Merge-Trees: Visualizing the Integration of Commits into Linux

by

Evan Wilde
B.Sc., University of Victoria, 2016

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

© Evan Wilde, 2018
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Merge-Trees: Visualizing the Integration of Commits into Linux

by

Evan Wilde
B.Sc., University of Victoria, 2016

Supervisory Committee

Dr. Daniel M. German, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Daniel M. German, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Departmental Member
(Department of Computer Science)

ABSTRACT

Version control systems are an asset to software development, enabling developers to keep snapshots of the code as they work. Stored in the version control system is the entire history of the software project, rich in information about who is contributing to the project, when contributions are made, and to what part of the project they are being made. Presented in the right way, this information can be made invaluable in helping software developers continue the development of the project, and maintainers to understand how the changes to the current version can be applied to older versions of projects.

Maintainers are unable to effectively use the information stored within a software repository to assist with the maintanance older versions of that software in highly-collaborative projects. The Linux kernel repository is an example of such a project. This thesis focuses on improving visualizations of the Linux kernel repository, developing new visualizations that help answer questions about how commits are integrated into the project. Older versions of the kernel are used in a variety of systems where it is impractical to update to the current version of the kernel. Some of these applications include the controllers for spacecrafts, the core of mobile phones, the operating system driving internet routers, and as Internet-Of-Things (IOT) device firmware. As vulnerabilities are discovered in the kernel, they are patched in the current version. To ensure that older versions are also protected against the vulnerabilities, the patches applied to the current version of the kernel must be applied back to the older version. To do this, maintainers must be able to understand how the patch that fixed the vulnerability was

integrated into the kernel so that they may apply it to the old version as well.
This thesis makes four contributions: (1) a new tree-based model, the Merge-Tree, that abstracts the commits in the repository, (2) three visualizations that use this model, (3) a tool called *L*invis that uses these visualizations, (4) a user study that evaluates whether the tool is effective in helping users answer questions related to how commits are integrated about the Linux repository.
The first contribution includes the new tree-based model, the algorithm that constructs the trees from the repository, and the evaluation of the results of the algorithm. the second contribution demonstrates some of the potential visualizations of the repository that are made possible by the model, and how these visualizations can be used depending on the structure of the tree. The third contribution is an application that applies the visualizations to the Linux kernel repository.
The tool was able to help the participants of the study with understanding how commits were integrated into the Linux kernel repository. Additionally, the participants were able to summarize information about merges, including who made the most contributions, which file were altered the most, more quickly and accurately than with Gitk and the command line tools.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **v**

List of Tables **vii**

List of Figures **viii**

Acknowledgements **xii**

1 Introduction **1**

1.1 Thesis Organization 3

2 Background and Related Work **4**

2.1 Related Work 8

2.2 Git 19

2.3 Directed Acyclic Graph 21

2.4 Linux 27

3 Merge-Tree Model **33**

3.1 Algorithm 37

3.2 Algorithm Evaluation 40

4 Design and Implementation **45**

4.1 Search 46

4.2 Summarization 46

4.3 Visualization 48

4.3.1 List Tree 50

4.3.2 Reingold-Tilford Tree 50
4.3.3 Pack Tree 51

5 Implementation Details 54
5.1 Composition 54
5.2 Database 56
5.2.1 Full Text Search 57
5.3 Web Interface 59

6 Evaluation 60
6.1 Methodology 60
6.1.1 Commit Selection 63
6.1.2 Part 1: Conceptual Study 65
6.1.3 Part 2: Summarization Study 67
6.1.4 User Opinions and Exit Interview 70
6.2 Participant Profile 70
6.3 Results 71
6.3.1 Conceptual Study Results 71
6.3.2 Summarization Study Results 75
6.3.3 User Opinions 84

7 Discussion 86
7.1 Interpreting the Results 86
7.2 Study Observations 87
7.3 Comments From a Release Manager 88
7.4 Threats to Validity 89
7.4.1 Internal Validity 89
7.4.2 External validity 90
7.5 Limitations 91
7.6 Future Work 92

8 Conclusion 94

Bibliography 96

List of Tables

Table 5.1 Search Attribute Ranking 58

Table 6.1 Conceptual Tasks 67
Table 6.2 Summarization Tasks 68
Table 6.3 User Opinion Questions 70
Table 6.4 Variance and Difference between correct answers and user responses in conceptual tasks in tasks T2 and T3 74
Table 6.5 Timing Results from the conceptual tasks T2 and T3 75
Table 6.6 Effect of merge size on correctness 76
Table 6.7 Aggregated Correctness Results comparing *L*invis and Gitk . . . 78
Table 6.8 Effect of merge size on accuracy 78
Table 6.9 Aggregated Accuracy results, including the Wilcoxon p-values and Cliff's Delta Effect size 80
Table 6.10 Effect of merge size on response time 81
Table 6.11 Aggregated time results, including the Wilcoxon p-values and Cliff's Delta Effect size 84

List of Figures

Figure 2.1 A depiction of the distinction between fast-forward merges and non fast-forward merges 7
Figure 2.2 View of Hipikat, listing bugs that are similar to the one being viewed 10
Figure 2.3 A screenshot of the search results on Hoozizat, an implementation of codebook.[2] 10
Figure 2.4 Construction of Fractal Figures[9] 11
Figure 2.5 Evolution Radar visualization[10] 12
Figure 2.6 A screenshot of the communication mapping tool by Heller et al.[17] 12
Figure 2.7 View of Gource file graph with users operating on a repository[5] 13
Figure 2.8 View of the Postgresql repository in Codeswarm[21] 14
Figure 2.9 Gitk interface, the graphical interface shipped with Git. 14
Figure 2.10 Screenshot of the main view in GitKraken[1] 15
Figure 2.11 Gitg interface from the Gnome project[16] 16
Figure 2.12 Screenshot of Giteye DAG view of a repository[23] 17
Figure 2.13 GitHub online network view of a repository[14] 17
Figure 2.14 GitLab online graph view[15] 18
Figure 2.15 Screenshot of commit metadata for a commit and merge in the repository of this thesis. 19
Figure 2.16 Example of a commit patch from the Linux kernel repository from commit *c03567a8e8d5*. 20
Figure 2.17 Example of a merge that only integrates a single commit 21
Figure 2.18 A small example of a sequence of commits and merges. The branch pointer A references commit 7, which merges the head of branch B, commit 6, into the original head of branch A, which was commit 5. Merge 7 is the most recent change to the repository. 23

Figure 2.19 The sequence of steps that are part of the foxtrot, from the point of view of each repository. Alice's commit (2) is pushed to the master branch but, as a result of the push by Bob, the master branch has been swapped with Bob's branch. This type of merge (4) is called a foxtrot. 25
Figure 2.20 The git graph visualization of two sections of the Linux repository. 26
Figure 2.21 Unique authors with contributions to each kernel version 29
Figure 2.22 Commits per release from Linux 3.1 to Linux 3.16 30
Figure 2.23 Merges per release from Linux 3.1 to Linux 3.16 30
Figure 2.24 Commits per merge into each release of Linux from 3.1 to 3.16 . 30
Figure 2.25 Distribution of Merge Sizes per Release Between Linux 3.1 and 3.16 31

Figure 3.1 An example sequence of events performed in different repositories. The horizontal axis represents time. The branches and repositories are aligned horizontally, and color-coded. Each commit points to its parent. The initial commit is at time t_0, and the head is at t_8. 34
Figure 3.2 DAG representation of the commits represented in Figure 3.1. The DAG loses information about which repository the commit is performed in and through which merges it has passed on its way to the master branch. The DAG does not even distinguish the master branch from other branches. 35
Figure 3.3 The Merge-Trees computed for each commit in Figure 3.2 showing the path that each commit takes to be merged into the master branch of the repository. This does not indicate how the events being merged are related. This figure retains the numerical order of the events, but the order is arbitrary. 35
Figure 3.4 Example of how merges record a subset of commits being merged. The commit only shows the first 20 one-line summaries messages for the 24 non-merge commits it merged. The ending "..." is part of the log and represents that other commits were merged. 41
Figure 3.5 Merge *186051d70444* graph view, showing that the merge into master is immediately followed(above) by a non-merging commit. 42

Figure 3.6 The largest Merge-Tree, made by Linus Torvalds into Linux 3.16. This visualization clusters nodes based on the parents. The visualization is explained in more detail in Section 4.3.3. 44

Figure 4.1 Two Merge-Trees returned from the query for “net-next”. The top search result contains multiple entries with the search term in the title, whilst the second result contains a single entry with the search term in the title. The groups provide a link to the root at the top, and the relevant commits in the table below. . 47
Figure 4.2 Table showing the modified files in a merge, with the second entry expanded to show the commit that makes the changes. . . 47
Figure 4.3 Table showing the modules involved in a merge, listing the commits that modify this module. 48
Figure 4.4 Table showing the authors who made changes in a merge. The entry for Randy Dunlap is expanded, showing the files that Randy modified in this merge. 49
Figure 4.5 The List Tree Visualization 49
Figure 4.6 The Reingold-Tilford tree visualization with the root currently selected. The root is at the top, the leaves are depicted as the white circles with no children. 50
Figure 4.7 The pack tree visualization; the root depicted as the outer-most circle, containing all other nodes, the leaves depicted as white circles containing no other nodes. The currently selected node shown in pumpkin orange. 52

Figure 5.1 Webserver Architecture, showing the protocol used for communication between the modules. 55

Figure 6.1 Two examples of DAG to Merge-Tree conversions used in explanation during evaluation 64
Figure 6.2 The visualizations of commit 1 by Gitk and *L*invis respectively. 66
Figure 6.3 The visualizations of commit 2 by Gitk and *L*invis respectively. 66
Figure 6.4 An example of a drawn diagram for the integration of commit 2 compared with the correct answer. 72
Figure 6.5 An example of a drawn diagram for the integration of commit 2 compared with the correct answer. 74

Figure 6.6 Difference between commits in the correctness of responses to task T5. 76
Figure 6.7 Aggregated Correctness of the summarization results 77
Figure 6.8 Difference in accuracies in responses to task T9 between Commit 1 and Commit 2. 79
Figure 6.9 Aggregated Accuracy of the summarization results 80
Figure 6.10 Difference in time taken to respond to task T7 between merge sizes 82
Figure 6.11 Difference in time taken to respond to task T8 between merge sizes 83
Figure 6.12 Aggregated Time to respond to summarization tasks 83

Figure 7.1 Updating the Merge-Tree shows the order that commits are created, while retaining the merges that the commits pass through. 92

ACKNOWLEDGEMENTS

I would like to thank everyone who has contributed to this work, to my undergraduate degree, and the completion of my masters degree.

I am grateful for the opportunity to study with support from my family, guidance from my profssors, and assitance from the staff in the department office. Without these people, I would not have been able to complete this work.

I am also very thankful for the participants of the study. Without their cooperation, it would not have been possible to evaluate this work. They took time from their work to assist me with mine. The information provided by them enabled the completetion of this thesis.

There have been challenges, but I look forward to the opportunities that lie ahead as a result of the investment of time and energy from everyone involved.

I would like to specifically thank:

My parents for their unconditional support and encouragment.

Daniel German for putting up with my indecision.

Chapter 1

Introduction

A version control system records the changes being made to the files of a project, enabling users to view previous versions of the files, view the individual changes made, and restore the files back to a previous state if necessary. The version control system also maintains a log of who made changes and when those changes were made. By storing this information, the version control system stores the history of the project. Presented in the right way, there are many opportunities to use this information to help users understand the evolution of the software system. A version control system can be used in any context where file history is needed, it is usually used in the software development process.

Git is a version control system (VCS) used by the Linux kernel project. Git was designed by Linus Torvalds for the Linux project as a replacement for BitKeeper. In order to handle the number of people contributing to the kernel from different locations, git was designed to be distributed. Unlike in centralized version control systems, where users must re-synchronize with the server, a distributed version control system provides each user with a full first-class repository. This allows the user to have additional flexibility, and means that a user has to synchronize their local repository with the remote repository less often. Furthermore, each user has access to the entire history of the repository, including all branches and commits that were part of the original repository. Users are able to combine and re-order commits before making the changes publicly available, which alleviates issues with synchronization between developers. To make it more useful to the Linux project, git was designed to allow easy branching. Branches allow users to work on a logically separate part of the repository, then merge the changes into the repository once the feature is finished or the bug is fixed. This lets git users work independently on a feature, taking full

advantage of version control, without needing to worry about synchronization until the feature is ready to be integrated. To support these features, git uses a directed acyclic graph to represent the structure of the commits in the repository. The nodes of the graph represent the commits, containing the changes being made and metadata about when and who made the changes. The edges of the graph represent the parent relationship between commits.

Visualizations of this graph are used to answer questions about the development of the software including what changes are being made into various branches, how the changes to the code are grouped, and who is working with whom, among others. Maintainers use these visualizations to understand what changes are being made to the current version of the software in order to apply the necessary fixes to older versions of the software to keep them secure and performing correctly. This requires understanding how a commit is integrated into the repository, and other commits that are merged with that commit. In large, active, software repositories this task is not trivial. The graph can be large and complicated, making these visualizations difficult to understand. The difficulties in understanding how commits are integrated into the Linux kernel repository drives the overarching question behind this thesis.

Overarching Research Question: How can we effectively visualize the graph of the Linux repository in a way that gives insight into how commits are integrated?

To answer the overarching question, this thesis makes four contributions. First, a new tree-based model, called the Merge-Tree, is abstracted from the underlying graph of the repository. Compared to the graph, trees are relatively easy to visualize, and there are many visualization metaphors that take advantage of different properties of trees. The Merge-Tree abstracts the repository commit graph into a set of trees, each rooted at a merge into the master branch of the repository. The leaves of the tree are the commits, and the inner nodes of the tree represent the merges leading to the integration of the commits. Second, this thesis proposes three visualizations that take advantage of the Merge-Tree model. Third, an implementation of the visualizations in a tool called *L*invis. Fourth, the last contribution of this thesis is an evaluation of the tool.

Thesis: Trees are more effective for visualizing and summarizing the integration of commits into the Linux kernel repository than the DAG.

1.1 Thesis Organization

This thesis is organized as follows. Chapter 2 contains background information about the motivation for this work and the structure of git repositories.

Chapter 3 introduces the Merge-Tree model. This chapter includes a description of the model, an algorithm to convert from the DAG to a set of Merge-Trees, and an evaluation of the resulting trees built from the Linux repository graph. At the end of the chapter is a summary of the information found in the Linux repository, including the number of authors contributing, the number of commits, and the average number of nodes per Merge-Tree.

Chapter 4 introduces *L*invis, providing the use-cases that were being targeted. This chapter also includes the features that were implemented into *L*invis, including its search engine, summarization tables, and tree visualizations. More details on how the tool was implemented are included in Chapter 5.

Chapter 6 is the empirical evaluation of *L*invis, and include the methodology and results of this two-part study. The first part evaluates user comprehension of the DAG and the second part compares visualizations and summarizations of the DAG in Gitk against the visualizations and summarizations of the Merge-Tree in *L*invis.

Chapter 7 discusses the results of the study providing more insight on the results. The chapter includes observations from the study, and the comments from one of the members of the study who had worked as a release manager, and a description and algorithm for an updated Merge-Tree that takes into account the comments and observations from the study. The chapter concludes with the limitations of the work and the future work.

Chapter 8 concludes that paper, reiterating the problem addressed by this thesis and how it was solved.

Chapter 2

Background and Related Work

A version control system tracks files, and how they are changed over time. While it can be used for storing any digital information, version control is usually used in the context of software development; it is used for storing and managing the files of the project, including the project's source code and binary assets. Version control has two primary purposes: first as a means of storing files, and second as a means of retrieving historical versions of those files. In addition to fulfilling the two primary purposes, a version control system maintains a log of who is making the changes and when the changes were made.

Early version control systems, such as Revision Control System (RCS), were designed for local development and provided very little support for collaboration. As software projects grew, the model quickly became outdated, being replaced with a centralized server model. Concurrent Version System (CVS) and Apache Subversion (SVN) are two examples of centralized version control systems. The centralized version control system provides means of collaboration through a client-server interface. The repository is stored in a central server. Developers use a client to check out parts of the repository, choosing parts that pertain to the part of the codebase that they are editing.

In large open source projects, the centralized architecture becomes a burden. Common tasks such as committing and changing branches requires re-synchronization with the central server. To work with the repository, the developer must always have access to the central server. To maintain the atomic properties of committing, the server will momentarily lock the repository to ensure that no other changes happen while a commit is being processed or a conflict resolved.

Due to the limitations of a centralized architecture, the Linux kernel uses a dis-

tributed version control system. Until April of 2005, the kernel project used BitKeeper. In April 2005, the licensing became too restrictive and Linux was forced to change version control systems[1]. Git was written as the replacement for BitKeeper, and was designed to maintain a similar level of patch granularity as in BitKeeper[2]. The first version of git was roughly 1300 lines of code and was written and self-hosted in less than two weeks[3].

Both BitKeeper and git are distributed version control systems (DVCS). In distributed version control, the entire repository is mirrored on the developer's local computer instead of copying parts of the project. As a result, the local copy is much larger on disk than with a centralized repository, but the developer has the freedom to make changes to the code and to the structure of the repository without needing to re-synchronize with a central server.

It is often desirable to have the features of version control before a feature is ready to be made available in a public-facing repository. It is also desirable that the commits into the master branch of the master repository leave the project in a state that will both compile and operate correctly. Distributed version control makes this possible; developers can combine, split, and edit commits locally before pushing their changes into the central repository.

A clone refers to a copy of a repository, cloning occurs when the developer makes a copy of a target repository, the target is recorded as a remote repository. It is possible for repositories to have many remotes, or have none. The process of updating a remote repository with the changes made in the local repository is known as pushing. Conversely, updating the local repository with the changes made to a remote are referred to as pulling. By default, the remote repository that was cloned will be given the label *origin*, which is the repository where git will push to and pull from, unless another remote is specified.

After changes are made to a remote, the local repository must resynchronize with the remote in order for those changes to be propagated. This resynchronization can be done in one of two ways. The normal way of resynchronizing is through the *pull* command, which fetches the changes in the origin repository, then merges the branches of the origin repository into the corresponding branches in the local repository. The second way of resynchronization breaks the process into two steps,

[1] `https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git`

[2] initial announcement of git on the mailing list `https://marc.info/?l=linux-kernel&m=111280216717070`

[3] From the git mailing list `https://marc.info/?l=git&m=117254154130732`

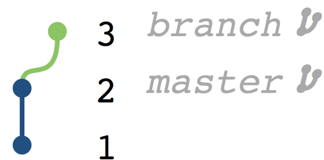
manually issuing the *fetch* command and then manually merging or rebasing branches. Rebasing is the process of moving one or more commits from one ancestor to another.

The process of updating a remote repository with the changes made in the local repository is known as pushing. If there are no merge conflicts, the merge can perform a fast-forward merge, shown in Figure 2.1b, which flattens the changes made in the origin repository into the master branch of the local repository. Fast-forward merging hides the fact that git would otherwise consider the two repositories to be separate branches. Without fast-forward merging, any resynchronization would result in the addition of a new merge node. If there is a merge conflict, or the user has specified that the merge should not fast-forward, a merge commit is created, as shown in Figure 2.1c.

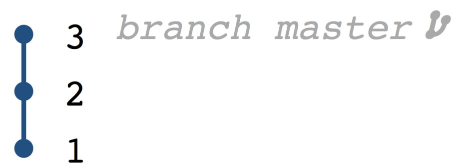
Distributed version control gives developers more flexibility with their local repository and requires the developer to synchronize their local copy of the repository with the public master repository less often than with centralized version control. The public master repository can be thought of as being the equivalent of the central repository in a centralized system. Instead of it being enforced by the version control system, it is a socially agreed upon location where the official version of the code exists. Unlike with the centralized version control though, there is no requirement for the developer to ever push their changes back to the public repository; the local repository is completely standalone. Developers can make changes that would otherwise break the workflow of other developers because they have a standalone repository. These changes include rebasing branches, re-ordering commits, splitting commits, and squashing commits into one. Once the developer is happy with their set of commits, they may push them to the remote repositories.

Git is designed to handle multiple repositories, with many developers working simultaneously. It is also be able to support the ability to move commits between branches, re-order commits, combine multiple commits into a single commit, and split a commit into multiple commits.

To support these feature, Linus chose to use a directed acyclic graph to represent the commits and the relationships between them. The graph imposes relatively few constraints on what a developer can do. The only requirement is that there is not a cycle in the graph of the commits, that is, a change cannot depend on itself. Git does not impose the requirement of a master branch. SVN always has a well-defined trunk, the SVN equivalent of a master branch. The graph structure also supports relatively cheap branching compared to other version control systems, which makes



(a) The repository contains two commits that are part of the master branch, with one commit that is part of a separate branch waiting to be merged.



(b) A fast-forward merge does not create a merge commit, and instead moves the branch pointer forward to the commit that is being merged into the branch.

(c) A merge commit is created when the merge cannot be made cleanly, or *–no-ff* is passed to the merge command.

Figure 2.1: A depiction of the distinction between fast-forward merges and non fast-forward merges

it possible for developers to create more branches without having to worry about consuming excessive resources. Git places fewer constraints on the structure of the commit graph than many other version control systems. For example, most version control systems need a branch to take the role of a main, or master, branch, whereas git has no such requirement. With fewer constraints on the structure of the commit graph, tools are unable to make as many assumptions when abstracting the graph. Many tools avoid this by not abstracting the graph and visualizing other properties of the repository, such as the file structure. Tools that do visualize the graph do only minimal abstraction, creating a visualization of the graph itself.

The graph of large and active repositories is very complicated. It is very difficult to understand the relationship between commits, and how the commits are integrated into the project from a visualization of the graph. This poses a problem for maintainers who must understand how a commit is integrated into the master branch of a project and the other commits that are integrated with that commit. Maintainers must sift through thousands of commits to determine which changes being made to the current version of the software pertain to the area of the software that they are maintaining. Specifically, maintainers must be able to answer two questions:

- How is a commit integrated into another branch?
- What other commits are integrated with the commit?

The remainder of this chapter includes related work, a description of git and how it's used, the directed acyclic graph that underlies git repositories, and an explanation of why this work focuses on the Linux kernel repository.

