Cohesive and Isolated Development with Branches

Earl T. Barr[1], Christian Bird[2], Peter C. Rigby[3], Abram Hindle[4],
Daniel M. German[5], and Premkumar Devanbu[1]

[1] UC Davis, Davis CA, USA
[2] Microsoft, Redmond WA, USA
[3] McGill University, Monteal QC, Canada
[4] University of Alberta, Edmonton AB, Canada
[5] University of Victoria, Victoria BC, Canada

Abstract. The adoption of distributed version control (DVC), such as Git and Mercurial, in open-source software (OSS) projects has been explosive. Why is this and how are projects using DVC? This new generation of version control supports two important new features: distributed repositories and histories that preserve branches and merges. Through interviews with lead developers in OSS projects and a quantitative analysis of mined data from the histories of sixty project, we find that the vast majority of the projects now using DVC continue to use a centralized model of code sharing, while using branching much more extensively than before their transition to DVC. We then examine the Linux history in depth in an effort to understand and evaluate how branches are used and what benefits they provide. We find that they enable natural collaborative processes: DVC branching allows developers to collaborate on tasks in *highly cohesive* branches, while enjoying *reduced interference* from developers working on other tasks, even if those tasks are *strongly coupled* to theirs.

1 Introduction

Version control (VC) is tool support for concurrent, collaborative software processes. VC allows developers to create a *branch*, an isolated workspace, from a particular state of the source code. They can share this branch and work on their tasks within it without impacting the rest of the project and later merge (or integrate) their changes back into the main line of development.

Intuitively, branches should be *cohesive* (*i.e.* collect related changes [26]) allowing a team to work together on a focused task and *isolated* from the rest of the project so that rapid and volatile development is not interrupted or impacted by external changes. The rich history provided by recent VC and their adoption by a number of projects provide a unique opportunity to address these intuitions and quantitatively measure how cohesive and isolated branches are in practice.

The evolution of VCs is marked by *increasing fidelity of the histories they record*. A *commit* is the write of a change into VC history. First generation VC, such as RCS, record the history of individual file commits. This enabled rolling back changes to a single file and reviewing file-specific changes. Second generation, or centralized VC (CVC), such as Subversion, stored sets of file changes committed together (*i.e.*, a *changeset*) 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.

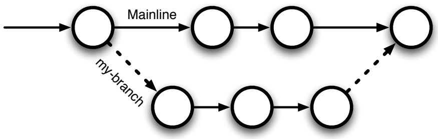


Fig. 1: DVC history preserves branches and merges.

Recently, a new generation of VC, distributed version control (DVC), has transformed the use of VC and has achieved widespread adoption. In DVC, every copy of a project is a repository, with its own history and the power to exchange source code changes with other repositories. In contrast with CVC, DVC is distributed in the sense that it allows the change of changesets unmediated by a central repository. DVC also preserves the history of branches after their promotion into the mainline of development. Consider Fig. 1 in which circles represent commits to the repository. Arcs denote the temporal ordering of commits. "Mainline" denotes the main line of development from which releases are made and to which features, like "my-branch", are merged. The dashed edges depict relationships that were untracked, and forgotten in CVC [6]. In DVC, a commit always tracks its immediate predecessor commits, across both branches and merges; for DVC, the dashed edges are indistinguishable from the edges along a branch. This branch history allows us to augment developer studies with quantitative studies of branch cohesion and isolation. We can use this branch history to crosscheck qualitative results on branch usage. We can also use these measures to shed light on whether differences in how a project uses branches correlate with defect rates or schedules delays.

Open-source software (OSS) projects have rapidly adopted DVC. Our first research question, RQ1, asks "Why did OSS projects rapidly adopt DVC?" We use interviews to show that developers had previously wanted to make heavier use of branches but were dissuaded by "merge pain", the difficulty of resolving conflict that arises during branch integration, and buttress this observation by showing that branch usage has markedly increased in those projects that made the transition from CVC to DVC. We also note that almost all projects making the switch have continued to use a centralized repository, calling into question the conventional wisdom that DVC's support for distributed workflows has been the principal cause of the rapid transition to DVC.

