Merge-tree: Visualizing the Integration of Commits into Linux

Evan Wilde, Daniel German
Department of Computer Science, University of Victoria, Canada.
Email: etcwilde@uvic.ca, dmg@uvic.ca

***Abstract*—With an average of more than 900 top-level merges into the Linux kernel per release, many containing hundreds of commits and some containing thousands, maintenance of older versions of the kernel becomes nearly impossible. Various commercial products, such as the Android platform, run older versions of the kernel. Due to security, performance, and changing hardware needs, maintainers must understand what changes (commits) are added to the current version of the kernel since the last time they inspected it in order to make the necessary patches.**

Current tools provide information about repositories through the directed acyclic graph (DAG) of the repository, which is helpful for smaller projects. However, with the scale and number of branches in the kernel the DAG becomes overwhelming very quickly. Furthermore, the DAG contains every ancestor of every commit, while maintainers are more interested in how and when a commit arrives to the official Linux repository.

In this paper, we propose the merge-tree, a simplified transformation of the DAG of the Linux git repository that shows the way in which commits are merged into the master branch of Linux. Using the merge-tree, we build *L*invis, a tool that is designed to allow users to explore how commits are merged into the Linux kernel.

***Index Terms*—Linux, git, data structures, tree data structures**

I. Introduction

Between 50k and 70k commits are added to the Linux kernel per year requiring maintainers of older versions of the kernel to sift through thousands of commits and merges with tools that are unable to filter and effectively visualize projects at the scale of the kernel. Older versions of the kernel are used in embedded systems and mobile phones; for security purposes, performance needs, and changing hardware requirements, maintainers must be able to understand the changes being made in the current version of the kernel in order to produce the necessary patches for the older versions of the kernel. Tools like Gitk use a directed acyclic graph (DAG) model of the repository, showing all commits and merges in chronological order by when the commit was authored, not by when it arrived in the official Linux repository.

This representation works in smaller projects; it enables users to see when changes are made, when these changes are merged, how each branch is interacting, and the point where a branch forks from the master branch. In large modular projects, like the Linux kernel, the DAG becomes a mess of merges and commits (Figure 1) losing its visual meaning. In some cases, the Linux kernel is simply too large for the system to generate a visualization; Github provides a DAG view for many projects, but is unable to display the visualization for projects at the scale of the Linux of the kernel (Figure 2). Between 60k and 70k new commits are created for the Linux project every year; according to our previous work [1], a commit takes a median of 30 days from the time it is authored until it arrives in the official repository. The snapshot of the kernel tomorrow may be different than the snapshot from today, containing new commits authored in the past; distinguishing these new commits from the commits in the snapshot from today is not trivial.

One major challenge with visualizing the arrival of commits to a repository is that Git does not store the date that a commit was merged into another branch, including the master branch. To complicate the problem, the DAG only has references to the ancestors of a commit (a model necessary for the operation of Git), but maintainers would prefer knowing the path a commit followed to reach the master repository. Tracing a path that any commit followed to the master repository would imply that for any given merge, it would be possible to know which commits were merged. A user could inspect the commits that arrived into the master branch within a given time-frame by checking which commits were merged during that time-frame.

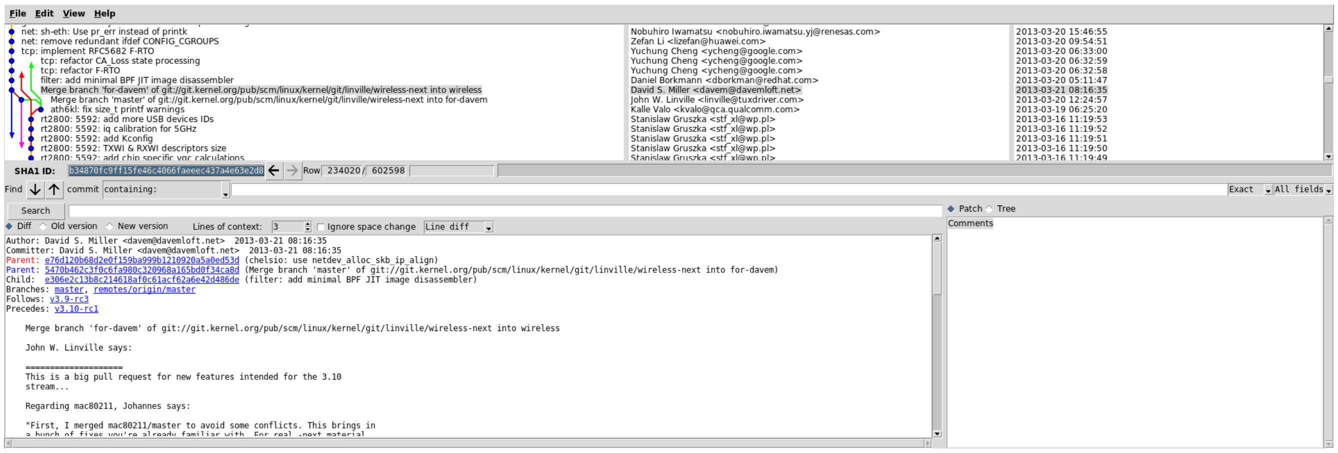
This paper makes two contributions; first, we describe a method of converting the DAG of the Linux repository into a tree, or *merge-tree* of the repository, that represents the path used by a commit to reach the master branch; second, we present a method to inspect and visualize the history of merges in the Linux project using the merge-tree model.

These methods and visualizations are implemented in a web-based tool called *L*invis[1]. Our visualizations and tool provide information about the location of any given commit or merge in its respective merge-tree, the files edited, the modules edited, and the commit message. *L*invis allows users to apply various filters, including the release version, along with a keyword or phrase from the log preview, the name of the author, or the commit ID. The user can request all merges made by Linus that contain a commit or inner merge that matches the search query, or just the commits and merges that match the query.

II. Merge-Tree model

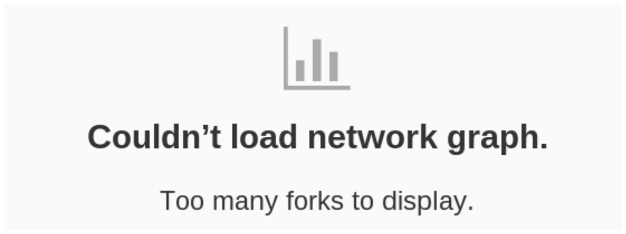
Git uses a directed acyclic graph (DAG) as its main data model. In this model, a commit has one or more ordered parents (ancestors), except the root commits of a repository

