Exploring Git Repository Commit Sizes

Drew Guarnera, Heather Michaud, Evan Purkhiser
Department of Computer Science
The University of Akron
{dtg3,hmm34,emp36}@zips.uakron.edu

Abstract*—Version control is an essential modern software engineering tool. 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, the understanding of which can improve the discipline of software development. Many version control systems exist, however this work specifically focuses on Git, a modern system that is rapidly gaining ground within the development community. Since Git follows a distributed version control model, any source code repository with a publicly accessible Git endpoint may be cloned by anyone utilizing the tool. Changes made to each repository can exist independently or be pushed or pulled between clones. Due to this flexibility, Git truly sets itself apart from older and more traditional version control tools by offering users an immensely customizable workflow.

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

***Keywords*—DVCS; Git; Source Control; Metrics; Mining Software Repositories**

I. Introduction

Throughout the course of the software life cycle, changes made to a project are recorded in a chosen version control system. When Git is utilized, modifications are made to the developer's local repository by committing the changes. These local modifications are placed in the central repository by pushing the changes. Through proper use of a version control system, the entire history of a project can be obtained via commit records.

Modern software practices encourage small commits to decrease the likelihood of introducing bugs. Git, in particular, motivates these practices with the commit-push workflow. Smaller changes can be committed and the larger aggregation of changes may then be pushed.

We believe that by studying repositories using a modern version control system, we can gain a better understanding of common software development practices in terms of commit sizes. In previous studies [Arafat09] [Alali08], size metrics were obtained from repository histories and it was found that less than 15% of commits were considered large. In [Alali08] specifically, the only projects that were studied were using SVN. Our ultimate goal is to determine if large commits are a frequent and problematic occurrence in repositories using Git, which encourages modern practices and stresses small commits in its workflow. To accomplish this goal, we examined three size metrics utilizing a custom tool developed for traversing repository commit histories:

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

All metrics were obtained using the libgit2[1] library with pygit2[2] Python bindings, which exposes the core functionality of Git. For the purposes of this study, the number of files that have been modified represent the coarsest level of granularity that can be achieved. The number of lines of code that have been added, deleted, or modified represent the finest level of granularity. The measure of hunks lie between these two levels. A hunk is a consecutive block of lines that have been modified together within a file. Unlike the previous work, we eliminated the context lines from each hunk to only compare lines that were modified. In this fashion, we measure commit sizes with varying levels of granularity.

The study presented here examines thirteen open source software projects. After collecting data for each project's entire revision history, the calculated metrics for lines, files, and hunks per commit are separated into categories of extra-small, small, medium, large, and extra-large. Overall trends and relationships between the extra-small to extra-large categories and the size metrics were studied by computing correlation coefficients.

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

[1] http://libgit2.github.com
[2] http://www.pygit2.org

II. Related Works

Arafat and Riehle [Arafat09] analyzed commits for over 9,000 open source projects and over 8 million commits using an analytics tool called Ohloh[3]. Each commit was separated into three categories including single focused commits, aggregate team contributions, and repository refactorings. Single individual developer commit contributions pertain to changes that ideally fix only one semantic issue, ranging from 1 to 100 SLOC in the commit size. Aggregate developer contributions are similar to Git push, and are comprised of a set of commits that either a single developer or a team of developers made. These commit sizes can range anywhere from 101 to 10,000 SLOC. Repository refactorings surpass 10,000 SLOC in commit size and typically deal with integrating an entire library or the branching of a project. Categories were formed based on the size of the commit measured in SLOC. The study concluded that single commits constitute 83% of the sample population. The authors suggested that the dominance of small commits is a result of modern open source software development processes, such as the dictator and lieutenants workflow. The authors proposed that large commits were caused by copy and paste, initial check-in of existing projects, or merging.

Hindle, German, and Holt [Hindle08] created a taxonomy of large commits which expanded Swanson's categories of changes [Swanson76], and manually classified the largest 1% of commits in nine open source projects. Traditionally, large commits are often explained by merges, copyright changes, and license updates. However, this study found that the most frequently occurring large commits involved not just merges, but also feature additions, documentation, adding a module, initializing a module, code cleanup or refactoring, token replacement, bug fixes, and changes in the build or configuration settings. A theme of large commits was also considered for each project.

Hindle, German, Godfrey, and Holt [Hindle09] manually classified large commits from nine open source projects into categories that were an extension of Swanson's categorization [Swanson76], then created a machine learning tool to automatically classify the changes based on commit messages.