Without branches, developers must share a single mainline and deal with the conflicts that sharing entails. In practice, projects developed workflows to avoid or mitigate conflict, such as baton passing or the "commit bit". We can demonstrate the benefit of branching by simulating a lack of branching. We observe that branched history of a DVC can be linearized onto a single "mainline" in which the conflicts and interruptions that branching avoids become manifest. This linearized history overapproximates the actual conflict and allows us to bound the cohesion and isolation that branches afford.

Ideally, when a task is identified, developers create a branch to work on the task together. But does this occur in practice? Is work performed in a branch more cohesive than all changes across the repository during the same time period? Thus, RQ2 is "How

[6] This limitation was addressed in Subversion 1.5.

cohesive are branches?" To investigate this question, we use directory distance of the files modified in a branch to measure its cohesion. Then we compare actual branches in the Linux history against the baseline, background cohesion of linearized sequences of commits. If actual branches are no more cohesive than these commit sequences, then branches are either unlikely to be cohesive or directory distance is a poor proxy for branch cohesion. To form these commit sequences, we picked a random starting point on the linearized branch history. In §4.2, we found that actual, observed branches are significantly more cohesive than background commit sequences.

RQ3 asks "How successfully do DVC branches protect developers from interruption?" VC is good about flagging syntactic conflict; *semantic conflict* occurs when mainline has changed in such a way as to invalidate assumptions made during the development of a branch. Cross branch coupling causes semantic conflict. To merge a feature branch into mainline is to *promote* that branch. When promoting a branch, programmers must review mainline to try to find semantic conflict. To measure semantic conflict, we measure the number of commits in a branch being considered for promotion that modified a file that has also been modified in mainline, since the branch forked from mainline. Against a linearized DVC history, we measure and bound how often the semantic conflict would interrupt a developer in the absence of branching or procedures to ameliorate it.

We make three principal contributions in this paper: 1) We present compelling evidence from study of sixty projects (RQ1) that branching and not distribution has driven the rapid adoption of DVC; 2) We define two new measures: branch cohesion and distracted commits, a type of task interruption that occurs when integration work intrudes into development; and 3) We apply these measures to the Linux history and (RQ2) quantify the cohesiveness of branches and (RQ3) the effective isolation they provide against the interruptions intrinsic to concurrent development.

2 Theory

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 [33], indicating that it has achieved widespread acceptance and adoption[7]. VC has a profound effect on workflow, and adoption of a new VC is not a trifling matter [12], 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. For examples see GNOME's move to Git [25], Python's move to mercurial [7], and the project that KDE created solely to evaluate and eventually create tools for a migration to Git [13].

[7] The data we report here comes from the repository that contains the Debian packaging scripts. In practice, we observe that for the majority of projects, this repository is indistinguishable from the upstream repository.

Research Question 1: Why did OSS projects rapidly adopt DVC?

In §4.1, we present compelling evidence that DVC support for branching drove the transition to DVC. Our interviews show that the impetus is cohesion and isolation. But how cohesive are branches and how well do they isolate developers?

Cohesion If developers use branches to isolate tasks, we expect to find that branches are cohesive and encapsulate related changes. Two reasons to expect developers to work with cohesive branches is that their histories are easier to understand when faced with maintenance tasks and they are easier to revert if the branch has a problem. On the other hand, developers could be using branches merely to isolate their development work, without separating that work into cohesive tasks.

Research Question 2: How cohesive are branches?

Coupling and Interruption Developing a new feature often requires making changes to modules that are coupled to other modules. If different features, under simultaneous construction by different developers, affect coupled modules, the tasks may require coordination, as one developer's work can cause other developers' code to become unstable. Ideally, uninvolved developers should be isolated 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 [2] 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.* [24] 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. The use of branches allows a developer to control and minimize the frequency of such interruptions.