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



```
net: sh-eth: Use pr_err instead of printk                                       Nobuhiro Iwamatsu <nobuhiro.iwamatsu.yj@renesas.com>   2013-03-20 15:46:55
net: remove redundant ifdef CONFIG_CGROUPS                                      Zefan Li <lizefan@huawei.com>                          2013-03-20 09:54:51
tcp: implement RFC5682 F-RTO                                                    Yuchung Cheng <ycheng@google.com>                      2013-03-20 06:33:00
    tcp: refactor CA_Loss state processing                                      Yuchung Cheng <ycheng@google.com>                      2013-03-20 06:32:59
    tcp: refactor F-RTO                                                         Yuchung Cheng <ycheng@google.com>                      2013-03-20 06:32:58
    filter: add minimal BPF JIT image disassembler                              Daniel Borkmann <dborkman@redhat.com>                  2013-03-20 05:11:47
    Merge branch 'for-davem' of git://git.kernel.org/pub/scm/linux/kernel/git/linville/wireless-next into wireless    David S. Miller <davem@davemloft.net>    2013-03-21 08:16:35
      Merge branch 'master' of git://git.kernel.org/pub/scm/linux/kernel/git/linville/wireless-next into for-davem    John W. Linville <linville@tuxdriver.com>    2013-03-20 12:24:57
      ath6kl: fix size_t printf warnings                                        Kalle Valo <kvalo@qca.qualcomm.com>                    2013-03-19 06:25:20
    rt2800: 5592: add more USB devices IDs                                      Stanislaw Gruszka <stf_xl@wp.pl>                       2013-03-16 11:19:53
    rt2800: 5592: iq calibration for 5GHz                                       Stanislaw Gruszka <stf_xl@wp.pl>                       2013-03-16 11:19:52
    rt2800: 5592: add Kconfig                                                   Stanislaw Gruszka <stf_xl@wp.pl>                       2013-03-16 11:19:51
    rt2800: 5592: TXWI & RXWI descriptors size                                  Stanislaw Gruszka <stf_xl@wp.pl>                       2013-03-16 11:19:50
    rt2800: 5592: add chip specific vgc calculations                            Stanislaw Gruszka <stf_xl@wp.pl>                       2013-03-16 11:19:49

SHA1 ID: b34870fc9ff15fe46c4066faeeec437a4e63e2d8   Row 234020 / 602598
Find  commit containing:                                          Exact  All fields
Search
Diff   Old version   New version   Lines of context: 3   Ignore space change   Line diff        Patch   Tree
Author: David S. Miller <davem@davemloft.net>  2013-03-21 08:16:35                            Comments
Committer: David S. Miller <davem@davemloft.net>  2013-03-21 08:16:35
Parent: e76d120b68d2e0f159ba999b1210920a5a0ed53d (chelsio: use netdev_alloc_skb_ip_align)
Parent: 5470b462c3f0c6fa980c320968a165bd0f34ca8d (Merge branch 'master' of git://git.kernel.org/pub/scm/linux/kernel/git/linville/wireless-next into for-davem)
Child:  e306e2c13b8c214618af0c61acf62a6e42d486de (filter: add minimal BPF JIT image disassembler)
Branches: master, remotes/origin/master
Follows: v3.9-rc3
Precedes: v3.10-rc1

    Merge branch 'for-davem' of git://git.kernel.org/pub/scm/linux/kernel/git/linville/wireless-next into wireless

    John W. Linville says:

    ====================
    This is a big pull request for new features intended for the 3.10
    stream...

    Regarding mac80211, Johannes says:

    "First, I merged mac80211/master to avoid some conflicts. This brings in
```

Fig. 1: A view of the Gitk interface centered on merge b34870fc9ff15fe46c4066faeeec437a4e63e2d8 by Miller. Commits point toward their ancestors and there is no clear path from the commit to the merge with the master branch. Neither Gitk nor Git are capable of showing the commits in master.



Couldn’t load network graph.

Too many forks to display.

Fig. 2: Github Failure Message showing the DAG of the Linux Project and its relationship to other forks.

which do not have ancestors–Linux’s git repository has two such commits. Commits are divided into merge commits, merging two or more branches, and non-merge commits. Non-merge commits have only one parent, while merge commits have two or more. The order of the parents matter: the first parent is the branch in which the merge is being done, while the rest indicate the branches being merged (Git allows merging multiple branches simultaneously). The second parent is the first branch merged, the third parent is the second branched merged, etc.

Once a commit is created, it is never changed. Git allows operations to alter commits or reorder them, but it changes the commit ID in the process, effectively replacing it with a new commit. This makes commits unable to record the traversal of merges from the commit to the merge into the master branch.

A short example: assume the commits represented in Figure 3 show the sequence of events in a repository. In this case, commits are performed in various repositories and branches. The DAG representation of the commits is shown in Figure 4. Notice that the DAG loses information about the master branch and the repository that the master branch is part of. The merge-tree view of this DAG is visible in Figure 5. Note that the direction of the edges of the DAG have been inverted, instead of pointing from the child to the ancestors, it points from the ancestor to its successors, forming a path to the master branch. Also note that the DAG has been simplified, showing only a single edge on the path to master for any commit.

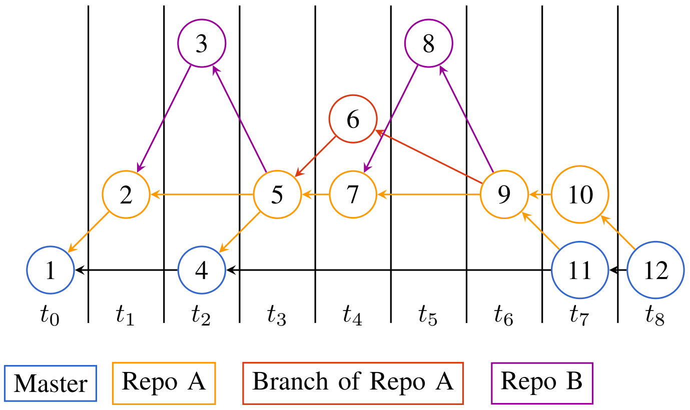


Fig. 3: Example of a sequence of events performed in different repositories. The horizontal axis represents time. Each horizontal section represents a different branch and/or repository. Each commit points to its ancestor.

A. *Computing the merge-tree of the DAG of Linux*

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. First, the master branch of Linux is maintained by Linus Torvalds, and only Linus has write access to it. We have verified this assertion in previous research [1]. We have developed a heuristic that is presented in Algorithm 1. In short, the algorithm first identifies the commits made directly to the master branch; whereafter it recursively determines the shortest path (in terms of time), using the DAG, from each commit to the master branch using the inverted DAG.

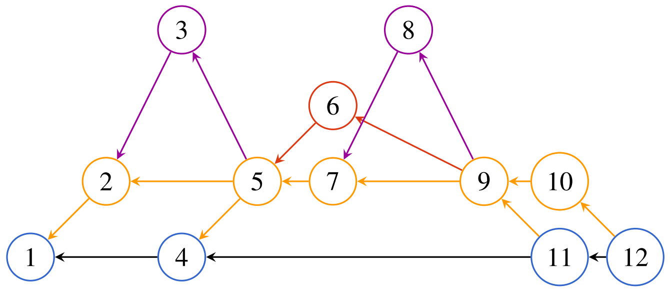


Fig. 4: DAG representation of the commits represented in Figure 3. The DAG loses information about which repository the commit is performed in and through which merges it has passed on its way to the master branch. The DAG does not even distinguish the master branch from other branches.

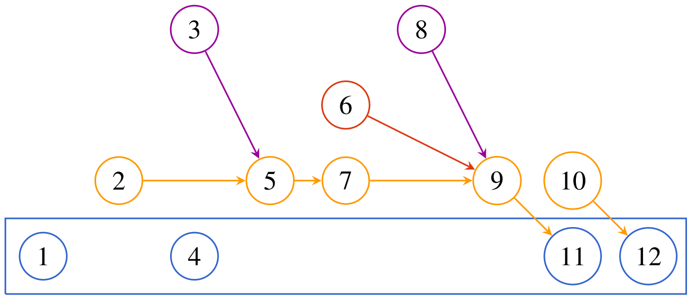


Fig. 5: Merge-tree view of the commits represented in Figure 3 showing the path they followed to reach the master branch. In this model the successors of each commit represents the path followed by that commit to reach the master branch.

