to. This mirrors the SVN way of working in that branching is rare and commits are made and pushed to a central location.

1. Reorganizing Commit Workflow

Because Git commits are made to the local repository it is possible to, at any given time, “go back in history” and reorder, remove, break up, or “squash” commits together. These features allow for a unique workflow in which the user can spend extra time organizing their commits after they have already been made, allowing for a cleanly organized history where each commit makes logical sense.

1. Branched-Merging Workflow

A common branching strategy with Git is to create a “topic branch” for each new feature to be added to the project. This is allows for work on a particular feature to be done independent of any work on the master branch and can offer a clean separation of workflow concerns. When changes are completed in a branch and are ready to be brought into the master branch of the project it is possible to create a “merge” commit. This is a special type of commit that has more than one parent commit reference. All changes made since the most recent ancestral commit of both parents will be reflected in this commit; however both lines of work are kept track of.

1. Branched-Squash Workflow

Similar to the Branched-Merge workflow, it’s also possible to squash multiple commits on a branch down into a single commit when merging it back into the mainline branch. This will discard entire branches commit history and merge the changes in as a single commit. This workflow is often preferred for projects where topic branches tend to be smaller and more concise features, as opposed to long running  feature branches.

# Tool

## Approach

In order to properly replicate the initial study [typical commit], multiple Git repositories would need to be cloned to a local machine for the required analysis. The commits of each repository would then require analysis, and some of the repositories utilized in this study could contain anywhere from around ten thousand to several hundred thousand commits. To process the commits and obtain the required code change size metrics, a diff would need to be calculated between all consecutive pairs of commits starting with the initial commit and ending with the most recent commit to the repository, i.e.

git diff c1 c2, git diff c1 c2, etc.

The metrics of interest from each commit pair diff are the number of files modified, number of hunks, and number of lines modified between the commit. These values will be totaled separately per diff and grouped into the categories of extra small, small, medium, large, and extra-large based on their respective number of occurrences.

With the amount of data in need of analysis, tool assistance would be an absolute necessity in order to compile the metrics for this study. The initial approach taken toward tooling was to avoid reinventing the wheel and seek out an existing tool, and while there was no shortage of tools for software metrics, Cloc [cloc] and Ohcount [ohcount] initially stood out. Unfortunately while vetting the tools during the early testing and planning phase, it became clear that while Cloc and Ohcount would indeed provide software size metrics such as SLOC and LOC, these tools were unable to assist with providing metrics based on revision differences since the granularity was limited to the calculation of size metrics for an entire file or source code project directory.

When the search for off the shelf tooling proved unsuccessful, the Git command line client was then examined. This option was more promising as the Git command line tool has a very convenient option for its diff functionality called ‑‑shortstat. When combined with two revision identifiers the command provided a very clean single line diff response that detailed the number of files changed and the number of line additions and deletions. The only shortcoming of this approach was that finding the number of hunks in each commit was troublesome as that information is not included in the abbreviated diff report information. Using the standard diff option and parsing the full output of the diff report while not impossible, seemed inefficient and time consuming, so the search for a cleaner and more comprehensive solution lead to the libgit2 [libgit2] library (covered in the following section).

The initial tool while functional, was quite basic. Test repositories were hard coded into the application, and upon execution, the repositories were cloned from GitHub [github]. Once a repository was downloaded, an array containing all of the commits, represented by cryptographic SHA hashes, was created and ordered from oldest to newest commit. These hashes were fed in pairs to a differencing function in the library which performed the internal Git differencing between the two revisions of the repository, and provided access to a data structure containing information on the files, hunks (consecutive blocks of code changes), and lines all altered between the two revision commits. This first endeavor proved to be a solid base for the project, but the performance of the approach proved inefficient. Upon closer analysis of the raw data being processed, it became more clear as to why the first attempted failed.

While a large amount of data was expected from the diff process, it was unclear during early testing of the tooling just how large some of the diffs would be. One particular repository, FreeBSD caused significant resource strain on the tool due to the sheer size of the diff object produced. As an example, one particular diff contained approximately seven thousand file modifications, consisting of nearly 250,000 line additions and 500,000 line deletions. Large commits such as this had serious ramifications on the tool’s performance, and ultimately exhausted all of the RAM in our two data collection systems, each equipped with eight gigabytes of RAM and crashed the OS (Ubuntu 12.04 64-bit and Arch Linux 3.12.1 64-bit). After observing this limitation first hand, adjustments were made in an effort to improve the efficiency and stability of the tool.

