Assessing the Value of Branches with What-if Analysis

Christian Bird
Microsoft Research
Redmond, WA, USA
cbird@microsoft.com

Thomas Zimmermann
Microsoft Research
Redmond, WA, USA
tzimmer@microsoft.com

ABSTRACT

Branches within source code management systems (SCMs) allow a software project to divide work among its teams for concurrent development by isolating changes. However, this benefit comes with several costs: increased time required for changes to move through the system and pain and error potential when integrating changes across branches. In this paper, we present the results of a survey to characterize how developers use branches in a large industrial project and common problems that they face. One of the major problems mentioned was the long delay that it takes changes to move from one team to another, which is often caused by having too many branches (branchmania). To monitor branch health, we introduce a novel what-if analysis to assess alternative branch structures with respect to two properties, isolation and liveness. We demonstrate with several scenarios how our what-if analysis can support branch decisions. By removing high-cost-low-benefit branches in Windows based on our what-if analysis, changes would each have saved 8.9 days of delay and only introduced 0.04 additional conflicts on average.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*Version Control;* D.2.9 [Software Engineering]: Management—*Software Configuration Management.*

General Terms: Measurement, Management, Human Factors

Keywords: Concurrent Development, Branches, Teams, What-if Analysis, Branch Refactoring, Coordination

1. INTRODUCTION

As software projects grow ever larger, both in terms of development teams and code size, coordinating work and changes to the system without causing undue harm or hindering others unnecessarily becomes a challenge. Thousands of developers making substantial changes to the contents and interfaces of hundreds of subsystems can quickly lead to disaster. One common solution to this problem is to use branches within the source code management system (SCM) [1]. Branches provide developers a facility for working individually or in teams on the source code of a software project independent of the changes being made by others. The branch provides a workspace where changes can be made, designs explored, and code tested in parallel with other teams working in other branches. Once a work task has been completed and the software is judged to be of sufficient quality (via testing or some other method), these changes can be integrated (also known as merged) into other branches (and their corresponding features) as they move towards a common branch (sometimes called "trunk", "master", or "root" in different SCMs) from which the product is released.

The practice of using branches to divide teams and tasks is used extensively at Microsoft for projects with large codebases, multiple concurrent releases undergoing development, and large teams. In recent years, with the advent of SCMs that facilitate easy branching and merging such as Git, Mercurial, Darcs, and Bazaar, many open source projects have begun using branching increasingly in their development practices. Prominent examples include the Linux kernel, Python, Perl, Ruby on Rails, X.org, and GNOME. Of the projects reporting their SCM in Debian, 61% indicated that they used next generation SCMs that facilitate branching [2]. Branching is a practice that is only becoming more prominent.

Branches do not come without a price, however. Since a change is initially only visible to the team working within its branch, it must be integrated into other branches before it can be seen by the rest of the project. The process of integrating changes from multiple branches can be difficult and error-prone, especially if changes on different branches conflict, either syntactically or semantically. In addition, this process takes time, which can slow teams on different branches that are dependent on each other or features which are related. Thus, branches incur an overhead in both developer effort and time, which, if not monitored and managed, can have severe impact on the project in the form of missed deadlines and increased failures.

In an effort to identify the extent of the cost of branching we surveyed developers at Microsoft to determine the difficulty and time associated with integrating changes from multiple branches as well as tools and practices used to verify such work. We also included questions to determine how often common problems with branches (also called "anti-patterns", initially identified by Appleton et al [3]) are encountered and what their severity is. Based on the survey and follow-up discussions with developers, we found that branch awareness and decisions surrounding branches are important pain points for many software projects, especially for SCMs that contain many branches, leading to many large integrations and long delays in moving changes across teams (anti-patterns known as "Branchmania" and "Big Bang Merge").