2.1 Related Work

A Version Control System (VCS) tracks the development of a software project, recording each change as it happens. By tracking the changes, the VCS contains the entire history of the software, rich with information about who the authors are, what files

are being modified, and the changes being made. This makes the VCS vital in providing information about how a software project is being developed and how the software is structured. In order to use the information stored in the VCS, users must be able to gain a clear understanding and summarization of the changes being made, and how they interact with the rest of the source code. While there has been extensive research on visualizing software repositories, previous work does not focus on how commits and merges are structured in the repository graph, and in extension, how commits are integrated into a repository.

The literature on repository visualization and summarization can be broken down into three academic subcategories: communication[8, 2], aspect-oriented visualization[9, 4, 10], and visualizations of naturally occurring phenomena[21, 5]. A fourth industrial category exists, including tools like GitKraken and SourceTree. The goal of the industrial tools is not to extract or synthesize new information from the repository, but to act as a user-friendly client on top of what git already provides.

Many tools focus on addressing the issue of communication between developers in inter-team collaborative work. Hipikat[8] investigated communication between developers, focusing on assisting with the integration of new developers into a project though communication, providing the new developer with searchable artifacts of the changes being made, and where to find them. The artifacts may include files or bug information, shown in Figure 2.2. Codebook[2] also focused on communication, but while Hipikat focused on assisting new developers find artifacts, Codebook assists developers with finding who was responsible for creating the artifact. Codebook used a data-mining technique to determine the developer of a piece of code, the program manager who wrote the specification for the code, and the program managers and developers on the team who were working together. A screenshot of Hoozizat, an implementation of Codebook, is shown in Figure 2.3. Hoozizat and Hipikat use the version control as the archive of artifacts that are being queried. Neither tool is designed with the goal of providing information on the topological structure of a source code repository, nor are these tools designed for visualization purposes, but they do draw information from the contents of the version control system.

Most visualization systems provide information about a certain aspect of the contents in the repository. The goal of Fractal Figures[9] is to show the division of work between contributors. The project is represented as a square. The square is then subdivided based on the proportion that a given contributor contributed to the project, shown in Figure 2.4. The visualization makes it easy to see where work is evenly

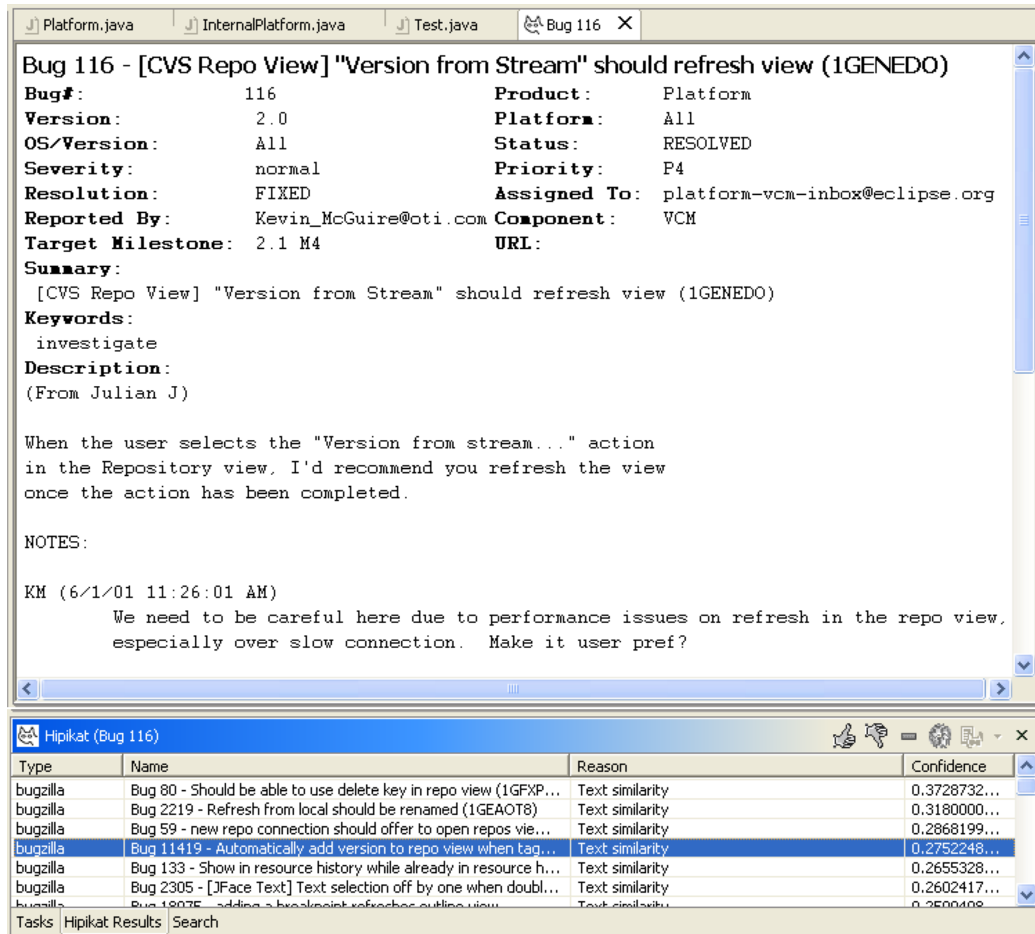


Figure 2.2: View of Hipikat, listing bugs that are similar to the one being viewed

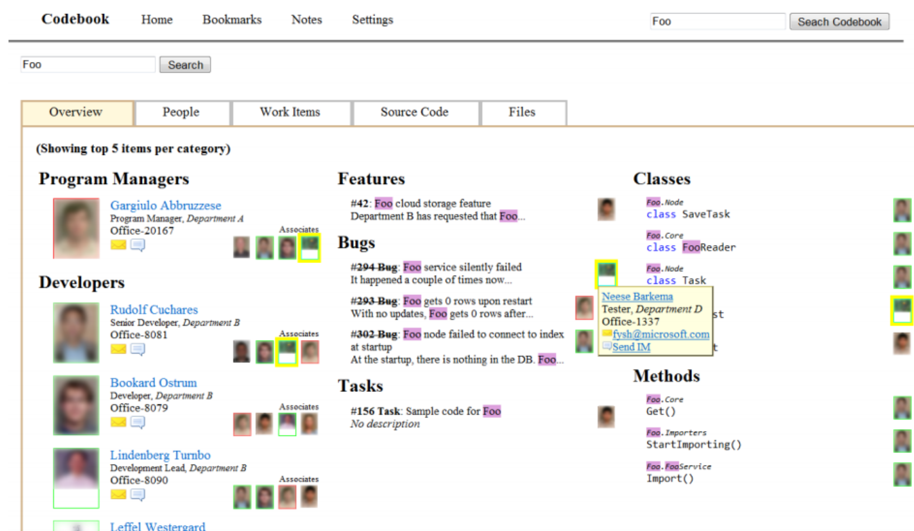


Figure 2.3: A screenshot of the search results on Hoozizat, an implementation of codebook.[2]

divided versus the projects where a single contributor is doing most of the work.

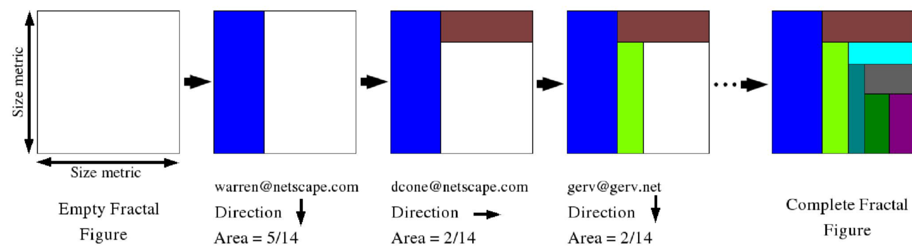


Figure 2.4: Construction of Fractal Figures[9]

EPOSee[4] and Evolution Radar[10] use the information from the version control system to determine which files are edited together. These tools are designed to help a user identify the degree to which two files are coupled. Two files are edited and committed together frequently are said to be more tightly coupled. This makes it possible to determine when two classes are semantically related. The evolution radar shown in Figure 2.5 places points on a circle based on the name and how tightly coupled they are. The files are arranged around the circle based on the file name, including the full file path. This has the effect of grouping files that are from the same directory. The distance from the center of the circle is dependent on how tightly coupled the file is to the file be analyzed. A more tightly-coupled file will be positioned more closely to the center of the circle.

Hoozizat, Hipikat, Fractal Figures, EPOSee, and Evolution Radar all extract data from CVS repositories. Our goal is to provide information about git repositories. Fewer tools are available for generating visualizations and summaries of git repositories.

The visualizations of naturally occurring phenomena show patterns in cooperation and communication that arise within a software project. Heller et al.[17] plots communication on a map. This visualization show patterns in communication as they arise, and how these communication channels operate internationally within a software project, depicted in Figure 2.6.

The visualizations proposed in Gource[5], shown in Figure 2.7, shows which files contributors are working on. Using this, it is possible to draw conclusions about which parts of a project a given contributor is working on and the group of contributors working on a given area. Gource uses a graph metaphor structure to represent the file structure of a repository. Files in the same directory cluster together to form a node. Edges between the directory clusters represent which directory contains

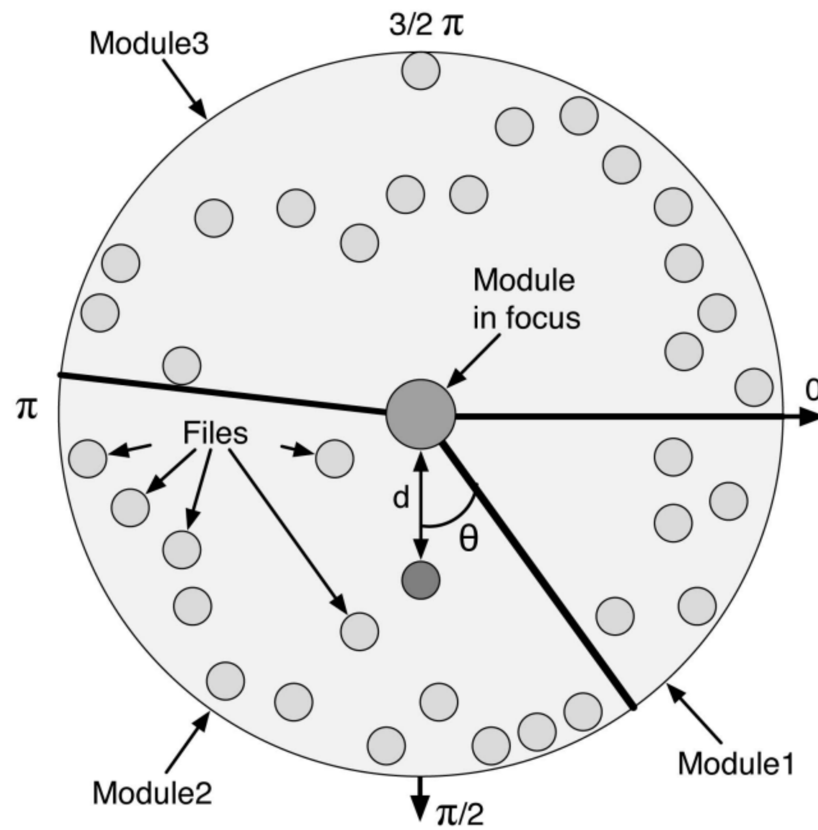


Figure 2.5: Evolution Radar visualization[10]

Figure 2.6: A screenshot of the communication mapping tool by Heller et al.[17]

another, although there is no way to determine the direction of the relationship. User avatars move around the graph emitting different beams of colored light depending on the change being made to the file. Green indicates the creation of a new file, yellow indicates a modification, and red indicates the deletion of a file. The visualization is animated to show how a project grows over time. Codeswarm[21], shown in Figure 2.8, is similar to Gource, using a timelapse approach to visualizing the events in the repository. Unlike Gource, which constructs a graph from the directory structure of project, Codeswarm does not have a graph structure; developers are the center of the visualizations. When a developer makes a change to a file, the file lights up and flies toward the developer. As a developer makes more changes, the files that the developer is modifying will form a ring around their avatar. If multiple developers are modifying a file, the developer nodes are drawn together.

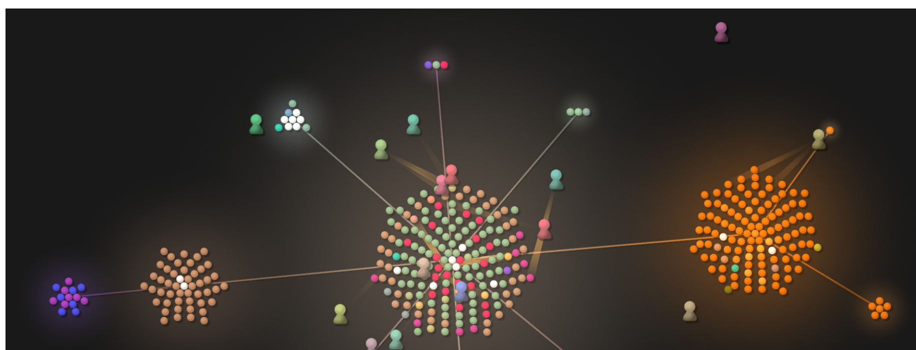


Figure 2.7: View of Gource file graph with users operating on a repository[5]

There are many non-academic tools that are designed as an interface to git. While not all of these programs provide visualizations, those that do use a visual metaphor of the DAG to show topological relationship between commits. While they ultimately show the same information, the topology of the repository, the organization of that information is different.

Gitk is the graphical interface that is shipped with Git, shown in Figure 2.9. The interface is fairly complex, and looks a little dated. The program displays all of the information that is stored in a commit, giving what is likely the most complete view of the information stored. Unfortunately, the presentation makes the interface appear somewhat overwhelming.

GitKraken [1], shown in Figure 2.10, is a popular commercially-written git interface that aims to be efficient, elegant, and reliable, according to it's official website. On visual inspection, it appears to satisfy these goals. Overall, the interface is clean

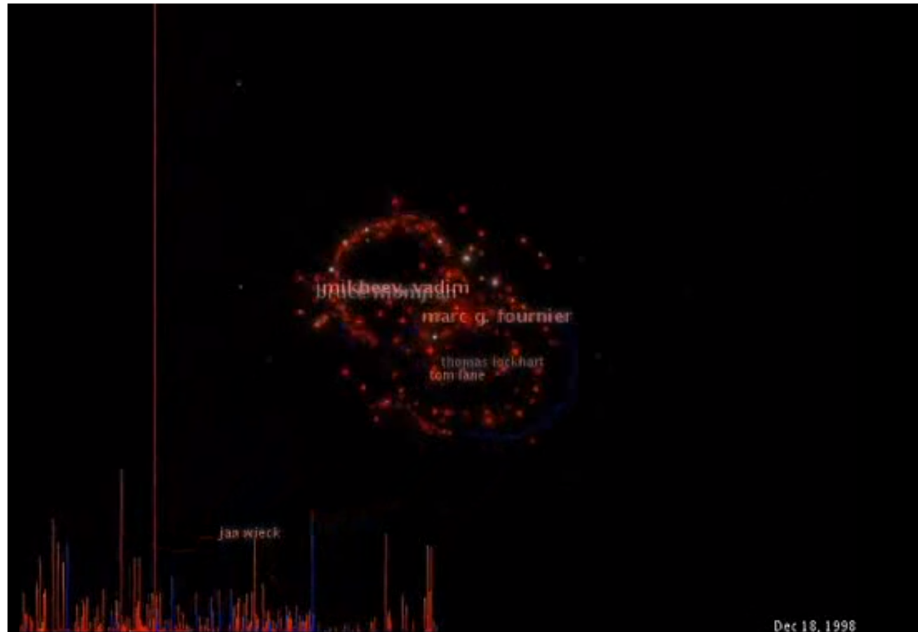


Figure 2.8: View of the Postgresql repository in Codeswarm[21]

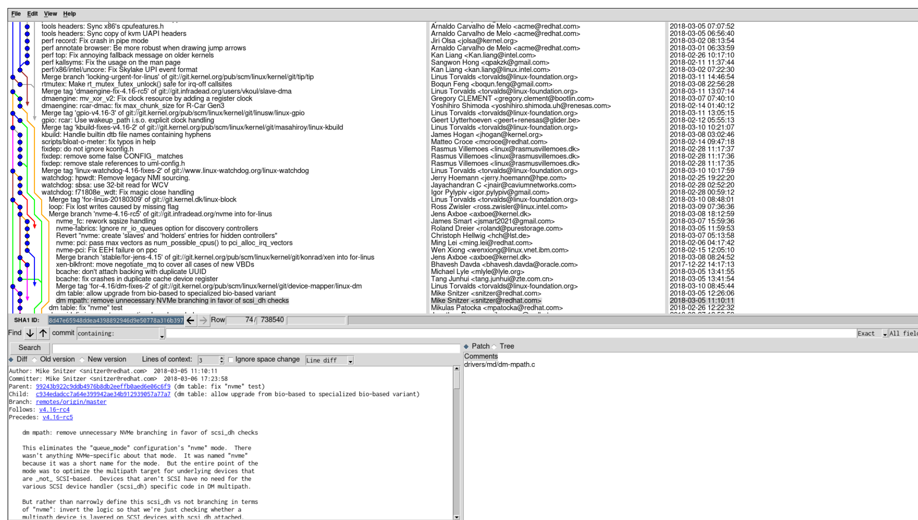


Figure 2.9: Gitk interface, the graphical interface shipped with Git.

and most actions that are possible with the git command line are available in the graphical interface. The tool is effective and garners online approval from users. The graph of the commits is shown in the center of the main view and provides users with the same information as the graph visualization in gitk and the git command line, though it may be visually more appealing.

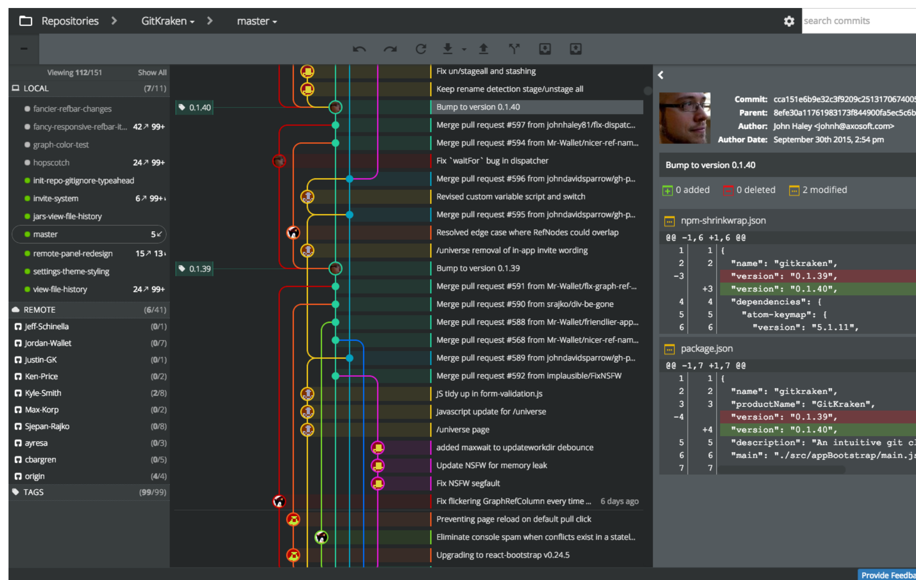


Figure 2.10: Screenshot of the main view in GitKraken[1]

In January of 2018, the Gnome project released a replacement for Gitk. Gitg[16], shown in Figure 2.11, is the git GUI client for the Gnome environment. The visualization is relatively clean, and it is able to produce a visualization of the Linux repository quickly. Like in Gitk, Gitg uses arrows to indicate that a branch has been cut. Unlike in Gitk, the arrows do not hyperlink which makes it difficult to find the parents of a commit. There is no apparent way to find the other side of the branch, as the interface does not provide information about the parents or children of the commit.

Giteye[23] and most of the other visualizations are relatively conventional, simply acting as a cleaner version of Gitk. GitLab[15] and GitHub[14] are both online repository hosts, with visualization and summarization provided as well. While the GitLab visualization does not appear to provide any additional information, the visualization provided by GitHub takes advantage of additional internal knowledge to display information about forks. Through this visualization, GitHub displays the branch history of the repository network, including the branches of the main repository and forks

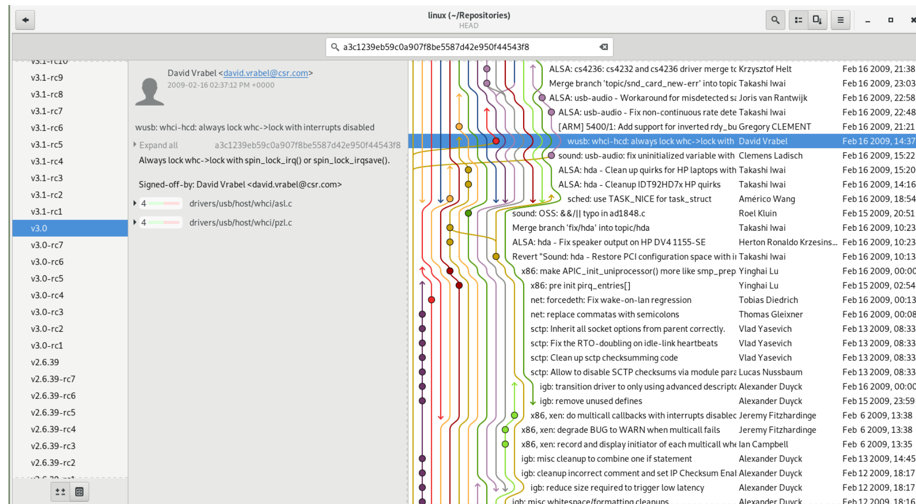


Figure 2.11: Gitg interface from the Gnome project[16]

from that.

With the exception of Gitk and Gitg, no GUI visualizers are able to produce a visualization for the Linux repository, due to its size: the GitHub visualizer displays an error message, stating that there are too may forks to display; the GitKraken interface will freeze and eventually crash while trying to load the repository; Giteye and the other visualizers will consume all of the system memory before they are able to produce a visualization. The Gitk interface is the least polished, but is able to produce a visualization of the repository.

