First-Class Branching Facilitates Collaboration

Earl T. Barr† Christian Bird‡ Peter C. Rigby⋆ Abram Hindle†
Daniel M. German⋆ Premkumar Devanbu†

†Department of Computer Science
University of California, Davis, USA
{etbarr,ajhindle,ptdevanbu}@ucdavis.edu

‡Microsoft Research
Microsoft, Redmond, USA
cbird@microsoft.com

⋆Department of Computer Science
University of Victoria, Victoria, CA
{pcr,dmg}@cs.uvic.ca

ABSTRACT

The adoption of third generation version control systems (3gVC), such as Git and Mercurial, in open-source projects has been explosive. Why is this? How are these 3gVC's being used? This new generation supports two important new features: *distributed repositories*, and *first-class branching and merging* where branching is easier, faster, and more accurately recorded. Pundits and the blogosphere have largely assumed that *distribution* has driven the rapid growth of 3gVC. Using a mixed methods approach (interviews and data analysis), we show that enhanced branching, **not** the support for distribution, has driven 3gVC adoption. We then studied how developers are using branching. We find that projects that have made the transition are using branches much more heavily to enable natural collaborative processes: first-class branching allow developers to collaborate on task in a *highly cohesive* branch, while enjoying reduced interference from developers working on other tasks, even if those tasks are *strongly coupled* to theirs.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Version Control; D.2.8 [**Metrics**]: Process Metrics; D.2.9 [**Management**]: Life Cycle, Programming Teams, Software Configuration Management

Keywords

Configuration management, empirical software engineering, collaboration

1. INTRODUCTION

Collaboration is fundamental to software engineering. Most interesting problems need the coordinated effort of teams of programmers working concurrently to solve them. Well-managed, collaborative teams aim to parallelize as many tasks as possible, while controlling co-ordination overhead. Typically, work is split up into subtasks that can be assigned to smaller teams or even individuals. These subtasks, however, are often interdependent. With well-designed, modular systems, these interdependencies are manageable (by dint of modular software design with low coupling), allowing developers to break off and work in isolation and in parallel, whilst also integrating their work periodically. Another example of task parallelism is the simultaneous and independent exploration of alternative and competing design choices; the final choice emerges from among the alternatives.

Version control (VC) is tool support for collaborative software processes; developers, especially those working on large software engineering projects, have long relied on it to work in parallel on (relatively) well-separated tasks [34] and then to integrate their results. VC tools have been in use for several decades now, at least since 1975 [33]. There are publicly available VC repositories for open source projects, which have provided a valuable source for studies of collaborative practices in software engineering [25, 44]. Recently, a new generation of systems supporting distributed version control (DVC) and first-class branching has transformed the use of VC. These systems, and the practices that have emerged from their use, are the subject of this paper.

In April, 2005, development simultaneously began on two open source DVC systems, git and mercurial. Their popularity has exploded, and by 2011, a large portion of open source projects have already migrated to a DVC. According to Debian (a Linux distribution), of the 55% of projects that report their VC (9,132 projects), 44% (3994 projects) use DVC [42], indicating that it has achieved widespread acceptance and adoption. VC has a profound effect on workflow, and adoption of a new VC is not a trifling matter [16]. Indeed, changing VC can be a major disruption, as evidenced by the amount of discussion surrounding decisions to change, the work required to move from one to another, and the change in project workflows, all of which we have observed in OSS projects (e.g. GNOME's move to git [32], Python's move to mercurial [9], and the KDE project created solely for its adoption of git [17]). Why did projects rapidly adopt DVC?

Pundits and the blogosphere claim that support for distributed, as opposed to centralized, development is the root cause of this rapid transition [10, 27]. We have observed something different. The vast majority of these projects do not appear to be making much, if any, use of distribution. We have found through surveys that most have continued to use a *single, centralized* repository. Of the 60 projects whose VC use we surveyed, all but Linux continue to use a centralized model with single public repository, except the xemacs and gnome projecs which publish two repositories. In contrast, our findings show an extensive increase in the use of branching (§5.1).

We contend that advances in the support for branching, not distribution, have driven this transition and are the most significant contribution of the 3^{rd} generation of VC (3gVC). In this new generation, *first-class branching* is an elegant, easy-to-use, and well-tracked operation that facilitates collaborative software processes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

We observed significant positive effects on project workflows arising from the use of 3gVC systems. First, the excellent support for branching in 3gVC does in fact lead to more prolific use of branching (§5.1). Certainly, this would not occur if the integration that follows the branch, *i.e.* merging, were unduly painful. Thus, we investigate how 3gVC has reduced merge pain in §5.2. Two questions then arise "what are the characteristics of these branches?" and "how are developers choosing to use them?".

During development, we find that developer teams create branches in 3gVC to construct a cohesive set of changes (§5.3). We also find that branches are chosen to work in "splendid" isolation, protected from interference from others working on the same files (§5.4). This isolation is "splendid" because it protects developers from contending with the interruptions of constantly dealing with the changes being committing by developers working on other tasks. It defers that integration work until after the work on the branch is complete, the branch is being merged and the developer can concentrate on integration.

The transition to 3gVC was a volcanic event, rapid and transformative. We embarked on this study to understand the conditions that preceded it and the new working styles have emerged as a result. By elucidating the software processes that projects have adopted since the transitions, we have tried to shed light on the impediments that stymied them. Understanding these impediments may help us identify others. Once found, we part company with the volcanologist: we will try to precipitate the next event and improve software processes. Our contributions follow.

- We find that, contrary to conventional wisdom, the most popular aspect of the 3gVC are not DVC features, but rather first-class branching, for three reasons. First, most projects continue to use a single repository, as discussed above. Second, the use of branching has exploded (RQ1: posed in §3.1 and answered in §5.1). Finally, the pain of merging has been alleviated (RQ2: posed in §3.1 and answered in §5.2).
- We find that first-class branches 1) are used to undertake cohesive development tasks (RQ3: posed in §3.2 and answered in §5.3) and 2) are strongly coupled: many interruptions and high co-ordination overhead could result in the absence of branching (RQ3: posed in §3.3 and answered in §5.4).
- We define three new measures: branch cohesion (§5.3) and two types of task interruption that occur when integration work intrudes into development — intervening files and distraction (§5.4).

2. BACKGROUND

Large-scale software development processes make intimate use of VC tools, to help handle co-ordination and conflict that invariably arise in distributed, collaborative work. Before we discuss the process implications of modern VC tools (such as Git or Mercurial), we present some background on VCs. Expert users of Git or Mercurial may wish to skip this section, or refer back here "lazily" when needed. At the outset, we note that the change history recorded in VC supports three crucial functions: *sharing* changes, *rolling back* to previous versions, and enabling a *historical understanding* how each part of the system came to be in a particular state.

The evolution of VCs is marked by *increasing fidelity of the histories they record*. First generation VC, such as RCS, record the history of individual file changes. This enabled rolling back changes to a single file, or understanding its specific change trajectory. Second generation VC (2gVC), such as SVN 1.4, stored sets of file changes committed together (*i.e.*, transactions) in its history. This allows a related set of changes to be rolled back, and also enables the conjoint history of a set of related files to be reconstructed. 2gVC also supports branches, or sequences of sets of co-committed file changes; however branching and re-merging in 3gVC is inconvenient, and not recorded in the history.