To address scenarios involving monitoring and making decisions about branches, we present a what-if analysis which serves to characterize individual branches in terms of *isolation*—how many conflicts they prevent—and *liveness*—how quickly changes made on the branches are conveyed to other teams. Our approach separates branches which exhibit the expected benefits (which we term

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

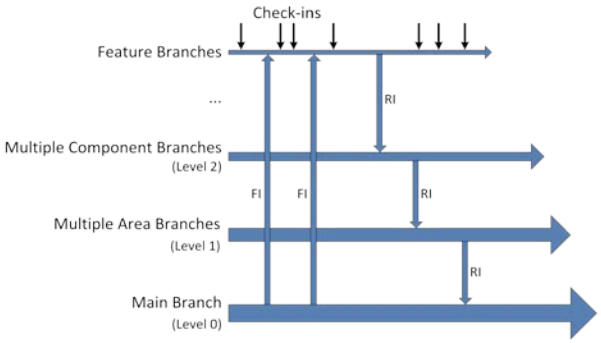


Figure 1. Branches in Windows development. FIs (forward integrations) move changes from parent to child branches. RIs (reverse integrations) move changes in the opposite direction.

high-benefit-low-cost branches), from those that slow development without providing high levels of isolation (low-benefit-high-cost). This analysis can aid developers by alerting them to parts of the branch structure that are "unhealthy" and hindering development or indicating which branches should be considered for removal.

Throughout this paper we demonstrate and evaluate the utility of our technique by illustrating its use on Windows. For example, we found that removing high-cost-low-benefit branches based on our analysis, changes would each have saved 8.9 days of delay and only introduced 0.04 additional conflicts on average. Over the past year, we also applied this analysis to Windows Mobile and Bing with similar results.

We make the following contributions in this paper

- Results of a survey of conducted with Microsoft developers on branching practices and issues encountered with branch use (Section 3).
- Technique for measuring the isolation and liveness of branches via what-if analysis (Section 4 and 5).
- Decisions scenarios supported by this technique and demonstration of these scenarios in the context of Windows development (Section 6).

2. BRANCHES AT MICROSOFT

Let us first illustrate how the Windows development process works. As shown in Figure 1, the Windows software development takes place in various branches of the version control system with tightly integrated schedules for code integration and comprehensive builds. There are several parts of Windows, each of which are developed in individual branches. (An example feature could be "Sound" in the component "DirectX" in the area "Multimedia".) Each of these individual branches can work as though the rest of the code base to be frozen, except for their own evolving features.

Engineers check-in their code to the feature branches. To ensure that the newly developed code in the feature branch maintains compatibility with the other changes committed to the main branches, the feature branches continually synchronize with the main branch (also called forward integration, or simply FI). After passing quality gates (for example, code coverage or static analysis) the code moves to the parent branch in the tree (reverse integration, RI). Once the code reaches the main branch (level 0) it is automatically integrated (FI'ed) with the rest of the code base to ensure that other code being developed is compatible with these changes. This process ensures stability in the main Windows branch, with a working version of Windows always available for system test and other purposes—however, this isolation also comes at a cost: transit time is increased as changes are only visible to other teams after several integrations. An example could be Multimedia branches, where changes have first to be integrated to the main branch before they are visible in the Networking branches.

Q1: Approximately how many hours per month do you spend on branching operations such as creating branches and integrating changes from other branches?

Number (decimals okay)

Q2: How much time does an integration take on average, including the time to verify that changes have been integrated correctly? Please provide also the unit of time.

Comment

Q3: How do you validate correctness of an integration?

Comment

Q4: Based on your experience how many errors are caused by incorrect integrations and in which areas do these errors occur?

Comment

Q5: Based on your experience how many times have you encountered the following branch anti-patterns?

► List of anti-patterns by Appleton et al. with description (see Figure 3).

Never | Once or twice | Occasionally | Frequently | No opinion

Q6: Based on your experience how large is the impact of the following branch anti-patterns on productivity?

► List of anti-patterns by Appleton et al. with description (see Figure 3).

No impact | Small impact | Moderate impact | Large impact | No opinion

Figure 2. The survey questions.

The branch structure in Windows (and other products at Microsoft) is typically chosen at the beginning of a release and remains mostly unchanged during the development of the release. Thus, there is not a strong notion of short-lived vs. long-lived branches. With this paper, we introduce an approach to quantitatively assess cost and benefit of branches to inform branch decisions.