B. Evaluation

Merges that do not have conflicts provide information to verify this heuristic. If a merge does not contain a conflict, it records a summary of the commits that it merges. See Figure 6 for an example. This summary contains a list of the first 20 non-merge commits in the merge, including their one-line log description, the full logs of the merge commits that merge this subset, and the total number of non-merge commits in the merge.

```
Merge: 8cbd84f fd8aa2c
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
[... edited for the sake of space]
md: split out md_rdev_init
md: be more careful setting MD_CHANGE_CLEAN
md/raid5: ensure we create a unique name for kmem_cache...
...
```

Fig. 6: 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.

Algorithm 1 Computing the merge-tree of Linux Git's DAG

```
function COMPUTEMERGETREE(DAG): tree
    # Compute the tree from the DAG of Linux repository.
    # Returns $Tree$, a graph containing every commit
    # in DAG with the path it followed to master.
    $head \leftarrow$ Head of master of git repository
    $master \leftarrow$ traverse DAG from head using
            first ancestor until reaching root
    $nodes(Tree) \leftarrow nodes(DAG)$

    function DISTANCE2MASTER(cid) : seconds
        # Helper function
        # Recursively compute shortest distance to master
        # setting cid's successor (next) in its way to master.
        # This function should be memoized. Otherwise it
        # would run in exponential time.
        if cid in master then
            return 0
        end if
        $d \leftarrow \infty$
        # Traverse the inverted DAG
        for $c \in children(cid, DAG)$ do
            if $c \in master$ then
                $d_1 \leftarrow commitTime(c) - commitTime(cid)$
            else
                $d_1 \leftarrow distance2Master(c)$
            end if
            if $d_1 < d$ then
                $next \leftarrow c$
                $d \leftarrow d_1$
            end if
        end for
        # $c$ is the commit that follows $cid$
        # in its way to master
        add edge $(cid, next)$ to $Tree$
        return $d$
    end function
    # Compute the distance for each commit
    # discarding result
    for $c \in nodes(DAG)$ do
        $distance2Master(c)$
    end for
    return $Tree$
end function
```

We used this information to evaluate the accuracy of the merge-tree model extracted from the DAG. The method we followed started with the extraction of the commit history up to July 20, 2016. We computed the merge tree of every commit until then. Since Linus Torvalds mostly does merging directly into master, we assumed that every merge by him is the root of a merge-tree. As described above, the log of a merge-commit usually contains the number of commits in the merge the first 20 summaries of commits being merged. We extracted merges by Linus Torvalds using the command `log -merges -author='Torvalds` and compared the number of commits according to the log with the number of commits in the merge-tree rooted in this commit. We also used the summaries of the commits found in the merge (not necessarily all—see above) to make sure those commits were in their corresponding merge tree. For example, for the merge in Figure 6 we would expect that the merge tree rooted at `8cbd84f` contains 24 commits, and the one-line summaries corresponds to commits in that merge-tree. We also inspected those with differences to make sure they were true errors. The results can be summarized as follows:

- Five merges were false-errors because their logs did not contain accurate information (were probably edited by hand). For example in `42a579a0f...` one commit summary was missing (the line was empty), in `c55d267` the summaries were reordered.
- The heuristic correctly identified that 79 of Linus merges (between Jun 7, 2014 and Jun 2, 2014) were made to a branch (not master). This branch was merged at (`3f17ea6d..` which contained 6809 commits.
- The heuristic worked perfectly until Sept 4, 2007, the earliest date that it could be verified. Before this date, and until Dec 12, 2006, merges did not include a summary of the commits they included, hence making it impossible to verify; during this period, however, we correctly identified the merges by Linus into master.
- Before Dec. 12, 2006 (1542 merges) our heuristic breaks due to the presence of a *foxtrot* commit (`c436688...`), which confounded the true master branch of a repository (see http://bit-booster.blogspot.ca/2016/02/no-foxtrots-allowed.html for a description of the issue).

In summary, of the merges after Sept 4, 2007, in 100% of them (16,860) our heuristic was correct. It failed in 1,542 commits before Dec. 12, 2006 and in 836 it appears to be correct (Dec 7, 2006 to Sept 4, 2007).

III. Visualizing the merge-tree of Linux

The goal of *L*invis is to simplify the navigation of the kernel commit information, specifically focusing on merges. This is done by leveraging the merge-tree to inspect how commits are merged on the path to the master repository.

A. Use cases

We designed *L*invis with two use-cases in mind, though a user may switch between the cases as they work.

Use-case 1: top-to-bottom approach
These are users that are maintaining a section of the kernel and would like to pick a merge (including all the commits that it merges) and merge it directly into their current repository. This is useful for reducing the amount of re-implementation work. For these users, it is important to have the ability to aggregate metadata about files and modules being effected by the merge. Also, it is important for these users to be able to navigate from the root of the merge-tree toward the leaves.

Use-case 2: bottom-to-top approach
These are users that start with a known merge or commit and would like to see what other changes are being made in commits that are in the same merge, including knowing the merge-tree they belong to. This is useful to see what other commits are related to the current commit and how they get collated into merges that eventually end in the master branch. This is primarily for maintainers that need to perform some specific cherry picking of commits. We must provide these users a mechanism for navigating from a single commit toward the master branch, allowing them to see other commits that might be related to their original commit.

B. Data Model

In our visualizations, we leverage the merge-tree model described in Section II. In this model each commit is either already in the master branch, or is part of a tree which is rooted in the merge that merged it into the master branch. Each commit, whether a merge or non-merge, has only one successor; the root of each tree has none as it was made by Linus Torvalds directly into the master branch. Non-merge commits contain the metadata for the changes made. This metadata includes the files changed, the lines added and removed from each file, the author, the date the commit was merged into the merge that led to being merged into the kernel, the date the commit was authored, the patch, and the commit log. Merges contain less metadata, only storing the author of the merge, the log, the commit date, and the author date, and potentially, changes necessary to address conflicts during the merge. The details of the model are outlined in [1].

IV. Design and Implementation

To navigate and inspect the merge-tree view of the kernel we created a web-based tool called *L*invis. Creating a web-based tool enables users to use the system without having to install additional software or store a large database, making it more accessible, more easily maintainable, and platform independent. *L*invis uses the following mechanisms to reach our goals of better navigation and better explanation of the selected changes.

- Filter by searching
- View files edited
- View modules
- Tree viewer

A. Searching

Searching (depicted in Figure 7) allows a user to filter commits and merges that are irrelevant. The search mechanism breaks down the results by release version. A user can further narrow down the search by specifying a range of dates in which such commits were merged by Linus into the master branch—not when the commits were created (author date and/or commit date). This distinction is important. We have observed commits that have taken years to arrive into the master repository after they were originally created.