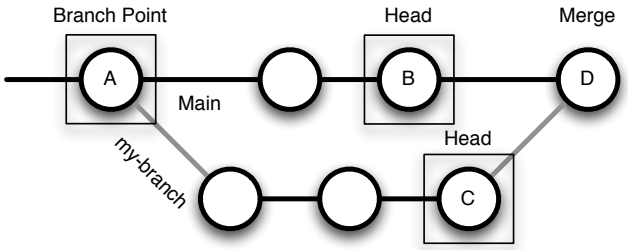


Figure 1: 3-way merging.

```
$ svn copy trunk branches/my-branch
$ svn commit -m "Created␣private␣branch."
$ cd branches/my-branch
  <work in my-branch>
$ cd ../../trunk
$ svn merge -r 100:200 \
  http://test.com/repo/branches/my-branch
```

Figure 2: non-first-class branching (SVN 1.4 syntax).

Branching and merging in 2gVC and 3gVC use the same underlying 3-way merge algorithm[1]. Figure 1 depicts the history (time flowing from left to right) of creating `my-branch` from `main`, work occurring in both, then merging the `my-branch` back into `main`. To create the merge commit, we must discover the changes made on each branch, *i.e.* `main` and `my-branch`, then compare those differences to produce a patch that applies `my-branch`'s changes to `main`. The changes on `main` are the differences between the branch point A and `main`'s head B; the changes on `my-branch` are the differences between branch point A and `my-branch`'s head C. Thus, computing a 3-way merge depends on knowing the least common ancestor (LCA) of the two branches, here commit A.

3gVC provides sophisticated branching and merging features; we We believe that it is precisely these features that have driven their explosive growth. This growth, and the emerging collaborative practices adopted by teams that leverage this feature, are the core subject of this paper. So we dwell upon branching and merging in the rest of this section. 3gVC improves branching and merging in 3 ways: under 3gVC, these operations combine 1) an improved interface with 2) an efficient time and space implementation while 3) preserving when branches and merges occur in the history.

Improved Merging Interface Prior to release 1.5, SVN did not support first-class branching and merging; we realize the scenario in Figure 1 using SVN 1.4 syntax in Figure 2. By changing directory to trunk before merging, we make trunk the implicit merge target: the parameters to the '-r' switch identify the branch point and the head of main. Figure 3 contains the equivalent commands in git[2].

These examples highlight the key difference between non-first-class and first-class merging — manually identifying vs. computing

[1] 3-way refers the three commits needed to define the relationship between related branches: their branch point, or least common ancestor, and the heads of the two branches. It is perhaps an unfortunate name, however, when thinking in terms of the two branches merging; nonetheless, we use it here because it is the industry standard.

[2] We have used git as our example, but other recent version control systems such as mercurial support first-class branching and merging. In addition, since its 1.5 release in June 2008, subversion tracks information (such as the LCA) that makes merging branches much less painful.

```
$ git branch my-branch
  <work in my-branch>
$ git checkout master
$ git merge my-branch
```

Figure 3: First-class branching (Git syntax).

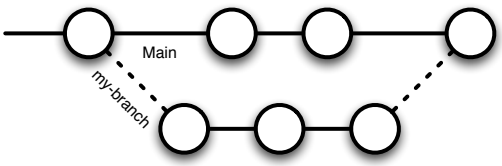


Figure 4: History-preserving branching and merging: Under 2gVC, the dashed edges are not recorded; under 3gVC, the dashed edges are recorded and indistinguishable from edges along a branch.

the least common ancestor, needed for the 3-way merge. The hallmark of non-first-class branching is the requirement to manually track the LCA (revision 100 in the merge command in Figure 2), in the commit message by convention [30]. Failure to track the LCA effectively prevented merging because discovering it was often tedious prior to 3gVC. Cherry-picking is the practice of copying a critical change, but not all the changes, from one branch into another. Cherry-picking introduces shared commits before the LCA, thus complicating finding it. 3gVC computes the LCA, even in the presence of shared commits. Git, for instance, uses content-addressability [36]: it identifies each commit by its hash, so it automatically discovers and handles shared commits.

Such user interface issues can be crucial. In an industrial study of tool adoption, Jacky *et al.* [16] found that the user interface to configuration management tool "plays a very important role in tool adoption ... Software engineers are submitted to time pressure and will not use a tool if they can't get easily the result they need."

Efficient Branching and Merging 1gVC and 2gVC distinguish centralized *repositories*, where version histories reside, and local *working copies*, which are just current snapshots under change. 3gVC effaces this distinction, providing a local repository with local version history; this maintains local evolution history, *sans* interactions with a central repository. This allows branch creation and commits to that local branch entirely unmediated a remote repository. In the non-first-class branching in a 2gVC as illustrated above, the central remote repository is intimately involved in both sharing the branch and committing to the branch. Under 3gVC, a branch is no more than an identifying tag on a commit; creating a branch is trivial when contrasted with the explicit copying of the working copy, as in the 2gVC example above. Although local history is a side-effect of DVC, CVC can support it, as subversion currently partially does by caching the state of HEAD at a client.

History-preserving Branching and Merging Consider Figure 4. With 2gVC, the first commit to a fresh new branch does not record its parent, leading to the merge complications described above. At the other end of a branch's lifecycle, when it is merged, there is another loss — the merge commit does not track its parent in the source branch. The dashed edges in Figure 4 depict these untracked, and forgotten relationships. In 3gVC, a child always tracks its parents, across both branches and merges[3]; for 3gVC, the dashed edges are indistinguishable from the edges along a branch. History preservation is important when trying to find the commit that introduced a bug, as when bisecting. Smaller, more focused commits, in the context of their original branch are often easier to understand, which can facilitate approaches like delta debugging [43].

These advanced branching and merging features are crucial to the burgeoning popularity of 3gVC. In the next section, we discuss in more detail how they enable some common workflows, and present a conceptual framework for analyzing the work practices enabled by these features.

[3]All parents in the case of n-way merging, which we do not discuss further. Git is not history preserving when the least common ancestor is the head of the merge target. In this case, the target branch is "fast-forwarded" with the commits from the source branch and no merge commit is created. Git partisans might argue that no interesting history is lost in this case.

3. THEORY

Large-scale, collaborative development is always a trade-off between delegation and co-ordination. One seeks to delegate subtasks to maximize division of labour while moderating of required co-ordination between tasks. Typically large team development exhibits *episodic collaboration*: people work individually on their tasks for a while, share their work to check proper integration with others' work, and then go back to individual tasks, and so on. While software design [2] is a crucial factor in task structure and assignment, anther critical factor is the VC facility. VC tools enable sharing, co-ordination and recording of changes.

Consider two very common collaborative workflows: *1) task parallelization*, and *2) experimentation*. In the first, a task is divided, perhaps using modularization, into individually cohesive subtasks, each assigned to several co-operating developers. These developers ideally work separately, and in parallel, then merge their results. In the second scenario, several different alternative implementations (perhaps of a module) are tried, usually by different developers working relatively independently, and finally one solution is chosen.