Abdulkareem Alali et al [Alali08] focused on the version histories of nine open source software projects that use SVN. Commit sizes were measured in LOC, hunks, and number of files modified using GNU diff. This approach allowed for varying levels of granularity with respect to the amount of changes made to the system while also respecting the cohesive properties of the. Each size metric was separated into categories ranging from extra-small to extra-large. After all commits had been categorized, a correlation coefficient was calculated for lines and hunks, files and hunks, and files and lines. While no significant correlation was found between line and file or file and hunk measures, a substantial co-relationship was found between line and hunk measures. As opposed to SVN, our paper aims to replicate this study using Git. In addition, this study also examined the vocabulary of the commits compared to the sizes and established the most frequent words utilizing those two data points.

III. Background

Git is a distributed version control system (DVCS) offering speed and flexibility not found in other well-known version control systems, such as SVN or CVS. Originally developed in 2005 by Linus Torvalds for the management of Linux kernel source code, it has since taken the software development community by storm and gathered a significant following. With the advent of social coding platforms such as GitHub[4], Bitbucket[5], and Gitorious[6], Git has seen widespread growth and adoption by both large organizations and small independents. GitHub alone hosts over 6 million repositories.

A. *Distributed Version Control*

Git offers a distributed approach to source control. Most well-established version control systems follow a centralized model, where the source code exists in a central location and all changes to the source are made to that central repository. Instead of *checking out* a local copy of a remote repository's working tree, Git allows for a complete copy to be made, known as a *clone*. Each clone can be thought of as a completely independent to the source from which it was cloned but maintains all ancestral commits. When committing to a clone, only the changes are available to that particular clone. Git offers the ability to share these modifications by using the concept of *pushing* and *pulling* to exchange any changes.

Branching is a first class feature and a core concept of Git. It offers the ability to rapidly split off of the current tip of a cloned repository, creating a new branch that references the same commit. Commits to a newly created branch are only available on the new branch, while the original branch remains unchanged. By default, all repositories are initialized 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* of the repository from which it was cloned, known as the *origin*. It is also possible to have a remote branch of any cloned repository, as Git makes no distinction between remote branches. Changes between any two branches can be pushed and pulled between local branches, remote and local branches, or even between two remote branches.

B. *DVCS Workflows*

Due to the distributed nature of Git, the user has the ability to completely customize their workflow with respect to their usage of Git and for the entire project team. Because of their relevance in this study, the following Git workflows have been taken into consideration:

1) Incremental Commit

A common way for new users to work 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. If working in a team environment, there will often only be one central repository to which committers push their changes. This mirrors the approach of

[3] http://www.ohloh.net

[4] https://github.com

[5] https://bitbucket.org

[6] https://gitorious.org

SVN, in that branching is rare and commits are made and pushed to a central location.

2) Reorganizing Commit Workflow

Because commits are made to the local repository, it is possible to “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 additional time rearranging their commits after they have already been made, allowing for a cleanly organized history where each commit has been logically structured in a sequential context.

3) Branched-Merging Workflow

A common branching strategy is to create a *topic* branch for each new feature addition to the project. This is conducive to work on a particular piece of functionality to be completed independent of any work on the master branch, which 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 special type of commit has more than one parent commit reference, with all changes made since the most recent ancestral commit of both parents; however both historical timelines are preserved. Because merges usually have a substantial amount of changes, the size of the commit is typically larger.

4) Branched-Squash Workflow

Similar to the Branched-Merge workflow, it is also possible to squash multiple commits on a branch down to a single commit when merging it back into the mainline branch. This action is destructive and will discard the entire branch 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 large, long running feature branches.

The comprehension of these workflows is crucial to the analysis of the results obtained from the thirteen projects using Git which are the central focus of this study.

IV. SRCSTAT

A. Approach

In order to properly replicate the initial study [Alali08], multiple Git repositories would need to be cloned to a local machine for analysis. The commits of each repository would then need processing, and the projects utilized in this study could contain anywhere from ten thousand to several hundred thousand commits. In order to 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 commits. 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. While there was no shortage of tools for software metrics, Cloc[7] and Ohcount[8] initially stood out. However, 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 tooling proved unsuccessful, the Git command line client was then examined. This option was more promising due to the fact 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 because that particular information is not included in the abbreviated diff report. Using the standard diff option and parsing the full output of the diff report seemed inefficient and time consuming. Thus began the search for a cleaner and more comprehensive solution, leading to the libgit2 library (covered in the following section).

The initial tool was a basic prototype. Test repositories were hard coded into the application, and upon execution, the repositories were cloned from GitHub. Once a repository was downloaded, an array containing all of the commits, represented by cryptographic SHA hashes, was initialized and ordered from oldest to newest commit. These hashes were fed in pairs to a function in the library that performed the internal Git differencing between the two revisions of the repository. The resulting object from the diff operation then provided access to a data structure containing information on all the files, hunks, and lines modified between the two revisions. This first endeavor proved to be a solid base for the project, but the performance of the approach was inefficient. Upon closer analysis of the processed raw data, it became clearer as to why the first attempted failed.