Integration interruptions are a form of task interruption. Prior literature has shown that task interruptions seriously impact 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 risk inserting bugs. In a study at Microsoft [16], 62% of developers said that recovering from interruptions is a substantial problem. Van Solingen [28] found that interruptions are most problematic 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 [10]. Parnin *et al.* [22] instrumented Visual Studio and Eclipse to observe the time taken to resume development tasks. While they found some strategies for mitigating the effects, developers began editing within a minute of restarting a task only 10% of the time and took over 30 minutes in 30% 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 3: How successfully do DVC branches protect developers from interruption?

3 Methodology

We used a mixed method research strategy [9] in our study of branches in DVC. We began with interviews of developers (the qualitative phase) to help develop hypotheses regarding the motivations for DVC adoption and then empirically evaluated these hypotheses by gathering data and performing statistical analyses (the quantitative phase). For us, the advantage of a mixed method approach is that the qualitative investigation allowed us to collect answers to fundamental questions related to the "how" and "why" of DVC adoption. The answers then provided insight and added meaning to our quantitative results that might otherwise have been missed in a purely quantitative study. This increased our confidence in the findings and provided a richer context that can aid in understanding whether our results generalize.

In an effort to understand what has motivated the rapid transition to DVC from an operational point of view, we observed the development activities in projects that switched to DVC and interviewed a number of lead developers from these projects regarding their switch. We sent personalized requests for fifteen minute interviews to the three most active developers in a number of large and mature projects that had used CVC for multiple years and had recently moved or decided to move to DVC. 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 DVC, the perceived benefits and drawbacks of the switch, and (in cases where the projects have used DVC for some time) how it has affected the policy and development process of the projects. The data mining of the VC history and developer mailing lists allowed us to provide quantitative evidence of the effects of DVC. 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 [17]. Semi-structured interviews are often used in an exploratory context when there are clear research questions [17,31]. 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 [9] for coding. The interview guide that we used

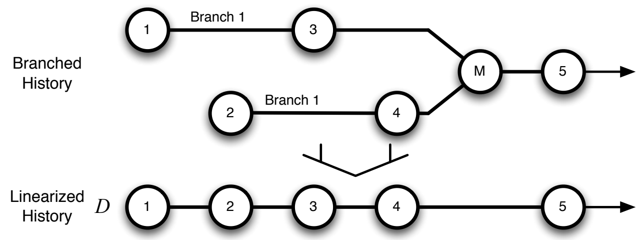


Fig. 2: Branches projected onto D, a single timeline by date. The merge change M that joins the two branches falls out since the work to merge each change occurs, piecemeal, as each change is recorded.

can be found at `http://www.cabird.com/public/vcinterviewquestions.pdf`[8]. 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 [17].

For the quantitative mined data, we developed measures and modified existing ones to best examine the impact of DVC in the context of our dimensions. The data used, the definition of the measure, and attendant threats to validity are discussed in §4. We chose to examine 60 projects that had transitioned to DVC. 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 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 DVC usage because the Linux kernel project has never used a CVC and its developers are generally very experienced with history-preserving 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.

4 Evaluation

In this section, we answer each of our research questions. To begin, we introduce our branch linearization technique on which much of our analysis rests. To linearize a branched DVC history, we project the concurrent sequence of changes in a DVC history onto the single timeline D, as shown in Fig. 2. The commits along this timeline represent concurrent work that actually occurred *across* branches. Conflict or interruption, that

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

occurs along this timeline, bounds the work needed to avoid conflict or recover from interruption. This work was previously largely unobservable (apart from mutterings in mailing lists/interviews/change-log messages), handled by policies and procedures such as baton passing and patch rework on a project's mailing list [32]. To measure the cohesion (§4.2) and isolation (§4.3) of branches, we compare the cohesion and isolation of their *within* branch changes against that of *across* branch changes, in the form of simulated branches drawn from D.

4.1 Rapid DVC Adoption

Pundits claim that support for distributed (changeset flows unmediated by a central repository), as opposed to centralized, development is the root cause of this rapid transition [23,8]. We have observed something different. The vast majority of these projects do not appear to be making use of distribution. Of the sixty projects whose VC use we examined, all but Linux continue to use a centralized model organized around a single public repository, except the `xemacs` and `gnome` projects which publish two repositories. Although these projects continue to use a centralized style of development, we *have* observed a dramatic shift in their use of branches.