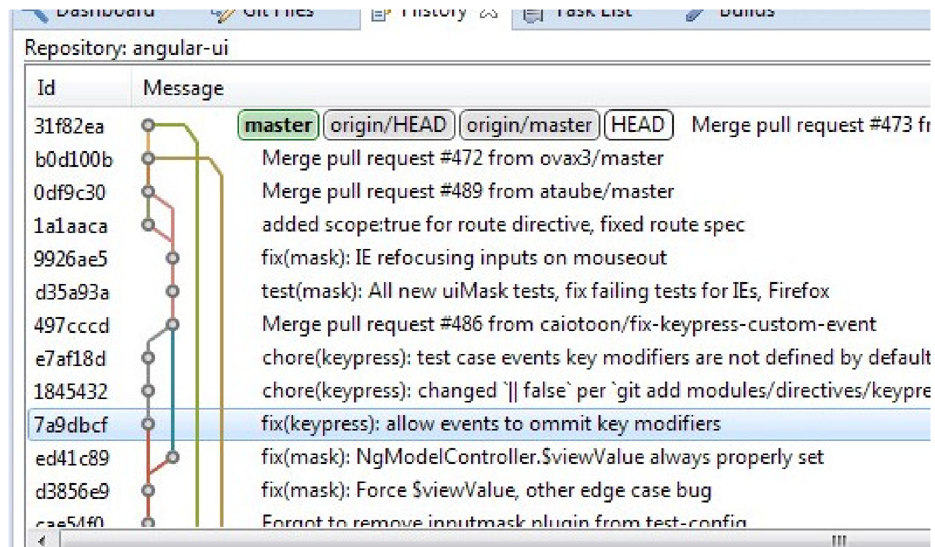


Figure 2.12: Screenshot of Giteye DAG view of a repository[23]

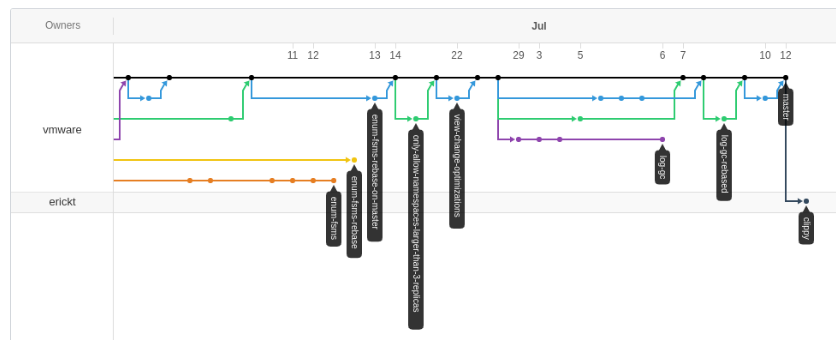


Figure 2.13: GitHub online network view of a repository[14]

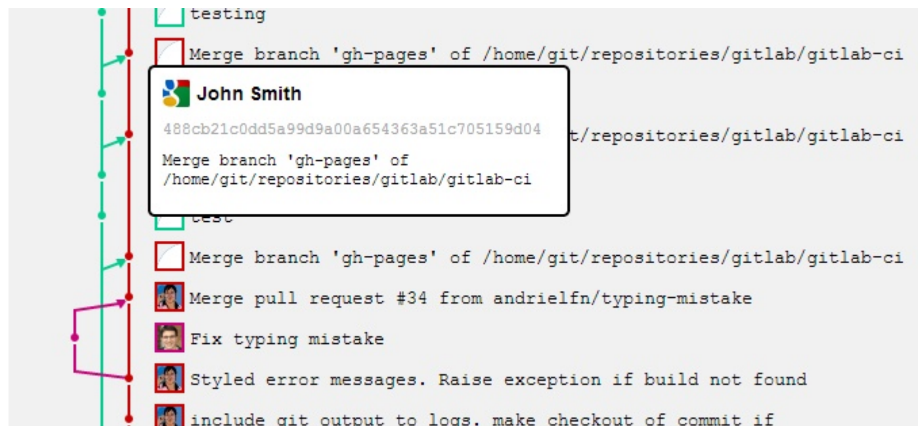


Figure 2.14: GitLab online graph view[15]

2.2 Git

Commits are the core of git repositories, storing the patch representing the changes being made to the files in the repository, and metadata about when the change was made, and who made the change. In the metadata, commits store an author, an authordate, a committer, and a commit date, and an ordered parent list of the commit hashes. The author is the person who first created the patch and issued the commit command. The author date is when the commit was first created. The committer and commit date contain the person who most recently updated the commit, and when that update was made. This metadata is exemplified in Figure 2.15 for a commit and merge.

```
commit 333fc72d451e57557860447893a483186f5767d8
Merge: 45cca4c 192ac7c
Author:     Evan Wilde <etcwilde@uvic.ca>
AuthorDate: Sat Feb 10 21:03:59 2018 -0800
Commit:     Evan Wilde <etcwilde@uvic.ca>
CommitDate: Sat Feb 10 21:03:59 2018 -0800

    Merge branch 'background' into editing

commit 192ac7cf992623aeabef32d04520f3594bbfad1e (background)
Author:     Evan Wilde <etcwilde@uvic.ca>
AuthorDate: Sat Feb 10 20:59:08 2018 -0800
Commit:     Evan Wilde <etcwilde@uvic.ca>
CommitDate: Sat Feb 10 20:59:08 2018 -0800

    [Background] Adding remainder of edited background
```

Figure 2.15: Screenshot of commit metadata for a commit and merge in the repository of this thesis.

Commits are immutable; a commit cannot be modified once created. When a commit needs to be updated, a new commit is created, the original metadata is copied to the new commit, the relevant changes are made, the comitter and commit date are updated, and the original commit is deleted. If other commits have the original commit as one of their parents, they are updated in the same manner to reflect the new parent. This happens recursively until all descendants of the original commit have been updated to reflect the new commit hash.

The patch contains the changes being made, including the filenames, the line numbers, and the actual change, as shown in Figure 2.16

Rebasing will also change the committer and commit date.

```
diff --git a/include/linux/compiler.h b/include/linux/compiler.h
index eca8ad75e28b..043b60de041e 100644
--- a/include/linux/compiler.h
+++ b/include/linux/compiler.h
@@ -517,7 +517,8 @@ static __always_inline void __write_once_size(volatile void *p, void *res, int s
 # define __compiletime_error_fallback(condition) do { } while (0)
 #endif

-#define __compiletime_assert(condition, msg, prefix, suffix) \
+#ifdef __OPTIMIZE__
+# define __compiletime_assert(condition, msg, prefix, suffix) \
  do { \
  bool __cond = !(condition); \
  extern void prefix ## suffix(void) __compiletime_error(msg); \
@@ -525,6 +526,9 @@ static __always_inline void __write_once_size(volatile void *p, void *res, int s
  prefix ## suffix(); \
  __compiletime_error_fallback(__cond); \
  } while (0)
+#else
+# define __compiletime_assert(condition, msg, prefix, suffix) do { } while (0)
+#endif

 #define _compiletime_assert(condition, msg, prefix, suffix) \
  __compiletime_assert(condition, msg, prefix, suffix)
```

Figure 2.16: Example of a commit patch from the Linux kernel repository from commit *c03567a8e8d5*.

The parents of a commit are the next commits toward the initial commit. The first parent in the list is the commit that is on the same branch as the commit that is being created, i.e, the branch that the other branches are being merged into. The remaining commits are the branches being merged, in the order that they are specified in the merge. A non-merging commit will only have one parent.

To clarify the difference, non-merging commits are referred to as commits and merging commits as merges. In the scope of this thesis the term, repository event or simply event is used to refer to either a commit or a merge.

Integration is the process by which the changes in a commit are propagated to the master repository. A small change with few dependencies is easier to integrate than large changes. Many times, small changes that are localized contain bug fixes or small changes to documentation. An example of this from the Linux repository is shown in Figure 2.17.

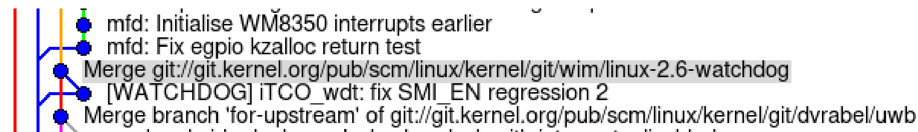


Figure 2.17: Example of a merge that only integrates a single commit

Large changes may be broken down into smaller changes and committed separately. The set of these commits combined represent the full implementation of the change, which represents a logical separation from the rest of the code, and should be merged separately. A large set of changes may be necessary to implement an entire feature. Each of these large changes may be merged into a feature branch before being integrated into the project.

To understand how a commit is integrated, it is necessary to understand the merges that the commit was propagated through, and which commits are integrated with it. In order for a commit to be integrated, it must be propagated to the master branch. Merging commits is the process of integrating them. The other commits that are merged with the commit are also necessary for the given commit to be integrated in a meaningful way.

2.3 Directed Acyclic Graph

To allow for the flexibility needed for a distributed version control system, git uses a directed acyclic graph (DAG) to model the relationship between events. The repository

events make up the nodes in the graph, and the child-parent relationship represents the edges. Commits will have a single parent, which is the repository event that is at the head of the current branch at the time that the commit is created. Merge nodes have an ordered list of parents[4], each parent is the head of each branch being merged. The first parent is the head of the current branch, and the other parents are the heads for the other branches being merged, in the order that they are specified in the merge command. Every repository will have at least one initial commit, which will have no parents, but it is possible for repositories to have multiple initial commits. Furthermore, it is possible for the graph of a repository to be disconnected; branches that do not interact with the master branch are referred to as orphaned branches.

The model is simple, but flexible. The flexibility of the model makes it more difficult to reason about, stricter models are easier to reason about since the model must follow more rules.

For example, many version control systems have a well-defined notion of the master branch. In SVN, this is referred to as the "trunk" branch. There is a single trunk branch, and it is well-defined, it won't be confounded with another branch. The DAG model in git does not explicitly define a master branch, or even enforce the requirement that one exists. Instead, the idea of the master branch is a social construct used to identify where releasable code should be merged into, and where the final product will be released from. This relies on the discipline of the people committing code to the repository to maintain a well-defined master branch.

The convention in git is that the first parent of the current commit was made to the same branch as the commit. Using this definition it is possible to define the set of commits in the branch as those that are long the first-parent path up to the first place where the first-child of the first-parent is not a commit of the branch. Using the example in Figure 2.18, branch B consists of nodes 6, 4, and 2. Branch A consists of nodes 7, 5, 3, and 1.

Git has no internal safe-guards to protect branches from obfuscation. When certain conditions are met, it is possible to perform an action on the repository which results in commits to appear as if they were performed as part of a different branch. The series of steps to swap branches is called a foxtrot[5].

It is necessary for multiple repositories to be interacting for a foxtrot to occur.

[4] It is possible for a merge to have many parents, commit 2cde51fbd0f3 has 66 parents

[5] See `http://bit-booster.blogspot.ca/2016/02/no-foxtrots-allowed.html` for a full description of the issue

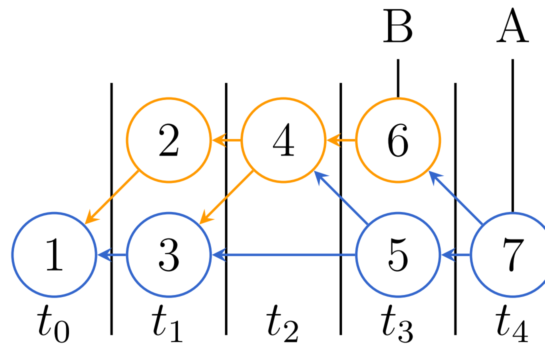


Figure 2.18: A small example of a sequence of commits and merges. The branch pointer A references commit 7, which merges the head of branch B, commit 6, into the original head of branch A, which was commit 5. Merge 7 is the most recent change to the repository.

The following is a short example of the series of steps in the foxtrot, also shown in Figure 2.19. Bob and Alice have both made local clones of a remote repository and are making changes to the master branch of their local repository. Bob and Alice both make local changes to the same file in the repository and commit those changes into the same branch. Alice pushes her changes to the repository first, which results in a fast-forward merge of the remote branch. Alice's commit is clearly pushed to the master branch. Bob attempts to push, but the push fails as his repository is not in sync with the remote branch anymore, so Bob pulls. The pull merges the difference in the remote branch into the local branch. Alice and Bob edited the same file creating a merge conflict, so Git cannot perform a fast-forward merge. Bob resolves the conflict and a merge commit is created to store the resolution. The head of Bob's local branch at the time of the pull is the first parent of this merge commit, and the changes made by Alice are the second parent. With the merge conflict resolved, Bob pushes the changes back to the remote branch. Prior to Bob pushing his changes to the remote repository, Alice's commit was at the head of the master branch. After Bob's push, this information is lost, and it appears that Alice's commit was merged into the master branch by Bob. This sequence of operations swaps the branches: the commits that were in remote's master now appear to be made to a separate branch and merged into the master branch, while Bob's commits appears as if they were made to the master branch. The merge commit that merges the remote master branch into Bob's master branch is the foxtrot merge.

The effect of this can have different repercussions depending on the project. In the best case, the repository visualizations will not give an accurate visualization of how commits were integrated into the project, which may lead to confusion. In a more serious situation, a specific branch is considered to be the stable branch, where only code that has been reviewed and tested is accepted. When a regression occurs, the project may need to revert back to a previous stable state, where the regression is not present. This requires the ability to find and track the master branch, which may be confounded by a foxtrot. Picking the incorrect commit to revert to could lead to serious consequences. In the Linux project, the merges to the master branch are the code that Linus has reviewed and accepted into the mainline kernel.

As mentioned earlier, the nodes in the DAG are immutable; once a commit or merge is created, it cannot be changed. Git allows operations to alter the events and re-order them, but this will create a new event with a new commit hash. This property makes it impossible for nodes to store information about their children, and

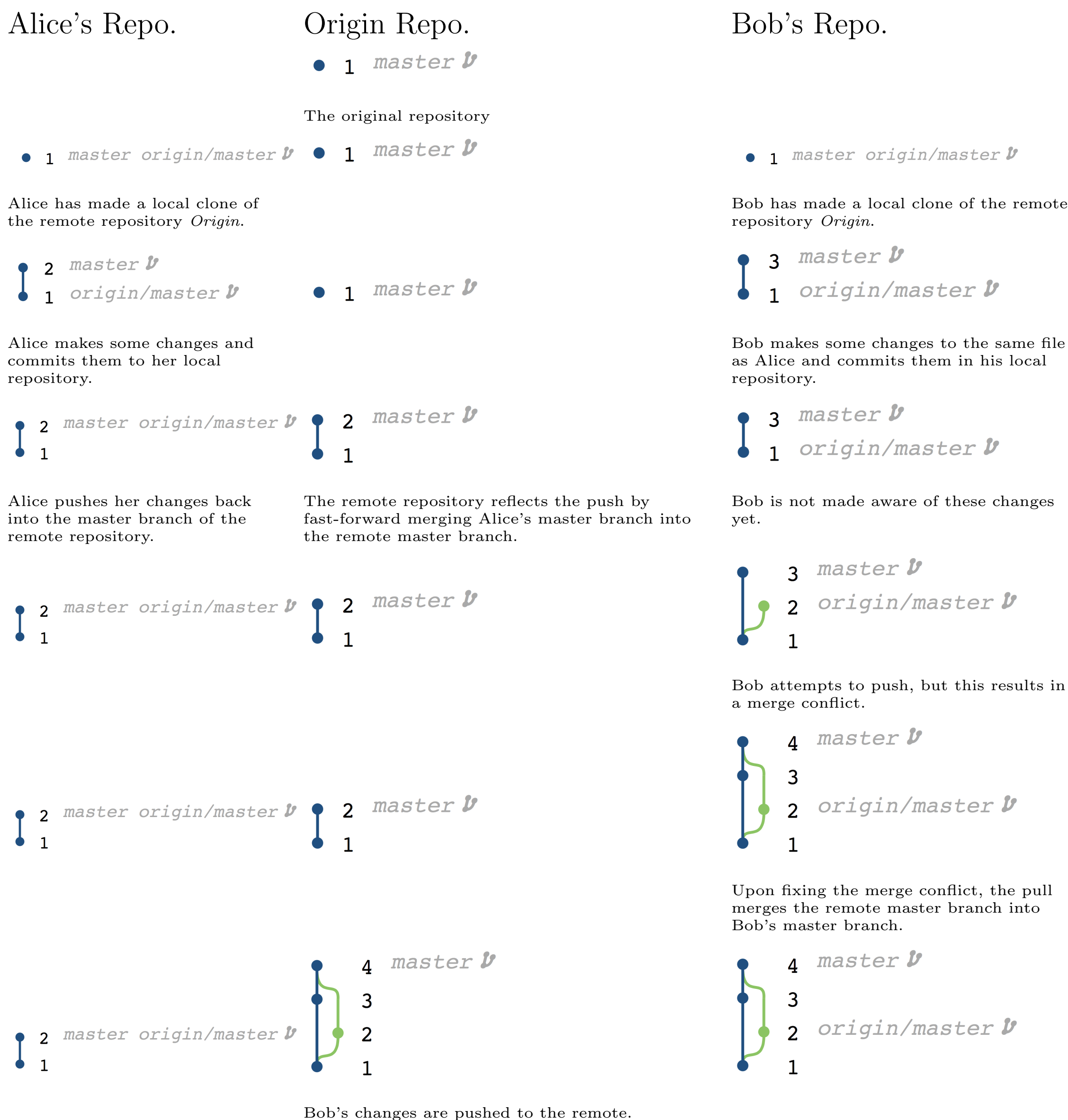


Figure 2.19: The sequence of steps that are part of the foxtrot, from the point of view of each repository. Alice's commit (2) is pushed to the master branch but, as a result of the push by Bob, the master branch has been swapped with Bob's branch. This type of merge (4) is called a foxtrot.

(a) In the simple case, it is relatively easy to see that *a231d* is merged into *6bf99*, which is then merged into *545ea*.

(b) As the number of merges that a commit passes through increases, it becomes more challenging to understand how the commit is integrated.

Figure 2.20: The git graph visualization of two sections of the Linux repository.

in extension, how the commit is being merged, as this information is not available when the commit is created. Git provides the command `git log --children`, which traverses the DAG and inverts the edges. The next child on the path of children that is a merge is the first merge on the path to the commit being integrated.

The graph combined with the child information gives most of the information necessary for understanding how the commit is merged into the master branch; however, information about the specific merge into the master branch is still missing. Furthermore, the graph itself is not always easy to understand, as shown in Figure 2.20. This figure contrasts the levels of complexity that can be found in a given section of the Linux kernel repository.

Most repositories are simple enough that it is possible to identify how commits are integrated using the visualizations of the DAG that are available with the current tools. Difficulties arise in larger repositories. The master branch can be confounded due to *foxtrot* merges, making it difficult to identify merges to the master branch. Sheer number of commits being added to various branches at a given time can make it difficult to understand which branch a commit is being added to.

2.4 Linux

The Linux repository itself is complex, containing tens of thousands of commits and thousands of merges per year. Older versions of the kernel are used in a wide variety of situations including various Linux desktop distributions, IOT device firmware, web servers, spacecrafts[6], and in mobile devices as the kernel of the Android platform. These kernels are sometimes modified forks of the official Linux kernel, made to be more suitable for the specific needs of the application. Due to these application-specific modifications, it is not feasible to update to the latest version of the kernel. While it may not be feasible to update to the next version of the kernel, the changes being made to the official version are necessary as they fix bugs, patch security issues, and improve performance. Due to the sometimes critical nature of the patches being merged into the current version of the kernel, it is necessary for maintainers working on an application-specific fork of the kernel to sift through the commits coming into the official version, looking for changes that may impact the kernel that they are maintaining.

[6] Linux is used heavily at SpaceX `https://lwn.net/Articles/540368/`

Linux itself follows strict development practices, which reflect in the structure of the repository. Linus Torvalds is the only contributor with write-access to the master branch and is able to accept or reject commits and merges as he chooses. This ensures the quality of the commits into the kernel, as well as maintaining a high level of consistency in how commits are merged. Under Linus are a handful of primary maintainers who accept changes related to a specific subsystem of the kernel. For example, Andrew Morton manages the memory management of the kernel, while David Miller handles the changes for networking subsystem, as well as the changes to the SPARC implementation. The primary maintainers collect patches that are related to the part of the kernel that they are maintaining, verify that the patches meet the quality requirements of the kernel, and pass them to Linus, who merges them into the master branch.

This structure is reflected in the repository graph. Linus merges commits based on the subsystem that the commits are changing. Within the merge that groups changes to the networking subsystem are commits related to networking and a merge for wireless networking. In the merge for wireless networking are additional merges that group commits that make changes to wireless networking technologies like bluetooth and mac80211. Merges are used in the repository in the same way that directories are used in a file system, grouping related information.

The model, visualizations, and tool presented in this thesis take advantage of the consistency of the repository, as well as the structure. While the work presented herein is designed for the Linux repository, other repositories with nested merging and highly-consistent merging practices can take advantage of this work.

The remainder of this section provides a summary of the Linux kernel repository. The analysis of the repository involves all merges into the master branch between April 18, 2005 and August 14, 2014. This corresponds to the merges added to the kernel between versions 2.6.11 and Linux 3.16. This thesis does not attempt to analyze the commits to the Linux repository prior to the switch to Git in 2005. The commits collected from the repository include commits authored between September 17, 2001 and December 6, 2014. There are 4 commits in the dataset that are beyond this range due to the date being incorrectly set on that developer's machine. There is one incorrect date that is dated January 1, 1970, authored by Ursula Braun, and three commits dated after 2014 (these commits are dated April 5, 2019, October 14 2030, and April 25 2037, authored by Len Brown, Yanmin Zhang, and Daniel Vetter, respectively). Commits are not necessarily merged immediately after being created,

all commits were merged into the kernel between April 15, 2005 and October 14, 2014. This breakdown of the kernel data focuses on the commits integrated into kernel versions Linux 3.1 to Linux 3.16, translating to the merges between July 21, 2011 and August 3, 2014.

As expected, the Linux kernel is highly collaborative and is very active. Between 1000 and 1500 authors have contributions accepted into the official kernel per release (shown in Figure 2.21). These authors contribute between 8000 and 14000 commits per release (Figure 2.22). Between 275 and 400 merges integrate the commits into the master branch of the kernel per release (Figure 2.22). The Linux kernel repository is a prime example of a successful open source project, exemplifying the collaborative nature of modern software development. The sheer number of commits being contributed make the task of filtering the important or relevant commits difficult.