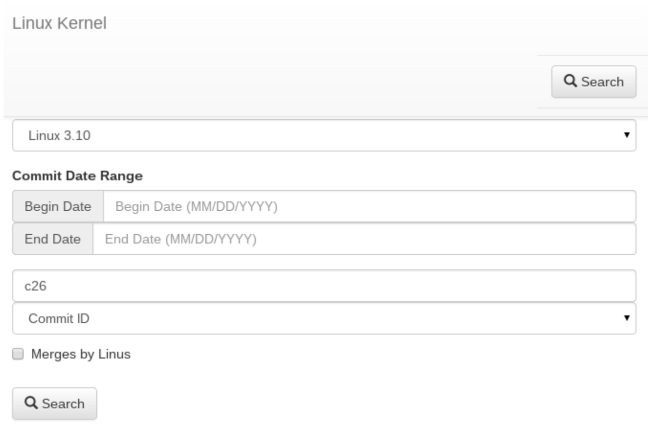
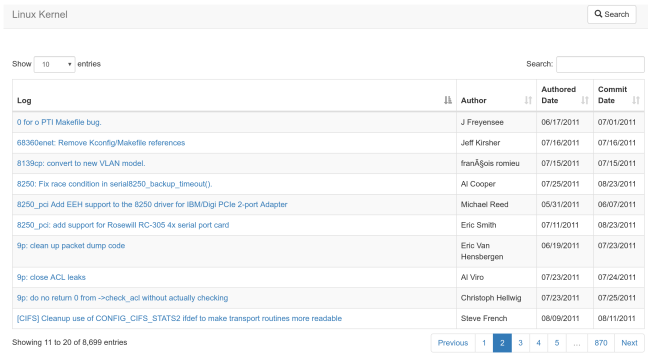


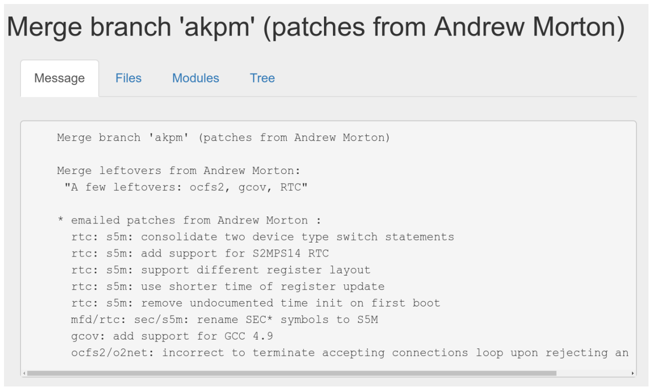
Fig. 7: Search View allows filtering commits that were merged in a given period, filtering by author, keyword, or commit ID.



Log	Author	Authored Date	Commit Date
0 for o PTI Makefile bug.	J Freyensee	06/17/2011	07/01/2011
68360enet: Remove Kconfig/Makefile references	Jeff Kirsher	07/16/2011	07/16/2011
8139cp: convert to new VLAN model.	franÃ§ois romieu	07/15/2011	07/15/2011
8250: Fix race condition in serial8250_backup_timeout().	Al Cooper	07/25/2011	08/23/2011
8250_pci Add EEH support to the 8250 driver for IBM/Digi PCIe 2-port Adapter	Michael Reed	05/31/2011	06/07/2011
8250_pci: add support for Rosewill RC-305 4x serial port card	Eric Smith	07/11/2011	08/23/2011
9p: clean up packet dump code	Eric Van Hensbergen	06/19/2011	07/23/2011
9p: close ACL leaks	Al Viro	07/23/2011	07/24/2011
9p: do no return 0 from ->check_acl without actually checking	Christoph Hellwig	07/23/2011	07/25/2011
[CIFS] Cleanup use of CONFIG_CIFS_STATS2 ifdef to make transport routines more readable	Steve French	08/09/2011	08/11/2011

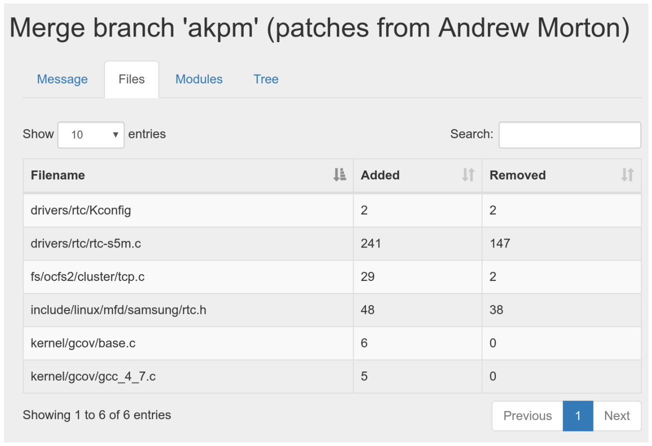
Showing 11 to 20 of 8,699 entries

Fig. 8: Search Results. Each table entry is a commit merged in the desired merged window.



```
    Merge branch 'akpm' (patches from Andrew Morton)

    Merge leftovers from Andrew Morton:
     "A few leftovers: ocfs2, gcov, RTC"

    * emailed patches from Andrew Morton :
      rtc: s5m: consolidate two device type switch statements
      rtc: s5m: add support for S2MPS14 RTC
      rtc: s5m: support different register layout
      rtc: s5m: use shorter time of register update
      rtc: s5m: remove undocumented time init on first boot
      mfd/rtc: sec/s5m: rename SEC* symbols to S5M
      gcov: add support for GCC 4.9
      ocfs2/o2net: incorrect to terminate accepting connections loop upon rejecting an
```

Fig. 9: Panel showing the Commit Message of a commit.



Filename	Added	Removed
drivers/rtc/Kconfig	2	2
drivers/rtc/rtc-s5m.c	241	147
fs/ocfs2/cluster/tcp.c	29	2
include/linux/mfd/samsung/rtc.h	48	38
kernel/gcov/base.c	6	0
kernel/gcov/gcc_4_7.c	5	0

Showing 1 to 6 of 6 entries

Fig. 10: Panel showing the files modified by all the commits that are part of this merge.

A user may then provide a search text, filtering by the author name, the commit ID, or keyword from the log. Any part of the author name may show up in the results, including searching by email address.

If the user is searching by commit ID, the ID can be specified by using any of its unique prefixes. For example, the commit `3f17ea6dea8ba5668873afa54628a91aaa3fb1c0` is returned when the user searches for a commit ID of `3f17e` in the 3.16 Linux kernel.

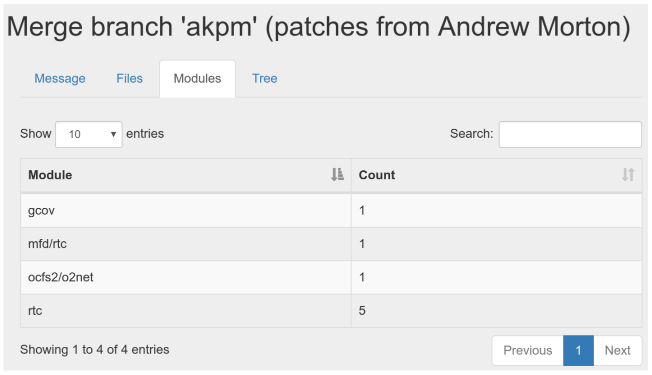
In the search results (seen in Figure 8), the user is presented with the one-line log message preview, the author's name and email, the date the commit was authored, and the date the commit was last committed.