## Final Tool Design and Implementation

After experimentation with various tooling options, the authors’ arrived at a combination of libgit2 with Python for the core implementation of the final tool supplemented by output from the Git command client’s diff --shortstat option. Libgit2 is used to expose the core functionality of Git allowing for third party applications to gain programmatic access to the details of a Git repository through the use of its API. Since the library is utilized in production by companies such as GitHub and Microsoft [libgit2], this lent the library additional credibility and made the choice fairly obvious for the authors. While the library is written in C, there are bindings for fifteen additional languages [libgit2], one of which being the Python programming language. While this made the development language decision easier, the authors’ did consider various languages before choosing Python. In the end, it was thought that the due to the authors’ previous experience with Python development and the ease of quickly prototyping in the language that Python would prove to be a safe choice. Another nice feature of Python is the comprehensive standard library that provides many additional convenient features with no additional configuration or external libraries required. Once initial setup of the library and language bindings were complete, development with pygit2, libgit2 and Python proved to be was relatively painless.

The final iteration of our data collection tool resulted in a straightforward command line application capable of cloning a repository from GitHub and totaling and categorizing the size metric data for each commit pair diff comparison in the entire history of the repository. The tool supports command line options for processing an individual repo with the following command:

srcstat --repo [repo-uri]

Additionally, and entire list of repos can be processed by providing a text file consisting of a list of repository URLs with the following command:

srcstat --repolist [repo-list-file]

Once the tool has collected a repository, the pygit2 library is used to retrieve the entire set of cryptographic SHA hash that represent each individual commit, and the hashes are returned in a list sorted from oldest to most recent. These hashes are run through a python sub process instance of the git command line tool to run a git diff --shortstat. While the application incurs an execution time running this external process it allows for a significant reduction in the amount of data objects that need to be traversed to acquire the number of lines deleted and added per commit. While this data accounts for two of the three required code size metrics, collection of the hunks metric is a more complicated matter. Collection of hunks requires the pygit2 diff function which accepts the same two SHA hashes as the git diff --shortstat command. The result of the function is a complex data structure that allows for iteration over changed files, hunks, and lines of code. Since lines are already captured, our tool only need to traverse the changed files to find all hunks within each file and collect the total number of hunks per commit diff calculation. Once all the totals for any given diff are calculated, the each metric for

change files, hunks and lines of code are then evaluated into extra small, small, medium, large, or extra-large categories based on their respective number of occurrences. When the entire repository is processed the end result is a file that contains a count the total number of modified files, hunks, and lines that fall into the aforementioned categories. It is this file and the values it contains that is used for the evaluation of our study. While this final revision of the tool was faster than the previous incarnations, it still struggled with the same memory consumption issues of its predecessor on the FreeBSD repository. With this in mind, the authors determined that a higher performance language such as C or C++ would most likely be necessary to provide the maximum efficiency necessary repositories with a combination of very large commits and a high overall commit history.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Quartiles** | **Number of Files** | | | **Number of New Lines** | | | **Number of Hunks** | | |
| **Q0 (Min)** |  | | |  | | |  | | |
| **Q1** |  | | |  | | |  | | |
| **Q­2 (Median)** | Did we do these? | | |  | | |  | | |
| **Q3** | We definitely didn’t, did we | | |  | | |  | | |
| **Q4 (Max)** |  | | |  | | |  | | |
| **Boxplot** | **Range** | **Freq.** | **Ratio** | **Range** | **Freq.** | **Ratio** | **Range** | **Freq.** | **Ratio** |
| **x-Small** | 1-1 | 16324 | 13% | 0-5 | 25339 | 20% | 0-1 | 17462 | 14% |
| **Small** | 2-4 | 65300 | 51% | 6-46 | 74267 | 58% | 2-8 | 79091 | 62% |
| **Median** | 5-7 | 18613 | 15% | 47-106 | 14532 | 11% | 9-17 | 14226 | 11% |
| **Large** | 8-10 | 7290 | 6% | 107-166 | 4809 | 4% | 18-26 | 5196 | 4% |
| **x-Large** | 11-19543 | 19543 | 15% | 167-8043 | 8043 | 6% | 27-11095 | 11095 | 9% |

Fig. 3‒gcc commits categorized by size for each of the three metrics

# Evaluation

## Setup

Using the tool, 13 open source software projects with varying sizes and domains were cloned from Git, as shown Figure 1. Several repositories were included from a previous study [typical commit], such as gcc, Ruby, and Python. Each repository’s commit history was traversed to obtain three size metrics:

1. Lines of code (LOC) added, deleted, or modified in a commit
2. Number of files added, deleted, or modified in a commit
3. Number of hunks added, deleted, or modified in a commit

Merge commits were also taken into account during the commit measures. Lines of code changes were calculated by taking the sum of line changes for each file changed. Hunks are sections of line changes that have been modified together within the fileEach metric was separated into a category of sizes ranging from extra-small to extra-large, as defined by [typical commit] in a previous study. The table representing how each range was categorized is shown in Figure 2.