Lead developers from prominent open source projects (§3) indicated that, prior to using DVC, branches were "painful and difficult" to integrate:

> " The biggest complaint associated with Subversion is associated with branching and merging. The one feature that Git has that our users would really like is a really fast and simple merge. "
>
> Richards, CEO WANdisco [14]

In some cases, two branches would grow so far apart, they had to abandon one of them altogether. Prior to DVC, branches were typically created only for releases and *not* new features. For instance, Koziarski from Ruby on Rails states: "We had branches for versions [releases]. Feature branches were very rare for us"[20]. A preliminary empirical investigation showed that few branches were created pre-DVC. Of the examined 60 projects that switched to DVC, 1.54 branches were created on average per month per project before using DVC; after switching to DVC, the average rose to 3.67. A Wilcoxon rank sum test shows that the two populations are statistically different[9]($p \ll 0.01$)

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 contained multiple, sometimes unrelated changes, and it was impossible to "consider each on their own merits without having to swallow the whole thing" (Turnbull, XEmacs [27]). 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 [21]) feature branches were effectively reduced to working in a time before version control.

[9] A Wilcoxon test was used rather than the standard t test due to the heavily skewed distribution of branches.

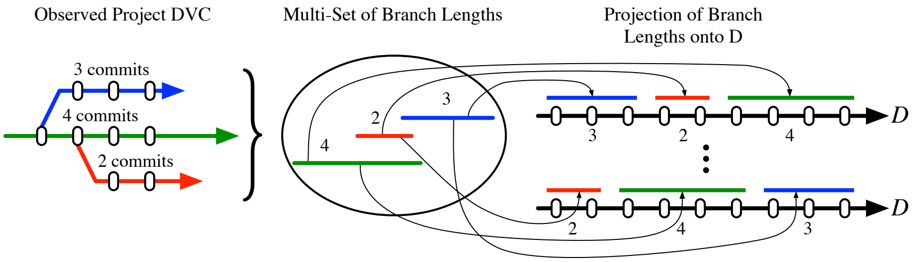
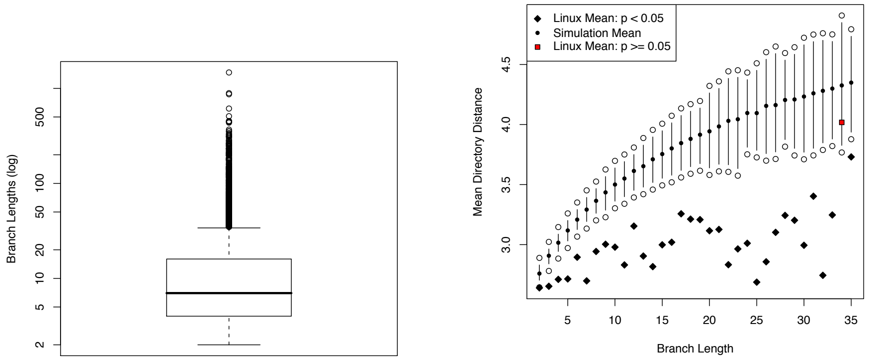

Observed Project DVC
Multi-Set of Branch Lengths
Projection of Branch Lengths onto D
3 commits
4 commits
2 commits
4
2
3
3
2
4
D
2
4
3
D

Fig. 3: Depiction of the 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 simulations.

Fig. 4: Linux branch lengths: observed and simulated.

> “ 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 [21]

In summary, projects continue to use a centralized repository and project maintainers have stated that the DVC branch and merging facilities was a principal motivation, so we find that the answer to RQ1 is branching, not distribution.

4.2 Cohesion

Large systems, like the Linux kernel, structure their files in a modular manner. Files that perform similar or related functions are close in the directory hierarchy [5], thus the directory structure loosely mirrors the system architecture. To determine how "cohesive" a set of changes is, 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 [5] found that header files for the entire system often are located in one directory.