Once a user selects a commit or merge to investigate, they are presented with a tabbed pane allowing them to view the full commit log, the files edited, the modules involved, and the merge-tree view.

The first tab displays the full commit log (Figure 9). From this, a user is able to see what they would see had they searched for the commit using Git log. This doesn't provide additional information to the other tools, but helps to complete the functionality of *L*invis. The commit log provides a user with the information about the content of the commit and who has signed-off on the commit to ensure that it is of good quality. The message for merges may contain a summary of the commits being merged. The information within these messages is highly variable, and is completely dependent on the author's style. As the user moves toward the root-level merge, the quality of these messages generally improves.

The second tab is the files tab (Figure 10). This tab provides information on what files have been edited, how many lines were added, and how many lines were removed in a given commit. For non-merges, this functionality is similar to the other tools available. Our tree-based design model allows us to extend this functionality to merges by aggregating information about all the commits that are children of the merge in the merge-tree, which other tools are unable to show. To find the number of lines added to a file in a merge, we take the sum of the lines added to that file in each of the children of that merge. We do the same for calculating the number of lines removed.

The modules tab (Figure 11) shows the modules that are contained within the commit. Modules are not natively recognized by Git, and are not going to be present in all repositories. In the Linux repository, authors put the module they are



Merge branch 'akpm' (patches from Andrew Morton)

Module	Count
gcov	1
mfd/rtc	1
ocfs2/o2net	1
rtc	5

Showing 1 to 4 of 4 entries

Fig. 11: Panel showing the modules changed by all the commits in this merge.

working on in the one-line summary of the log-message; for example: the log *gcov: add support for FCC 4.9* has updated the *gcov* module (the coverage testing tool of the kernel). We heuristically extract the module by taking all text in the log summary of commits until the first colon. Modules are logical partitions of the information in the kernel. Depending on where the author was working, modules can be general, such as “bluetooth” and “wireless”, or can be quite specific for individual hardware, such as “ath9k_hw” and “wl1251”. In a few cases, the author of a commit does not correctly follow this format and the heuristic approach fails. As with the Files panel, non-merge commits show their corresponding information, but for merge commits we aggregate all the modules changed in all the commits that are part of that merge. The output of this view is shown in Figure 11.

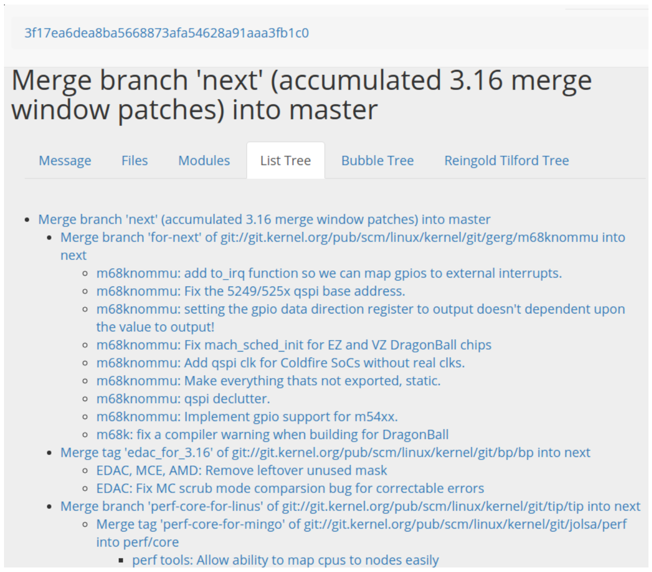
Finally, we have the Tree view tab. The tree view is designed for providing easy navigation of the commits within the merge-tree that is rooted in the current merge. It also provides a clear topological view of the merge and the submerges it includes. We have experimented with various tree designs to find a design that allows for easy navigation and visualization of both large and small trees. We discuss this panel in the next subsection.

B. Merge-Tree Views

The merge-tree view is what makes *L*invis unique to other tools that inspect the DAG of a Git repository. With it, a user can inspect how commits are merged on their way to the master branch. We have experimented with various types of trees:

1) List trees are a text-based representation of the merge-tree, and are easy to search and navigate.
2) Reingold-Tilford trees provide a clear visual representation of the tree structure of the merge-tree.
3) Bubble trees organize the data hierarchically by having the parent node contain the child nodes similarly to tree maps, but clearly showing the parent-child relationships between commits and merges.

1) List Tree: The list tree viewer (Figure 12) is in the form of nested lists, and is designed to more closely model the



3f17ea6dea8ba5668873afa54628a91aaa3fb1c0

Merge branch 'next' (accumulated 3.16 merge window patches) into master

- Merge branch 'next' (accumulated 3.16 merge window patches) into master
  - Merge branch 'for-next' of git://git.kernel.org/pub/scm/linux/kernel/git/gerg/m68knommu into next
    - m68knommu: add to_irq function so we can map gpios to external interrupts.
    - m68knommu: Fix the 5249/525x qspi base address.
    - m68knommu: setting the gpio data direction register to output doesn't dependent upon the value to output!
    - m68knommu: Fix mach_sched_init for EZ and VZ DragonBall chips
    - m68knommu: Add qspi clk for Coldfire SoCs without real clks.
    - m68knommu: Make everything thats not exported, static.
    - m68knommu: qspi declutter.
    - m68knommu: Implement gpio support for m54xx.
    - m68k: fix a compiler warning when building for DragonBall
  - Merge tag 'edac_for_3.16' of git://git.kernel.org/pub/scm/linux/kernel/git/bp/bp into next
    - EDAC, MCE, AMD: Remove leftover unused mask
    - EDAC: Fix MC scrub mode comparsion bug for correctable errors
  - Merge branch 'perf-core-for-linus' of git://git.kernel.org/pub/scm/linux/kernel/git/tip/tip into next
    - Merge tag 'perf-core-for-mingo' of git://git.kernel.org/pub/scm/linux/kernel/git/jolsa/perf into perf/core
      - perf tools: Allow ability to map cpus to nodes easily

Fig. 12: List tree view of merge 3f17ea6

tree view found in file browsers. This tree only contains the commits and merges that are within the subtrees of the current merge. A commit will never have any items in this tree as it is a leaf. To accompany the tree, we include breadcrumbs at the top of the page to enable a user to navigate both from the root to the leaves and from a leaf to the root. The last item in the breadcrumb list is the current commit, the previous item is the parent of the current commit, and the first item is the root merge into the kernel.

2) Reingold-Tilford Tree: The Reingold-Tilford tree [2] (Figures **??** and **??**) allows the visualization and navigation of the entire merge-tree in an intuitive representation of the tree. This illustrates a clear notion of root and leaves, and how to navigate in either direction. Some merge trees are very large, containing thousands of commits and merged. While the tree is capable of producing a visualization, it becomes far more difficult to understand. For example, the merge, 3f17ea6 performed by Linus Torvalds June 8 2014, contains 7217 commits and merges. [2]

The user is initially greeted with their current node centered on the screen. They are able to zoom the tree by scrolling the mouse, and clicking and dragging to pan the tree. They can see more details about a node by clicking on it, which will provide them a link to the specific page for that commit.