3. SURVEY ON BRANCH USAGE

Many best practices exist in software configuration management such as the work by Berczuk [4] and Aiello [5], which also discuss how branches should be handled. As an example, a white paper from Perforce Software presents five best practices related to branches based on the authors' experience in deploying SCM systems [6]: branch only when necessary; don't copy when you mean to branch; branch on incompatible policy; branch late; branch instead of freeze. Appleton et al. present 32 patterns (best practices) for managing branching in parallel development projects [3]. They further present 12 common traps and pitfalls in branching that they call anti-patterns (see Figure 3). Examples of such anti-patterns are: creating too many branches *(Branchmania),* deferring branch merging and then attempting to merge all branches simultaneously *(Big Bang Merge),* or stopping all development activities while branching and merging, permitting only activities focused on shipping the impending release *(Development Freeze).*

In order to characterize branch usage, we sent an online survey to 370 Microsoft engineers in January 2011. For the design of the

Merge Paranoia — avoiding merging at all cost, usually because of a fear of the consequences.
Merge Mania — spending too much time merging code instead of developing it.
Big Bang Merge — deferring branch merging and attempting to merge all branches simultaneously.
Never-Ending Merge — continuous merging activity because there is always more to merge.
Wrong-Way Merge — merging into the wrong branch.
Branch Mania — creating too many branches.
Cascading Branches — branching but never merging back to the main line.
Mysterious Branches — branching for no apparent reason.
Runaway Branches — branching for single purpose evolves to multi-purpose branch for unrelated tasks.
Volatile Branches — branching with unstable files merged into other branches.
Development Freeze — stopping all development activities while branching and merging.
Integration Wall — using branches to divide the development team members, instead of dividing work.
Spaghetti Branching — integrating changes between unrelated branches.

Figure 3. Branch anti-patterns identified by Appleton et al [3] in the order they appeared in the survey.

survey, we followed Kitchenham and Pfleeger's guidelines for personal opinion surveys [7]. Our survey consisted of 12 questions (all optional) of which 5 were related to branching and are shown in Figure 2. The survey was anonymous as this increases response rates [8] and leads to more candid responses.

Since we wanted to solicit the opinions of people well-versed in working with the SCM system and dealing with branches, we chose our survey participants as the top 10% of people who had either created most branches, integrated most changes, or submitted most edits within the 12 months before the survey date. Participants were invited with a personalized email and could enter their names into a raffle of two US $50 gift certificates. We received 124 responses (a 33.6% response rate) without sending any reminders; other online surveys in software engineering have reported response rates ranging from 14% to 20% [9]. For the write-in questions the completion rate was between 93% and 98%. Almost all respondents were developers and most were fairly experienced, with a median of 11 years of work experience in the software industry and 7.25 years at Microsoft.

3.1 Integrations

On average the survey respondents spent 5.45 hours per month creating branches or integrating changes from other branches; the median was 3 hours **(Q1)**. These numbers may appear low, but often teams select a single person to be in charge of integrations and maintain a branch; this observation is supported by the 95th percentile of 15.45 hours and several of the free form comments in the survey. The time spent on branching operations depends also largely on the work area: build engineers spend significantly more time than developers.

For the time that integrations take on average **(Q2)**, we solicited responses in the form of comments rather than numbers because we wanted to know more about the specific context of the integrations. The time varied widely across responses, ranging from minutes for simple integrations to days for more complicated integrations. Most of the time is spent on resolving conflicts and verifying correctness. The total time spent depends largely on the presence of conflicts but also on "the size of the payload, how well the branches are partitioned in terms of work going on inside them, and how far back is the baseline". Several people and teams had developed custom tools and scripts to help them speed up the integration and its validation.

To validate the correctness of integrations **(Q3)**, respondents used a wide spectrum of techniques: manual inspection using diff tools, historic change information, custom tools and scripts, builds, program executions, test runs, and also code review. Several people stressed the importance of social communication, especially when there are merge conflicts and the resolution is not clear. Survey respondents pointed out that the validation often depends on the type of branch (private, one-man branches vs. public, team based and aggregation branches) and the complexity of the changes to be integrated. In Windows and other systems, feature branches typically have quality gates that must be met before a change can move to a different branch [3,10].