Branching assists these workflows in critical ways. First, branches allow individual developers to easily undertake each task in a separate, isolated, yet still tracked, version history. While the task *per se* may be cohesive, it may involve modules that being concurrently worked on by others engaged in other tasks. The branch allows developers to work in relative isolation, confident in the fact that branching is not resource-intensive and also that subsequent merging is relatively simple. Finally, the fact that histories are preserved in 3gVC allows one to freely experiment with alternative implementations in branches, confident that the branch-preserving history allows one to easily roll back to a convenient earlier snapshot if/when one alternative is abandoned.

3.1 Branching Use and Ease-of-Use

The discussion above clearly indicates that branching is much easier in 3gVC than in 2gVC; thus we can expect that branching is more popular in 3gVC; this a natural, first research question:

> **Research Question 1:** Are branches used more often in 3gVC?

In §5.1, we demonstrate that branching is indeed much more heavily used in 3gVC. Beyond the fact that branches have proliferated in 3gVC, we also investigate other evidence of the utility of first-class branching and merging and ask

> **Research Question 2:** Is there less evidence of "merge pain" after transitioning to 3gVC.

Here, *merge pain* is defined as the difficulty, effort and obstacles one faces when one uses merging in a VC.

Having shown that branching and merging are easier to use and more prevalent, we turn to investigating their benefits. To support

our theory, we provide evidence that branches are indeed highly cohesive, measured against a background of branchless, concurrent development. If branches simultaneously exhibited high cohesive and low coupling, branching would not be needed to isolate developers who could simply share a mainline without fear of interference. To show the importance of isolation, we quantify the degree to which branches isolate developers from interruptions. We discuss these benefits of first-class branching in greater detail below.

3.2 Cohesion

Cohesion measures the extent to which related aspects of an artifact are kept together and unrelated aspects are kept out. If branches are being used to isolate development in the context of episodic collaboration, we should find them to be cohesive; they should encapsulate related changes. A highly cohesive branch is easier to revert if there is a problem and to backport to an older release with which it shares a common ancestor. Branch cohesion is so desirable that git includes a feature, called *rebasing*, for "perfecting" the history of a branch and to combine, break apart, remove, and modify commits so that the history is easier to understand [18].

Research Question 3: Are first-class branches cohesive?

3.3 Coupling and Interruptions

Developing a new feature often requires making changes to modules that may be coupled to other modules. If different features, under simultaneous construction by different developers, affect coupled modules, the two tasks are coupled and may need co-ordination. One developer's work can cause other developer's code to become unstable. Ideally, uninvolved developers should be insulated from these changes until the feature has achieved some degree of stability. At the same time, a developer working on a new feature should still have access to VC to commit incremental changes, and rollback, as necessary. Berczuk [3] makes this point in his discussion of *configuration management patterns*, where he argues that developers should checkpoint changes at frequent intervals to a location separate from the "team version control," and that only tested and stable code should be integrated. When the feature is ready, its integration must not be too difficult or the productivity gained from working on an isolated branch is lost. Indeed, Perry *et al.* [28] claim that tool support for integration is important because "integration too often is painful and distracting" and because development lines diverge when parallel development goes on too long.

When branches are not used, all changes occur on the mainline and a developer may need to merge and integrate changes that are unstable and transitory or only tangentially related to her work. The attendant interruptions can slow development. When first-class branches are available, a developer can isolate related changes, such as those changes required to implement a feature, into branches to minimize such interruptions.

Integration interruptions are a form of task interruption. Prior literature has shown that task interruptions have a serious effect on developer productivity. Recovering from interruptions can be difficult and time-consuming: developers must mentally juggle goals, decisions, hypotheses, and interpretations related to their task, or will risk inserting bugs. In a study at Microsoft [19], 62% of developers said that recovering from interruptions is a substantial problem. Van Solingen [37] found that interruptions are the most problematic when they occur during programming activities, which is most often the case when a developer is checking in changes or updating their working code base. DeMarco observed that resuming after an interrupt often takes at least 15 minutes [13]. Parnin *et al.* [26] instrumented Visual Studio and Eclipse to observe the time taken to resume development tasks. While they found some strategies for mitigating the effects, developers were able to begin editing within a minute of starting a task only 10% of the time and took over 30 minutes in 30% percent of the cases. While these papers consider the effect of interruptions in broader terms, they do support the claim that task interruptions diminish productivity.

Research Question 4: Do (first-class) branches in 3gVC protect developers from possible interruptions due to concurrent work in other branches?

4. METHODOLOGY

Todo 4-1 (for Chris): *Discuss and extol mixed methods.*

In an effort to understand what has motivated the rapid transition to 3gVC from an operational point of view, we observed the development activities in projects that switched to 3gVC and interviewed a number of lead developers from these projects regarding their switch. Following these interviews, we gathered data from the development history of these projects and quantitatively evaluated hypotheses based on their responses.

Interviewing project leaders was critical in understanding why people switched to 3gVC, the perceived benefits and drawbacks of the switch, and (in cases where the projects have used 3gVC for some time) how it has affected the policy and development process of the projects. The data mining of the VC history and the email lists (communication) allowed us to provide less biased evidence of the effects of 3gVC. We interleave quotations from interviews and numerical findings from data mining to triangulate and provide a balanced perspective.

We conducted semi-structured interviews of four projects and six people. Semi-structured interviews make use of an *interview guide* that contains general groupings of topics and questions rather than a pre-determined exact set and order of questions [20]. Semi-structured interviews are often used in an exploratory context when there are clear research questions [20, 40]. The responses from these interviews help develop hypotheses and focus quantitative analysis. We extracted themes from the interviews using a modified version of Creswell's guidelines [12] for coding. The interview guide that we used can be found at `http://janus.cs.ucdavis.edu/~cabird/vcstudy/vcinterviewquestions.pdf`[4]. We minimally copy-edited the quotes for readability. We eliminated false starts and superfluous crutch words; we used standard notation, delimiting clarifying comments with brackets and marking the suppression of unnecessary phrases with an ellipsis [20].

For the quantitative mined data, we either developed metrics or modified existing ones to best examine the impact of 3gVC in the context of our dimensions. The data used, the definition of the metric, and threats to validity are discussed in the section in which the metric is used.

We chose to examine 60 projects that had transitioned to 3gVC. These projects were drawn from lists of projects using DVC on Wikipedia and GitWiki and include such notable projects as Wine, Samba, Perl, Ruby on Rails, and the Glasgow Haskell Compiler. These projects vary in age from 21 years (in the case of Perl) to 6 months (pthreads-stubs in X.Org) with a median of 4.5 years. The number of contributors as recorded by the repositories ranges from 1462 (Wine) to 1 (dri2proto in X.Org). The commits to these projects number from 139,187 (Samba) to just 6 (pthread-stubs in

[4]At the request of the participants, the interviews in their entirety are confidential.

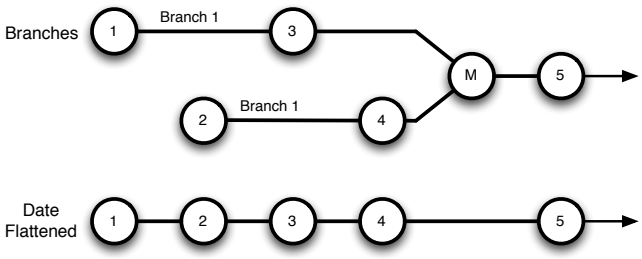


Figure 5: Branches flattened by date.