Let $d : F \times F \to \mathbb{N}_0$ denote the directory distance of two files. Each commit defines a set of modified files, or changeset. When F is the set of files in a source code repository and C is the set of commits, $f_m : C \to (2^F - \emptyset)$ returns the changeset of a commit; f_m cannot return the empty set because a changeset cannot be empty. 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, $B = c_1, \cdots, c_n$, where c_1 is not a merge commit and c_n is either a leaf (*i.e.* HEAD) or the parent of a merge commit. Thus, one branch includes and continues through a branch commit, while each child of a merge commit starts a new branch, rather than continuing one of the merged branches. For the branch B, let B_d be the multiset of directory distances formed over the union of all its changesets:

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

Definition 4.1 (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 need a baseline to compare the cohesion of branches because we have no *a priori* notion of what the range of good and branch cohesion values may be. Thus, we need to establish the background distribution of cohesion, as a baseline for comparison. To do so, we measure the cohesion of branches over D, the linearized history of a project (Fig. 2), which captures concurrency work as a free-for-all on a single, shared mainline. Specifically, 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 linearized history, D, using Monte Carlo simulation. Fig. 3 depicts this simulation. We first measure the length of each branch in the observed Linux kernel history (Fig. 3 left) and extract their multiset of branch lengths (Fig. 3 middle). We then randomly tile these branch lengths (which do not contain merge commits and sum to *precisely* the length of D) onto D to form simulated branches (Fig. 3 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 Fig. 4(a). We then compute the branch cohesion for each simulated branch. If developers generally work together on cohesive sets of files in branches then the branch cohesion for branches of length n in the observed DVC history will be higher than the cohesion for sequences of commits with length n in D. We generated 1000 tilings in our simulation.

Fig. 4(a) is a boxplot of the lengths of observed branches in the history of the Linux kernel. As Fig. 4(a) makes evident, the distribution is positively-skewed. Since 90% of the Linux kernel branches have length less than 35 commits, we truncated Fig. 4(b) at 35. Branches longer than 35 commits had fewer than 25 instances, giving too small a sample to produce meaningful results. Fig. 4(b) plots the mean branch 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 1000 tilings 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 of files along D, the linearized history, 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 Fig. 4(b), as 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 increases with the number of changes.

Our interviews are consonant with this result: branches are not created only for releases. In projects that have moved to DVC, 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, we find that the answer to RQ2 is that branches are highly cohesive.

4.3 Coupling and Interruptions

Using data mined from the Linux kernel, we construct its linearized history D, as defined in Fig. 2, and quantitatively establish an upperbound on the number of integration interruptions that a developer avoids through the use of branching. 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 non-merge ancestor.

Consider a developer working on a new feature on a branch. When she promotes a feature branch to master, she must not only resolve any syntactic conflict that arise, but, more generally, look for potential *semantic conflicts*, conflicts that occur when mainline changes in a way that violates the assumptions on which a feature branch rests. For instance, her branch may rely on a global variable whose range of allowed values has changed in master, because her branch is coupled to other branches promoted since her branch began. Such verification can be subtle and time-consuming. This work is inherent

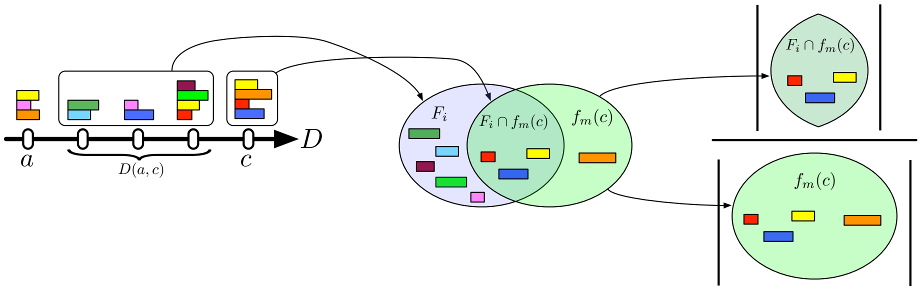


Fig. 5: Depiction of the formalisms described: The straight line indicates D, the linearized sequence of commits (ovals). The stacked colored rectangles above a commit represent its changeset. Here, c is a commit whose nearest, non-merge ancestor in DVC is a. $D(a, c)$ are the commits made by other developers 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* δ that we vary in our definition of distraction.

to concurrent development, but previously handled out-of-band by policy and procedure. To upperbound this work, we consider the work to search for semantic conflict that would occur along D where the distraction of integration work potentially intrudes into feature development work at each commit. This measures how often the integration work, ideally deferred to merge time, would instead intrude into feature development in the absence of an isolation mechanism, such as that provided by DVC.