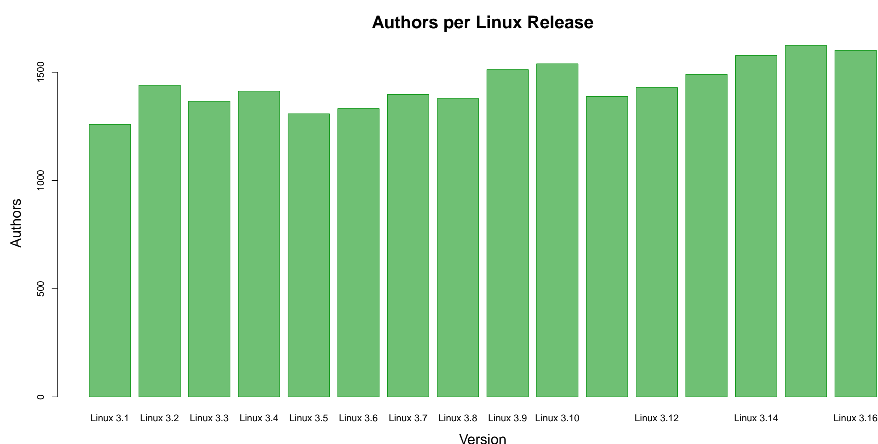


Figure 2.21: Unique authors with contributions to each kernel version

While the number of integrating merges into the master branch appears to be decreasing slightly per release, the number of commits per release is increasing. The average (mean) number of commits per merge per release has increased from slightly over 20 commits per merge into the master branch in Linux 3.1 up to 50 commits per merge in Linux 3.16 (Figure 2.24).

Grouping the commits by the merge that integrates the commit into the master branch and taking the median number of commits per merge shows a different view of the kernel repository. Each individual merge contains relatively few commits; 25% of the merges integrate only a single commit, and 50% of the merges merge at most 7 commits and merges (Figure 2.25).

The results of this digression show that breaking the commits based on the merge

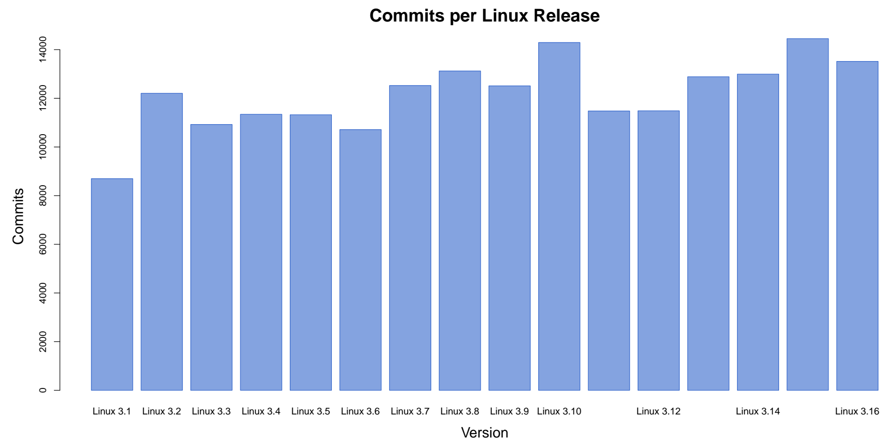


Figure 2.22: Commits per release from Linux 3.1 to Linux 3.16

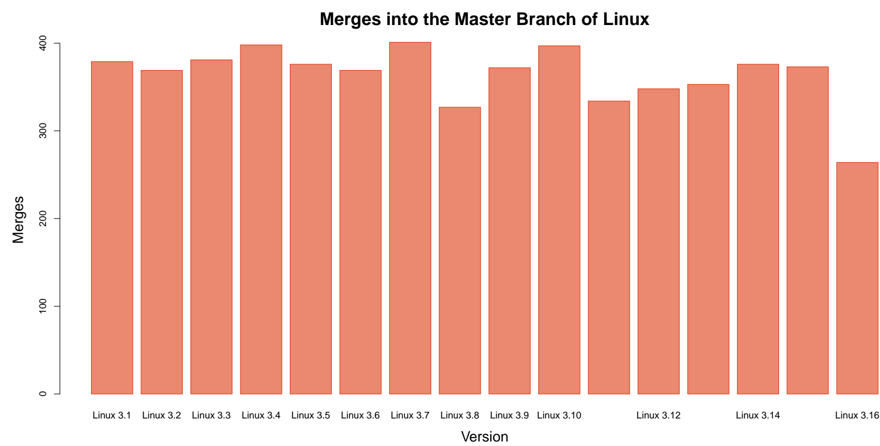


Figure 2.23: Merges per release from Linux 3.1 to Linux 3.16

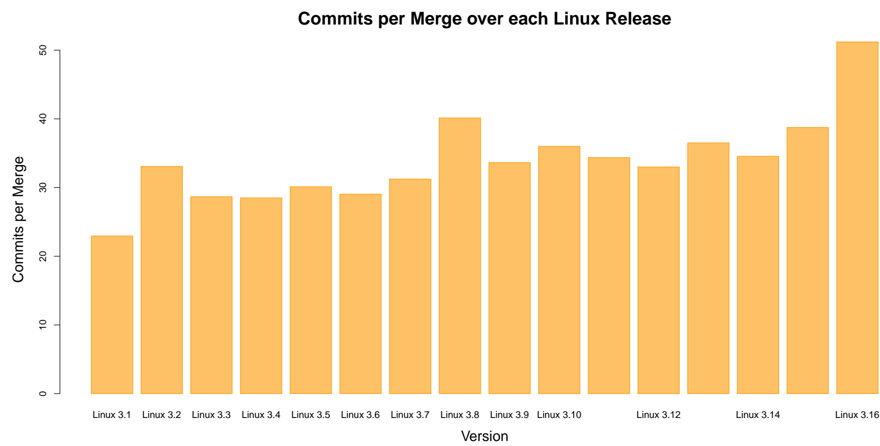


Figure 2.24: Commits per merge into each release of Linux from 3.1 to 3.16

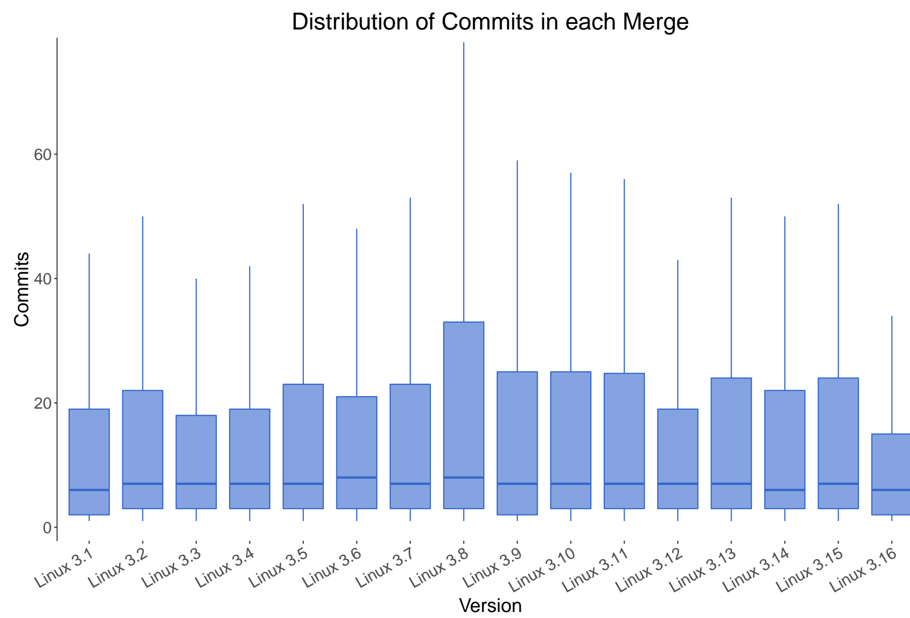


Figure 2.25: Distribution of Merge Sizes per Release Between Linux 3.1 and 3.16

should provide a means of creating useful visualizations. Seven commits is trivial to visualize, while attempting to visualize the entire graph is not.

Chapter 3

Merge-Tree Model

This chapter introduces the Merge-Tree model, the algorithm used to convert the DAG into a set of Merge-Trees, and an evaluation of the results produced by the algorithm. The model is designed to show how a commit is integrated into the Linux repository. In order to determine how a commit is integrated, it is necessary to identify the integrating merge into the master branch, the merges that the commit passes through on the way to the master branch, and the other commits that are merged with it. The model and algorithm take advantage of properties of Linux kernel repository that likely won't generalize beyond this repository. The algorithm will still run and produce results, but the results will likely not be meaningful.

Linus enforces a strict merging discipline in the Linux kernel repository. This ensures that the first parent of the merge commit is the head of the branch that is being merged into at the time the merge commit is created. That is, if a branch is being merged into the master branch, the first-parent of the merge commit will be the previous head of the master branch. If this property is broken, the results will not be meaningful as the trees will show how commits are integrated into a non-master branch. Repositories where a weaker merging discipline is used may include foxtrot merges, described in Section 2.3.

The Linux kernel project merges commits in logical groups, similar to the files in a directory structure. This results in many layers of merging, where each merge can be used as a means of filtering unrelated commits. Many repositories, including the OCaml and LLVM repositories, commit everything directly to the master branch, like how commits are made in SVN repositories. The algorithm will produce trees for these repositories, but the results will be a single tree for every commit, as they are all integrated directly into the master branch without passing through any merges.

The Linux repository is large; thousands of commits are merged into the Linux kernel repository per year. Directly visualizing all of the information in a meaningful way is difficult, or potentially impossible. Dividing the commits based on the merge into the master branch results in groups that are relatively small, the median size of the merges is seven items. This is visualized in Figure 2.25. A merge into the master branch is atomic, all of the context necessary for integrating a commit is available at the merge into the master branch. If a previous commit needs to be fixed after being integrated, the fix will be integrated in a future merge. If the issue is caught before the initial commit reaches the master branch, the fixing commit may be one of the commits that was integrated with the original commit. Showing only the commits that are merged into the master branch together filters the number of events down to a manageable size, while still containing all of the information necessary to determine how a commit is integrated, and other commits that are integrated with it.

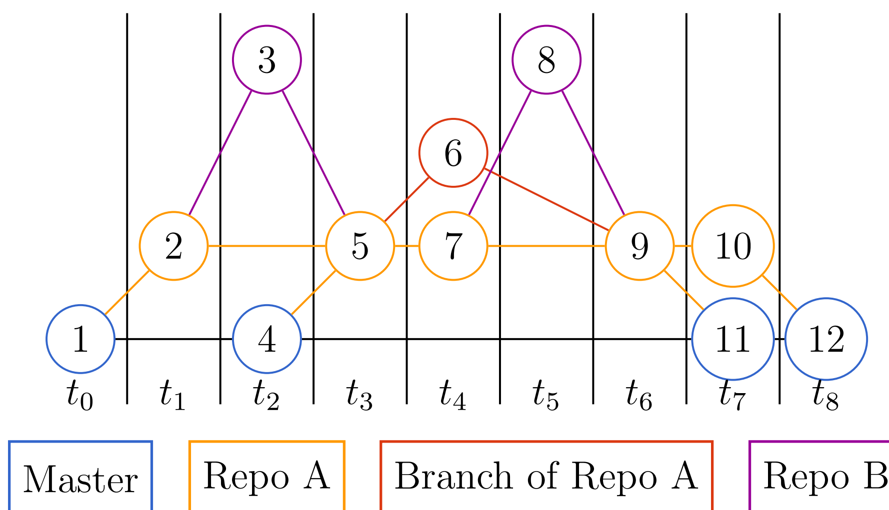


Figure 3.1: An example sequence of events performed in different repositories. The horizontal axis represents time. The branches and repositories are aligned horizontally, and color-coded. Each commit points to its parent. The initial commit is at time t_0, and the head is at t_8.

The Merge-Tree model is a tree structure, rooted at the merge into the master branch. The leaves of this tree are the commits and the merges are the inner nodes. The parent of a node is the next merge on the path to the root. Merge trees are constructed recursively. Starting at a commit, we walk up the children of the DAG until the first merge is found. All of the nodes that were traversed on the path to that merge are children of the merge in the Merge-Tree. Commits can be merged in

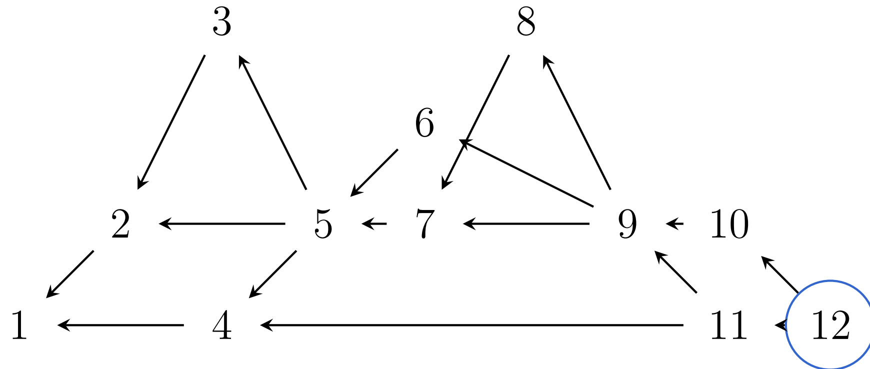


Figure 3.2: DAG representation of the commits represented in Figure 3.1. The DAG loses information about which repository the commit is performed in and through which merges it has passed on its way to the master branch. The DAG does not even distinguish the master branch from other branches.

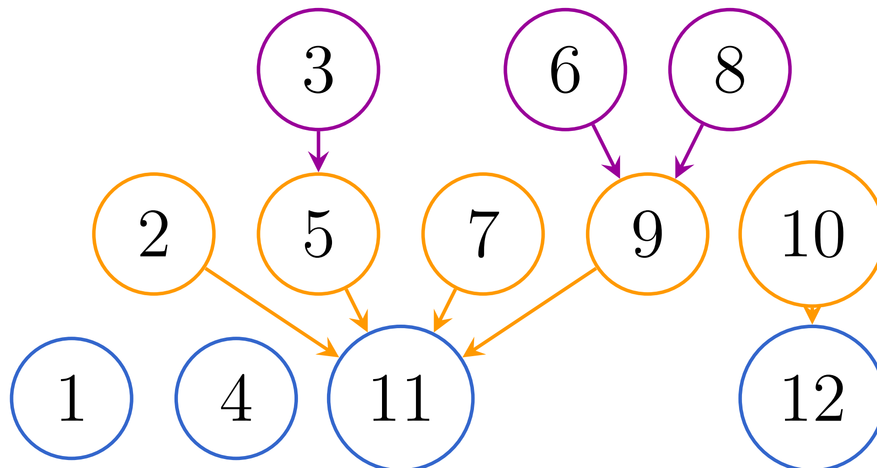


Figure 3.3: The Merge-Trees computed for each commit in Figure 3.2 showing the path that each commit takes to be merged into the master branch of the repository. This does not indicate how the events being merged are related. This figure retains the numerical order of the events, but the order is arbitrary.

multiple places. The true parent of a commit in the tree is defined by the shortest path. In the case that there are the same number of intermediate nodes between the commit and the root, shortest distance in time between when the commit was created and merges is used as a tie-breaker. The structure recursively groups commits that are merged together, making it easy to identify the commits that are integrated together. Commits have a single path to reach the master branch, which makes identifying the series of merges to the master branch and, in extension, how the commit is integrated into the master branch of the master repository, much easier.

The structure makes it easier to understand how commits are grouped and integrated. This model simplifies and prunes the information from the DAG, it also inverts the parent-child relationship. The parent of a node in the DAG is the child of that node in the Merge-Tree.

In addition to identifying the path that a commit took to being merged, it is possible to aggregate commit metadata at merges. Merges in the Merge-Tree are aware of their children, which is information that is not available in the DAG. Recursively traversing each child and aggregating the metadata from each commit produces an aggregated summary of the merge. The merges in the DAG are not aware of their children.

To illustrate this model I will use a small example: assume the commits represented in Figure 3.1 show the sequence of events in a repository. The sequence starts with the initial commit in the master branch of the master repository at time t_0. Repository event 1 is a commit, which gets forked into a separate repository, *Repo A*, where another commit is made, event 2. Event 5 is a merge event, merging events 2, 3, and 4 into *Repo A*. A branch created from event 5 and commit 6 happens in the new branch, while commit 7 is added simultaneously to the original branch in *Repo A*. Events 11 and 12 are both merge events, merging changes made in *Repo A* into the master branch of the master repository. As every repository is a first-class repository, including local copies and forks, git does not distinguish between forked repositories and branches, and in neither case does it explicitly record where a commit was made. In this case, commits are performed in various repositories and branches. The DAG representation of these events is shown in Figure 3.2.

The commit nodes do not preserve branch information, which allows users to rename branches and repositories without having to update the preceding commits. This is at the expense of maintaining a consistent history. It is desirable to reconstruct all of the branch and repository information, shown in Figure 3.1, from the

information in the DAG, shown in Figure 3.2, but this may not be possible. Git does not retain information the branch that a commit comes from, relying entirely on the order of the parent list. A foxtrot will confound this list, making it impossible to correctly re-generate the series of events to produce the repository. Instead, the focus is placed on finding the next merge that leads toward the integration of a commit. Depicted in Figure 3.3, is the first version of the Merge-Tree. It does not completely rebuild the lost information, but is able to show the sequence of merges that a commit follows to be integrated, and the commits that were involved with the integration. This is the version of the Merge-Tree that is used in the visualizations, in the construction of *L*invis, and in the user study. This tree does not preserve the ordering of the commits within the merge, and while the algorithm presented in Section 3.1 preserves this information, the visualization in *L*invis does not make use of it.

Using the depth of the node from the root of the tree, the branch information is reconstructed. In our events, nodes 2, 5, 7, and 9, are all on the same branch, and are merged into node 11. Nodes 5 and 9 are merge nodes, 5 merges a single commit into the branch, and 9 merges two nodes into the branch. The traversal may not find first integrating merge for a given commit. Node 9 is merged into 11, though the traversal through 9 will eventually reach node 12 as well. There is one merge to integrate 9 into either node 11 or node 12, so the shortest distance through time is used to break the ambiguity, selecting node 11.

3.1 Algorithm

Computing the Merge-Tree from a DAG for any repository may not be possible; however, certain features of the development process of Linux make it feasible to compute the Merge-Tree for the Linux repository. Linus Torvalds is the only one with merge access to the master branch of the Linux kernel repository, verified by German [13]. Linus enforces a strict merging discipline, which limits the number of foxtrot merges entering the master branch. By keeping a clean master branch, the Merge-Trees are rooted correctly in the master branch. The heuristic for determining which commits are along the master branch relies on this property being true.

In short, the algorithm first identifies the commits made directly to the master branch, where after it recursively determines the shortest path, using the DAG, from each commit to the master branch using the inverted DAG.

The algorithm has two phases. The first phase identifies the commits along the

Algorithm 1 Computing the Merge-Tree of Linux from the DAG

```
 1: function ComputeMergeTree(DAG): tree
 2:     head ← Head of master of git repository
 3:     master ← traverse DAG from head using
 4:             first parent until reaching root
 5:     nodes(Tree) ← nodes(DAG)
 6:
 7:     function MergeAtMaster(cid)
 8:         # Returns (depth, merge, next)
 9:         # Helper function
10:         # Compute the closest merge into master,
11:         # setting the children on the way to master.
12:         if cid in master then
13:             return (0, cid, ∅)
14:         end if
15:         d ← ∞
16:         # Traverse the inverted DAG
17:         for c ∈ children(cid, DAG) do
18:             (d_c, merge_c, next_c) ← MergeAtMaster(c)
19:             if IsMerge(c) then
20:                 fp ← FindFirstParent(c)
21:                 if fp ≠ cid then
22:                     d_c ← d_c + 1
23:                     next_c ← c
24:                 end if
25:             end if
26:             # Find the shortest path
27:             if d_c < d then
28:                 (d, m, next) ← (d_c, merge_c, next_c)
29:             else if  d_c = d  then
30:                 # Use the time as a tie-breaker
31:                 if cTime(merge_c) < cTime(m) then
32:                     (m, next) ← (merge_c, next_c)
33:                 end if
34:             end if
35:         end for
36:         # c is the commit that follows cid on it's way to master
37:         add edge (cid, next) to Tree
38:         return (d, m, next)
39:     end function
40:     # Compute the distance for each commit discarding result
41:     for c ∈ nodes(DAG) do
42:         MergeAtMaster(c)
43:     end for
44:     return Tree
45: end function
```

master branch. This is done by traversing the first parent from the master branch reference to the commit that has no parents. Multiple initial commits will not have an effect on this phase. The initial commit that is along the first-parent traversal of the master branch will be the one selected.

The second phase is encompassed by the function $MergeAtMaster$ which determines, for each commit, which merge the commit is merged at, the depth (as variable d in the algorithm), and the next merge on the path to the master branch. The function $MergeAtMaster$ has two parts, the first for determining the depth, from the master branch, that the repository event is at. The second phase determines the merge into the master branch, and the next merge on the way to the master branch. The distance is computed by shortest path close to the master branch as possible. If there is a tie between two paths, the path that merges into the master branch sooner is taken.

An example is used to demonstrate the behaviour of the algorithm, computing the merge at commit 5 in Figure 3.1. $MergeAtMaster$, recurses along the children of the nodes it visits. Eventually every child of every node along the path will be visited at least once. Without loss of generality, suppose that the path recursed along is from node 5 to 6, 9, 10, and finally 12.

The depth for each, except 12 (a merge into the master branch), is initialized to infinity, the merge into master is blank, and the next merge is blank. Merges into master trivially have a distance of 0 from the master branch, and it merges itself into the master branch. The recursion at 12 returns the triple $(0, \varnothing, 12)$ to the call from 10. 12 is a merge commit and 10 is not the first parent, so the temporary depth, d_c, is incremented to 1 and the temporary next merge, $next_c$, is changed to 12. 1 is less than infinity, so the depth is set to 1, the merge to 12, and the next to 12. This returns the triple $(1, 12, 12)$ to the call from 9. 9 is the first parent of 10, so no changes are made to the temporary variables.

The call to 9 recurses to the second child, 11. 11 is a merge into the master so it returns $(0, \varnothing, 11)$ to the call from 9. 9 is not the first parent of 11, so the d_c is incremented to 1 and $next_c$ is changed to 11. The distances d_c and d are the same, so time is used to break the tie. 12 was merged after 11, so 11 replaces 12 as the merge into the master branch for 9, as well as being the next merge. The call for 9 returns the triple $(1, 11, 11)$ to the call for 6. 6 is not the first parent of 9, so d_c is incremented and $next_c$ is changed to 9, as 9 merges 6. 2 is less than infinity, so the d is changed to 2, the merge to 11, and the next merge to 9. The call to 6 returns the

triple $(2, 11, 9)$ to the call for 5.