3) Bubble Tree: Bubble trees are useful for providing a clear visualization of wide, hierarchical data [3]. The tree structure is represented by the nesting of nodes; the largest circle is the root, containing all the other nodes. The smallest circles do not contain any nodes and therefore represent the leaf nodes.

Our implementation of the bubble tree (Figure 15) provides the user with a clear picture of where a commit is located in

[2]This commit can be inspected at http://li.turingmachine/org/commits/3f17ea6dea8ba5668873afa54628a91aaa3fb1c0

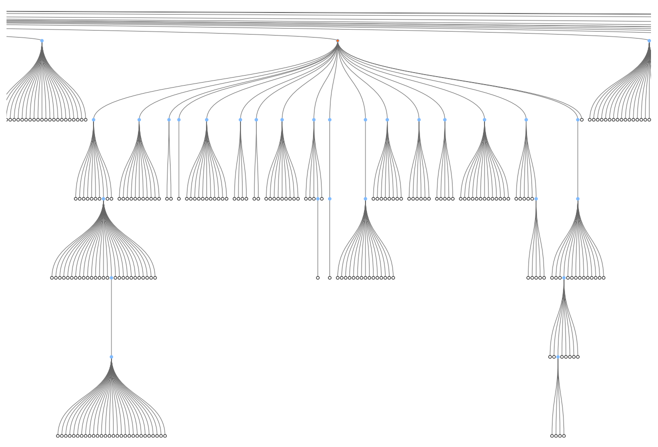


Fig. 13: Merge 4dc4226 is a subtree of 3f17ea6, for the power-management module of the kernel

the merge tree. We highlight the selected commit or merge in red. Non-selected merges are in a shade of blue determined by the depth of a node in the tree. The root is the lightest shade of blue, while the contained merges are progressively darker; the commits are white.

The bubble tree doesn't have an implicit way of providing additional information, placing any text near the nodes makes the tree impossible to read, so we include a separate pane in the web page. When a user hovers over a node, the pane shows additional information about the author and the commit message and a link to the detailed page for that commit or merge. If the user clicks on a node, the tree will zoom to that node and the information in the info pane will persist, enabling the user to click the link. This tree provides an easy mechanism for users to navigate from the root to the leaves and vice versa.

V. Implementation Details

The front-end uses asynchronous requests to gather the information for the trees and tables. This enables third-parties to implement new front-ends, trying other designs, though this interface may change in the future. The tree, file, and module information is accessible through http://li.turingmachine.org/data/tree/JSON/`<cid>`, http://li.turingmachine.org/data/files/JSON/`<cid>`, http://li.turingmachine.org/data/modules/JSON/`<cid>`, respectively.

The response for the tree is a single object. This object is the root node of the tree, and may contain an object of children sub-tree objects in the `children` field, which are in the same format as the root object. The children object uses the commit ID as the key.

The tree node objects have the following fields:

Field name	Data type	Description
`cid`	string	Commit ID
`name`	string	One-line log summary
`mlinus`	string	Root merge commit ID
`author`	string	Author name and email
`mnext`	string	Parent merge commit ID
`children`	object	Object of tree objects

The file responses contain only the files that the selected merge or commit works with. The response is a single object in the form of a tree. This object is the root of the tree, and represents the current merge or commit. If the current position in the tree is an inner node, the response will contain the child nodes in the `children` field, otherwise `children` will be an empty list. The format of file response is as follows,

Field name	Data type	Description	
`cid`	string	Commit ID	
`mnext`	string	Parent merge commit ID	
`children`	list	List of tree objects	
`files`	list of tuples	`Filename`	string
		`Lines added`	uint
		`Lines removed`	uint

VI. Discussion

From a developers point of view, the are two major disadvantages of the DAG model of Git: a) its edges point backwards, i.e. commits point to their ancestor, not to the commit that succeeds them on the path to being integrated; and b) the DAG contains many more edges that are necessary to understand how integration occurred. We addressed these two issues with the creation of the merge-tree model from the DAG of Linux. Effectively, the merge-tree recovers the details of how each branch—and the commits they contain—was merged into the master branch.

In our experience, no other Git repository reaches the level of DAG complexity that Linux has. In most Git repositories, most merges are not nested—most merges merge directly into the master repository, and these merges only merge a few commits. Even in these simplified cases, the merge-tree can provide a valuable summary of how commits are integrated into the master branch, specifically since the time of integration may be very different from the time the commits were authored.

The biggest challenge is computing the merge-tree of any given repository. It is likely that our heuristic for computing the merge-tree will not work with other repositories. The heuristic requires that there are no fast-forward merges into the master branch of the repository (a majority of repositories), and that there are no foxtrot merges (a practice that is starting to be considered desirable by git users[3]). We are able to use our heuristic with the Linux kernel because of the strict integration model imposed by Linus. To validate the results of the heuristic, we use the continuous mining technique described in [1]. Continuous mining of the repository requires foresight and planning.

[3]See http://devblog.nestoria.com/post/98892582763/maintaining-a-consistent-linear-history-for-git and https://developer.atlassian.com/blog/2016/04/stop-foxtrots-now/