X.Org). As such, our selection of projects for analysis spans a broad spectrum of OSS projects in terms of size, age, and development activity. All projects have used DVC for at least 5 months at the time of this study; the majority of them for over one year.

We use Linux to evaluate hypotheses and questions regarding advanced 3gVC usage because the Linux kernel project has never used a 2gVC and its developers are generally very experienced with 3gVC branching. Linux started using git in 2005; we have 3.5 years of Linux VC data and the corresponding data from Linux kernel Mailing List (LKML). Over this period, there were 4K developers, 118K commits, and 443K mail messages for Linux.

5. EVALUATION

In this section, we answer each of our research questions. To begin, we introduce a technique we use to date-flatten a branch and merge preserving history in order to project the concurrent sequence of commits in that history onto a single branch. We use this date-flattened branch to capture baseline of branchless, concurrent development that underlies a merge-preserving history. To measure the cohesion (§5.3) and isolation (§5.4) of 3gVC branches, we compare them against simulated branches drawn from the date-flattened history.

Date-flattening history Our approach is to *flatten* a 3gVC history onto a single workflow line to produce the commit sequence D, the system history's branch-free twin. An initial, parallel, and independent workflow against a single mainline is date-ordered, or precisely D. D does not model a CVC workflow, which has adapted to handle the conflicts and problems that arise from naïvely working without coordination against a single mainline. D does, however, capture the work to resolve conflict and avoid integration interruptions that is unobservable in a CVC history, like the reworking of a large patch into more manageable chunks on a project's mailing list [41].

Figure 5 illustrates our flattening technique. At the top of the figure is the original, branched history of the system; the bottom depicts the result of projecting the commits that comprise the system's history onto a single line in temporal order[5]. The merge commit M that joins the two branches falls out since it is meaningless in the date-flattened mainline, where the work to merge the two branches occurs, piecemeal, as each commit is written. We discuss the threats to the construct validity of date-flattening in §5.5.

5.1 The Prevalence of Branching in 3gVC

We investigated RQ1, whether branching has become more prevalent in 3gVC, both qualitatively and quantitatively. First, interviewees indicated that, prior to using 3gVC, branches were "painful and difficult" to integrate. In some cases, two branches would grow so far apart, they had to abandon one of them altogether. Prior to 3gVC, branches were typically created only for releases and *not* new features.

> "We had branches for versions [releases]. Feature branches were VERY rare for us."
>
> Koziarski, Ruby on Rails [23]

Quantitatively, we see that few branches were created pre-3gVC. Of the examined 60 projects that switched to 3gVC, 1.54 branches were created on average per month per project before using 3gVC; after switching to 3gVC, the average rose to 3.67. A Wilcoxon non-parametric test of means shows that the difference is statistically significant. There are some interesting anomalies. For instance, the Samba project used branches extensively before migrating to 3gVC, creating on average of 5.35 branches per month. Nonetheless, their use of branching grew dramatically to an average of 41.42 per month after switching to git. In contrast, the Wine project has only 1 branch in its repository created after the switch to a 3gVC and none before. Thus, we conclude that the answer to RQ1 is affirmative: *branches are used more often in 3gVC*.

5.2 Merge Pain

Without easy branch and merging facilities, our interviewees reported that developers would "pass around large patch sets" or "brain dump" a mega-patch that was almost impossible to review. These large patch sets would contain multiple, sometimes unrelated changes, and it was impossible to "consider each on their own merits without having to swallow the whole thing" (Turnbull, XEmacs [35]). This problem was compounded for new developers who did not have commit access and so could not work and commit incremental work in the course of making large changes. Under CVC, developers without commit privileges, as well as core developers who refused to use "painful" (Sperber, XEmacs [24]) feature branches were effectively reduced to working in a time before version control. The following quotation illustrates problems with mega-commits.

> " Because we'd have these large changes that would go in all at once, it would be really difficult to find the source of problems. For example, if you wanted to find a change that was responsible for certain problems, you would often go back [in history] . . . and pretty soon you'd find one of these 'mega' patches . . . that would essentially change every file in the system and would lump together sets of unrelated changes ... [these mega changes made it] really, really difficult to track down what change was responsible for a given problem, it makes software maintenance really difficult. "
>
> Sperber, XEmacs [24]

These large changes were observable within the XEmacs repositories, for example large features such buffer character encodings had to be merged from the external Emacs repositories. Often these mega-commits were often followed by numerous revisions that attempted to fix issues caused by such a large merge. Other projects, such as Samba, also suffered from merge pain prior to DVC adoption.

The Samba project made heavy use of branches for release management and feature development. It seemed that heavyweight branching was not enough for the Samba contributors as after they switched from CVS to SVN in 2004 they were still experimenting with integrating 3gVC such as SVK and Mercurial to handle their own branches until eventually switching to git in 2008. Examples of merge pain extracted from Samba included mentions of bad merges, manual merges, multi-stage merges and mega-commits.

[5]For simplicity, we assume no ties. In the case of git, timestamps record seconds, so ties are rare in practice and can be deterministically broken by treating a commit's id (a hash) as an ordinal.

Once Samba had switched, most merge messages were automatically created with little complaint.

The Perl project used Perforce, which supports lightweight branching, prior to their switch to git. Within Perl, the adoption of new merging features available in Perforce was noticeable. Perl was developed using numerous parallel branches, thus a common commit to see was a merge or integration to mainline. Perl merge pain was not very visible before and after the switch from Perforce to git as Perforce handled merges well.

Quantitatively, we found mixed support for these observations and developer views. We looked at three measures of mega-commits: the median size of all commits, the median size of the largest commits, and the number of commits until a mega-commit was reached (a deadend).

Commit size. Although statistically significant, the median size of a commit was only two lines smaller after switching to a DVC [4]. This difference is so small as to be uninteresting. This result is not unexpected as OSS projects already require commits to be "small, independent, and complete" before they are committed [31]. The median of 15 changed lines is so small that it is unlikely that commits will become even smaller as a result of using a DVC.

Largest commit size. We did, however, expect the size of the largest commits (i.e., mega-commits) to be smaller post-DVC than pre-DVC because with DVC commits can be individually inserted onto a branch instead of lumped into one large patch. To examine this we looked at the top 5% and 10% of commits pre- and post-DVC. We examined them at monthly intervals because the pre-DVC history was typically longer and would, by virtue of including a longer history, have a larger mega-commit size. Although some projects bore out our expectation, many did not, leaving an inconclusive result.

Deadends. As the previous quotation illustrates, when looking back through history for a defect, a mega-commit became a stopping point in the search — a deadend. To assess whether DVC produced fewer deadends, we count the number of commits before a mega-commit. Based on the above quotation, we would expect pre-DVC to have fewer human understandable sized commits before a deadend. We varied the size of a mega-commit from 100 to 1000 changed lines. Again our results were inconclusive. Many projects did not have a sufficiently long history using DVC to yield statistically significant results at all mega-commit sizes. When the results were statistically significant, there was no clear trend.