Fig. 5 illustrates the formalisms we introduce to measure the integration interruptions that occur along D. The line at the left represents D, the linearized 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. Specifically, c is a commit whose nearest, non-merge ancestor in the original DVC history is a, and $D(a, c)$ represents the commits, not including a or c, that developers made to other branches in that history in the intervening time. Definition 4.2 formalizes the set of files changed in a sequence of commits.

Definition 4.2 (Intervening Files) *The files modified in $D(a, c)$ "intervene" between c and a, its nearest, non-merge ancestor in D. These files therefore change the state of the project 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 $f \in F_i$, a syntactic or semantic conflict could occur. Semantic conflicts can be more distracting than syntactic conflicts as c's author must review each file in $f_m(c) \cap F_i$ to ensure their absence, since VC catches syntactic conflicts. For instance, one of the commits in $D(x, c)$ could have changed the semantics of a function used in c. Intuitively, the commit c is *distracted* if commits fall between it and its nearest, non-merge branch ancestor on D and one of those intervening commits changed a file that c also modified. In Fig. 2, all the commits except commits 1 and 5 are potentially distracted, depending on the set of files each commit changes. Definition 4.3 captures this intuition.

Definition 4.3 (Distraction) *The commit* $c \in D$ *is* distracted *if*

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

We cannot know how often files changed in both c and $D(a, c)$ will actually cause a conflict or require the developer committing c to understand a change that occurred in $D(a, c)$. We capture this uncertainty in the threshold δ, an *index of similarity*, or fraction of the size of the intersection of c's changeset and the changesets in $D(a, c)$ over the size of c's changeset. Each setting of δ represents a different assumption about how likely concurrent changes are to generate integration work in order to write the current changeset and form the commit c. At the right of Fig. 5, the fraction of the number of files in the intersection divided by the number of files in c pictorially depicts this index of similarity that we use to measure integration interruptions.

In Fig. 6, we plot the proportion of commits in the linearized history of the Linux kernel that are distracted as δ varies. At zero, we print the percentage of the time there are intervening files ($F_i \neq \emptyset$), regardless of whether they intersect with c's changeset. Even at $\delta = 1$, *i.e.* when we require $f_m(c) \subseteq F_i$, 2.8% commits are distracted, *i.e.* may encounter conflict or require review to ensure that no semantic assumption have been violated. 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 Fig. 6 with an index of similarity of 0.1. All of the files in the changeset of a commit c are distracted (index of similarity 1.0) with a confidence interval of 2.47% to 2.93%. Thus, a non-empty overlap occurs approximately once every 22 commits and a complete overlap every 35 commits.

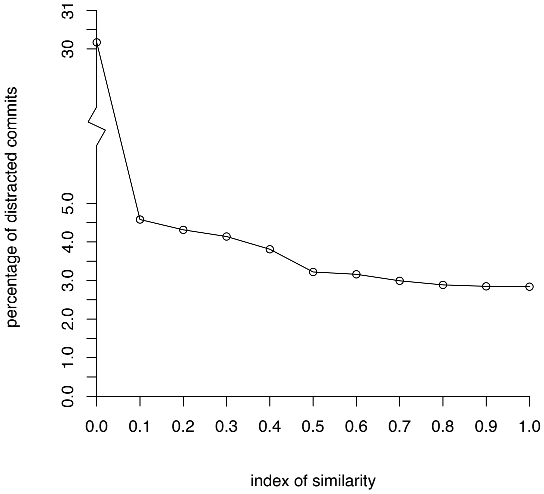


Fig. 6: Commits that require integration work as δ varies when the Linux kernel history is linearized.