Some of the diffs that were processed were incredibly large. 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 the available RAM in the two data collection systems, each equipped with eight gigabytes of RAM (Ubuntu 12.04 64-bit and Arch Linux 3.12.1 64-bit), ultimately causing forced process termination. After observing this limitation, adjustments were made in an effort to improve the efficiency and stability of the tool.

[7] http://cloc.sourceforge.net/

[8] https://github.com/blackducksw/ohcount

B. *Final Tool Design and Implementation*

After experimentation with the various tooling options, the authors arrived at a combination of libgit2 and Python for the final tool, with the core implementation 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 companies such as GitHub and Microsoft utilize libgit2 in production, this lent the library additional credibility and made the choice fairly obvious for the authors. While the library is written in C, bindings exist for fifteen additional languages, one of which is the Python programming language, making the development language decision easier. In the end, it was thought that due to the authors' previous experience with Python development, and the ease of quickly prototyping using the language, Python would be an appropriate choice. Another beneficial feature of Python is that the comprehensive standard library provides many additional convenient features with no superfluous configuration or external libraries required. Once initial setup of the library and language bindings were complete, development with pygit2, libgit2 and Python 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, as well as 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, an 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 SHA1 hashes that represent each individual commit, and sorted from oldest to most recent. These hashes are run through a python sub process instance of the Git command line tool in order to run a `git diff --shortstat`. Even though the application incurs an execution time running this external process, this sacrifice allows for a significant reduction in the amount of data objects that need to be traversed in order 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.

The collection of hunks requires pygit2's internal diff function, which accepts the same two SHA1 commit identifiers 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 needs 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 are calculated, each metric for changed 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 text file that contains a count of the total number of modified files, hunks, and lines that fall into the aforementioned categories. These files and the values they contain are used for the evaluation of this study. The final revision of the tool was faster than the previous incarnations, though 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 required to process repositories with a combination of very large commits and a high overall commit history.

V. Evaluation

A. *Setup*

Utilizing the srcstat tool, data was collected from thirteen open source software projects that were cloned from GitHub, each with varying sizes and domains. Several repositories were included from a previous study [Alali08], such as gcc, Ruby, and Python. Each repository's commit history was traversed to obtain three size metrics:

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

Merges were also taken into account during the commit measurements. Lines of code changes were calculated by taking the sum of line changes for each hunk modifed. Hunks are contiguous sections of line changes that have been modified together within the file. For example, when lines 1-10 have been modified and lines 23-30 have been modified within the same file, two hunks exist of size 10 and 8 respectively. Each metric was separated into a category of sizes ranging from extra-small to extra-large, as defined by [Alali08] in a previous study. The representation of how each range was categorized is shown in Table 1.

Metric	x-Small	Small	Medium	Large	x-Large
No. Files changed	1-1	2-4	5-7	8-10	11-81880
No. New lines	0-5	6-46	47-106	107-166	167-54794
No. Hunks	0-1	2-8	9-17	18-26	27-58817

Table 1. Range Categorizations for each commit size metric.

B. *Examined Projects*

The thirteen open source projects that were collected from GitHub are listed in Table 2, along with their corresponding project duration in years, total number of commits throughout the project, total lines of code per repository, and the total number of files in the project.

System	Duration	Total Commits	Total LOC	Total Files
Django Web Framework	8 Years	16476	1021488	4939
Express Web Framework	4 Years	4233	52060	347
GNU Compiler Collection (gcc)	25 Years	127070	21523093	82630
GNU Image Manipulation Program (GIMP)	16 Years	33588	4873393	5650
Apache Hadoop	5 Years	8469	3025318	6219
jQuery JavaScript Library	8 Years	5450	116980	235
Libgit2	5 Years	5915	247723	2504
Linux	8 Years	413028	26422160	44984
Mono	13 Years	94728	10237555	45707
Node.js	5 Years	9397	2985235	9747
Python Language	23 Years	41239	1880911	3971
Ruby Language	15 Years	33789	2338318	4514
XBMC Media Center	4 Years	25383	6559947	15701

Table 2. Thirteen open source projects analyzed by the srcstat tool.

Each repository is briefly described here in terms of its purpose and usage within the development community.

1) Django Web Framework

Django, written in Python, is a web applications framework for rapid development of database driven web applications using the Model View Controller (MVC) paradigm. Originally released in 2005, Django has seen a steady rise in usage since.