With respect to RQ2, *we observed less qualitative evidence of merge pain after transitioning to a 3gVC* . While the difference in maturity, activity and total time were different, we did not witness the same amount or frequency of complaints about merging and integration after switching to a 3gVC version control system. Thus, qualitatively via interviews and via manual inspection of commit messages we found that there was some evidence of merge pain suffered by developers using version control systems that did not support lightweight branching. For RQ2 *our empirical quantitative results were relatively inconclusive* but gave some positive indicators to support our hypothesis: the median commit size was less and some projects had smaller mega-commits after DVC adoption.

5.3 Cohesion

Large systems, like the Linux kernel, structure their files in a modular manner. Files that perform similar functions are closer in the directory hierarchy than files that perform dissimilar functions [6], thus the directory structure loosely mirrors the system architecture. To determine how "cohesive" a set of changes are, we measure how far source files are from each other in the directory tree. Two files in the same directory have a distance of zero (*i.e.* the highest level of cohesion), while the distance for files in different directories is the number of directories between the two files in the hierarchy. We only include '.c' source files as Bowman [6] found that header files for the entire system often are located in one directory.

Let $d : F \times F \rightarrow \mathbb{N}_0$ denote the directory distance of two files. Each commit defines a set of files, its changeset. The cohesion of a single commit is the multiset of directory distances formed from the files in its changeset. A branch is a "straight line" sequence of commits, $B = c_1, \cdots, c_n$, without a merge or a branch. For the branch B, let B_d be the multiset of directory distances formed over the union of all its changesets in Equation 1:

$$B_d = \{d(f, f') : f, f' \in \bigcup_{c \in B} f_m(c)\}. \quad (1)$$

DEFINITION 5.1 (BRANCH COHESION). *The* branch cohesion *of B is the average of the directory distances in B_d:*

$$B_c = \sum_{d \in B_d} \frac{d}{|B_d|}.$$

To determine if developers use branches to isolate cohesive changes, we compare the cohesion of observed branches in the history of the Linux kernel against the cohesion of simulated branches of equivalent length over the date-flattened line, D, using Monte Carlo simulation. Figure 6 depicts this simulation. We first measure the length of each branch in the observed Linux kernel history (left) and create a multi-set of branch lengths (middle). We then randomly project these branch lengths (which sum to *precisely* the length of D) onto D to partition D into simulated branches (right). Thus, the distribution of branch lengths is exactly the same as the observed distribution of branch lengths in the Linux kernel history; specifically, this is the distribution shown in Figure 7a. We then compute the branch cohesion for each simulated branch of each length. If developers generally work together on cohesive sets of files in branches then the branch cohesion for branches of length n in the observed branch and merge preserving history will be higher than the cohesion for sequences of commits with length n in D. We performed $1,000$ rounds in our Monte Carlo simulation.

Figure 7a is a boxplot of the lengths of observed branches in the history of the Linux kernel. As Figure 7a makes evident, the distribution is positively-skewed. Since 90% of the Linux kernel branches have length less than 35 commits, we truncated Figure 7b at 35. Branches longer than 35 commits had fewer than 25 instances, giving too small a sample to produce meaningful results.

Figure 7b plots the mean of cohesion of observed Linux kernel branches (black diamonds) against the mean of the means of the cohesion of the simulated branches (black circles). We report the mean of the means at each branch length for the $1,000$ simulations and provide a 95% confidence interval (the vertical lines). With the exception of branch length 34, which is not statistically significant (red square), the observed branches are more cohesive than the simulated branches at each length with $p < 0.05$.

Examining the magnitude of the differences in cohesion, we see that at branch length two (the minimum), pairs of files committed on observed branches are 0.12 directories closer together on average than pairs files in date-flattened commits, while the difference is 1.5 directories at branch length 32 (the maximum). These differences may appear small, but note that a difference of 1 means that for *each pair of files* the distance between them is at least one directory further apart in the code base on a simulated branch than on the observed branch. This effect looms larger when one recognizes that most branches modify tens to hundreds of files.

This point is further underscored by correlating this difference to the branch length. As can be seen from Figure 7b, as branches

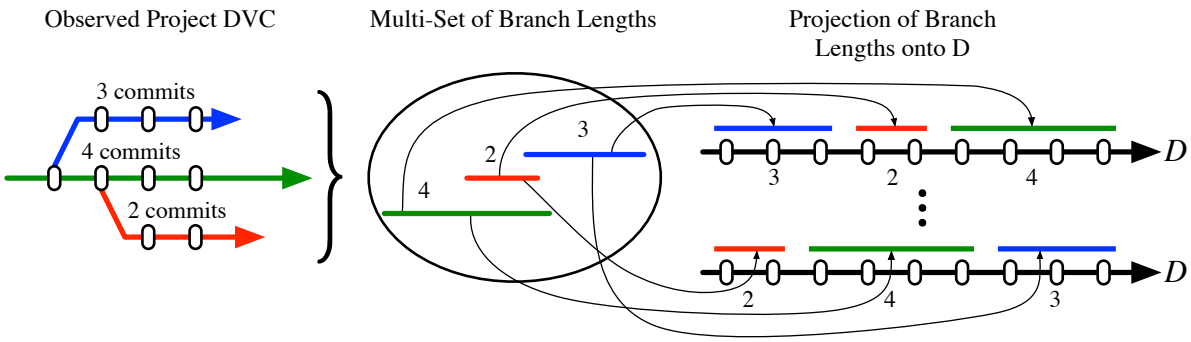
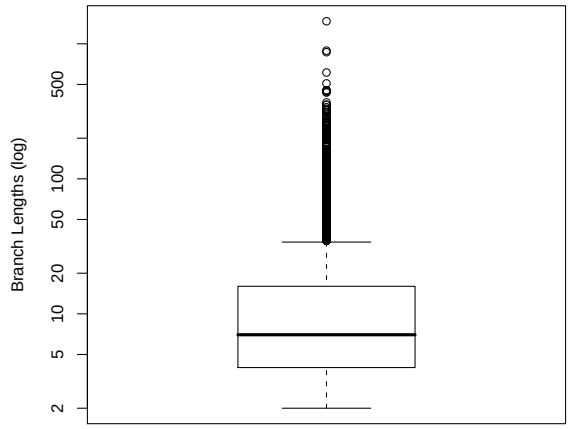
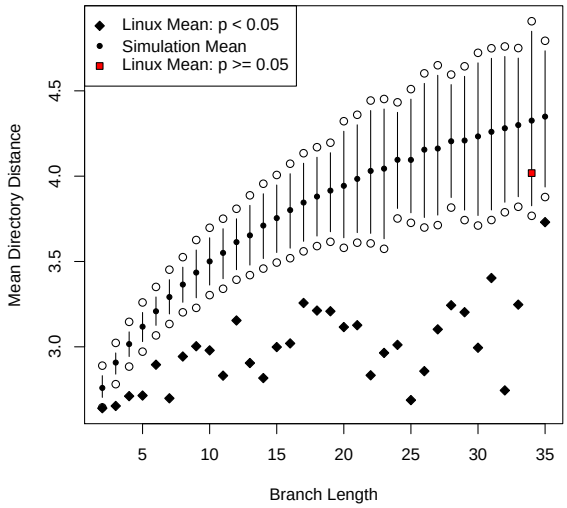


Figure 6: Depiction of selection of branches for the Monte Carlo simulation.



(a) Distribution of branch lengths in the Linux kernel.