We also asked about how many errors are caused by incorrect integrations **(Q4)**; again we solicited responses in the form of comments rather than numbers. The consensus among respondents was that errors "happen from time to time" but relatively rarely because changes are validated extensively (e.g., through quality gates). However, errors "tend to be subtle because [if they happen] they often aren't noticed for a while when totally bizarre behavior occurs and it takes a long time to track down what happened." Frequent causes for integration errors are merge conflicts that were not resolved correctly or partial integrations missing a file. Errors often occurred in files that were not source code, such as XML files or build manifests, and are difficult to compare.

3.2 Anti-patterns

Figure 3 contains the descriptions of the anti-patterns that were presented to the surveyed developers. With respect to anti-patterns we focused on two aspects:

- **Frequency (Q5).** In the survey, we asked how many times each anti-pattern had been encountered by a person. For the question, we used an ordinal scale with four levels *Never, Once or Twice, Occasionally,* and *Frequently*. To avoid any guesswork by participants we provided an explicit option for *No opinion*.

  For the analysis of the responses, we followed the advice by Kitchenham and Pfleeger [7] and dichotomized the ordinal scale to avoid any scale violations. More specifically, for each anti-pattern k we computed the percentage $P_F(k)$ of the response "Frequently" among all responses (excluding responses that had no opinion).

- **Severity (Q6).** In addition, we asked which anti-patterns had the highest impact on productivity. We used an ordinal scale with four levels *No Impact, Small Impact, Moderate Impact,* and *Large Impact*; again we offered an explicit option or *No opinion*.

  We computed for each anti-pattern k the percentage $P_S(k)$ of the response "Large Impact" among all responses (excluding responses that had no opinion).

To identify anti-patterns with both a high frequency and a high severity, we additionally computed for each anti-pattern the product $P_{FS}(k)$ of the percentages $P_F(k)$ and $P_S(k)$.

$$P_{FS}(k) = P_F(k) \times P_S(k)$$

In Figure 4 we show a bubble chart of the frequency and severity of the branch anti-patterns. Each anti-pattern k is represented as a bubble; the position on the x-axis corresponds to the percentage $P_F(k)$ of responses that selected "Frequently" in Q5 and the position on the y-axis corresponds to the percentage $P_S(k)$ of responses that select "High Impact" on productivity in Q6. The bubble size corresponds to the combined percentage $P_{FS}(k)$. (Intuitively the product $P_{FS}(k)$ is the area of the rectangle spanned by the zero point and the point representing the anti-pattern.) To increase readability, we show full names only for the four anti-patterns with the highest P_{FS}-values; the other patterns are identified with numbers. The *Mysterious Branches* anti-pattern is not shown because no developers indicated that it had high severity and therefore the area is 0. The four highest ranked anti-patterns in Figure 4 are Development Freeze, Big Bang Merge, Integration Wall, and Branchmania.

- *Development Freezes* allow only activities that are focused on shipping the next release; work on subsequent releases is blocked until the software is released. Appleton et al. [3] discuss several solutions for this problem such as having parallel release and development lines.
- The anti-pattern *Integration Wall* means that branches are used to divide the development team members rather than the work itself. In a prior work related to this anti-pattern, we presented a preliminary study of how branches are used to organize goals and teams [11].
- For this paper, however, we focus on the anti-patterns *Big Bang Merge* and *Branchmania*, which are related: too many branches often lead to large integrations. Branchmania may also lead to longs delays

To recover from and prevent Branchmania, project members need awareness of how different branches are affecting their work. If developers can identify what branches are posing problems or are not actually needed, they can make decisions, such as where to integrate changes more frequently or which branches to remove, proactively. In this paper we provide a methodology to empirically assess branches and show how our approach can be used in a number of branch decision scenarios by illustrating them on Windows.

4. LIVENESS AND ISOLATION

It is important to understand how developers and other project members view branches within a project and what qualities are important to them.

At Microsoft project members care deeply about making fast and continuous progress. One aspect of this progress is how quickly changes made on branches are being seen by the rest of the project. We term this general property of how fast changes are being integrated into the rest of the project as ***liveness***. One specific measure of liveness is the amount of time that it takes for a change on a branch to reach the root. This is important because even if a developer has completed a feature or a bug-fix, the task is not considered complete until the change has reached the root without error. Furthermore, some defects do not manifest until changes from different branches reach each other and their interaction leads to problems; a high liveness helps to reveal these problems faster.