2) Express Web Framework

Express is a relatively new web applications framework that utilizes the Node.js scripting language. Similar to Django, Express provides a way to rapidly create complex database driven MVC applications. Express is designed to be especially minimal in its approach and primarily handles HTTP logic. Development began just months before the initial Node.js release in January 2009.

3) GNU Compiler Collection (gcc)

The GNU Compiler Collection is a compiler suite produced for the GNU project that offers support for various programming languages. It is widely popular, available on almost all Linux distributions, and is the official compiler for the GNU operating system. Originally developed in 1987, gcc has played a large part in the growth of open source software.

4) GNU Image Manipulation Program (GIMP)

The GNU Image Manipulation Program is a cross platform photo and image manipulation tool that offers high end features commonly only available in commercial photo editing software. Development started in 1996 and like gcc this is also part of the GNU project.

5) Apache Hadoop

Hadoop is an open source framework developed by the Apache foundation in 2005 used for the processing and storage of data sets on a very large scale. Hadoop is split up into multiple modules, for the purpose of this project we will be focusing on the "Hadoop Common" module.

6) jQuery JavaScript Library

JQuery is a client side JavaScript library. It is primarily concerned with providing a simple API interface to the Document Object Model, but it also has a strong focus on normalizing browser quirks. JQuery originally began development in 2006 and has since become the most popular JavaScript library used today.

7) Libgit2

Libgit2 is a library which implements and exposes Git functionality through a C API. Libgit2 was originally developed in 2008 and is now used by the likes of GitHub, Beanstalk, and Microsoft. Language bindings are also offered for various other development languages.

8) Linux

Originally developed by Linus Torvalds in 1995 as a spinoff of the Minx kernel, the Linux Kernel provides the foundational backbone for many open source and commercial operating system distributions. Its proven effectiveness and stability has earned Linux its place as one the most popular operating systems in the world.

9) Mono

Mono is a compiler that is compatible with C# and the .NET framework. It also includes the common language runtime required to execute programs compiled with Mono. It originally began development in 2004 by Novell, but has since been taken over by Xamarin.

10) Node.js

Node.js is a relatively new software platform used for creating highly scalable network applications. The platform utilizes the JavaScript scripting language and uses an event based non-blocking event loop. Development only recently begun in 2009, but it has since gained a lot of traction due to the need for scalable real-time web applications.

11) Python Language

Python is a powerful dynamic programming language with a syntax designed to be clear and readable. An extensive library, including a wide array of “out of the box” provided functionality, backs it. It has proven to be quite applicable for rapid application development and prototyping.

12) Ruby Language

Publicly released in 1995, Ruby is functional programming language that strives to provide a clean object oriented language that is easy to comprehend and powerful in practical application. When combined with the Rails web framework, Ruby can be used to code web services and applications.

13) XBMC Media Center

Created in 2003, XBMC is a cross platform entertainment management and presentation application. Designed to provide simple and attractive playback for both audio and video multimedia, XBMC is frequently used as a HTPC (Home Theatre Personal Computer) interface. XBMC has received the support of over 450 software developers worldwide, and is available in more than 65 languages.

	Number of Files		Number of New Lines		Number of Hunks	
Categories	**Frequency**	**Ratio**	**Frequency**	**Ratio**	**Frequency**	**Ratio**
x-Small	348790	43%	174793	21%	195791	31%
Small	308401	38%	358576	44%	421850	37%
Median	56167	7%	100408	12%	80899	9%
Large	20049	2%	39095	5%	26593	3%
x-Large	81974	10%	145893	18%	93632	20%

Table 3. Aggregate commits from all 13 projects examined, categorized by size for each of the three commit size metrics.

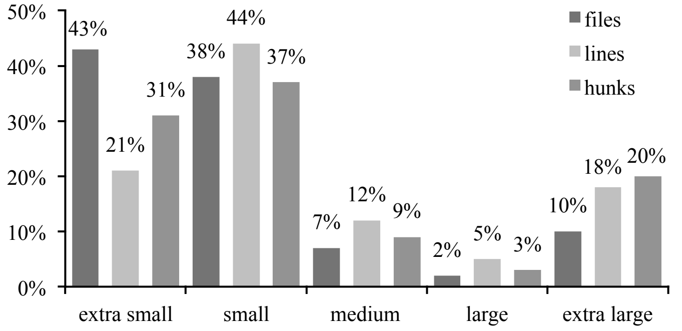

Figure 1. Histogram of commits from all 13 projects by size

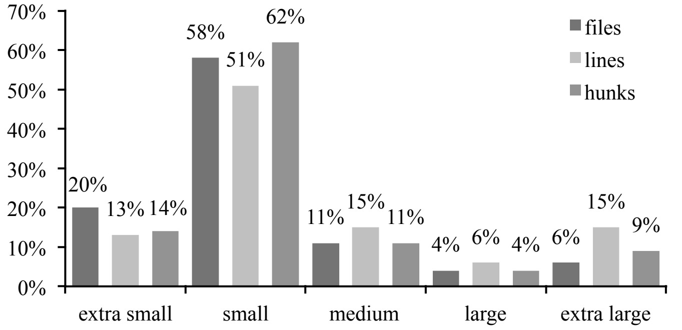

Figure 2. Histogram of commits from the *gcc* project by size

C. 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 average size of commits for each of the thirteen open source project separated by categories is shown in Table 3 and Figure 1. 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, given even when using modern software development practices and Git.