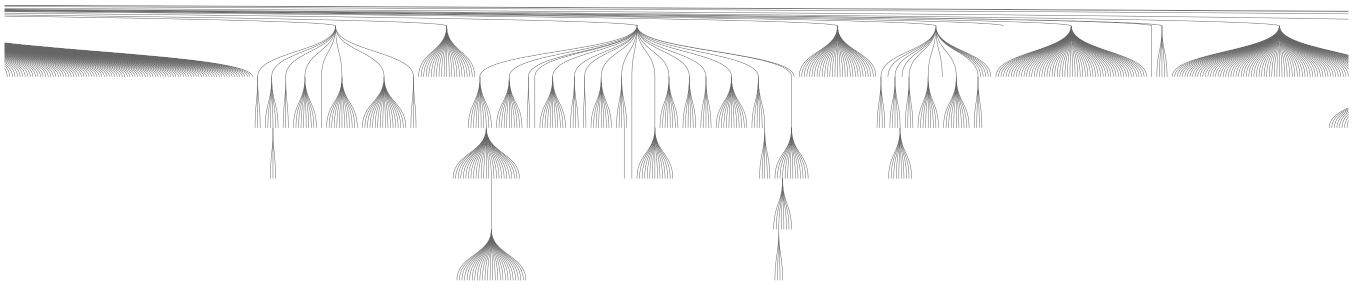


Fig. 14: Zoomed out view of the Reingold-Tilford tree for merge 4dc4226

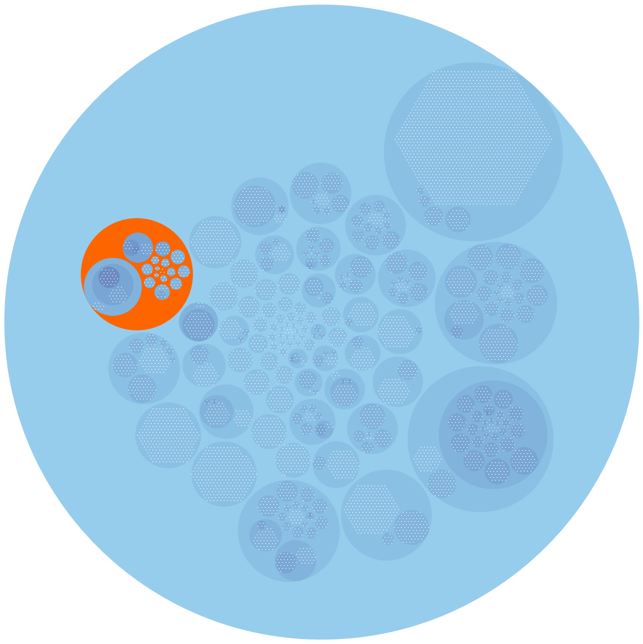
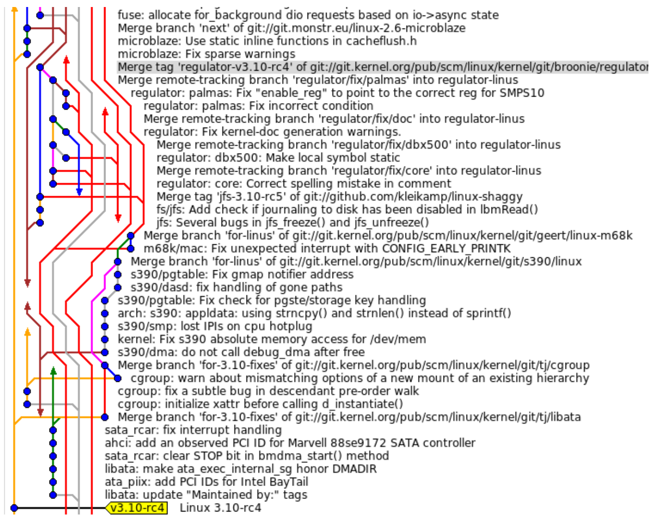


Fig. 15: Bubble Tree of merge 3f17ea6, merge 4dc4226 is the currently selected merge and is highlighted in orange.



```
fuse: allocate for_background dio requests based on io->async state
Merge branch 'next' of git://git.monstr.eu/linux-2.6-microblaze
microblaze: Use static inline functions in cacheflush.h
microblaze: Fix sparse warnings
Merge tag 'regulator-v3.10-rc4' of git://git.kernel.org/pub/scm/linux/kernel/git/broonie/regulator
Merge remote-tracking branch 'regulator/fix/palmas' into regulator-linus
  regulator: palmas: Fix "enable_reg" to point to the correct reg for SMPS10
    regulator: palmas: Fix incorrect condition
    Merge remote-tracking branch 'regulator/fix/doc' into regulator-linus
    regulator: Fix kernel-doc generation warnings.
      Merge remote-tracking branch 'regulator/fix/dbx500' into regulator-linus
      regulator: dbx500: Make local symbol static
      Merge remote-tracking branch 'regulator/fix/core' into regulator-linus
      regulator: core: Correct spelling mistake in comment
      Merge tag 'jfs-3.10-rc5' of git://github.com/kleikamp/linux-shaggy
      fs/jfs: Add check if journaling to disk has been disabled in lbmRead()
      jfs: Several bugs in jfs_freeze() and jfs_unfreeze()
    Merge branch 'for-linus' of git://git.kernel.org/pub/scm/linux/kernel/git/geert/linux-m68k
    m68k/mac: Fix unexpected interrupt with CONFIG_EARLY_PRINTK
  Merge branch 'for-linus' of git://git.kernel.org/pub/scm/linux/kernel/git/s390/linux
  s390/pgtable: Fix gmap notifier address
  s390/dasd: fix handling of gone paths
s390/pgtable: Fix check for pgste/storage key handling
arch: s390: appldata: using strncpy() and strnlen() instead of sprintf()
s390/smp: lost IPIs on cpu hotplug
kernel: Fix s390 absolute memory access for /dev/mem
s390/dma: do not call debug_dma after free
Merge branch 'for-3.10-fixes' of git://git.kernel.org/pub/scm/linux/kernel/git/tj/cgroup
  cgroup: warn about mismatching options of a new mount of an existing hierarchy
cgroup: fix a subtle bug in descendant pre-order walk
cgroup: initialize xattr before calling d_instantiate()
Merge branch 'for-3.10-fixes' of git://git.kernel.org/pub/scm/linux/kernel/git/tj/libata
sata_rcar: fix interrupt handling
ahci: add an observed PCI ID for Marvell 88se9172 SATA controller
sata_rcar: clear STOP bit in bmdma_start() method
libata: make ata_exec_internal_sg honor DMADIR
ata_piix: add PCI IDs for Intel BayTail
libata: update "Maintained by:" tags
v3.10-rc4   Linux 3.10-rc4
```

Fig. 16: Merge Dag View

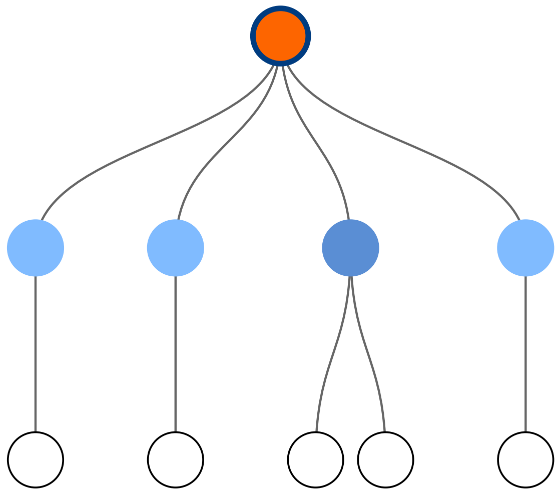


Fig. 17: Merge tree topology for the same merge as in Figure 16.

Because *L*invis leverages the merge-tree model of the kernel, it provides mechanisms and visualizations that other tools are unable to produce. It gives us the ability to see how commits are being merged into the master branch, allowing us to better understand how code is integrated into the kernel.

Our goal was to build a tool to enable maintainers to effectively navigate and browse the changes performed to the kernel over the period of a release. Achieving this goal includes removing information that does not pertain to the area of the kernel that the user is interested in. In Figures 16 and 17 we can visually compare the results returned from Gitk and our tool for the top-level merge "042dd60ca6dec9a02cefa8edd67de386e35755d6" from kernel version 3.10. Note that Gitk has no way to show or recognize that this is a merge into the master branch.

The log information to this merge-commit in both of these figures is identical, but the presentation drastically changes our ability to comprehend what we are seeing. The primary difference is the removal of irrelevant information. In Gitk (Figure 16), the visualization results include commits and merges from other components of the kernel, while our tool (Figure 17) only includes the results that are specific to the component of the kernel that we are interested in.

Even though this merge is relatively small, containing only four sub-merges, three of which only contain a single commit, and the last containing two commits, it is already hard to visualize in Gitk. With *L*invis we are able to immediately see what section of the kernel these commits and merges pertain to. The DAG view of this provides almost no explanation, furthermore, users must work to determine where this merge ends and the next one begins.

*L*invis further enhances our understanding by summarizing the files and modules that were edited in the entire branch being merged. We are able to determine that three files, "palamas-regulator.c" had two lines added and two lines removed, "dbx500-prcmu.c" had 12 lines added and 12 lines removed, and "core.c" had 5 lines added and 2 lines removed. Finally, we are able to determine that only the "regulator" module was modified in this merge and was modified by 5 commits.