Clearly, using a branch reduces distractions by delaying the need to resolve conflicts until merging the branch back into its parent. But how often does the use of branching actually avoid potential distractions in practice? Quantifying exactly how much distraction is avoided depends on how likely it is for concurrent changes to a single file to generate integration work. First, there is the rate, reported above, of non-empty intersection. That is, how often concurrent edits on different branches touch the same file. Second, there is the cardinality of that intersection; how many files are edited concurrently by different branches. Finally, there is the probability that the concurrent changes to a file in different branches actually generate integration work, at the very least in the form of confirming the changes made to the file are semantically noninterfering. We have established that on average, non-empty intersections occur once in every 22 commits. To be conservative, we assume that these intersections contain only a single file and that 90% of the time the programmer must examine the out-of-branch change made to it. To answer RQ3, we therefore conclude that working on branches protects a programmer from unexpected, unwanted semantic conflicts once in every $24.\bar{4}$ commits on average, across all branches that a developer works on.

4.4 Threats to Validity

The main threat to the external validity of our cohesion and distractions results is their dependence on Linux Git history, which may not be representative. Further, Git history can be perfected via “rebasing”, an operation that allows the history to be rewritten to merge, split or reorder commits [3]. Repositories hosted locally by developers are also not observable until branches are merged elsewhere.

Many projects we surveyed did not have a long enough DVC history (*i.e.* sample-size) to produce statistically significant results in all of our measures. Developers are still adjusting to DVC and may not have adopted history-preserving branching to break apart larger commits. As well, many contributions, even to DVC-using projects, are still submitted as large patches to the mailing-list, diluting, at least in the short term, the impact of DVC adoption.

D, the linearization of a DVC history that projects all branches onto a single, shared mainline overapproximates the integration interruptions faced by a developer, but we do not know by how much. Our use of directory distance as a cohesion measure 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. Our analysis assumes that all integration interruptions waste time, which may not always be the case.

5 Related Work

Version control systems have a long and storied past. In this paper, our concern is primarily the introduction of history-preserving branching and merging, and the resulting rich histories. The importance of preserving histories, including branches, has been well recognized [11]. The usefulness of detailed histories for comprehension [1] and for automated debugging [34] are by now well accepted. Some have even advocated very fine-grained version histories [18] for improved understanding and maintenance. Automating the acquisition of information, such as static relationships or why some code was commited, from accurate and rich VC history might improve developer productivity [15].

Branching in VCs have received a fair bit of attention [11]. Some have recommended "patterns" of workflows for disciplined use of branching [29]. Others advocate ways of branching and merging approaches [6] that mitigate the difficulties experienced with the branch and merge operations of earlier version control systems. Merging is a complex and difficult problem [19], which, if anything, will become more acute as a result of the transition to DVC and the corresponding surge in the use of branching we have shown. Bird *et al.* [4] developed a theory of the relationship between the goals embodied by the work going on in branches and the "virtual" teams that work on such branches.

Perry *et al.* [24] study parallel changes during large-scale software development. They find surprising parallelism and conclude "current tool, process and project management support for this level of parallelism is inadequate". Their conclusion anticipates the rapid transition to DVC that we chronicle in this paper.

The influential work of Viégas et al [30] uses a visualization methodology to study the historical record of edits in Wikipedia, and report interesting patterns of work (such as "edit wars"). To our knowledge, our paper is the first detailed study of the impact of DVC and its history-preserving branching and merging operations on the practice of large-scale, collaborative software engineering.

6 Conclusion and Future Work

Contrary to conventional wisdom, branching, not distribution, has driven the adoption of DVC: most projects still use a centralized repository, while branching has exploded (RQ1). These branches are used to undertake cohesive development tasks (RQ2) and are strongly coupled (RQ3). In the course of investigating these questions, we have defined two new measures: branch cohesion and distracted commits, a type of task interruption that occurs when integration work intrudes into development.