(b) Observed branches compared to simulated branches over D from 1,000 Monte Carlo simulations.

Figure 7: Cohesion of first-class branches.

become longer, the observed branches become increasingly more cohesive relative to the simulated branches (Spearman correlation: $r = .69, p \ll .001$). It is clear that developers group related changes on branches and that this grouping becomes more important as the number of changes in a branch increases.

Our interviews are consonant with this result: branches are not created only for releases. In projects that have moved to 3gVC, branches comprise non-trivial, cohesive changes such as features or localized bug fixes and maintenance efforts. Three of our interviewees indicated that previously, such non-trivial changes would either have been avoided or created "off-line" and then committed to the VC in a single, disruptive mega-commit. Thus, our findings confirm RQ3: first-class branches are highly cohesive.

5.4 Coupling and Interruptions

Using data mined from the Linux kernel, we quantitatively evaluate the number of integration interruptions that a developer avoids through the use of 3gVC. By analogy to numeric intervals, $D(x, y)$ denotes the subsequence of commits between x and y in D. For the commit c, let a denote its most recent ancestor, either its branch parent or its closest grandparent on D when its branch parent is a merge commit. In Figure 5, commit 4's ancestor is 2, while commit 5's ancestor is its grandparent 4, since its parent M is a merge commit.

Reviewing Changes Since Last Commit One simple form of integration interruption is the cost of finding your place after you have been interrupted. Consider the developer who commits intermediate work on Friday and, on Monday, reviews the VC history to find her place. If her colleagues worked through the weekend, she might have to wade through a number of commits before finding her Friday commit. After restarting work, she eventually makes another commit, c. The number of commits a developer would have to scan in this case is precisely the length of $D(a, c)$, or $|D(a, c)|$. These commits define a set of files that is a superset of the set of files that may actually interfere with the work a developer is doing. Precisely defining that set is undecideable due to reachability problem.

Table 1 shows the proportion of commits for which $|D(a, c)| > 0$ across a selection of our projects, chosen because they have the largest code bases and communities and are widely-used and well-known. Nonetheless, these projects are representative of all projects examined. Across all projects analyzed, the average proportion of commits for which $|D(a, c)| > 0$ was 15.8%. Almost one third of the commits in Linux and `X.org` histories require more work for the developer to "find his place" in D, the flattened history. GNU Autoconf and Coreutils use branching very little and are not under very active development, which may explain the small proportion of

Repository	Commits for which one one must review changes
X.org XServer	30.5%
Linux kernel	30.1%
Glasgow Haskell Compiler	28.1%
Ruby on Rails	27.3%
Perl	18.0%
Samba	17.8%
XMMS 2	15.8%
KDebug	13.0%
Yum	9.7%
Syslinux	5.0%
GNU Autoconf	0.5%
GNU Coreutils	0.2%
Wine	0.1%

Table 1: Proportion of commits for which one must review changes since one's last commit.

commits that require finding one's place. Wine is an anomaly: its use of branching has changed very little since moving to a 3gVC.

Verifying Assumptions When a developer is not isolated on a branch, *i.e.* when she must work on D, she faces the distraction of integration work intruding into feature development work. Even when overt conflict does not occur, she still must verify any assumptions she made about code on which her new feature depends.

Figure 8 illustrates the formalism we introduce to measure these interruptions. The line at the left represents D, the date-flattened history. Ovals on D represent commits. Each commit c defines a changeset, a set of files that it modifies. In the figure, these modified files are the rectangles stacked above each commit. When F is the set of files in a source code repository, $f_m : C \rightarrow 2^F$ extracts the changeset from a particular commit.

Specifically, c is a commit whose nearest, non-merge ancestor in the original branch and merge preserving history is a, and $D(a, c)$ represents the commits that developers made to other branches in that history in the intervening time. In particular, each commit w in $D(a, c)$ defines a changeset. Definition 5.2 formalizes the set of files changed in a sequence of commits.

To model the integration interruptions a developer may encounter when committing c, we examine the intersection c's changeset, $f_m(c)$, and the files in $D(a, c)$, F_i. At the right of Figure 8, the fraction of the number of files in the intersection divided by the number of files in c is the *index of similarity* we use to measure integration interruptions. If the index of similarity exceeds a threshold, δ, then c is distracted, as specified in Definition 5.3.

DEFINITION 5.2 (INTERVENING FILES). *The files modified in $D(a, c)$ "intervene" between c and a, its nearest ancestor in D. That nearest ancestor a is either the branch parent of c when the branch parent of c is not a merge commit, or a is one of the branch grandparents. These files therefore change the state of the file system into which c is written. The set of* intervening files *is*

$$F_i = \bigcup_{w \in D(a,c)} f_m(w).$$

If c modifies $f \in F_i$, an outright conflict could occur. Lack of overt conflict may be even more distracting as c's author must review each file in $f_m(c) \cap F_i$ to make sure that the non-conflicting changes do not invalidate an assumption on which c depends. For instance, one of the commits in $D(x, c)$ could have changed the semantics of a function used in c. This definition requires a commit's context to have changed in D: If c is adjacent to its parent or one its grandparents on D, $D(a, c) = \emptyset$ and therefore so does F_i.

Intuitively, the commit c is *distracted* if commits fall between it and its branch parent (or grandparents if its branch parent is a merge commit) on D and one those intervening commits changed a file that c also modified. In Figure 5, all the commits except commits 1 and 5 are potentially distracted. Definition 5.3 captures this intuition.

DEFINITION 5.3 (DISTRACTION). *The commit $c \in D$ is* distracted *if $F_i \neq \emptyset$ and*

$$\frac{|F_i \cap f_m(c)|}{|f_m(c)|} \geq \delta, \textit{ for } \delta \in [0..1].$$

We require $c \in D$ to eliminate merge commits, as the work they represent has already occurred in D, the date-flattened line. In Figure 5, all the merge work M required has already occurred by the time 4 has been written in D. A commit cannot be distracted by commits whose changesets do not overlap its changeset. Here, we have defined an index similarity relative to c's changeset vs all the commits on D between c and its nearest ancestor a because the point is to try capture the work to commit c given the state of the file system in D when c would have been written into D, which includes the specified commits and their changesets.

Our metric naturally follows from this definition of distraction: we plot the proportion of commits in the date-flattened line that are distracted as δ varies. So at $\delta = 0$, we plot proportion of all commits whose $F_i \neq \emptyset$, after which we show the proportion of all commits that satisfy each setting of δ. Figure 9 depicts the resulting curve for the Linux kernel. The y-axis jumps between 5 and 29 to span an order of magnitude in order to capture $\delta = 0$ when we ignore whether F_i and $f_m(c)$ overlap while zooming the majority of the data points in $0 < y < 5$ where δ ranges from 0.1 to 1. Even at $\delta = 1$, *i.e.* when we require $f_m(c) \subseteq F_i$, 2.8% commits are distracted. After calculating the 95% confidence intervals, we find that a commit c modifies a file that intervenes between c and its ancestor a on D with a confidence interval of 4.47% to 4.69% of the time. This corresponds to the point in Figure 9 with an index of similarity of 0.1. All of the files in a commit c intervene between c and its ancestor a with a confidence interval of 2.47% to 2.93% of the time. This corresponds to the point with index of similarity 1.0. Thus, a non-empty overlap occurs approximately once every 22 commits and a complete overlap every 35 commits. This rate is an under-approximation, as D is created from the perfected repository that does not include commits that were rejected after review or sequences of commits by the same author that were collapsed [4]. Although not all of these overlaps cause an overt conflict, they represent changes that a developer needs to examine and possibly correct, forcing a developer to interrupt development work with integration work. Clearly, branches reduce the distraction of resolving a conflict or verifying assumptions about changed code. Therefore, we affirm RQ4 and conclude that 3gVC branches better insolate developers than concurrent, branchless does.