The ratios of commit sizes for the gcc project are shown in Table 4 and represented as a histogram in Figure 2. As seen in Table 4, 64% of commits have been categorized as small or extra-small based on the lines of code added, deleted, or modified, while 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.

D. 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. Inversely, a positive value for r implies that as x increases, y increases. The closer the coefficient is to +1 or -1, the stronger the relationship. When $r = 0$, no linear relationship exists between variables [Johnson98]. A strong correlation is determined with r -values surpassing 0.8, while weak correlations are considered to have r-values less than 0.5. The equation used to calculate the correlation coefficient is as follows:

$$r = \frac{n\Sigma xy - (\Sigma x)(\Sigma y)}{\sqrt{n(\Sigma x^2) - (\Sigma x^2)}\sqrt{n(\Sigma y^2) - (\Sigma y^2)}}$$

The coefficient of determination, r^2, is a statistical calculation that indicates how well the relationship between variables fits into a linear regression model, ranging [0, 1]. It measures the strength of the correlation between variables. For example, if $r = 0.821$, then $r = 0.674$, meaning that 67.4% of the total variation between x and y can be explained by the linear relationship defined by r. Thus the remaining 32.6% of the variation remains unexplained by a linear model.

	Number of Files		Number of New Lines		Number of Hunks	
Categories	**Frequency**	**Ratio**	**Frequency**	**Ratio**	**Frequency**	**Ratio**
x-Small	16324	13%	25339	20%	17462	14%
Small	65300	51%	74267	58%	79091	62%
Median	18613	15%	14532	11%	14226	11%
Large	7290	6%	4809	4%	5196	4%
x-Large	19543	15%	8043	6%	11095	9%

Table 4. *gcc* commits categorized by size for each of the three commit size metrics. The duration of the commits was over 25 years.

		files × lines	files × hunks	hunks × lines
Django	***r***	0.821	0.891	0.986
	r^2	0.674	0.794	0.973
Express	***r***	0.885	0.947	0.985
	r^2	0.782	0.898	0.969
gcc	***r***	0.963	0.988	0.989
	r^2	0.928	0.977	0.979
GIMP	***r***	0.694	0.907	0.903
	r^2	0.481	0.823	0.816
Hadoop	***r***	0.806	0.994	0.809
	r^2	0.650	0.989	0.655
jQuery	***r***	0.853	0.876	0.995
	r^2	0.728	0.767	0.991
Libgit2	***r***	0.893	0.873	0.941
	r^2	0.798	0.762	0.886
Linux	***r***	0.580	0.606	0.982
	r^2	0.336	0.367	0.965
Mono	***r***	0.966	0.991	0.986
	r^2	0.934	0.982	0.971
Node	***r***	0.721	0.836	0.949
	r^2	0.520	0.699	0.900
Python	***r***	0.648	0.762	0.983
	r^2	0.421	0.580	0.965
Ruby	***r***	0.976	0.989	0.997
	r^2	0.953	0.978	0.994
XBMC	***r***	0.840	0.848	0.988
	r^2	0.706	0.720	0.976

Table 5. Correlation coefficient, *r*, and coefficient of determination, r^2, as calculated for all projects.

The correlation coefficient and coefficient of determination for gcc are shown in Table 5. It can be seen that all correlation coefficients for gcc are above 0.96. This means that all relationships between files and lines, files and hunks, and hunks and lines are strongly linearly correlated. Over 92% of the variants can be explained by this linear correlation.

All thirteen projects showed a strong correlation between hunks and lines size metrics, which is also shown in Table 5. Only Hadoop contained a correlation coefficient under 0.9, holding a correlation of 0.809. All other repositories had *r*-values exceeding 0.9, indicating a very strong positive linear correlation between hunks and lines modified in a commit.