| System | History Age | Total Commits |
| --- | --- | --- |
| Django Web Framework | 8 Years | 16476 |
| Express Web Framework | 4 Years | 4233 |
| GNU Compiler Collection (gcc) | 25 Years | 127070 |
| GNU Image Manipulation Program | 16 Years | 33588 |
| Apache Hadoop | 5 Years | 8469 |
| jQuery JavaScript Framework | 8 Years | 5450 |
| libgit2 | 5 Years | 5915 |
| Linux | 8 Years | 413028 |
| Mono | 13 Years | 94728 |
| Node Language | 5 Years | 9397 |
| Python Language | 23 Years | 41239 |
| Ruby Language | 15 Years | 33789 |
| XBMC Media Center | 4 Years | 25383 |

**Fig. 1‒ Systems by** the srcstat tool

| Metric | x-Small | Small | Medium | Large | x-Large |
| --- | --- | --- | --- | --- | --- |
| # Files Changed | 1-1 | 2-4 | 5-7 | 8-10 | 11-81880 |
| # New Lines | 0-5 | 6-46 | 47-106 | 107-166 | 167-54794 |
| # Hunks | 0-1 | 2-8 | 9-17 | 18-26 | 27-58817 |

**Fig. 2‒Range Categorizations for each metric**

## Quintessential Commit Sizes

Here we determine into which size range most commits are categorized. This will demonstrate the most frequent size of changes that have been committed. The ratio of commit sizes for the gcc project are shown in Figure 3, and the ratio of the average commit size for each open source project and category are shown in Figure 4. As seen in Figure 3, 64% of commits have been categorized as small or extra small based on the lines of code added, deleted, or modified. 78% have been categorized as small or extra small based on the number of files changed, and 76% of commits were classified as small or extra small based on the number of hunks modified.

The average commit size for each of the 13 open source project separated by size categories are shown in Figure 4. Aggregating all open source projects studied, 23% of the lines of code modified, 12% of files modified, and 23% of hunks modified were classified as large or extra-large. With the exception of the files modified, all other commit size metrics examined contained over 20% of larger commits. Based on these results, large commits are still a common occurrence, even given the modern software development practices when utilizing Git.

\*\*\* INSERT TABLE FOR ALL XS-XL % \*\*\*

## Correlation between Characteristics

In order to determine relationships between characteristics, the correlation coefficient is calculated between each metric. The correlation coefficient r, ranging [-1, 1], attempts to calculate if a linear co-relationship exists between variables. A negative value for r implies that as x increases, y decreases. A positive value for r implies that as x increases, y decreases. The closer the coefficient is to +1 or -1, the stronger the relationship. When r=0, no linear relationship exists between variables. [applied multivariate statistical analysis]. A strong correlation is determined with r values surpassing 0.8, while weak correlations are considered to have r values less than 0.5. For the purposes of this study, the conventional p-value of 0.05 was used to determine statistical significance. For all correlations that were calculated, results stand with 95% confidence.

# Threats to Validity

Certain threats to validity includes human error during development and data collection. In addition, the number of commits per project varied slightly than when the Alil paper [typical commit] was published due to new releases and bug fixes since publication. The additional data keeps the study up to date, but could skew the results to vary from those found by Alil et. al. Our tool is also missing various commits from thrown exceptions rooted in pygit2’s library calls. The exceptions are ignored and the commit is skipped over, though limited data is lost.

An additional threat to validity is merged branches. Some of the repositories examined in the original paper [typical commit], such as gcc, Ruby, and Python, were using SVN. However, these repositories were also found in GitHub, meaning that the project was moved over to Git at a later date. Thus these projects did not fully take advantage of the Git-specific utilities. For example, the projects developed using the SVN version control system commit changes use the SVN commit. Projects developed using Git follow the commit-push workflow, where a developer’s changes are committed to his local repository, then the aggregate of all commits are pushed to the central repository. In this way, Git encourages smaller changes which can then be pushed as a larger change, whereas SVN changes are possibly equivalent to the size of Git pushes. Therefore those projects which were originally developed using SVN and later merged to Git will resemble development patterns and practices that closer resemble SVN styles as opposed to Git styles, which is the focus of this paper.

# Future Work

Future work involves making an improved version of our tool that is quicker, more memory efficient, and

Given the results of the study, future work involves developing an automated commit splicing tool that will recognize a large commit based on various size metrics and splice it into one or more smaller, logically structured commits. Such a tool would ameliorate the software development process, especially given our findings.

# Conclusion

This is where we shall write our amazing conclusion

##### References

1. Cloc, http://cloc.sourceforge.net/. 12/9/13
2. Ohcount, https://github.com/blackducksw/ohcount. 12/9/13
3. libgit2, http://libgit2.github.com. 12/9/13
4. Python, http://www.python.org. 12/9/13
5. pygit2, http://www.pygit2.org. 12/9/13
6. GitHub, https://github.com. 12/9/13