We intend to investigate how projects select branches to merge. The isolation that branches afford 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 history-preserving branching on the use of named stable bases [2].

References

1. D. Atkins. Version sensitive editing: Change history as a programming tool. *System Configuration Management*, pages 146–157, 1998.
2. S. Berczuk. Configuration Management Patterns. In *Third Annual Conference on Pattern Languages of Programs*, 1996.
3. 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 *Proc. 6th MSR*, 2009.
4. C. Bird, T. Zimmermann, and A. Teterev. A Theory of Branches as Goals and Virtual Teams. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*, 2011.
5. I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proc. ICSE 1999*, 1999.
6. J. Buffenbarger and K. Gruell. A Branching/Merging Strategy for Parallel Software Development. *System Config. Management*, 1999.

7. B. Cannon. PEP 374: Choosing a distributed VCS for the Python project, 2009. http://www.python.org/dev/peps/pep-0374.
8. I. Clatworthy. Distributed version control — why and how. In *Open Source Developers Conference (OSDC'07)*, 2007.
9. J. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage Publications, Inc, 3rd edition, 2009.
10. T. DeMarco and T. Lister. *Peopleware: productive projects and teams*. Dorset House Publishing Co., Inc. New York, NY, USA, 1987.
11. J. Estublier. Software configuration management: a roadmap. In *Proc. of the Conf. on The future of Software engineering*. ACM, 2000.
12. J.-M. F. Jacky, J. Estublier, and R. Sanlaville. Tool adoption issues in a very large software company. In *Proc. of 3rd Int. Workshop on Adoption-Centric Software Engineering*, 2003.
13. KDE. Projects/MoveToGit - KDE TechBase, November 2009. http://techbase.kde.org/Projects/MovetoGit.
14. S. M. Kerner. Subversion 1.7 released with some git-esque merging. *developer.com*, 2011.
15. A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proc. of the 29th ICSE*. IEEE, 2007.
16. T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proc. of the 28th ICSE*. ACM, 2006.
17. T. Lindlof and B. Taylor. *Qualitative communication research methods*. Sage, 2002.
18. B. Magnusson and U. Asklund. Fine grained version control of configurations in COOP/Orm. *Software Configuration Management*, pages 31–48, 1996.
19. T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.
20. Michael Koziarski. Personal interview, April 5 2009. rubyonrails.org.
21. Michael Sperber. Personal Interview, April 3 2009. xemacs.org.
22. C. Parnin and S. Rugaber. Resumption strategies for interrupted programming tasks. In *Proc. of 17th ICPC 2009*. IEEE, 2009.
23. R. Paul. DVCS adoption is soaring among open source projects. *ars technica*, January 7 2009.
24. 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.
25. L. Rocha. GNOME to migrate to git, March 2009. http://http://mail.gnome.org/archives/devel-announce-list/2009-March/msg00005.html.
26. A. Sarma, Z. Noroozi, and A. Van der Hoek. Palantír: raising awareness among configuration management workspaces. In *Proc. of 25th ICSE*, 2003.
27. Stephen Turnbull. Personal Interview, April 7 2009. xemacs.org.
28. R. van Solingen, E. Berghout, and F. van Latum. Interrupts: just a minute never is. *Software, IEEE*, 15(5):97–103, Sep/Oct 1998.
29. S. Vance. Advanced SCM branching strategies, 1998. http://www.vance.com/steve/perforce/Branching_Strategies.html.
30. F. Viégas, M. Wattenberg, and K. Dave. Studying cooperation and conflict between authors with history flow visualizations. In *Proc. of the SIGCHI conf. on Human factors in computing systems*. ACM, 2004.
31. R. S. Weiss. *Learning From Strangers : The Art and Method of Qualitative Interview Studies*. Free Press, November 1995.
32. P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *Proc. of the 2008 Int. W. Conf. on Mining software repositories*. ACM, 2008.
33. S. Zacchiroli. (declared) VCS usage for Debian source pacakges, February 2011. http://upsilon.cc/~zack/stuff/vcs-usage.
34. A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Software Engineering—ESEC/FSE'99*, pages 253–267. Springer, 1999.