Most projects had a strong correlation between files and hunks. Only Linux and Python had moderate correlations, containing 0.606 and 0.762 respectively. The inconsistency with Linux may be due to the nature of the project. Linux is a long running and stable project, which is based deeply on providing consistent and reliable functionality. With this in mind, changes would most likely be limited to certain sections within a file. Linux also has very rigid development restrictions that the maintainers placed on contributors to keep commits small and focused. Additionally, the moderate correlation can be described with respect to commit distributions. In terms of the number of hunks modified per commit, 27% and 48% of commits were classified as extra small or small, respectively, as shown in Table 6. However, the number of commits classified as such when measured by the number of files changed is nearly reversed.

Multiple aspects can explain the minor inconsistency with Python's moderate correlation between files and hunks. One

	Number of Files		Number of New Lines		Number of Hunks	
Categories	**Frequency**	**Ratio**	**Frequency**	**Ratio**	**Frequency**	**Ratio**
x-Small	224237	54%	101299	25%	112204	27%
Small	102942	25%	163916	40%	190574	46%
Median	22553	5%	47802	12%	39474	10%
Large	7788	2%	18131	4%	11959	3%
x-Large	54794	13%	81880	20%	58817	14%

Table 6. *Linux* commits categorized by size for each of the three commit size metrics.

	Number of Files		Number of New Lines		Number of Hunks	
Categories	**Frequency**	**Ratio**	**Frequency**	**Ratio**	**Frequency**	**Ratio**
x-Small	29291	71%	13126	32%	16930	41%
Small	9304	23%	17916	43%	19314	47%
Median	1264	3%	4249	10%	2765	7%
Large	404	1%	1756	4%	845	2%
x-Large	895	2%	4192	10%	1385	3%

Table 7. *Python* commits categorized by size for each of the three commit size metrics.

	Number of Files		Number of New Lines		Number of Hunks	
Categories	**Frequency**	**Ratio**	**Frequency**	**Ratio**	**Frequency**	**Ratio**
x-Small	7702	23%	3967	12%	4327	13%
Small	16754	50%	12523	37%	15184	45%
Median	3297	10%	4487	13%	4929	15%
Large	1704	5%	2279	7%	2172	6%
x-Large	3772	11%	10332	31%	6976	21%

Table 8. *GIMP* commits categorized by size for each of the three commit size metrics.

	Number of Files		Number of New Lines		Number of Hunks	
Categories	**Frequency**	**Ratio**	**Frequency**	**Ratio**	**Frequency**	**Ratio**
x-Small	4589	49%	2285	24%	3015	32%
Small	3031	33%	3476	37%	4391	47%
Median	470	5%	1467	16%	735	8%
Large	203	2%	445	5%	243	3%
x-Large	1025	11%	1724	18%	1013	11%

Table 9. *Node* commits categorized by size for each of the three commit size metrics.

such explanation is that Python has fewer than 4,000 files and, according to our previous measurements, only 3% of all commits made to the project were considered to be large or extra-large based on the number of files modified. In addition, only 5% of all commits made to the project were categorized as large or extra-large based on the number of hunks modified in a commit, as shown in Table 7. An alternative viewpoint is that many of the modules in Python's standard library have a long-standing codebase which may require minimal architectural changes. This leads to the assumption that only small maintenance related changes are the focus of a majority of the commits. Another possible explanation of the results could be associated with a workflow that consists of development policies to distill large changes or maintenance work into a batch of smaller commits. This would directly correlate with Git's ability to easily reorganize commits as mentioned previously in Section 3.

Django, Express, gcc, Hadoop, jQuery, Libgit2, and Mono had a strong correlation between files and lines. GIMP, Linux, Node, and Python had r-values of 0.684, 0.58, 0.721, and 0.684 respectively, indicating moderate correlations between files and lines. GIMP's commit distributions, shown in Table 8, could have skewed the results of the correlation calculations because only 16% of the number of files modified in a commit were categorized as large or extra-large, where as 38% of the number of lines of code modified in a commit were categorized as such. The focus of the GIMP application is centered on user interaction and the ability to edit images and graphical projects. With user satisfaction and maintaining a rich feature set as primary objectives, it stands to reason that additional features and architectural modifications would be more frequent in order to maintain relevance among the application's user base. These changes translate to multiple line modifications mostly centralized to, or at least built on top of, a given module file containing the functionality.

In addition, the size distribution of Node's commits varied greatly. Shown in Table 9, fewer than 25% of the commits were considered extra-small based on LOC modifications, where as nearly 50% of commits were categorized as extra-small based on the number of file modifications. Oddities in the correlation between files and lines of Node could also be associated with the fact that Node has yet to reach its 1.0 milestone release, and is still under heavy development. This could make pattern determination difficult due to the potential volatility of the project.

Similarly as previously speculated, the inconsistencies identified with Python's correlation between file and line sizes can be explained by the commit size distributions in addition to workflow patterns. The number of commits to Python that were classified as extra-small and small were 71% and 23% respectively when measuring based on files modified. However, only 32% and 43% of commits were classified as such as when measuring based on lines modified.