The call for 5 recurses on the second child of 5, calling on 7, which calls 8, and then 9. 9 can continue, but if the implementation of the algorithm uses memoization, the call to 9 can immediately return the triple $(1, 11, 11)$ to the call for 8, and avoid an exponential runtime. 8 is not the first parent of 8, so d_c is incremented to 2 and $next_c$ is changed to 9. 2 is less than infinity, so d is changed to 2, $merge$ to 11, and $next$ to 9. The call to 8 returns the triple $(2, 11, 9)$ to the call for 7, which recurses on the second child of 7, 9. 9 returns the triple $(1, 11, 11)$. 7 is the first parent of 9, so the depth is not incremented. d_c is less than d, so d is changed to 1, m to 11, and $next$ to 11, returning $(1, 11, 11)$ to the call for 5. d_c is less than d, so d is changed to 1, m to 11, and $next$ to 11. There are no other children, so the function halts.

3.2 Algorithm Evaluation

The merge trees generated by the algorithm must be validated to ensure that they are an accurate representation of the events occurring in the repository. Evaluation poses some issues, as there is no easy way to accurately gather this information directly from the DAG. Further inspection of the merges provides some insight; Linus Torvalds adds useful information about the content of the merges. For each branch being merged, Linus includes either the first 20 commit titles and the total number of commits being merged (see Figure 3.4 for an example), or if there are 20 or fewer commits being merged, the list of commit titles.

This information is used to verify that the results of the Merge-Tree algorithm are consistent with the information explicitly stated in the merge log. A program extracts the merges made by Linus into the master branch of the repository. The merges are gathered by two commands;
`git log --merges --author='Linus Torvalds' v3.16` collects the merges by Linus Torvalds, and `git log --format="%H" --first-parent --merges v3.16` collects the merges that are along the master branch, assuming that the master branch is not confounded. The intersection of the results of the two commands leaves the set of merges made by Linus Torvalds into the master branch. The results of the algorithm were collected up to version 3.16, which is why the set of merges must be limited to that version. For each merge in the intersection, the program extracts the number of commits, commit titles, commit date, and authorship date from the merge log and compares them against the corresponding generated Merge-Tree.

```
commit 3d30701b58970425e1d45994d6cb82f828924fdd
Merge: 8cbd84f2dd4e fd8aa2c1811b
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Tue Aug 10 15:38:19 2010 -0700

    Merge branch 'for-linus' of git://neil.brown.name/md

    * 'for-linus' of git://neil.brown.name/md: (24 commits)
      md: clean up do_md_stop
      md: fix another deadlock with removing sysfs attributes.
      md: move revalidate_disk() back outside open_mutex
      md/raid10: fix deadlock with unaligned read during resync
      md/bitmap:  separate out loading a bitmap from ...
      md/bitmap: prepare for storing write-intent-bitmap ...
      md/bitmap: optimise scanning of empty bitmaps.
      md/bitmap: clean up plugging calls.
      md/bitmap: reduce dependence on sysfs.
      md/bitmap: white space clean up and similar.
      md/raid5: export raid5 unplugging interface.
      md/plug: optionally use plugger to unplug an array ...
      md/raid5: add simple plugging infrastructure.
      md/raid5: export is_congested test
      raid5: Don't set read-ahead when there is no queue
      md: add support for raising dm events.
      md: export various start/stop interfaces
      md: split out md_rdev_init
      md: be more careful setting MD_CHANGE_CLEAN
      md/raid5: ensure we create a unique name for ...
      ...
```

Figure 3.4: Example of how merges record a subset of commits being merged. The commit only shows the first 20 one-line summaries messages for the 24 non-merge commits it merged. The ending "..." is part of the log and represents that other commits were merged.

The results of the analysis found that 14670 merges along the master branch were found in both the results of the Merge-Tree algorithm, and the git logs. Two merges, *186051d70444* and *5170a3b24a91* were found in the logs, but were not in the results of the algorithm, and 426 merges in the set of Merge-Trees, but not in the logs.

The two merges in the logs, but not in the Merge-Tree were likely due to a fast-forward merge on the master branch. As git does not maintain the information necessary for distinguishing commits added by a fast-forward merge from commits committed directly to the branch, the only way to verify is to ask the person making the merge, and given that these merges were made 13 years ago, the truth has likely been lost to time. The graph for merge *186051d70444* is show in Figure 3.5. Each commit in the merge will produce a separate Merge-Tree.

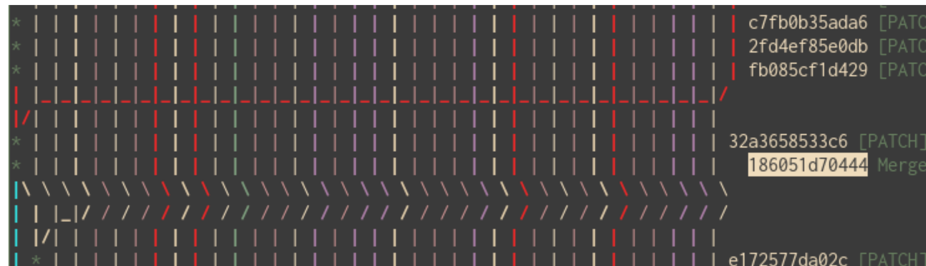


Figure 3.5: Merge *186051d70444* graph view, showing that the merge into master is immediately followed(above) by a non-merging commit.

Conversely, many of the merges and commits that were in our database but not found in the logs were due to the *foxtrot* merges. The commits were detected as being part of the master branch when inserting the commits into the database, but were not collected by our git log query since they were not authored by Linus Torvalds.

Beyond investigating why merges show up in either the database or logs, further analysis requires that the merges are present in both the database and the merge log. There are 14198 merges with these properties.

The results of the evaluation are summarized as follows:

- Five merges did not have matching commit counts between the database and the logs. Upon further investigation, four merges had incorrectly formatted logs. The fifth merge, *42a579a0f960*, is a *foxtrot* merge. One commit is on the first-parent of the merge, and is therefore not detected when building the Merge-Tree, but is included in the merge log.

- The heuristic worked correctly until September 4, 2007, the earliest date that

could be verified. Before this date, merge logs did not include a summary of the commits being merged, making it impossible to verify. Manual inspection indicates that the heuristic worked correctly for these commits, until December 12, 2006 where a *foxtrot* merge occurs.

- There is one merge after September 4, 2007 that does not have recorded commit logs. This is due to incorrect formatting. If it were correctly formatted, it would report having 15 commits integrated, which is consistent with the results in the database.

- The algorithm breaks on merges prior to December 12, 2006 due to a *foxtrot*. There were 1537 merges made by Linus prior to this date, according to the commit metadata. While we do not know exactly which commits, or how many were being integrated at a given merge, the commit retains metadata about who wrote it. In this case, these commits were not authored by Linus, who is the only one who has merge access to the master branch.

- 77 Merges were made by Linus into non-master branches after September 4, 2007. These merges were made into *3f17ea6dea8b*, which is exceptionally large containing 7217 repository events, 6809 of which are commits, shown in Figure 3.6.

There were 12837 merges after September 4, 2007. With the exception of the five merges, four with errors, and one as part of a *foxtrot*, all merges were correctly identified. The 835 merges between December 12, 2006, and September 4, 2007, appear to be correct, but cannot easily be verified. The algorithm breaks on 1537 merges prior to December 12, 2006.

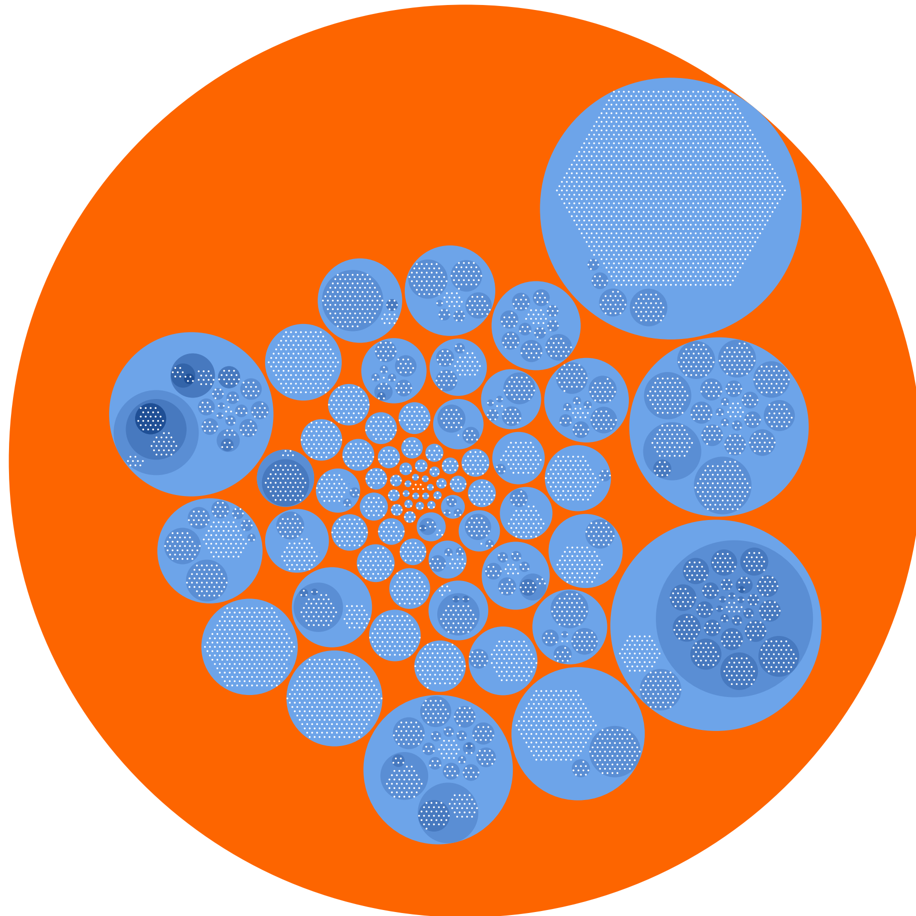


Figure 3.6: The largest Merge-Tree, made by Linus Torvalds into Linux 3.16. This visualization clusters nodes based on the parents. The visualization is explained in more detail in Section 4.3.3.

Chapter 4

Design and Implementation

A model is difficult to assess directly, instead it is possible to convert the model into visualizations, which can be assessed as a proxy to the effectiveness of the model. This chapter presents various visualizations that take advantage of the Merge-Tree model, and a tool, called *L*invis, that uses these visualizations.

*L*invis is designed with two uses-cases in mind, though a user may freely switch between the cases as they work. Both use-cases are designed with maintainers in mind.

Use-Case 1: top-to-bottom approach
These users are maintaining a portion of the kernel and would like to pick an entire merge, including all commits being merged, to inspect and potentially merge it directly into their version of the kernel.
These users do not have a specific commit in mind.

Use-Case 2: bottom-to-top approach
These are maintainers that start with a given commit and would like to understand what other changes are being made to integrate this commit. This is done by understanding the merges that the commit passes through toward integration, and finding the commits that are necessary for the integration of a given commit.
These users do have a specific commit in mind.

*L*invis uses full-text search to find the merges and commits that the user is interested in, and provides two summarization views, and three visualizations of the Merge-Tree for the repository events.

4.1 Search

*L*invis provides a search engine for navigating within the kernel repository. The search engine finds the merges and commits that are relevant for a given query. The results are then sorted based on the relevancy to the query. Relevancy is computed by weighted similarity, taking into account the contents of the commit log, the author name, the filenames, the commit hash, the date the commit was authored, and the date the commit was committed. More information about the design of the search engine are available in Section 5.2.1.

Before presenting the results, the commits and merges are grouped by Merge-Tree root. Each group of events has the link to the root at the top, followed by a table of the relevant commits and merges from that tree, shown in Figure 4.1. The table of results includes the relevancy rank assigned by the search engine, commit preview, author, commit date, and the authored date. The Merge-Trees are ordered by the mean of the relevancy scores.

4.2 Summarization

*L*invis uses seven tabs to present the information and visualizations for a selected repository event. The informative tabs are: messages, files, modules, and authors. The visualization tabs are: list tree, pack tree, and Reingold-Tilford tree.

The message tab shows the full commit log message. This does not include the patch, but given a commit hash, the patch can be found directly from the repository with the `git show` command, or using other tools.

The files tab shows the list of files that were modified in a merge or commit. The table includes the number of lines added, lines removed, total lines modified, and the difference between the number of lines added and removed. Each metric is aggregated across each commit within the merge. A details drop-down button allows a user to see exactly which commits make the changes, as shown in Figure 4.2.

The modules tab shows the modules modified in the Merge-Tree. Like the files tab, the modules tab uses a table to show the name of the module, the number of

Merge git://git.kernel.org/pub/scm/linux/kernel/git/davem/net — Linus Torvalds — Search:

Rank	Preview	Author	Commit Date	Authored Date
2.34026	net-next:asix: V2 Update VERSION	Grant Grundler	11/15/2011	11/15/2011
2.28766	net-next:asix:PHY_MODE_RTL8211CL should be 0xC	Grant Grundler	11/15/2011	11/15/2011
2.27013	net-next:asix: V2 more fixes for ax88178 phy init sequence	Grant Grundler	11/15/2011	11/15/2011
2.27013	net-next:asix: reduce AX88772 init time by about 2 seconds	Grant Grundler	11/15/2011	11/15/2011
2.25844	net-next:asix:poll in asix_get_phyid in case phy not ready	Grant Grundler	11/15/2011	11/15/2011

Previous 1 Next

Merge git://git.kernel.org/pub/scm/linux/kernel/git/davem/net-2.6 — Linus Torvalds — Search:

Rank	Preview	Author	Commit Date	Authored Date
2.27013	net-next-2.6 [PATCH 1/1] dccp: ccids whitespace-cleanup / CodingStyle	Gerrit Renker	09/14/2009	09/12/2009

Previous 1 Next

Figure 4.1: Two Merge-Trees returned from the query for "net-next". The top search result contains multiple entries with the search term in the title, whilst the second result contains a single entry with the search term in the title. The groups provide a link to the root at the top, and the relevant commits in the table below.

	File	Lines Added	Lines Removed	Total	Delta
>	sound/soc/soc-dapm.c	1	1	2	0
∨	sound/soc/at32/playpaq_wm8510.c	3	9	12	-6

Show 10 entries — Search:

Commit	Lines Added	Lines Removed	Total	Delta
cdbdd1676a	3	9	12	-6

Showing 1 to 1 of 1 entries — Previous 1 Next

	File	Lines Added	Lines Removed	Total	Delta
>	sound/pci/hda/patch_nvhdmi.c	1	0	1	1

Figure 4.2: Table showing the modified files in a merge, with the second entry expanded to show the commit that makes the changes.

commits that are in the Merge-Tree that work with the module, and a details button to provide the links to those commits, shown in Figure 4.3. Modules are not an inherent part of git, but are a property of most commit log previews in the Linux repository. The text up to the first colon in the commit log preview tends to indicate the subsystem of the kernel that is being modified. In Figure 3.4, the first commit is from the "md" subsystem, which virtualizes multiple physical devices into a single virtual device.

Show 10 entries Search:

	Module	Count
⌄	ALSA	5

- cdbdd1676a
- 7c2dfee848
- 1c85cc6445
- ec4e86ba06
- 9a3f371e99

Showing 1 to 1 of 1 entries Previous 1 Next

Figure 4.3: Table showing the modules involved in a merge, listing the commits that modify this module.

The authorship tab is similar to the files tab, but shows the authorship information. It shows the sum of the lines added, removed, modified, and the delta within the Merge-Tree. It also shows the number of files that were modified by the author. The details are organized slightly differently than in the files tab. Instead of showing the commits that make the modifications, *L*invis shows each file that was modified by this person, as well as the commit where the changes took place. As multiple files can be modified in the same commit, commit hashes are not unique in this table. The authors tab is shown in Figure 4.4.

4.3 Visualization

*L*invis provides access to three visualizations, each with a specific intention.

	Takashi Iwai	1	1	0	1	1
	Randy Dunlap	2	32	40	72	-8

Show 10 entries Search:

Commit	Filename	Lines Added	Lines Removed	Total	Delta
1c85cc6445	sound/core/pcm_native.c	8	16	24	-8
1c85cc6445	sound/core/pcm_lib.c	24	24	48	0

Showing 1 to 2 of 2 entries Previous 1 Next

Figure 4.4: Table showing the authors who made changes in a merge. The entry for Randy Dunlap is expanded, showing the files that Randy modified in this merge.

8

Merge branch 'for-linus' of git://git.kernel.org/pub/scm

Message Files Modules Authorship List Tree Pack Tree Reingold Tilford Tree

- Merge branch 'for-linus' of git://git.kernel.org/pub/scm/linux/kernel/git/tiwai/sound-2.6
  - Merge branch 'topic/asoc' into for-linus
    - ALSA: Fix debugfs_create_dir's error checking method for sound/soc/
    - ALSA: ASoC: Convert playpaq_wm8510 to bulk route registration API
  - Merge branches 'topic/asoc', 'topic/hda' and 'topic/misc-fixes' into for-linus
    - ALSA: Handle NULL jacks in snd_jack_report()
    - ALSA: kernel docs: fix sound/core/ kernel-doc
    - ALSA: hda - Fix PCM type of Nvidia HDMI devices

Figure 4.5: The List Tree Visualization

4.3.1 List Tree

The list tree, depicted in Figure 4.5, is constructed from nested lists. The commit or merge being inspected is shown first, and each child of the subtree is indented recursively. If the node is a commit, only the node will be shown.

This visualization is text-based, making searching simple through the browser text search functionality. This visualization works well with large and small trees, though it does not give an immediate impression of the structure of the entire merge. The visualization is designed to show the path that a specific commit takes to reach the master branch.

*L*invis also provides breadcrumbs showing the path that the current repository event took to being merged into the root of the Merge-Tree to ensure that a user can navigate both toward the root and toward the leaves using this visualization. In Figure 4.5, the merge being inspected is the root node, so the breadcrumbs only contain a single link, which is the small "8" in the top left corner of the image. The items in the breadcrumbs are the shortest unique prefix of the commit hash in the Merge-Tree. In small trees, this will usually be a single character.

4.3.2 Reingold-Tilford Tree

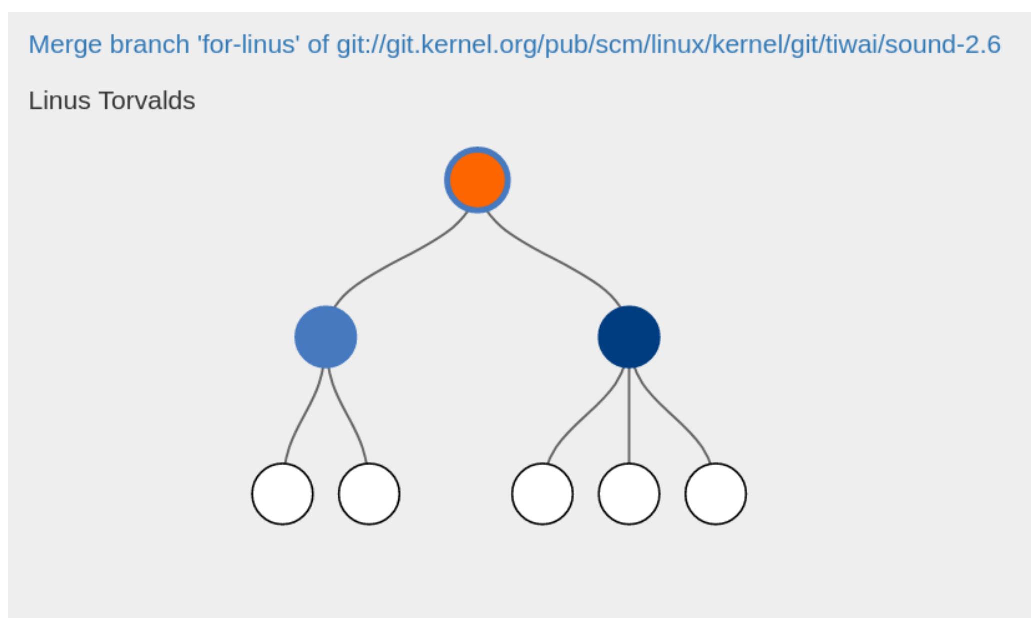


Figure 4.6: The Reingold-Tilford tree visualization with the root currently selected. The root is at the top, the leaves are depicted as the white circles with no children.

The Reingold-Tilford tree [22] visualization, depicted in Figure 4.6, is the classic tree visualization. The root is at the top, leaves are at the bottom, and the edges between the nodes showing the parent-child relationship.

The Reingold-Tilford tree was designed with the goal of producing trees that looked tidy. Proposed by Wetherell and Shannon[25], there were three aesthetic requirements behind the design of tidy trees:

- Nodes at the same depth in the tree must be aligned, and each level should be parallel.
- The left and right children must be positioned to the left and right of their parent.
- The parent should be centered above the two children.

These requirements are necessary, but not sufficient for producing visually appealing trees. Reingold and Tilford propose an additional aesthetic requirement; subtrees should be drawn identically regardless of position in the tree, and that the reflection of the tree should produce a mirror-image of the original tree. The entire goal is to produce trees that are aesthetically pleasing and easily understandable. This visualization works well with trees that are not exceptionally wide and can give a good impression of how commits are integrated in smaller trees.

In our visual metaphor, the white nodes indicate a commit, while the colored nodes indicate merges. The merges are colored in shades of blue to indicate the number of children of that node. Darker shades of blue indicate more nodes, while lighter shades indicate fewer nodes. The repository event that is currently being inspected will be filled with pumpkin orange, but the outline will remain the original color. Initially, the node representing the current repository event will be centered in the window, and the title and author of the node are presented above the visualization. Clicking on a node will center that node and update title and author. The title is hyper-linked to the page for that commit, clicking on the title will navigate to the node that was selected. Clicking and dragging will move the tree in the view, scrolling will zoom the tree.

4.3.3 Pack Tree

The Pack Tree[24] is used for quickly displaying an overview of large hierarchical data. The original use-case was the file system, where there are usually many files,

but few levels of directories, creating a very wide but shallow tree. This metaphor is directly comparable with the structure of a git repository. The commits are files, they contain the information, rather than being structural. The merges map to the directories, structurally grouping the commits in a logical fashion.