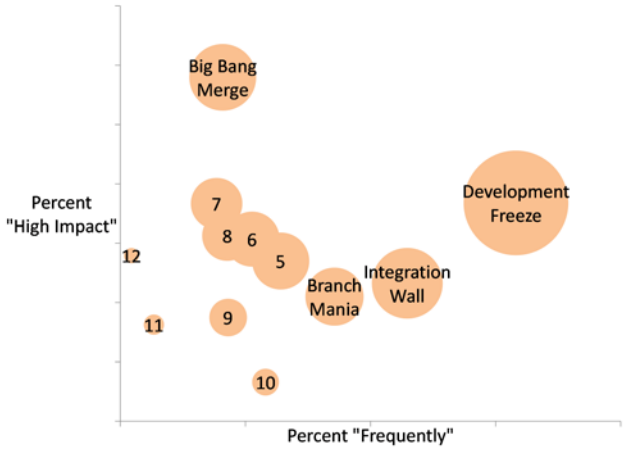


Figure 4. The frequency and impact of branch anti-patterns. To increase readability we used numbers to label some of the anti-patterns. 5: Merge Paranoia, 6: Never-Ending Merge, 7: Volatile Branches, 8: Merge Mania, 9: Spaghetti Branching, 10: Runaway Branches, 11: Cascading Branches, 12: Wrong-Way Merge, 13: Mysterious Branches

The interval between the checkin of a change and the time that it reaches the root branch is the *transit time* of the change.

Low transit times result in high liveness.

Many teams at Microsoft are interested in tracking the transit time of edits in their project. While transit time for individual edits is fairly straightforward to compute from SCM metadata, accounting which branches contribute the most to long transit times on the path to the root branch is more complex. Branches that increase transit times may represent a bottleneck and pose a severe barrier to project agility.

Developers are aided by branches because of the ***isolation*** that they afford. By making changes on a branch, developers are unaffected by others and need not worry about unduly impeding teams on other branches. Developers working in different branches can change the same file without immediate negative impact. Such activity means that the changes will eventually need to reach each other and their interactions will need to be resolved, but developers can wait until their changes are complete and stable beforehand. If a file is changed on two branches, A and B, then when the change from one is integrated into the other, or the two changes meet on another branch (perhaps the parent of A and B) there is a file-level conflict, which needs to be resolved.[1] We can measure how many conflicts occur when branches integrate with other branches, but many edits to the same file in two different branches may only result in one conflict if there is only one integration; consequently this measure does not accurately reflect isolation.

[1] Note that there are different levels of conflicts. Line-level conflicts are when two changes if they change the same lines in a file and require manual merging by developers. For this paper, we focus on file-level conflicts, which is when the same file has been changed on different branches. While some of these can be merged automatically, even these merges have caused enough problems that the integration still has to be validated by developers (for example via compilation and testing as indicated in our survey) to avoid errors based on bad merges [25].

(a) Initial history, H0, with branches A (parent) and B (victim).

(b) Move checkins from victim branch B to parent branch A.

(c) Remove the integrations between parent and victim.

(d) Final alternative history, H1.

Figure 5. The steps required to simulate the removal of a branch. In this figure, A is the parent branch, B is the child branch, solid circles represent checkins, hollow circles represent integration checkins between branches. Checkins are colored according to the branch that they were made on in the original history.

Accurately quantifying liveness and isolation via what-if analysis and using such data to aid project members' decision making is one of the main contributions of this paper.

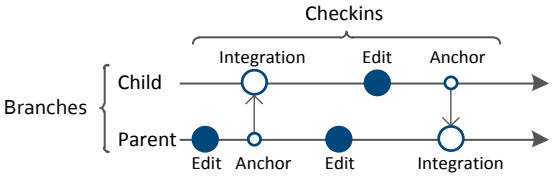
5. METHODOLOGY

Isolation and the liveness of a branch can provide valuable information to project members that can be used in a number of scenarios, from monitoring "branch health" to identifying branches that should be removed from the system. However, measuring isolation and liveness is not straightforward: How can we determine how many conflicts were avoided? How much code movement was slowed due to the use of a branch? To address such questions we introduce a what-if analysis as illustrated in Figure 6. We take an original development history H0 and apply several branch removal operations (Sections 5.2 and 5.3) to obtain an alternative history H1. We then compare how liveness and isolation change between H0 and H1 (Section 5.4). We demonstrate based on Windows 7 branches how such data can support several decision scenarios (Section 6).