In Linux, 54% and 25% of the commits are classified as extra-small or small, respectively, based on the number of files modified, where as almost inversely, 25% and 40% of the commits were categorized as extra-small or small based on the number of lines modified in a commit. These differences in categorizing commit sizes based on the size metric would skew the correlation coefficient.

All projects have a moderate to very strong positive correlation between all characteristics examined during the study. The number of lines of code changed and the number of hunks modified had the strongest correlation across all of the projects. The number of files and hunks modified had the next to strongest correlation, while the number of files and lines of code changed per commit had the weakest correlation, though still exhibited a moderate to strong correlation between the characteristics.

VI. Threats to Validity

Certain threats to validity include human error during development and data collection. In addition, the number of commits per project varied slightly when compared to the Alali paper [Alali08] due to new releases and bug fixes since publication. The additional data keeps the study up to date, but could cause the results to vary from those found by Alali 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 only a small amount of data is lost.

An additional threat to validity is merged branches. A limited number of the repositories examined in the original paper [Alali08], including gcc, Ruby, and Python, were originally developed using SVN. Specifically, *gcc* still uses SVN and the repository located on GitHub is only a mirror to the changes presently being made to SVN. However, the remaining 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 features. 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. It is speculated that 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 have commit histories that closer resemble an SVN style workflow.

In addition, the inter quartile ranges that were used to calculate the size ranges from extra small to extra large were calculated by Alali et al [Alali08] on *gcc* at the time of the paper. This study used those size ranges, though the study was performed on multiple repositories whose data points varied, and even the *gcc* repository is over twice as large as the original repository used in the previous study. Remediation of these issues remains as future work.

VII. Future Work

Future work involves making an improved version of our tool that is faster, more memory efficient, and capable of performing similar and additional commit size metric calculations on a large number of repositories.

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

The previous study [Alali08] used inter quartile ranges to calculate how the commit size data should be distributed. Calculating seven regions by outliers determined the size ranges. When shown in a box and whisper plot, these ranges constitute the sizes ranging from extra-small to extra-large. In future work, we intend to calculate the inter quartile ranges for each of the repositories studied.

VIII. Conclusion

The purpose of this study was to determine if large commits are a prevalent issue in modern software development practices. We believe that by studying repositories using modern version control system such as Git, our understanding of common software development practices in terms of commit sizes measured in terms of files, lines, and hunks can be improved. A previous study conducted in 2008 [Alali08] studied nine open source projects using SVN to determine the frequency of varying commit sizes and attempted to find a correlation between various forms of size metrics. In this work, Alali et al concluded that less than fifteen percent of commits were considered to be large. Our paper replicates that study, but differentiates itself from the original work with a larger dataset and focusing on projects that utilize Git as the primary version control system as opposed to SVN.

The choice of version control system is a crucial element of the software development process. Using SVN allows developers to commit their changes directly to the central repository, while the utilization of Git supports a commit-push workflow. This involves a developer making several commits local to their repository clone and then pushing those commits to a central repository. This paper examines the commit sizes of thirteen open source projects located in GitHub. Three size metrics were gathered including number of files, lines of code, and hunks added, deleted, or modified in a commit. These three size metrics were broken into five categories of extra-small, small, medium, large, and extra-large. The correlation coefficient between the different metrics and the coefficient of determination was calculated to determine if there was a co-relation between metrics, and how strongly a linear model could describe the relationship.

All size metrics were found to be fundamentally correlated with each other. Hunks and lines of code modified had the strongest correlation of the three measurements, as was observed in every one of the thirteen repositories studied. Files and hunks followed closely, with all but two repositories having strongly correlated values, and the remaining projects having moderately correlated values. Finally, files and lines did moderately well, with all but four projects having strongly correlated values. Ultimately, it was found that 12% to 23% of commits were classified as large or extra-large, depending on the size metric that was used. This leads us to conclude that large commits are still a prevalent and frequent element of modern software development, despite the fact that the Git workflow encourages small commits.

References

[Arafat09] Arafat, O. and D. Riehle. *The Commit Size Distribution of Open Source Software*. in *System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on*. 2009.

[Alali08] Alali, A., H. Kagdi, and J.I. Maletic. *What's a Typical Commit? A Characterization of Open Source Software Repositories*. in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. 2008.

[Hindle08] Hindle, A., D.M. German, and R. Holt, *What do large commits tell us?: a taxonomical study of large commits*, in *Proceedings of the 2008 international working conference on Mining software repositories*. 2008, ACM: Leipzig, Germany. p. 99-108.