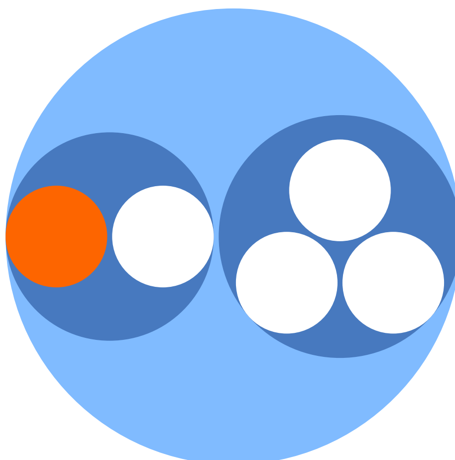


Figure 4.7: The pack tree visualization; the root depicted as the outer-most circle, containing all other nodes, the leaves depicted as white circles containing no other nodes. The currently selected node shown in pumpkin orange.

Our goal in providing this visualization is to quickly show the topology of the merge. With the Reingold-Tilford tree, it can be difficult to understand just how big the large Merge-Tree are, as they don't fit on the screen. The Pack Tree uses a set theory-like model for describing the hierarchical structure. Mapping the pack tree metaphor with the tree, leaves are the smallest nodes, containing no other nodes. Inner nodes contain other nodes within them and the root is the outermost circle, containing all other nodes within it.

The bubble tree[3] was seen as a possible visualization for the trees, as it uses a similar metaphor but due to limitations with the bubble tree, the pack tree seemed more suitable with our goals. The initial bubble tree visualization starts with a single circle, the root. Hovering over that circle shows the contents of the node. This works recursively, hovering the cursor over an inner node will reveal the contents of that node. While this is designed to prevent overwhelming users, the user cannot quickly determine what the tree looks like without hovering their cursor over each inner node. Due to this limitation, the pack tree was a better option.

An example of the pack tree visualization used in *L*invis is depicted in Figure 4.7. The outer-most circle that contains the other circles is the root node. Nodes that are

not leaves are colored in a shade of blue. Darker blue indicates that the node is deeper in the tree; the root is shaded with the lightest blue. The commits are colored white. The node that represents the repository event that is currently being inspected will be shown in pumpkin orange.

Chapter 5

Implementation Details

The implementation is broken into two parts, the *L*invis[1] and the database. The database uses PostgreSQL, an advanced opensource database management system. The tool backend uses Python and the Flask framework. Both the database and the *L*invis backend are containerized using Docker. This facilitates easy migration between the development environment and the server where the system is running. Both components are running Alpine 3.5, a simple, light, and secure distribution of Linux designed to run as the base of Docker containers.

This chapter contains details on the design of the database, the extraction of the data, and the implementation of the tool.

5.1 Composition

The database, discussed in Section 5.2, and website, discussed in Section 5.3, run in different containers, and must be linked together in order to communicate with each other. This section describes the architecture of the system, depicted in Figure 5.1, describing how the various components work together to produce the page.

The built-in Flask HTTP server is not designed for a production environment. It is not designed with performance or security in mind, it is designed for debugging. Requests must be passed through an actual web server. I chose Nginx to be the outward-facing server, it is modern, and designed with speed and security in mind. Nginx receives HTTP requests from a user over the internet. It converts the request from HTTP to WSGI, a high-performance binary protocol for python applications,

[1] *L*invis is available at `http://li.turingmachine.org`

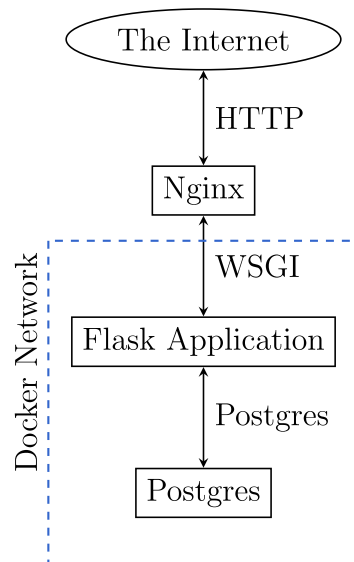


Figure 5.1: Webserver Architecture, showing the protocol used for communication between the modules.

and then forwards the WSGI request to the Flask application.

The Flask application runs in a Docker container, so Nginx cannot communicate directly with the application. Nginx sends requests to the Docker network, which applies the local equivalent of port-forwarding to pass the requests to the appropriate container. Docker is specifically designed for working with scalable systems, so as long as Nginx can connect to the Docker network, requests are passed to the container, regardless of where the container is running. In our case, everything is running on one host, but the architecture would not need to change if this were to change.

The Flask application and database are in separate containers. Since both are in the same Docker network, they are discoverable to each other using the container name as the host name. Again, the containers could be running on separate physical hosts, but the Docker overlay network would hide this fact, and the system architecture would not have to change. Requests are passed between the application and database using the PostgreSQL protocol.

Once the Flask application has generated a response, it is sent back to Nginx. Nginx generates the appropriate response header, setting the cache and MIME information, as well as caching the response, before sending the response back to the user.

5.2 Database

The database is made of seven tables:

- Baseline contains the commits that have been integrated into the master branch of the repository.

- Commits contains the metadata for each commit, including the author, committer, the date the commit was authored, and the date the commit was committed, as well as the associated patch.

- Filesmod contains the information regarding which files were changed, and how many lines were added and removed from each.

- Logs contains the subject and full log message for every commit.

- Releases contains information regarding which commits represent the split between versions of the kernel. It contains the version name, the commit, the

previous version, the commit for the previous version, the previous candidate and commit, and whether the version is a candidate.

- Search contains information necessary for the search engine. While this table is not necessary for the operation of the tool, it is used as a cache and index, which improves the performance of the search engine greatly. It contains the search term vector and the associated commit hash.

- PathToMerge contains the actual Merge-Tree structures, with references pointing from a given commit to the next merge on the way to integration.

5.2.1 Full Text Search

The search engine uses the full text search built into PostgreSQL to enable easy searching for commits in the database.

The engine uses pre-computed text vectors instead of re-computing them for each query. The search vectors are pre-computed and stored in the search table, creating a map between search terms and commit hash. The vectors include terms from the commit hash, log subject, full log message, commit author name, commit and authorship date, and list of files that were modified in the commit.

The search attributes are given different weight when calculating the search rank. The attribute weights are listed in Table 5.1. The rank calculation does not normalize against document length. The ranking system is designed to associate a higher weight with attributes of more importance. One drawback of using full text search is that terms must be identical to the term used in the attribute. This has two negative effects, the first is that typos will drop commits that are otherwise relevant, and second, the entire term must be entered. Trigraphs are a possibility for working around this. Trigraphs break the words in the attribute into groups of three. Using this on the commit hash, for example, would allow a user to perform a substring search, typing only a portion of the commit hash. Unfortunately, the Trigraphs are slower to compute, to search, and consume more memory, and therefore were infeasible for this project.

Postgres applies stop-word elimination when constructing the search vectors; stop words are terms that are common in a given language and won't help when discriminating items. These terms include words like “and”, “the”, and “a”. The default English stop-words built into Postgres version 9.6.2 are used to clean the queries.

Table 5.1: Search Attribute Ranking

Attribute	Weight
Commit Hash	A
Log Subject	A
Log Full Text	C
Author Name	A
Commit Date	B
Authorship Date	B
File names	B

5.3 Web Interface

The tool itself is implemented as a web-interface to enable people to use it without needing to install dependencies or download external files. As the compressed database image sits at 976MB, it would not be useful for many to download it.

The application is written with asynchronous processing in mind. The pages do not contain information that is specific to a commit or query. The pages are a template that is later filled with commit-specific information with Javascript. Using the asynchronous architecture has advantages and disadvantages. Since the main part of the page is the same for every commit, it can be cached. The requests are processed asynchronously. When a user clicks on a page, it takes nearly no time to get most of the page showing, since it should be cached at all hops on the way back to the server. The primary delay comes from the delay caused by the round-trip to the server to gather the commit-specific data.

The issue is that many requests must be passed between the client and the server. When the client requests the page, instead of getting a single response, the client will wait for the response, then send a request for the commit-specific information. There are different requests involved for collecting the tree, files, authorship, modules, and message information.

The backend itself is written in Python 3.5, and uses the Flask framework to handle routing requests and manage cookies. While flask contains a simple built-in HTTP server, I opted to use the uWSGI server, which is more secure and runs faster, supporting multiple worker threads. The frontend is written in Javascript, HTML, and CSS. Bootstrap is used to produce the clean interface, and D3 generates the tree visualizations.

Chapter 6

Evaluation

In June 2017, I conducted a study to quantitatively and qualitatively evaluate the effectiveness of the visualizations and summaries presented in *L*invis to the DAG-based visualizations found in Gitk and the git command-line. This chapter presents the methodology, a profile of the participants, and the results of the study. The study was performed in a controlled environment running Ubuntu 14.04. Participants were allowed to use Gitk and the git command-line tools for these tasks, I will refer to both tools as Gitk, when working with DAG-based visualizations and summarizations. I considered allowing participants to use any of the free tools for Linux suggested on the git website[1], but after attempting to use them, found that none of the tools listed at the time were able to operate on repositories that were as large as the Linux repository. *L*invis was used for evaluating the Merge-Tree-based visualizations.

The study has two primary goals: first determining if the DAG-based visualization is sufficient for conceptual understanding; second, comparing *L*invis and Gitk to determine which is more capable of providing users with a summarization of various metrics involved with integrating a commit into the repository. These were done in two parts of the same study. They were performed together as a single study for pragmatic reasons, but could have been done as separate studies.

6.1 Methodology

This section describes the evaluation, how it was performed, the methods used to ensure that things were kept consistent between participants, and setting of the study.

[1] `https://git-scm.com/downloads/guis`

The evaluation was performed as a single study, broken into two parts, using the same 12 participants and same 2 commits. The first part of the evaluation only used Gitk, while the second part used Gitk and *L*invis. The order that the participants studied each commit was randomized for each participant, but was kept consistent through each part of the study. Participants would complete part one of the study on both commits, the continue with part two, before answering a few questions about their opinions and experience in an exit interview. A screen-capture with audio was taken for the duration of the study with each participant.

Three metrics are recorded from each task: correctness, accuracy, and the time taken to respond. Correctness is simply whether the response was correct. Accuracy is how far from the response was from the correct answer. An Accuracy of zero indicates that the answer was correct. Time indicates how long the participant took to respond. Time is measured from the end of the question to the beginning of the final response. Since the time measures until the final modification to the answer, it is possible for times to overlap between tasks if the participant were to change their response after answering another question. Overlapping times occurred infrequently through the study, but more frequently with Gitk than with *L*invis.

Order bias was mitigated throughout the study through the use of randomization between participants. In both parts, the order that the participants worked with the commits, tasks, and tools was randomized. Participant performance, with regard to the recorded performance metrics, between different size merges, or merges that are merging different number of commits, is measured in both parts of the study. The hypothesis being that it is easier to locate commits for and summarize smaller merges. In the remainder of the thesis, merge size refers to the number of repository events being integrated at a given merge. Two merge trees are selected from the set of merge trees, and from each of those, a single commit is selected. The reasoning and method for selecting the trees and commits is detailed in Section 6.1.1. The order that the commits were inspected was randomized between participants, and were kept consistent between parts for each participant. Where applicable, the order that the tools were used by the participants was randomized between participants. Details on the randomization of the tasks are outlined with the methodologies for the specific parts of the study.

Statistical significance testing is performed to verify that the results are meaningful. An $\alpha = 0.05$ or a 95% confidence level is used. The first test is used to determine if the Merge-Tree size has an effect on the result.

If tree size does not have an effect on the correctness, accuracy, or time taken, there is nothing interesting to be gained from analyzing the results separately and the results are combined. This is permissible, as the data comes from the same distribution. If tree size does have an effect on the correctness, accuracy, or time taken, then the results must be analyzed on the individual trees. The Wilcoxon test[26] is applied between commits to determine if the size of the Merge-Tree effects timing and accuracy.

The second test is used to determine if there is a significant difference between responses when the participants are using *L*invis versus Gitk.

The McNemar χ^2 test[19] with continuity correction is used to test if there is a difference in the correctness of responses made by users when using *L*invis versus Gitk. The Wilcoxon test[26] and Cliff's effect size[7] are applied to the accuracy and timing metrics to determine the significance of the difference between the results in *L*invis and Gitk.

The resulting research questions are:

1. Are people able to draw a conceptual understanding from the visualization of the DAG in Gitk?
2. Does Merge-Tree size have an effect on the correctness and accuracy of people's ability to summarize aspects of a merge?
3. Does Merge-Tree size have an effect on the time taken to summarize aspects of a merge?
4. Does *L*invis have an effect on the correctness and accuracy of the responses?
5. Does *L*invis have an effect on the time taken to respond to questions about the merge?

The first research question evaluates understanding based on how the participants draw the graph and respond to questions about their drawing. The question does not

make comparisons either, so no baseline is set for testing against. The collected results are stated, but this question is left open.

The null hypothesis for the last four research questions are as follows:

- There is no difference in the correctness nor accuracy between Merge-Tree sizes
- There is no difference in the time taken to respond between Merge-Tree sizes
- *L*invis does not have an impact on the correctness or accuracy
- *L*invis does not have an impact on the time taken to respond

The study is broken into two parts, followed by a short exit survey. The first part has the participants perform tasks and answer questions that require understanding about how and where a commit is merged. The second part of the study answers the remaining questions by asking the participants to summarize various aspects about the commits being integrated with a given commit. The responses are recorded, determining if the response was correct, the how far from the right answer the incorrect answers were, and how long participants took before answering. Part one is discussed in detail in Section 6.1.2, and part two is discussed in Section 6.1.3. After completing part one and part two, the participants were asked for opinions in a short exit interview to collect information about their prior experience with version control, and their opinions on the tools and visualizations used in the study. The details of which are discussed in Section 6.1.4.

Prior to conducting the study, participants were introduced to the Merge-Tree model and the conversion from the DAG to the Merge-Tree. Two examples of how the conversion works were given, the examples are shown in Figure 6.1. Any questions about the model or conversion were answered at this time. Then participants were introduced to the tools, Gitk and *L*invis, and could ask questions about either interface. These introductions usually lasted no more than 10 minutes.

6.1.1 Commit Selection

Two commits are used in both the conceptual study and summarization study, with the goal of determining how well the DAG and Merge-Tree visualizations scale between merges integrating varying numbers of commits. The order that the two commits are presented to each participant is randomized. I analyzed 15096 Merge-Trees

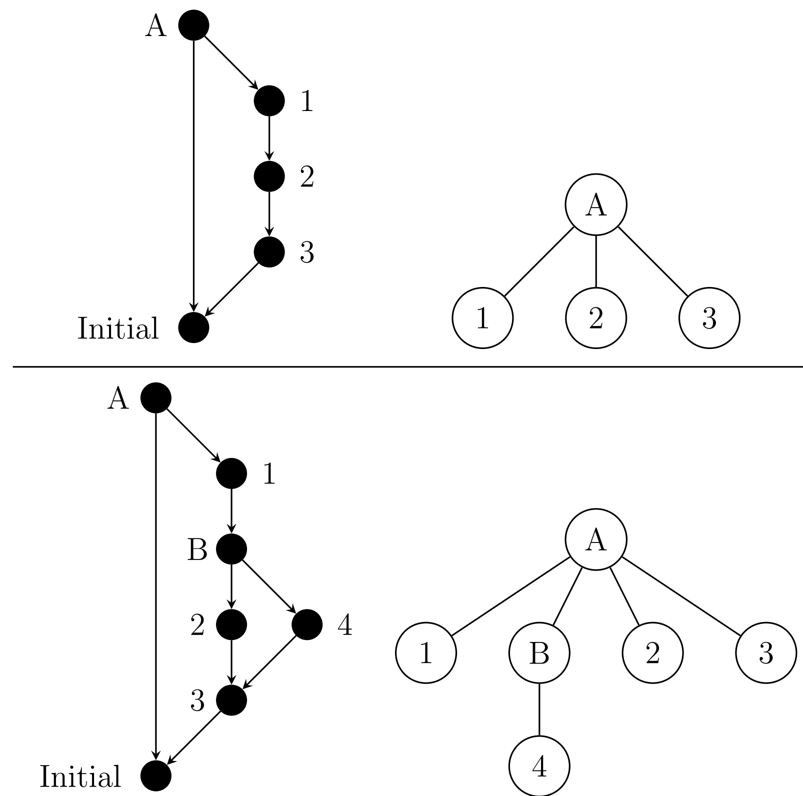


Figure 6.1: Two examples of DAG to Merge-Tree conversions used in explanation during evaluation

from the database that were candidates for the study, from April 16th 2005 to October 14th 2014, corresponding to Linux releases 2.6.12-rc3 to 3.17-rc1. A Merge-Tree could only be selected if it was not a foxtrot, and had correctly been identified. 25% of the trees contain at most a single repository event, not including the root, while 50% of the trees contain at most seven nodes. 75% of the trees contain at least 51 nodes, and the largest tree contains 7217 nodes. 8031 trees contained at least seven nodes, of these only 593 contained at least a single internal merge node. Trivially, trees with a single node cannot have any internal merge nodes. Of the 624 trees with seven non-root node, only 135 contained at least one inner node. Using this information, I chose a random tree from the 2008 trees in the first quartile, which merges a single commit into the master branch. These trees are trivial, but appear frequently in the repository. The second tree was chosen randomly from trees in the second quartile. These trees contain seven nodes, and to increase the complexity of the tree, I required that the tree contain at least one internal merge node. We limited the selected trees to the first two quartiles to ensure that the tasks would take a reasonable amount of time. The third quartile included trees with up to 51 nodes, and the fourth quartile included trees with more than 7000 nodes. Given that Gitk cannot provide aggregated summarizations due to the limitations of the graph, it is infeasible, in a reasonable time, to aggregate information about the larger trees, even with a conceptual understanding of which commits are involved.

Commits were selected randomly from the trees. The small tree was trivial as it contained only a single node. A node was chosen randomly from the medium-sized tree. The commits selected from the small tree and medium tree were *a3c1239eb59c* and *cdbdd1676a53* respectively. Commit 1, the commit from the small tree is visualized in Figure 6.2, showing the DAG visualization by Gitk and Reingold-Tilford tree visualization in *L*invis. The same is shown for commit 2 in Figure 6.3.

6.1.2 Part 1: Conceptual Study

The conceptual portion of the study seeks to answer the first research question, "Are people able to derive a conceptual understanding from the visualization of the DAG in Gitk". This part tests our initial assumption that the visualization of the DAG is unable to provide people with a conceptual understanding of the events in a repository. Participants in the study are asked to perform tasks that are related to understanding how a commit is integrated. The tasks for this part of the study are outlined in

Figure 6.2: The visualizations of commit 1 by Gitk and *L*invis respectively.

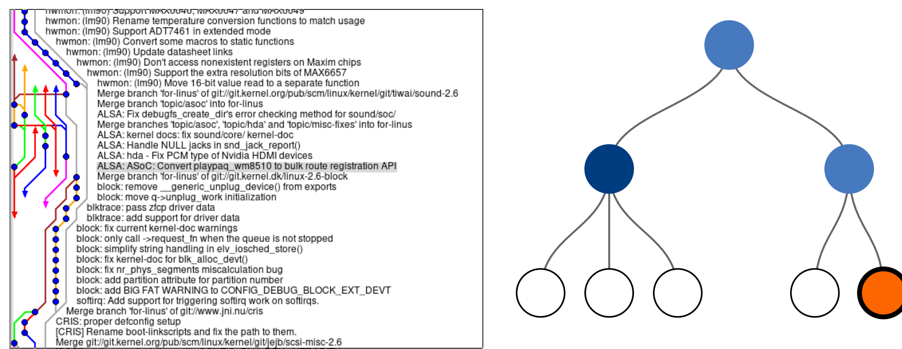


Figure 6.3: The visualizations of commit 2 by Gitk and *L*invis respectively.

Table 6.1.

Table 6.1: Conceptual Tasks

Task	Description
T1	Draw a diagram showing how this commit was merged into the master branch, along with any other related commits
T2	How many individual commits are related to this commit?
T3	How many merges are involved with merging this commit into the master branch?

Task 1 asks participants to draw a diagram showing the shortest path for a commit to be merged, including any commits that are related to it, or are necessary for the integration. Participants were given 10 minutes to draw the diagram, working with GitK and the git command line interface. The correct answer should look something like the Merge-Tree for that commit. The drawing from task 1 is then referred to in task 2 and 3. Building on the conceptual understanding built in task 1, task 2 and 3 ask the participant to determine how many commits are related, and how many merges. The order that task 2 and 3 are presented to participants is randomized between participants, to remove order bias, although it shouldn't matter as both questions are designed to build off of task 1 and the conceptual understanding constructed from analyzing the DAG visualization of that commit for 10 minutes. These questions enable us to find issues when users are comprehending the DAG visualizations.

The diagrams drawn by the participants in the first task help provide insight about how the participants are interpreting the DAG. I looked for patterns in the drawings to see if common issues arose, providing qualitative information about issues in comprehension. The results from task 2 and 3 are numerical results, the number of commits that are related, and the number of merges. These numbers are directly comparable with the correct numbers.

6.1.3 Part 2: Summarization Study

The summarization portion of the study compares the visualization and summarization capabilities of *L*invis and Gitk to determine if the visualization of the Merge-Tree is capable of providing a better understanding of the events in repository. This portion of the study requires the participants to switch between both Use-Case 1 and

Use-Case 2 strategies. The participants are provided an commit, since their goal is to summarize information about the entire Merge-Tree, they must navigate to the root node, making use of features for Use-Case 1. Once they are at the root, the participant must be able to summarize information about the authors, files, and modules. The tasks are outline in Table 6.2.

Table 6.2: Summarization Tasks

Task Set	Task	Description
Merge	T4	What is the series of merges involved with merging this commit?
	T5	What other commits are merged?
Authorship	T6	How many authors are involved?
	T7	Who contributed the most changes?
Files	T8	How many files were modified?
	T9	Which file had the most changes?
Modules	T10	Which modules does this Merge-Tree involve?

For each task, where specified, four statistical tests are applied. The first test determines if the results from the two commits are from the same distribution. If they are, it indicates that the results can be aggregated, otherwise, the results must be analyzed for each commit separately.