5.1 Terminology

We first introduce basic definitions needed to describe our what-if analysis. Where possible, we adhere to accepted terms from SCM parlance and only describe key terms and concepts that would otherwise be confusing or ambiguous. For a more detailed and formal description of our methodology, we refer the reader to the online appendix [12].

To model file histories we use branches, edits, and integrations as shown in the diagram below. Time flows left to right. Edits and integrations are referred to as checkins. In this paper, we use circles on a horizontal line to denote checkins on a branch.



We represent a ***branch*** by the list of subsequent checkins that have been made to the file on the branch. Branches form a hierarchy in which the main branch is called the *root* branch. Likewise there are *parent* and *child* branches. Checkins are integrated between branches and propagated towards the root branch. The depth of a branch in the hierarchy is also referred to as the *level*, with level 0 being the root branch.

An ***edit*** includes a direct modification of a file by a developer such as editting its content as well as adding or removing a file from the SCM. Edits are a type of checkin and we denote edits with a solid circle.

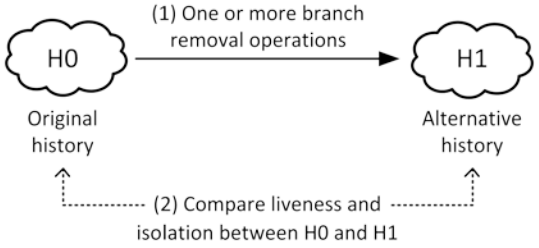


Figure 6. What-if analysis applies one or more branch removal operations to create an alternative history and then compare liveness and isolation.

An ***integration*** merges the contents of a file at a specific point in time on one branch (source) into another branch (target). In most cases, but not always, integrations occur between parent and child branches. Integrations are a type of checkin and we denote integrations with a large hollow circle. To model the state of the file at the "specific point in time" on the source branch, we introduce *anchors*, which are temporal placeholders on the source branch and contain no actual change to the file. Anchors are denoted with a small hollow circle. Note that the anchor and the corresponding integration have the same time:

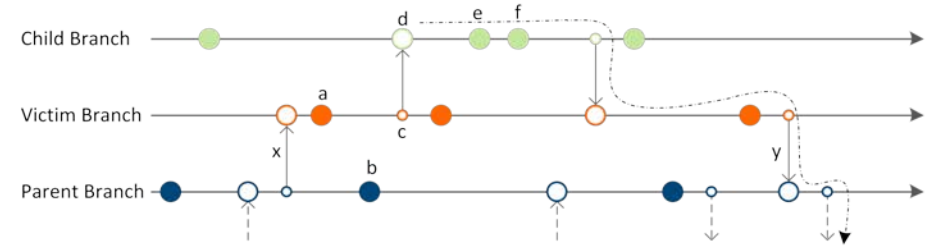
5.2 Simulated Removal of a Single Branch

The core part of our what-if analysis is removing a single branch. This allows us to explore a variety of alternative branch structures because scenarios where more than one branch is removed can be reduced to a series of single-branch-removal steps.

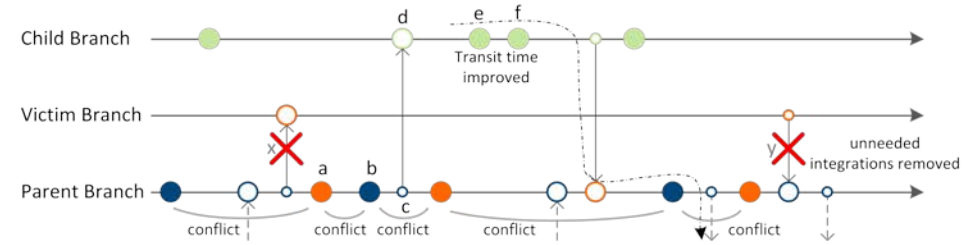
To simulate what would have happened if a branch was removed, we use the past development history and examine and modify the checkins and branch operations that involve the removed branch, the parent, and the children. Throughout this paper we refer to the branch being removed as the *victim* branch.