VII. Future Work

There are still many areas that can be improved before the full potential of our model can be realized. Here, we outline various areas of the tool that still need more attention.

A. Files

There currently is no functionality surrounding searching by filename. Users falling under use-case 2 may know what file they are editing and try to determine how it may work with the other commits and merges in the module. It is possible that a third use-case may arise, where the user wants to determine all commits that effect a given file. In both cases, bottom-to-top approach is applied.

There is limited functionality to the presentation of the file information. At a minimum, the patches for the commits can be displayed. From there, the patches can be used to piece together parts of the file to generate a single patch at a given merge rather than displaying each patch individually. The patches can also be used for determining what kinds of changes were made, if the lines are being added, removed, or being replaced.

B. Authorship

Our model can aggregate more information than what we have implemented. The authorship information is important for licensing purposes, and we can show all authors contributing to commits in a merge and how many commits they contributed to the merge. We could go further, providing information about what files they edited, how many lines they added and removed, and more.

C. Evaluation

At this time, we have no evidence that our tool is able to improve the work-flow of maintainers. We believe that the tool is able to improve the work-flow and performance of maintainers because it provides cleaner mechanisms of visualizing and presenting the commit information. It is able to provide more relevant information, while removing information that is irrelevant to a given module or set of merges.

We can either perform user-testing to show how a users workflow changes, or we could have maintainers evaluate and critique the tool and use their feedback to determine if the tool meets our goals.

VIII. Related Work

Version control systems monitor the development lifetime of software projects. This makes the version control system vital in providing information about how a software project is being developed. To our knowledge, we know of no Git repository visualization tool that builds a tree that maps the path a commit follows to the master branch from the DAG provided by Git. This may be because more information is required to generate the merge-tree model than what is stored by Git. However, there has been a lot of work in providing visualizations of various repositories.

Many tools work to address the issues in communication between developers in inter-team collaboration work. Codebook [4] uses 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. Hipikat [5] is another tool with a focus on communication. Where Codebook focuses on developers working on a project, Hipikat is focused on enabling easier integration of new developers to a project by providing them with easily-searchable artifacts of the changes made. Codebook is useful for pairing a contributor with the original developer; however, the developer may not have worked with the piece of code in years. A Hipikat program may provide more information to the maintainer as it records the artifacts of why certain design decisions were made when they were made using other tools like Bugzilla and CVS. Neither tool is sufficient in meeting our goals to provide a summary of the topology of the kernel repository through a visual tree.

Most visualizers provide a visual presentation of a certain aspect of a repository. Fractal Figures [6] uses a unit square to represent a portion of a project, then partitions the square based on the proportion of an author's contributions to that portion of the project. EPOSee [7] and Evolution Radar [8] perform further analysis, determine which files are made together, and what changes are made over a sequence of commits, though the goals behind these projects is different.

Codebook, Hipikat, Fractal Figures, EPOSee, and Evolution Radar all work with data from CVS repositories. Our goal is to provide information about Git repositories, specifically the Linux master repository. Fewer tools are available for generating visualizations and summaries of Git repositories, potentially due to the DAG model used by Git.

Gource is a tool for providing an interactive timelapse of the state of a repository [9]. In the timelapse, it shows who contributes and what type of contribution a developer is making. These contribution types are one of, adding a file, removing a file, and changing a file. While the timelapse is interesting to watch, it does not provide any additional explanation of the changes actually being made, only the frequency that they are being made and who is making them.

The current industry standard tool for Git repository visualization is Gitk. The Gitk interface is built around the central DAG viewer. The DAG view displays the commits on their respective branches, the author of the commit or merge, and the date the commit or merge was authored. A user can select a single merge or commit to view additional information. For both merges and commits, Gitk will present the user with the Git log. If the user selected a commit, Gitk will also present the user with the patch information and the names of the files edited. Gitk is unable to provide the patches and filenames information in merges, as it is unable to aggregate commit information in merges.

Our tool is primarily aimed at presenting the hierarchical structure of the Linux git repository. We use tables for presenting the summarized information of the commits and merges, but this information could also be presented in a graphical form. Various graphical forms for displaying file and authorship data exist, the principal forms being matrix views, city scapes, bar and pie charts, and networks [10]. Any of these data visualization metaphors are applicable to our system.

IX. Conclusion

Our tool, *L*invis, shows promising results in working toward our goals, building a tool that can more easily navigate the repository, and provide clearer explanations of the changes made. The filtering by searches and the various tree visualizations enable users to easily navigate through the kernel, finding the commits and merges that pertain to what they are interested in. The tree visualizations are able to both serve as a navigation piece, providing simpler navigation through the kernel, but also demonstrate which commits contribute to which merges, providing further explanation over the DAG. The tree model further improves the explanation by enabling the tool to aggregate metadata at the merges instead of requiring a user to manually aggregate the information. Our tool currently aggregates file information and the modules edited, but can be extended to include authorship information and aggregated patch information.

We use a heuristic method to generate the tree from the repository. The heuristic requires that the repository does not have fast-forward merges into the master branch or foxtrot merges, which obfuscate the master branch. In either case, the heuristic will break and result in a tree that does not accurately represent the repository.

These may pose as significant drawbacks to smaller projects; however, with a proper merge discipline, the heuristic is able to correctly convert the DAG into a merge-tree. The importance of clear visualizations in large projects outweighs the costs of enforcing a clean merge discipline. The tree-based model appears to have the ability to provide a clean visualization and more detailed explanations of the changes in the Linux kernel.

References

[1] D. M. German, B. Adams, and A. E. Hassan, "Continuously mining distributed version control systems: an empirical study of how linux uses git," *Empirical Software Engineering*, pp. 1–40, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-014-9356-2

[2] E. M. Reingold and J. S. Tilford, "Tidier drawings of trees," *IEEE Trans. Software Eng.*, vol. 7, no. 2, pp. 223–228, 1981. [Online]. Available: http://dx.doi.org/10.1109/TSE.1981.234519

[3] R. Boardman, "Bubble trees the visualization of hierarchical information structures," in *CHI '00 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '00. New York, NY, USA: ACM, 2000, pp. 315–316. [Online]. Available: http://doi.acm.org/10.1145/633292.633484

[4] A. Begel, Y. P. Khoo, and T. 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*, 2010, pp. 125–134. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806821

[5] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Trans. Software Eng.*, vol. 31, no. 6, pp. 446–465, 2005.

[6] 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*, 2005, pp. 1–6.

[7] M. Burch, S. Diehl, and P. Weißgerber, "Eposee - A tool for visualizing software evolution," in *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, Budapest, Hungary, September 25, 2005*, S. Ducasse, M. Lanza, A. Marcus, J. I. Maletic, and M. D. Storey, Eds. IEEE Computer Society, 2005, pp. 127–128.

[8] M. D'Ambros, M. Lanza, and M. Lungu, "Visualizing co-change information with the evolution radar," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 720–735, Sept 2009.

[9] A. H. Caudwell, "Gource: visualizing software version control history," in *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*, W. R. Cook, S. Clarke, and M. C. Rinard, Eds. ACM, 2010, pp. 73–74. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869554

[10] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster, "Visualizing software changes," *IEEE Transactions on Software Engineering*, vol. 28, no. 4, pp. 396–412, Apr 2002.