5.5 Threats to Validity

The main threat to the external validity of our cohesion, distractions and named stable bases results is their dependence on Linux git history, which may not be representative. For example, counting merges is confounded in git by the fact that fast-forward merges do not create merge records. Thus merges visible in other version control systems cannot be counted in git.

Many projects we surveyed did not have a long enough 3gVC history (*i.e.* sample-size) to produce statistically significant results in all of our measures, particularly in Section 5.2. Two other threats are feature adoption and project maturity. Developers are still adjusting to DVC and may not have adopted first-class branching to organize

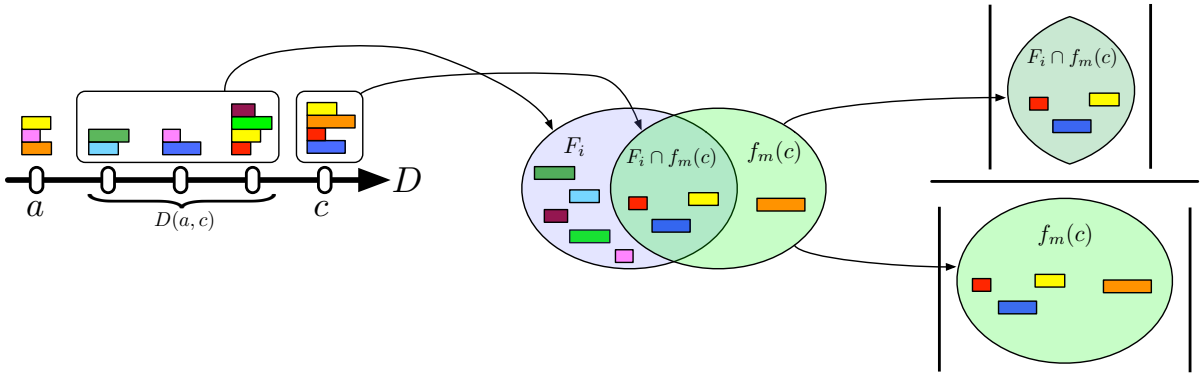


Figure 8: Depiction of formalisms described: The straight line indicates D, the date-flattened sequence of commits (ovals). The rectangles above commits represent files modified in a commit. Here, c is a commit whose nearest non-merge ancestor in DVC is a. $D(a, c)$ are the commits made by other developers in the intervening time. $F_m(c)$ is the set of files modified in c and F_i is the set of files modified in the commits in $D(a, c)$. The ratio of files in the intersection to files changed in c is the *index of similarity* that we vary in our definition of distraction.

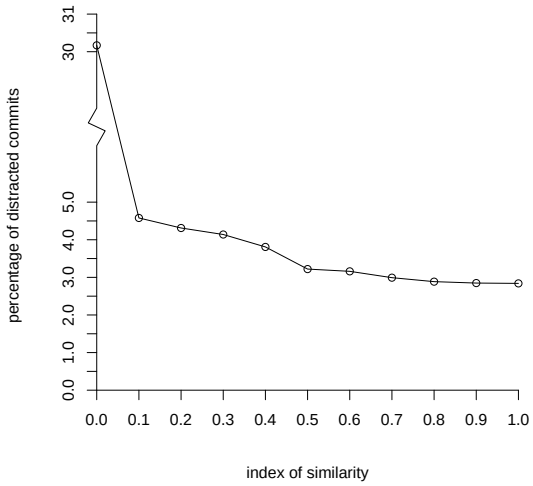


Figure 9: Commits that require integration work as δ varies when the Linux kernel history is date-flattened.

larger commits. As well, many contributions, even to 3gVC-using projects, are submitted in the form of large patches to the mailing-list. The maturity of a project might affect its process and its patterns of development, thus 3gVC adoption might not be noticeable. We discuss threats to date-flattening and specific results next.

Date-flattening As we quantified in **??**, interleaved branch commits leads to a fractured history D. Is this fractured history realistic baseline for branchless, concurrent development? This is debatable. The coordination cost of a shared mainline may not be great: the required workflow may be routinely handled by regular communication between developers. A developer using shared mainline may use her greater awareness of the activities of others to craft their changes to splint and heal rather than crack the shared history. This discipline would reduce both the number of integration interruptions and the level of dispersion that we report below. However, this awareness is itself an interruption and an annoyance. Because developers using a shared mainline are likely to commit less often to avoid conflict or breaking the build, our flattening technique, in the worst case, indicates how often developers would like to commit.

Cohesion This result depends on D, the date-flattened line, and inherits its threats to validity. Cross-cutting concerns, by definition, are rarely well-correlated with directory distance. Thus, our use of directory distance as a cohesion metric does not capture the cohesion of a cross-cutting change; however, the fact that we found a significant difference in spite of understating the history-preserving nature of lightweight branching strengthens our result.

Reviewing Changes As noted above, D may overstate the integration interruptions faced by a developer. Nonetheless, developers using branches have more freedom. There is little need to consider "when should I commit to cause the least amount of disruption to my work and the work of others?" Developers are freed to work in parallel and isolation. Further, they can work within VC on "experimental" or "speculative" code which may not ever end up being integrated into the main line. Our analysis assumes that all integration interruptions waste time, which may not always be the case. In Figure 5, 3 may benefit from 2's changes, *i.e.* if 2 deletes code that 3 would have had to uselessly modify when isolated on a branch. Our first metric, which reports the average length of $D(a, c)$, assumes that a developer, at least partially, relies on VC history to find their place. In practice, developers may turn to other means, perhaps as mundane as a hand-written notes or a "todo" email sent to self.

6. RELATED WORK

Todo 6-1 (for Earl): *Homage to andy ko*

While version control systems have a long and storied past, our concern in this paper is primarily the introduction of first-class branching and merging, and the resulting rich histories.

The importance of preserving histories, including branches, has been well recognized [14]. The usefulness of detailed histories for comprehension [1] and for automated debugging [43] are by now well accepted. Some have even advocated very fine-grained version histories [22] for improved understanding & maintenance. There have also been efforts to track histories at the object, or configuration, rather than at the file level, with more sophisticated data models tuned for software engineering products [15].

Branching in VCs have received a fair bit of attention [14]. Some have recommended "patterns" of workflows for disciplined use of branching [38]. Others advocate ways of branching and merging approaches [8] that mitigate the difficulties experienced in earlier version control systems with non-first-class branching.

In their work, Bird and Zimmermann [5] also examine branching from an empirical perspective. However, they're work is different from ours in that they identify branching anti-patterns in SCMs and

demonstrate methods of avoiding or removing these anti-patterns by evaluating branches through simulation. In contrast, our work examines the affect of 3gVC on software development practices.