The following are the null hypotheses for the first test for each recorded metric:

- There is no difference in the correctness of the response between merge sizes
- There is no difference in the accuracy of the response between merge sizes
- There is no difference in the time taken to respond between merge sizes.

The three remaining tests measure the difference in correctness, accuracy, and time between the results for *L*invis and Gitk. The null hypotheses for each are as follows:

- *L*invis does not have an impact on the correctness of the response
- *L*invis does not have an impact on the accuracy of the response
- *L*invis does not have an impact on the time taken to respond

The tasks are split into four task sets, based on the type of information that the task is investigating. The *Merge* tasks set focuses on detailed information about the

topology of the merge itself, looking at the specific merges and commits involved in the integration. Task T4 asks for the specific merges that merge the commit into the master branch. These must be the correct merges, and must be in the correct order. I use the edit distance between the response and the correct answer as a measure of how correct the response is. Adding new merges, removing extraneous merges, replacing merges, and swapping the order of merges are of unit cost. An edit distance of 0 indicates a correct answer. Task T5 asks for the other commits that are integrated with this commit. Again, I use an edit distance-like metric to evaluate the response. Order doesn't matter, but the correct commits must be indicated in the response. Adding commits, replacing commits, and removing commits are of unit cost.

The *authorship* task set involves finding information about the authors involved in the merge. Task T6 asks for the number of authors involved in the merge. These are authors of commits, not merges, as merges in the kernel repository do not include code, and are used for creating logical separation in commits. The accuracy is measured as the absolute difference between the response and the correct answer. Task T7 asks participants to identify the person who was responsible for contributing the most in the merge. While it is possible that two authors could have contributed the same number of changes, in the merges for both commits, there is an identifiable author who contributed the most. The answer to this task can either be correct or incorrect, so accuracy is not recorded for this task.

The *files* task set involves finding information about specific files being merged. Task T8 asks the participant to identify how many files are modified in a merge. Like task T6, the response to task T8 is a single number, so the accuracy is measured as the absolute difference between the response and the correct answer. Task T9 asks the participants to identify which files had the most lines modified in the merge. While this is similar to task T7, the files in the tree associated with commit 1 had the same number of changes. For this reason, I use the edit distance between the response and the correct answer, with addition, removal, and replacement of unit cost.

The *modules* task set only contains a single task, and involves determining the modules involved in a merge. Modules, or subsystems, refer to the component of the kernel that is being modified by a commit. This is not a property that is inherent to git repositories in general, but a property I noticed in the repository of the Linux kernel. Commit summaries are prefaced with the module, followed by a colon. For example the log summary *"ALSA: kernel docs: fix sound/core/ kernel-doc"* is in the *"ALSA"* module.

The order that the task sets are performed is randomized between participants, and the order that the tasks within a task set are performed is randomized as well. This keeps related tasks together, while still mitigating some of the order bias.

6.1.4 User Opinions and Exit Interview

The goal in this part of the study is to expose issues in the underlying assumptions made while writing *L*invis. Two questions were asked in this part of the study, outlined in Table 6.3. This portion of the study gives the participants to voice their opinions and observations that may not have been recorded or captured by the rest of the study.

Table 6.3: User Opinion Questions

Question	Description
Q1	Given these tasks again, which tool would you prefer?
Q2	Which aspects of each tool did you like and why?

Question Q1 allows users to express their opinions on tool preference for merge-summarization tasks. Neither *L*invis nor Gitk are perfect, participants may have complaints or aspects of each tool that they preferred, or aspects that assisted them in understanding the events in the repository. Question Q2 is meant to address this.

The exit interview is designed with the goal of collecting some information about our participants, and their experience with version control software and git. Three questions were asked in the exit interview portion of the study:

- For how long have you used git?
- For what kind of projects have you used git?
- How many commits, files, and collaborators were involved with the largest repository you have worked with?

6.2 Participant Profile

The study was conducted with 12 participants, all of whom were masters, PhD, or post-doc researchers in the field of software engineering. The participants had between

6 months and 10 years experience with git, with the median being 3.5 years. Most participants had additional experience with SVN and CVS. One of the participants in the study worked as a release engineer, studying merge practices to determine the best way to merge branches while minimizing the number of merge conflicts in SVN repositories. The participants worked with repositories ranging from around 10 commits up to 38000 commits, with the median being 350 commits. Two of the participants had never collaborated with anyone in a repository, while the rest had some experience with repositories being modified by multiple people, with the most being 219. The median number of collaborators was four. Participants had most experience with personal and academic repositories. Three of the twelve participants had experience with professional repositories.

All participants have had at least some experience with version control, branching in repositories, and git. The participants are from the same lab, and each participant worked with both tools in the study, thus, keeping the sample populations identical for both tools, with some variation between participants in experience with repositories.

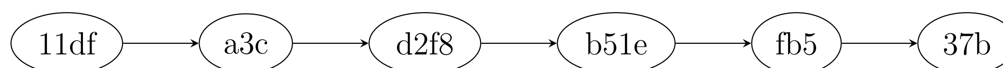
6.3 Results

This section presents the results from the user study. Section 6.3.1 presents the results for the conceptual tasks. The results for the summarization tasks, performed on both *L*invis and Gitk, are presented in Section 6.3.2. Lastly, the user opinions are presented in Section 6.3.3.

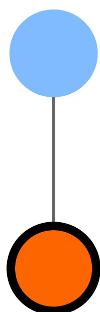
6.3.1 Conceptual Study Results

The conceptual study is performed with Gitk to determine if the participants are able to derive a conceptual understanding of how commits are integrated into the master branch from the graph. Investigating the drawings made in response to task T1, none of the results from this task are correct. The diagrams for commit 1 tend to resemble a linked list, showing all of the merges along the master branch. The participants would generally continue traversing the master branch using either the first parent or the child links in the Gitk interface, or simply following the branch in the visualization, until they either hit the branch tag `2.6.29-rc6`, or indicated that it was every merge along the master branch, or stopped at a seemingly arbitrary merge. An example of a diagram drawn by one participant is shown in Figure 6.4a. This drawing follows the

DAG backward, working from the commit, through the branch point, and traversing the DAG toward the initial commit. In another case, the participant was able to produce the correct drawing using the git command line tools, but then looked at the DAG visualization in Gitk and continued to draw a linked list, following the merges along the master branch toward the HEAD node.



(a) Example of a diagram drawn for the integration of commit 1



(b) The correct diagram showing the integration of commit 1.

Figure 6.4: An example of a drawn diagram for the integration of commit 2 compared with the correct answer.

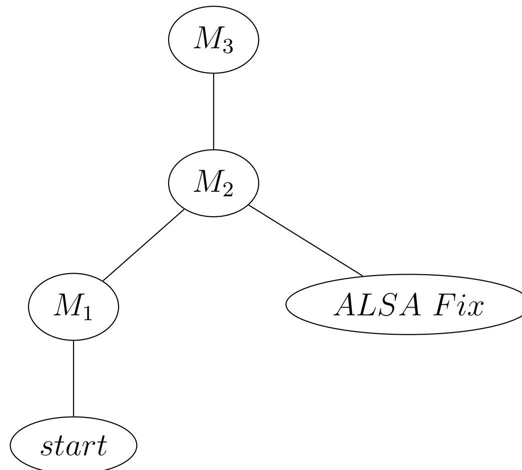
This indicates difficulties identifying the master branch in the visualization. Furthermore, the commit date should provide some indication on which direction the commits are being merged in, some participants did not understand this from the Gitk interface. This could be confusion looking at the interface, or in the terminology surrounding the parent-child relationship.

The resulting diagrams for commit 2 do not show any consistent patterns among participants. Many diagrams do not show the shortest paths from a commit to the master branch. Some participants were able to identify the master branch in this case. These participants had experience with SVN repositories, and were accustomed to the master branch being the first branch in the graph. In the visualization of the DAG for commit 2, the master branch returns to the first position in the graph, which some participants mentioned that this meant it should be the master branch. In this case, they are correct in that the first line in the visualizations is the master branch in this case, but the reasoning is not correct. The first branch is not necessarily the master branch, as is the case with the results from Commit 1, where the master branch is the third line.

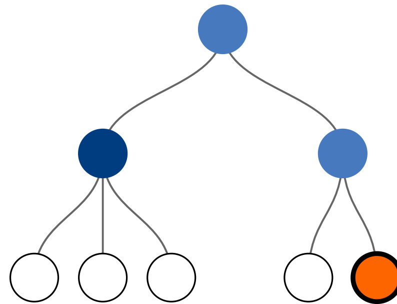
With some participants, there was confusion about the direction of the parent-child relationship. In most diagrams, the parents point toward the node of interest. In this case, the parents point toward the branch point, which is in the opposite direction of the merge, so the participant would actually need to follow the child to get to the merge point. The difference is alluded in Figure 3.2 and Figure 3.3. Notice that in Figure 3.2 that the edges point from the merge toward the commits, and eventually toward the branch point. Conversely, the edges in Figure 3.3 point from the commit toward the integrating merge.

One participant was able to identify the path to the master branch, but no participants were able to identify the commits that were necessary for integrating the given commit into the master branch. The diagram drawn is shown in Figure 6.5a, which shows a possible path for the commit to pass through on the way to the master branch. While it isn't the shortest path, it is a possible path. In the diagram, M_3 is the merge into the master branch, and *start* the commit that is being queried. The diagram includes one other commit that is being merged, but there are three additional commits that were not included. In the Merge-Tree representation, the starting commit and the $ALSA\ Fix$ commit are both merged into M_2, while three others are merged into M_1. M_1 and M_2 are merged directly into the master branch at M_3. In the other cases, the diagrams had little resemblance of the events occurring in the repository. Some diagrams appeared to be constructed from random commits, others did not conform to the tree structure as what would be constructed from using shortest paths.

The results show that participants were closer to the correct number of merges to integrate commit 2 into the master branch. This is counter-intuitive as commit 2 is from the larger, more complex Merge-Tree. Furthermore, the results show that they were also closer to determining how many other commits were integrated with it than with commit 1, shown in as seen in Table 6.4. The results reported in this table are in number of commits for task T2, and the number of merges for task T3. Commit 1, which was merged directly without any other related commits, was said to have a median of 6 related commits. The median number of merges was said to be between 6 and 7 merges. The correct answer should have been 1, the integrating merge. There was more disagreement when determining the number of commits, in both commits. Interestingly, there was more variance in the answers on the smaller merge tree. The variance in the answers for determining the number of merges was consistent between both trees.



(a) Example of a diagram drawn for the integration of commit 2. This diagram is the closest to being a possible representation of the events in the repository, but miss the other commits involved.



(b) The correct diagram showing the integration of commit 2.

Figure 6.5: An example of a drawn diagram for the integration of commit 2 compared with the correct answer.

Table 6.4: Variance and Difference between correct answers and user responses in conceptual tasks in tasks T2 and T3

Task	Commit	Median Difference	Average Difference	Answer Variance
T2	Commit 1	5	20	803.43
T3	Commit 1	5.5	7.8	56.18
T2	Commit 2	2	6.22	120.19
T3	Commit 2	1	3.67	48.5

While the results were closer with the larger Merge-Tree, the merges that were indicated as being the integrating merges and commits identified as being those integrated with it were incorrect in both cases. For commit 1, many participants indicated that the entire master branch up to the branch pointer was in the integrating merge. This dramatically increases the average and variance for commit 1. In the case of commit 2, participants selected merges and commits a more conservatively, though many of the selected merges and commits were incorrect, and in some cases appeared to be randomly selected.

Participants generally took longer to respond to questions about the larger merge tree, and the time taken was far more variable. The timing results are listed in Table 6.5. When interpreting these results, it must be noted that participants had spent 10 minutes working with the merge tree that they were summarizing prior to answering the questions. These times do not indicate the time to read the DAG, but the time taken to understand the conceptual image of the events in the DAG. It took the participants longest to determine the number of commits in both cases, but it took far longer in the case of commit 2, taking over half a minute.

Table 6.5: Timing Results from the conceptual tasks T2 and T3

Task	Commit	Median Time (s)	Average Time (s)	Time Variance (s)
T2	Commit 1	9	53.45	6382
T3	Commit 1	8	26.64	921
T2	Commit 2	35	114.45	58769
T3	Commit 2	15	71.36	32324

Users were able to more closely estimate the number of commits and merges in the larger tree, but generally took longer than the smaller tree. The tree with a single node resulted in more variability in the estimate of number of commits.

6.3.2 Summarization Study Results

Merge size does not affect **correctness** for merges in the first and second quartile of merge sizes; however, task T5, finding the other commits being integrated, and task T9, determining which file had the most changes, are very close to the p-value threshold. The results of the Wilcoxon test on correctness are shown Table 6.6. Most of the p-values are reasonably far from the threshold of 0.05; however, in tasks T5 and T9, the p-value is within 0.01 of the threshold. Further investigation reveals that

the differences in the distributions stem from the number of participants who were incorrect when using *L*invis. Inspecting the results from task T5, in Figure 6.6, no participants provided an incorrect response using *L*invis in the small merge, but four participants provided an incorrect response when using *L*invis for the larger merge. Similarly, the number of incorrect responses increased going from the small merge to the large merge with Gitk. The split results are similar in task T9.

Table 6.6: Effect of merge size on correctness

Task	p-value	Conclusion
T4	0.16	Do not reject H_0
T5	0.05	Do not reject H_0
T6	0.11	Do not reject H_0
T7	0.08	Do not reject H_0
T8	0.13	Do not reject H_0
T9	0.06	Do not reject H_0
T10	0.45	Do not reject H_0

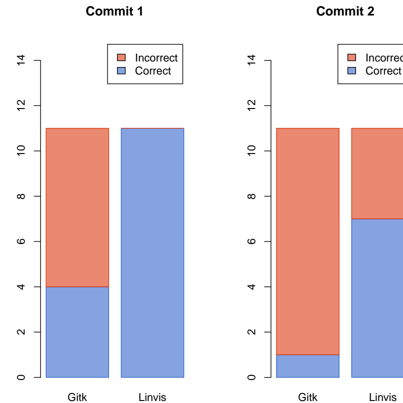


Figure 6.6: Difference between commits in the correctness of responses to task T5.

The results do not indicate that there is a difference in the distributions between the merge sizes, although one might arise with tasks T5 and T9 with more samples, the results for both merges are combined. The McNemar test is applied to determine if *L*invis has an effect on the correctness. The results of the McNemar test are presented in Table 6.7. For each task, except for task T10, determining which modules are involved in a merge, *L*invis has an effect on the results. Inspecting the results in Figure 6.7 shows that *L*invis effects results in a positive way, helping users to correctly

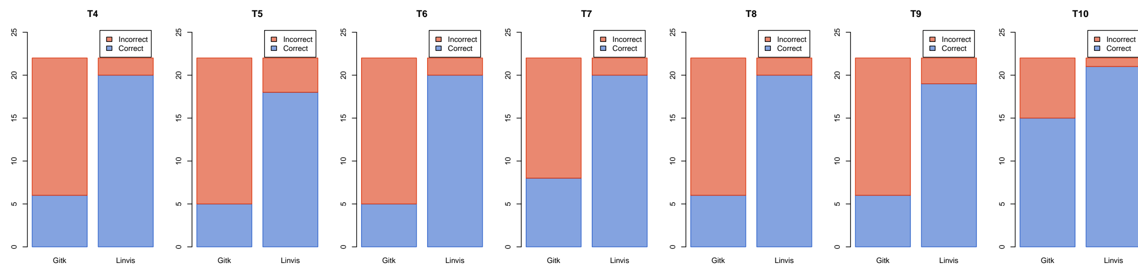


Figure 6.7: Aggregated Correctness of the summarization results

identify properties about the merge. In task T10, it is also evident that participants were able to determine the module involved using both tools.

Table 6.7: Aggregated Correctness Results comparing *L*invis and Gitk

Task	χ^2	p-value	Conclusion
T4	12.07	0.0005	Reject H_0
T5	11.08	0.0007	Reject H_0
T6	13.07	0.0003	Reject H_0
T7	10.08	0.0015	Reject H_0
T8	12.07	0.0005	Reject H_0
T9	11.08	0.0009	Reject H_0
T10	3.13	0.0771	Do not reject H_0

Merge size does not affect **accuracy** in any of the tasks, as seen in Table 6.8. The results in task T9 are very close to the threshold. Inspecting the distributions in Figure 6.8, the distribution for *L*invis appears to be different in the two commits. In commit 1, there is no variance, the entire distribution is at 0 files from the correct answer. In commit 2, the 3rd quartile spans from 0 files up to 1 file from the correct answer, and the fourth quartile, from 1 file to 2 files. This is consistent with the results found for correctness, although, unlike with correctness where task T5 showed more difference between merge sizes, task T9 shows more difference between merge sizes with accuracy. This indicates that those who were incorrect, were also further from the correct answer in the larger merge than the small merge in T9 than in task T5.

Table 6.8: Effect of merge size on accuracy

Task	p-value	Conclusion
T4	0.97	Do not reject H_0
T5	0.08	Do not reject H_0
T6	0.21	Do not reject H_0
T8	0.13	Do not reject H_0
T9	0.06	Do not reject H_0
T10	0.22	Do not reject H_0

The results do not indicate a difference in the accuracy distributions between commits. Again, this could change in tasks T5 and T9 with a larger sample size. The results for both merges are combined. The Wilcoxon test and Cliff's Delta effect size

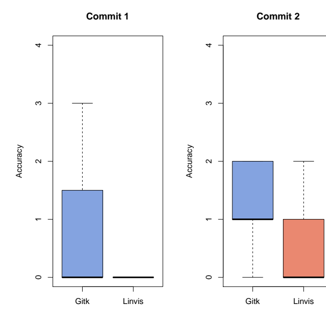


Figure 6.8: Difference in accuracies in responses to task T9 between Commit 1 and Commit 2.

are applied to each to determine if there is a difference in the accuracies of responses to the tasks between tools. The results of the tests are presented in Table 6.11. In all cases, there is a difference in accuracies between when the participants use *L*invis versus Gitk. In all cases except for task T9, which files had the most changes, and T10, which modules were involved, there was a large effect in favour of *L*invis. In task T9, there was a medium effect, and T10, a small effect. Looking at the results depicted in Figure 6.9, this makes sense. The variance in the accuracies of responses of Gitk is much smaller in tasks T9 and T10 than in the other tasks, furthermore, the median is much closer to zero than in the other tasks. In task T10, only the third and fourth quartiles are beyond 0, which indicates that participants were generally correct when using Gitk to determine the module. This is also consistent with the results measuring the correctness.

Table 6.9: Aggregated Accuracy results, including the Wilcoxon p-values and Cliff's Delta Effect size

Task	p-value	Delta Est.	Conclusion	Effect
T4	0.00017	0.596	Reject H_0	Large
T5	0.00019	0.616	Reject H_0	Large
T6	0.00000	0.675	Reject H_0	Large
T8	0.00079	0.534	Reject H_0	Large
T9	0.01195	0.412	Reject H_0	Medium
T10	0.00479	0.318	Reject H_0	Small

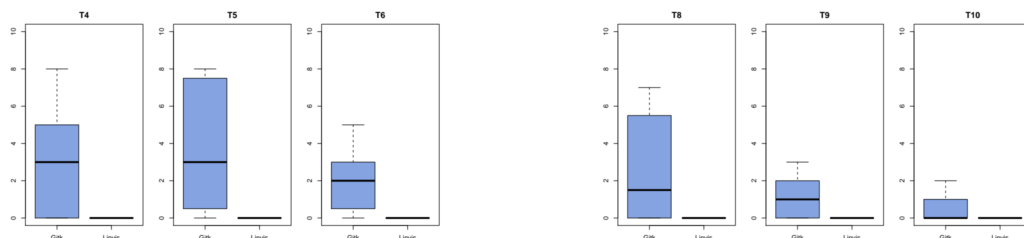


Figure 6.9: Aggregated Accuracy of the summarization results

The results indicate that in all tasks except for task T7, who contributed the most changes, merge size did not have a significant impact on the **time** taken to respond. The results are presented in Table 6.10.

Inspecting the results from task 7, depicted in Figure 6.10, the primary difference stems from the time used to respond when using *L*invis. Participants much longer

Table 6.10: Effect of merge size on response time

Task	p-value	Conclusion
T4	0.99	Do not reject H_0
T5	0.90	Do not reject H_0
T6	0.92	Do not reject H_0
T7	0.01	Reject H_0
T8	0.99	Do not reject H_0
T9	0.70	Do not reject H_0
T10	0.77	Do not reject H_0

to produce a response for the larger merge while using *L*invis than they did for the smaller merge. Comparing this with the timing results for task T8, in Figure 6.11 where participants took very little time to respond for both merges. Participants took considerably less time using *L*invis than they did with Gitk.

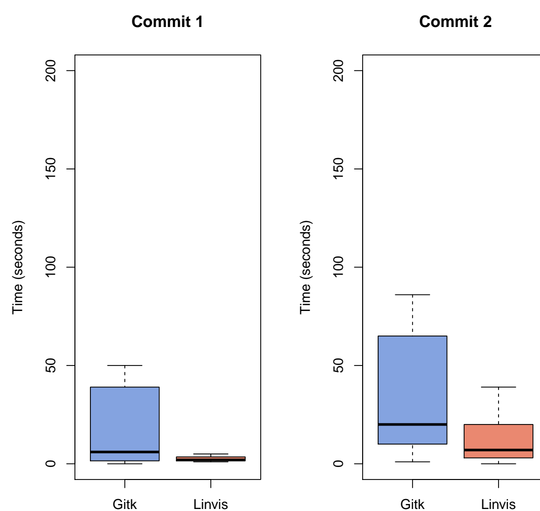


Figure 6.10: Difference in time taken to respond to task T7 between merge sizes

The results do not indicate that there is a difference in the time taken to respond between merge sizes for the other tasks. In all tasks, except for T7, *L*invis does have a statistically significant impact on the time taken to respond. *L*invis has a large impact on the time taken to respond to tasks T4, T8, T9, and T10, determine the merges that led to integration, the number of files modified, which files had the most changes, and the modules involved. *L*invis had a medium effect on the time taken to respond to tasks T5 and T6, what other commits are integrated with this commit, and the other number of authored involved. The direction of the effect is visible in Figure 6.12.