[Swanson76] Swanson, E.B., *The dimensions of maintenance*, in *Proceedings of the 2nd international conference on Software engineering*. 1976, IEEE Computer Society Press: San Francisco, California, USA. p. 492-497.

[Hindle09] Hindle, A., et al. *Automatic classication of large changes into maintenance categories*. in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*. 2009.

[Johnson98] Johnson, R.A.a.W., D. W., *Applied Multivariate Statistical Analysis*. 1998, Prentice Hall.

APENDIX

For completeness, the following tables represent the remainder of this study's analyzed data set for commits categorized by size.

	No. of Files		No. of New Lines		No. of Hunks	
Categories	**Freq.**	**Ratio**	**Freq.**	**Ratio**	**Freq.**	**Ratio**
x-Small	8427	52%	4885	30%	5956	36%
Small	5517	34%	7056	43%	7550	46%
Median	992	6%	1821	11%	1383	8%
Large	368	2%	666	4%	427	3%
x-Large	990	6%	2048	12%	1160	7%

Table 10. *Django* commits categorized by size for each of the three commit size metrics.

	No. of Files		No. of New Lines		No. of Hunks	
Categories	**Freq.**	**Ratio**	**Freq.**	**Ratio**	**Freq.**	**Ratio**
x-Small	19458	21%	12781	13%	13808	15%
Small	59577	63%	49515	52%	62259	66%
Median	7886	8%	13427	14%	10176	11%
Large	2768	3%	5165	5%	3254	3%
x-Large	4771	5%	13840	15%	5231	6%

Table 11. *Mono* commits categorized by size for each of the three commit size metrics.

	No. of Files		No. of New Lines		No. of Hunks	
Categories	**Freq.**	**Ratio**	**Freq.**	**Ratio**	**Freq.**	**Ratio**
x-Small	2093	52%	1411	33%	1795	42%
Small	1540	39%	2038	48%	2140	51%
Median	197	5%	448	11%	195	5%
Large	77	2%	106	3%	41	1%
x-Large	90	2%	230	5%	62	1%

Table 12. *Express* commits categorized by size for each of the three commit size metrics.

	No. of Files		No. of New Lines		No. of Hunks	
Categories	**Freq.**	**Ratio**	**Freq.**	**Ratio**	**Freq.**	**Ratio**
x-Small	12044	48%	7844	31%	8461	33%
Small	6809	27%	9355	37%	10127	40%
Median	1349	5%	2224	9%	2062	8%
Large	549	2%	833	3%	711	3%
x-Large	4162	17%	5127	20%	4022	16%

Table 13. *XBMC* commits categorized by size for each of the three commit size metrics.

	No. of Files		No. of New Lines		No. of Hunks	
Categories	**Freq.**	**Ratio**	**Freq.**	**Ratio**	**Freq.**	**Ratio**
x-Small	675	8%	912	11%	468	6%
Small	4644	55%	2975	35%	4527	53%
Median	1150	14%	1242	15%	1417	17%
Large	513	6%	669	8%	580	7%
x-Large	1429	17%	2671	32%	1477	17%

Table 14. *Hadoop* commits categorized by size for each of the three commit size metrics.

	No. of Files		No. of New Lines		No. of Hunks	
Categories	**Freq.**	**Ratio**	**Freq.**	**Ratio**	**Freq.**	**Ratio**
x-Small	9633	29%	6470	19%	7465	22%
Small	20606	62%	20272	60%	21661	64%
Median	1804	5%	3394	10%	2587	8%
Large	560	2%	1199	4%	806	2%
x-Large	854	3%	2454	7%	1270	4%

Table 15. *Ruby* commits categorized by size for each of the three commit size metrics.

	No. of Files		No. of New Lines		No. of Hunks	
Categories	**Freq.**	**Ratio**	**Freq.**	**Ratio**	**Freq.**	**Ratio**
x-Small	3167	58%	1887	35%	2160	40%
Small	1756	32%	2315	42%	2576	47%
Median	161	3%	518	10%	317	6%
Large	81	1%	184	3%	113	2%
x-Large	249	5%	546	10%	284	5%

Table 15. *jQuery* commits categorized by size for each of the three commit size metrics.

	No. of Files		No. of New Lines		No. of Hunks	
Categories	**Freq.**	**Ratio**	**Freq.**	**Ratio**	**Freq.**	**Ratio**
x-Small	2135	39%	1602	27%	1740	29%
Small	1654	30%	1919	32%	2456	42%
Median	512	9%	716	12%	633	11%
Large	225	4%	372	6%	246	4%
x-Large	900	17%	1306	22%	840	14%

Table 16. *libgit2* commits categorized by size for each of the three commit size metrics.