Todo 6-2: *Discuss Perry* et al. *[29]*

Todo 6-3: *Probably say something about our own promises and perils thing, which is a warning to people who write papers like this one?!?!?!??*

Todo 6-4: *Scan PC members for related papers. Scan last five years of ICSE and FSE for related paper. Chris, Prem: Other venues I should scan?*

The influential work of Viégas et al [39] uses a visualization methodology to study the historical record of edits in wikipedia, and report interesting patterns of work (such as "edit wars"). Our study is in a similar spirit, but focuses on issues specific to large-scale software development. Branching and merging are not very relevant to wikipedia; nor are software developers nearly as contentious. To our knowledge, our paper is the first detailed study of the impact of 3gVC and its first-class branching and merging operations on the practice of large-scale, collaborative software engineering.

7. CONCLUSION AND FUTURE WORK

Contrary to conventional wisdom, branching, not distribution, has driven the adoption of 3gVC: most projects still use a centralized repository, while branching has exploded (RQ1), without a concomitant increase in merge pain (RQ2). first-class branches are used to undertake cohesive development tasks (RQ3) and are strongly coupled (RQ4). In the course of investigating these questions, we have defined three new measures: branch cohesion (§5.3) and two types of task interruption that occur when integration work intrudes into development — intervening files and distraction (§5.4).

We intend to investigate how projects order and select which branches to merge when. The isolation that branches affords carries the risk that the work done on that branch may be wasted if the upstream branch evolves too quickly. We intend to investigate the impact of first-class branching on the use of named stable bases [3].

Although VC is itself an "essential" complexity of software development, Brooks [7] would call the transition to 3gVC a manifestation of "accidental" complexity. Nonetheless, 3gVC has potential to make releasing, developing, and coordinating large software projects much less painful than the generation of tools it is supplanting.

Todo list

4-1 Chris: Discuss and extol mixed methods. (*Page 4*)
6-1 Earl: Homage to andy ko (*Page 9*)
6-2: Discuss Perry *et al.* [29] (*Page 10*)
6-3: Probably say something about our own promises and perils thing, which is a warning to people who write papers like this one?!?!?!?? (*Page 10*)
6-4: Scan PC members for related papers. Scan last five years of ICSE and FSE for related paper. Chris, Prem: Other venues I should scan? (*Page 10*)

References

[1] D. Atkins. Version sensitive editing: Change history as a programming tool. *System Configuration Management*, pages 146–157, 1998.

[2] C. Baldwin and K. Clark. *Design rules: The power of modularity*. The MIT Press, 2000.

[3] S. Berczuk. Configuration Management Patterns. In *Third Annual Conference on Pattern Languages of Programs*, 1996.

[4] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *MSR '09: Proceedings of the 6th IEEE Working Conference on Mining Software Repositories*, 2009.

[5] C. Bird and T. Zimmermann. Avoiding antipatterns with branch refactoring, 2011. *Under submission to* ESEC/FSE.

[6] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 555–563, Los Angeles, California, 1999.

[7] F. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987.

[8] J. Buffenbarger and K. Gruell. A Branching/Merging Strtegy for Parallel Software Development. *System Configuration Management*, pages 792–792, 1999.

[9] B. Cannon. PEP 374: Choosing a distributed VCS for the Python project, January 2009. `http://www.python.org/dev/peps/pep-0374`.

[10] I. Clatworthy. Distributed version control — why and how. In *Open Source Developers Conference (OSDC'07)*, 2007.

[11] J. Coplien. *A Generative Development-Process Pattern Language*. Cambridge Univ. Press, 1998.

[12] J. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage, 2003.

[13] T. DeMarco and T. Lister. *Peopleware: productive projects and teams*. Dorset House Publishing Co., Inc. New York, NY, USA, 1987.

[14] J. Estublier. Software configuration management: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 279–289. ACM, 2000.

[15] J. Estublier and R. Casallas. The Adele configuration manager. *Configuration Management*, 2, 1994.

[16] J.-M. F. Jacky, J. Estublier, and R. Sanlaville. Tool adoption issues in a very large software company. In *Proceedings of 3rd International Workshop on Adoption-Centric Software Engineering*, pages 81–89, 2003.

[17] KDE. Projects/MoveToGit - KDE TechBase, November 2009. `http://techbase.kde.org/Projects/MovetoGit`.

[18] Kernel.org. `git-rebase`, March 2011. `http://www.kernel.org/pub/software/scm/git/docs/git-rebase.html`.

[19] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM.

[20] T. Lindlof and B. Taylor. *Qualitative communication research methods*. Sage, 2002.

[21] Linus Torvalds. Email to `dri-devel@lists.sourceforge.net`, March 29 2009.

[22] B. Magnusson and U. Asklund. Fine grained version control of configurations in COOP/Orm. *Software Configuration Management*, pages 31–48, 1996.

[23] Michael Koziarski. Personal Interview, April 5 2009. `rubyonrails.org`.

[24] Michael Sperber. Personal Interview, April 3 2009. `xemacs.org`.

[25] A. Mockus, J. D. Herbsleb, and R. T. Fielding. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, July 2002.

[26] C. Parnin and S. Rugaber. Resumption strategies for interrupted programming tasks. In *ICPC .09: Proceedings of the 17th IEEE International Conference on Program Comprehension*, Washington, DC, USA, 2009. IEEE Computer Society.

[27] R. Paul. DVCS adoption is soaring among open source projects. *ars technica*, January 7 2009.

[28] D. Perry, H. Siy, and L. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(3):308–337, 2001.

[29] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, 2001.

[30] red book.com. Subversion 1.4 best practices for merging, March 2011. `http://svnbook.red-bean.com/en/1.4/svn.branchmerge.copychanges.html#svn.branchmerge.copychanges.bestprac`.

[31] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: A case study of the apache server. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 541–550, New York, NY, USA, 2008. ACM.

[32] L. Rocha. GNOME to migrate to git, March 2009. `http://http://mail.gnome.org/archives/devel-announce-list/2009-March/msg00005.html`.

[33] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, Dec. 1975.

[34] A. Sarma, Z. Noroozi, and A. Van der Hoek. Palantír: raising awareness among configuration management workspaces. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 444–454, 2003.

[35] Stephen Turnbull. Personal Interview, April 7 2009. `xemacs.org`.

[36] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[37] R. van Solingen, E. Berghout, and F. van Latum. Interrupts: just a minute never is. *Software, IEEE*, 15(5):97–103, Sep/Oct 1998.

[38] S. Vance. Advanced scm branching strategies, 1998. http://www.vance.com/steve/perforce/Branching_Strategies.html.

[39] F. Viégas, M. Wattenberg, and K. Dave. Studying cooperation and conflict between authors with history flow visualizations. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 575–582. ACM, 2004.

[40] R. S. Weiss. *Learning From Strangers : The Art and Method of Qualitative Interview Studies*. Free Press, November 1995.

[41] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 67–76. ACM New York, NY, USA, 2008.

[42] S. Zacchiroli. (declared) VCS usage for Debian source pacakges, February 2011. `http://upsilon.cc/~zack/stuff/vcs-usage`.

[43] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Software Engineering—ESEC/FSE'99*, pages 253–267. Springer, 1999.

[44] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.