The results of task T7 in Figure 6.10 show that *L*invis appears to have an effect. The Cliff's delta effect size indicates that, in the case of the medium-sized merge, there is a medium effect. The Wilcoxon test, however, indicates that the null hypothesis should not be rejected, or at least that there is inconclusive evidence to show that there is a difference. This is likely due to the sample size; the Wilcoxon test is related to the sample size, while the delta effect size is not. Since the results in task T7 are not aggregated, the number of samples is effectively cut in half, from 22 down to

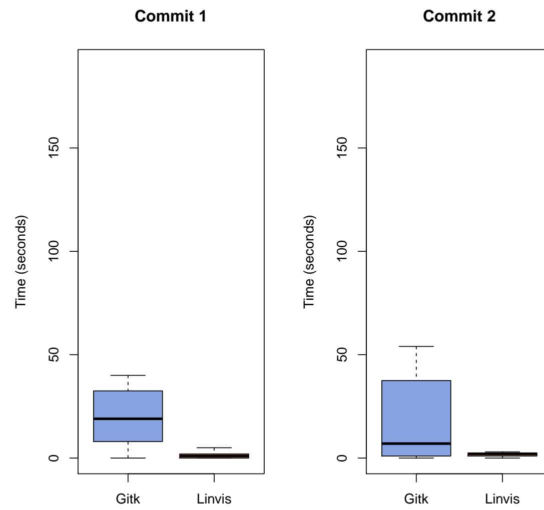


Figure 6.11: Difference in time taken to respond to task T8 between merge sizes

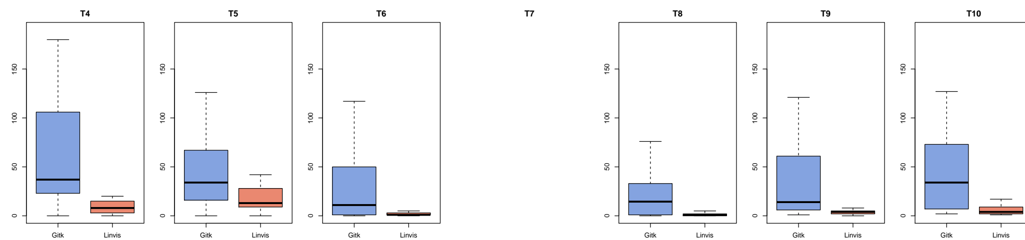


Figure 6.12: Aggregated Time to respond to summarization tasks

11, which is not enough to have confidence that the results are representative of the population. The effect size says that if our sample is representative of the population, this is the impact that could be expected. The results of the tests are presented in Table 6.11.

Table 6.11: Aggregated time results, including the Wilcoxon p-values and Cliff's Delta Effect size

Task	p-value	Delta Est.	Conclusion	Effect
T4	0.0006	0.605	Reject H_0	Large
T5	0.0241	0.399	Reject H_0	Medium
T6	0.0254	0.388	Reject H_0	Medium
T7	0.2586	0.289	Do not reject H_0	Small
	0.1146	0.405	Do not reject H_0	Medium
T8	0.0018	0.545	Reject H_0	Large
T9	0.0002	0.667	Reject H_0	Large
T10	0.0002	0.649	Reject H_0	Large

To summarize, *L*invis is able to assist users determine the series of merges a commit passes through to integration, the other commits integrated with it, the number of authors, how many files were modified, and which file had the most changes more quickly and accurately than with Gitk. *L*invis does not have an impact on the accuracy when determining the modules involved in the merge, but does have an impact on the time taken to respond. *L*invis does not appear to have an impact on the time take to determine who contributed the most changes, but does have an impact on accuracy and correctness. Interestingly, there is a medium effect on the time taken to respond in the case of the larger merge, but little confidence in this effect. More participants are required in order to determine if *L*invis makes a difference.

6.3.3 User Opinions

Among the 12 participants, there was nearly unanimous agreement that for conceptual understanding and summarization tasks, *L*invis was easier to use than Gitk. The participants cited the ability to abstract information about the merge from the clean summarization tables and simple visualizations as the primary reasons for preferring *L*invis to Gitk. Three participants suggested that someone with a professional understanding of Gitk and the git command-line may be able to extract a conceptual un-

derstanding from the DAG visualization and perform the summarization tasks. One of these three participants said that they would prefer to have both tools available, as they are able to complement each other. This is discussed further in Section 7.3.

Chapter 7

Discussion

This chapter provides further discussion on the results and observations from the study evaluating *L*invis. Included are observations that could lead to improvements in the current DAG visualizations, and the comments from a release manager.

7.1 Interpreting the Results

Overall, the results indicate that *L*invis is able to improve the correctness and accuracy of responses to various summarization tasks, and decrease the time taken to produce the results. This doesn't come as a surprise since the goal of the Merge-Tree model and the visualizations in *L*invis are to provide better conceptual understanding and summarizations of merges, while Gitk and DAG visualizations are designed to show the topology of the entire repository. Since there are no other tools for showing how a commit is integrated, and the topology of the DAG does contain this information, the DAG visualization is used as a proxy to show how a commit reaches the master branch.

One area of interest is the comparison of *L*invis and Gitk on correctness in task T10, determining the modules modified in a merge. Again, modules are not inherent to Git and are a property of the commits in the Linux repository, the module is found in the summary of the commit logs. In this task, the difference in the number of correct responses between *L*invis and Gitk was not statistically significant, and, while significant, the effect on accuracy was also small. This is interesting because *L*invis provided this information directly, while users would have to look at the commit logs to determine this information from Gitk. Further inspection of the merges show that

this was the only task where the correct answer was in the commit that was provided, and actually required no aggregation of the results.

Another area of interest are the time results for task T7. This is the only task where merge size had a significant impact on the performance of the participants. There was not a statistically significant difference in the time taken to respond to this task between the two tools; however, the effect size indicates that the tool has a medium effect on the time taken to respond. This is likely due to the sample size. In the other tasks, the responses 11 responses for both merges were combined, effectively doubling this number, creating 22 samples. Since there was a difference in the time taken to respond given the merge size, the results had to be analyzed separately. 11 samples is quite small, and is likely not enough to have a 95% confidence in the results.

Ultimately, the positive results are not entirely unexpected. *L*invis is specifically designed for handling the tasks that we presented to the participants in our study. Ideally, we would be able to compare against another tool that is specifically designed for the same tasks; however, one does not currently exist. Instead, we compare the results between *L*invis, and the tool that is currently in use for these tasks, Gitk.

7.2 Study Observations

Identifying the master branch was an issue that consistently came up among all participants during the study. Some participants assumed that the first line in the DAG visualization indicated the master branch, while others assumed that the next branch tag indicated the master branch. The DAG visualization provides no indication of which branch is the master branch. Furthermore, the visualization in Gitk is not consistent, branch colors and positions change between runs; identifying the branch once does not guarantee that it is identifiable after restarting Gitk.

Had the participants been able to easily identify the master branch, the results from the study would likely be very different. This would be most prominent in the summarization portion of the small merge, since summarizing a single item is trivial. The issue was identifying that there was only a single item. With more than 25% of the merges into Linux being single-commit merges, it is important that users are able to identify them and understand the changes being made within them. The structure of a single-commit tree is identical to the structure of a flat tree, all commits are merged directly into the master branch, passing through no other merges on the way.

Flat trees are the most common form of tree in the kernel repository. To improve the visualization of the DAG for providing an effective visualization for summarization and comprehension of flat trees, it would likely be sufficient to indicate which line represents the master branch. For the non-flat trees, a more powerful structure would likely be necessary, such as the Merge-Tree.

7.3 Comments From a Release Manager

One of the participants in the study had worked as a release manager for more than three years, working with both SVN and CVS repositories. The goal of a release manager is to determine how to merge the branches of a repository in such a way that it minimizes merge conflicts and maintains the meaning of the underlying source code. This section contains insights from this participant, providing comments on ways that could improve *L*invis and the Merge-Tree model.

Contributors making merges need to understand more than just what merges a commit was collected into before reaching the repository of the contributor. It is also important to understand the order that the related commits were made, as the order tells the story of what the developer was thinking as they were writing the changes. The visualization of the Merge-Tree in *L*invis does not order the commits, randomly ordering them in each level as atomic units.

This is the primary reason behind why this participant would ask to use both tools simultaneously. *L*invis is able to help with the aggregation of the information, and provide a better understanding of the next merge involved in integrating this commit, but the DAG visualization in Gitk provides the full story of the commit instead of hiding it behind a layer of abstraction.

The algorithm is capable of retaining information about which commit comes next on the path to the master branch. The adjustment simply requires that the returning commit be passed with the depth, next merge, and integrating merge. The update rules are the same for the next merge and integrating merge, so no other changes are required. Then instead of using the next merge toward integration, the next commit should be used in the visualization.

The comments from this participant were very insightful, and will help to improve the Merge-Tree model.

7.4 Threats to Validity

While precautions were taken to mitigate threats to the validity of this research, these threats must be taken into account when considering the results. The threats, the mitigation techniques, and the steps to minimize other threats are provided in this section.

7.4.1 Internal Validity

This section provides a summary of areas where bias may have influenced the results of the study and the steps taken to mitigate these issues. The biases may impact the design of the study, how the questions are presented, and how the results are interpreted. To ensure that this research was conducted in a way that was ethical, the study was approved by the university ethics board.

The design of the study has an impact on the results. I needed to make considerations about the merges that would be used during the study and the tasks that the participants were working with. The merges need to be representative of what is found in the Linux kernel repository, but must also not be too large such that it overwhelms the participants. The task selection is another challenge in this study, the goal is to limit the bias toward either tool while attempting to determine if *L*invis has the desired effect.

The tasks themselves are biased in favour of *L*invis, as the tool is specifically designed for answering questions surrounding conceptual understanding of how commits are merged. Ideally, we could compare *L*invis with another tool designed for the tasks specified, but such a tool does not exist. Instead, we compare against the tool that is currently in use for performing these tasks. The original thesis statement must be softened to accommodate this change though.

How the study is conducted can impact the results. The study has two manipulated variables, the tool being used and the number of commits being merged. The order that people work with these will have an impact on the results. As a means of combating the resulting order bias, the order that commits were analyzed, tasks performed, and tools used was randomized for each participant. To minimize the risk that an error is introduced by the randomization, a script was used to generate the exact text for each experiment. While this method proved to be very useful, I omitted one task during the course on one study. As a result, the information collected from this participant were removed from the analysis and final results.

There are many merges into the master branch of the repository. The number of commits being merged range from one to more than 6000. The goal is to select merges that do not give an advantage to either tool, are representative of the merges found in the repository, and are merging a different number of commits. The first commit studied came from a merge that was in the first quartile. All of the merges in the first quartile were merging only a single commit. One of these merges was randomly selected from the data set using a python script to avoid any biases. The first and second quartiles were selected to avoid overwhelming the participants. From prior experience, Gitk does not summarize the changes at a merge. To summarize the changes at a merge, a developer needs to visit each commit that is being merged and record the desired metric. For small merges, this is feasible. With large merges, this would become heavily burdensome on the participants of the study.

This biases the results in favour of the tool that is better-able to visualize small merges. Due to this bias, I had hypothesized that Gitk and *L*invis would have nearly identical results for the merge containing only a single commit, while the results would be in favour of *L*invis for the second commits. This was not the case, participants had a difficult time discerning which commits were being merged in both cases. Based on the comments and behaviours exhibited while performing the tasks, I don't believe that increasing the number of commits will improve the results of Gitk.

The main issue with Gitk appeared to stem from the difficulty in determining the set of commits that belonged to a merge. The answers provided by participants to earlier tasks were not taken into account when evaluating the correctness or accuracies to following tasks. The results of an incorrect answer may impact the results of the tasks that followed. If the participant was unable to determine the correct commits that were being merged, then none of the summarizations would be correct, even if the response was correct given the commits they identified. A future study could mitigate this by providing the correct set of commits that are being merged between the conceptual and summarization task sets to limit the propagation of errors.

7.4.2 External validity

While many online git resources, git graphical clients, and the git command line provide visualizations of the DAG, many participants were unfamiliar with the DAG. While other tools than Gitk and the command line may provide better summarizations and different visualizations, I am not aware of any. I investigated the use of

other GUI tools on the git website, but none were able to produce visualizations for repositories that are at the size of the Linux kernel repository, except for Gitk and the git command line at the time of the study. While this may have had an effect on the results, the tools listed on the website provide a very similar visual DAG metaphor to the visualization in Gitk.

The participants in the study were students, some with industrial experience. Most participants had worked with relatively few collaborators on academic projects. Many of the participants had worked with relatively large repositories while performing a research study. Even though the participants have worked with large repositories during the course of their research, professional developers are the target audience of this tool, so working with professional developers would provide more meaningful results.

7.5 Limitations

The model is designed with the Linux repository in mind. The viability of Merge-Trees to provide useful and accurate information relies on a few properties of the underlying repository. The repository must use a branch and merge structure. Some repositories, like the OCaml repository, commit directly into the master branch. At release time, a branch is created for the version being released. Patches to the version are added as necessary. The release branch is never merged back into the master branch. Since Merge-Trees are designed to show how a commit is integrated into the master branch, an Merge-Tree will not help with a repository with this structure.

Repositories cannot have foxtrots. A foxtrot confounds the master branch, making it impossible to properly determine where the integrating merge occurs. The algorithm will continue to process repositories containing foxtrots; however, the resulting Merge-Trees will not be meaningful.

Repositories should limit the use of fast-forward merging. The goal of Merge-Trees is to help understand how commits are grouped together, which is done at a merge commit. Fast-forward merges splice the changes directly into the underlying branch, hiding the fact that there was ever a branch. The original branch information is not retrievable and will result in many flat trees, where everything is merged directly into the master branch, or worse, the master branch contains only individual commits.

7.6 Future Work

In the evaluation of *L*invis, it was noted that some important information is lost in the conversion from the graph to the Merge-Tree. Commits usually build on the changes of it's ancestors; the changes in one commit usually build on the changes of the commits that came earlier in the graph. The proposed Merge-Tree model does not maintain the order of the commits in the graph, only that they share a merge. This leaves space for additional research to build on the Merge-Tree, extending the model to preserve the order of the commits in the graph. This extension would involve deciding on new visualizations, and performing a new study.

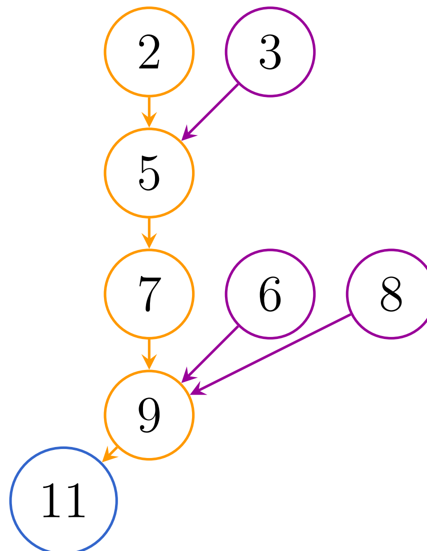


Figure 7.1: Updating the Merge-Tree shows the order that commits are created, while retaining the merges that the commits pass through.

In our model, the trees are short, but very wide, as there are usually very few merges that a commit will pass through, but there may be many commits that build on it before finally having the changes merged. In the current structure, all of these commits are on the same level, as they are merged together, which contributes to the short and wide structure. In the updated structure, the commits won't be on the same level, the parent of the commit will be the next commit in the chain. This structure leads to tall and narrow trees. The list-tree and Reingold-Tilford tree would likely visualize these effectively. The list-tree would only need to order the elements according to the order of the commits, but would otherwise be left unchanged. The Reingold-Tilford tree would become very tall, but this does not present any major

technical challenges. The pack tree visualization is specifically designed for wide and short trees. If a node is the only child of it's parent, the parent node is not drawn. With the structure of the trees, this does not present any major issues as there are generally multiple commits being merged by a given merge. If the model is adjusted so that the parent of a commit is the next commit on the path toward the master branch, this property changes. A commit may have one or zero children, so only the first commit of a branch will be rendered by the pack tree, omitting most of the information. An example of what the model may look like is shown in Figure 7.1.

Changing the model to include information about how commits are ordered will have an impact on how it is rendered. Some of the visual metaphors of the tree will be impacted more severely than others. A re-evaluation is required to verify that the updated model and new visualizations still fulfill the original goal, to enable people to easily determine the merges that a commit passes through toward being integrated, and the other commits that are integrated with it.

The work in this thesis addresses an issue with comprehending visualizations of the DAG. At no point is it verified that the work is applicable to the problems faced by practitioners in industry. While this is due to accessibility, future work should perform a study to verify that the Merge-Tree model, and the visualizations of the model are able to help solve issues in industry.

Chapter 8

Conclusion

The visualization of the DAG of a git repository is difficult to comprehend in large projects. This thesis investigates user comprehension of the DAG visualization in the Linux repository, and presents the design of a new model and a tool built from the model, with the goal of assisting users with comprehending how a commit is integrated into the master branch. Very few tools have the explicit goal of showing the topology of the repository. No academic tools that I am aware of attempt to do this. Most of the non-academic tools provide a visualization of the entire DAG, if they are able to produce a visualization at all. There is little variance in the DAG visualizations between these tools, which leaves room for improvement.

One major issue in understanding the DAG visualizations is the amount of information being presented. The DAG visualization provides information about all of the commits, but in the case of the Linux repository, the integrating merges in the master branch work nearly independently of each other. Only the commits that are merged together are related to each other are relevant, while commits that are not included in that merge are unrelated. The Linux repository adds thousands of commits per release, but only a few of these are related to each other. 50% of the merges are merging at most seven commits.

The Merge-Tree model takes advantage of this structure, breaking the commits into groups based on the merge into the master branch. The commits are then organized into trees, the parent of a node is the next merge on the way to integration, which shows the path that a commit takes to reach the integrating merge into the master branch. A commit may pass through multiple merges on the way to the master branch. An algorithm is devised, and evaluated. Through the evaluation of the algorithm, I found some interesting events in the repository. The logs for the

integrating merges contain the number of commits being integrated, and a listing of a subset of the commit log summaries being merged. This practice was put into place on September 4, 2007. A foxtrot merge occurred on December 12, 2006. I identified 507 merges, of the 1537 merges made prior to this date, that were confounded by the foxtrot.

I constructed a tool, *L*invis, around the Merge-Tree model. Leveraging the model, *L*invis is able to provide simpler visualizations and summarize additional information about the commits being merged, including the authors involved and files modified. Through *L*invis, I am able to further evaluate the effectiveness of the visualizations of the Merge-Tree compared to the visualizations of the DAG.

Using *L*invis, I conducted a 12-user study with two goals. One goal is to verify the assumption that the visualizations of the DAG are not able to convey information about how a commit is integrated into a project. The other goal is to compare the visualizations and summarizations from the DAG in git and Gitk to the visualizations and summarizations from the Merge-Tree. The participants were unable to accurately determine how commits were integrated from the DAG visualization. Furthermore, the visualizations and summarizations in *L*invis helped the participants answer questions about a merge more accurately and more quickly. Further information gathered from the study indicated that important information that was present in the DAG visualizations was lost in the Merge-Tree model. The order of commits with regard to each other tell the story of why a developer is making changes. This information is lost in the Merge-Tree, but is retained in the DAG visualizations.

Merge-Trees are a novel means of processing git repositories to be visualized and summarized in a more effective way. Participants in our study found visualizations of the Merge-Tree to be more enjoyable for summarization tasks than the visualizations of the DAG. The visualizations of the Merge-Tree model help users to more accurately summarize information about merges more quickly. We cannot definitively defend the original thesis statement as there are no other tools that attempt to provide information about how commits are integrated. Instead, we defend the revised thesis statement. The findings of the study show that the visualizations of the Merge-Tree are more effective at providing a conceptual understanding of how a commit is integrated and what other commits are integrated with it than tools that are currently available.

Bibliography

[1] Axosoft. Gitkraken. `www.gitkraken.com`, 2017.

[2] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 125–134, 2010.

[3] Richard Boardman. Bubble trees the visualization of hierarchical information structures. In *CHI '00 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '00, pages 315–316, New York, NY, USA, 2000. ACM.

[4] Michael Burch, Stephan Diehl, and Peter Weißgerber. Eposee - A tool for visualizing software evolution. In Stéphane Ducasse, Michele Lanza, Andrian Marcus, Jonathan I. Maletic, and Margaret-Anne D. Storey, editors, *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, Budapest, Hungary, September 25, 2005*, pages 127–128. IEEE Computer Society, 2005.

[5] Andrew H. Caudwell. Gource: visualizing software version control history. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 73–74. ACM, 2010.

[6] Per Cederqvist. *Version Management with CVS*. Free Software Foundation, 1.11.23 edition, 2008.

[7] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, 1993.

[8] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Software Eng.*, 31(6):446–465, 2005.

[9] M. D'Ambros, M. Lanza, and H. Gall. Fractal figures: Visualizing development effort for cvs entities. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6, 2005.

[10] M. D'Ambros, M. Lanza, and M. Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering*, 35(5):720–735, Sept 2009.

[11] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, Apr 2002.

[12] Jean-Daniel Fekete and Catherine Plaisant. Treemap visualization of the linux kernel 2.5.33, 2002.

[13] Daniel M German, Bram Adams, and Ahmed E Hassan. Continuously mining distributed version control systems: an empirical study of how linux uses git. *Empirical Software Engineering*, pages 1–40, 2015.

[14] Github. Github. `https://www.github.com`, 2017.

[15] Gitlab. Gitlab. `https://www.gitlab.com`, 2017.

[16] GNOME. Gitg. `https://wiki.gnome.org/Apps/Gitg/`, 2018.

[17] Brandon Heller, Eli Marschner, Evan Rosenfeld, and Jeffrey Heer. Visualizing collaboration and influence in the open-source software community. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 223–226, New York, NY, USA, 2011. ACM.

[18] kernel.org. gitk(1) manual page, 2015.

[19] Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, Jun 1947.

[20] Aras Moghaddam. Visualization of code contributions to linux kernel, 2010.

[21] M. Ogawa and K. L. Ma. code_swarm: A design study in organic software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1097–1104, Nov 2009.

[22] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Trans. Software Eng.*, 7(2):223–228, 1981.

[23] CollabNet VersionOne. Giteye. `https://www.collab.net/products/giteye`, 2017.

[24] Weixin Wang, Hui Wang, Guozhong Dai, and Hongan Wang. Visualization of large hierarchical data by circle packing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 517–520, New York, NY, USA, 2006. ACM.

[25] Charles Wetherell and Alfred Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, SE-5(5):514–520, Sept 1979.

[26] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.