We first describe our simulation. Figure 5 shows the changes to the history that are involved when simulating the removal of the victim. Figure 5.a shows the original history for a subset of development history as it actually happened in the SCM. In this figure, A and B are horizontal lines representing branches. B is the victim and A is the victim's parent. As before, solid circles represent edits and hollow circles represent integrations from one branch to another. In this diagram, the color of the circle indicates which branch the checkin occurred on in the original history (Figure 5.a). We use the following steps to simulate an alternative history with the victim removed.

First we identify all edits that occur on the victim. Since we are simulating what would have happened if the victim had not existed, the edits would have been made on the parent branch. Thus we move these edits to the parent branch, while preserving chrono-



(a) Original development history. There are four integrations between the victim, parent, and child branches. The dashed line indicates the path for two commits on the child brenach through victim and parent branches to the root.



(b) Alternative history with victim branch removed. Two integrations have been removed. Five potential conflicts (indicated by arcs) have been introduced. The velocity time was improved for two edit checkins on the victim and three on the child.

Figure 7. Simulating branch removal.

logical order. Figure 5.b illustrates this step by moving the checkins from B to A.

Second, we remove integrations between the parent and the victim. Since all edits now reside on the parent branch, the integrations to and from the victim branch are no longer needed. This is illustrated in Figure 5.c where the integrations and anchors between parent and victim are removed.

Third, all integrations from the victim to its children or any other branches are modified so that they now have the parent branch rather than the victim as the source. Likewise, all integrations that have the victim as the target branch are modified so that they have the parent branch as the target. This step is not shown in Figure 5 as these integrations do not occur in the simple history shown.

The final alternative history is shown in Figure 5.d. Note that although all checkins occur on the parent branch, A, we can still determine which branch each checkin was made on in the original history. This is required for our branch metrics.

A more complex example is shown in Figure 7 (original history in 7.a; alternative history with victim removed in 7.b). The edits on the victim are moved to the parent branch in the alternative history. Integration and corresponding anchor checkins have either been removed (*x* and *y*) or rerouted (e.g., *c*→*d*). While more complex, this illustrates the effect of simulated branch removal:

- The path (shown as a dashed line in both histories) from the two edit checkins *e* and *f* is different in the original and alternative histories. In the alternative history, the edits reach the parent branch and leave towards the root earlier. The difference in transit time to the root branch is the **delay** caused by the victim.
- On the other hand, some edits that originally occurred on different branches are now subsequent, conflicting edits on the parent branch, as indicated by the arcs between edit checkins in Figure 7.b (for example, *a* in conflict with *b*). These conflicts characterize the **isolation** provided by the victim branch in the original history (where *a* and *b* were isolated on different branches).

5.3 Alternative Branch Structure Scenarios

Based on the single-branch-removal step described above, we perform what-if analysis for a wide spectrum of alternative branch structures to address different scenarios. Examples include:

What if a single branch is removed? – Is there a particular branch that is causing problems and should receive attention? This scenario simply applies the step described in the previous section. In Figure 7.b we illustrate the history produced when this step is applied to the history shown in Figure 7.a.

What if an entire branch subtree is removed? – Are there sections of the branch structure that aren't actually needed? This scenario selects a victim branch and removes the entire subtree rooted at the victim. For each branch removed from the subtree, we follow the process described in Section 5.2.

What if we only had branches up to level N? – Several teams asked how liveness and isolation would change if the depth of the branch hierarchy is restricted. This would limit the maximum number of hops for changes to reach the root branch and thus may maintain progress within a project. To assess this scenario, we remove all branches on levels greater than *N* with the process described in Section 5.2.

If a scenario requires removing multiple branches, the actual order of the branch removals does not affect the results. We record two branches for each checkin: the branch that the checkin occurred on in the *original history* (which is never changed throughout the entire analysis) and the branch that it was made on in the *alternative history* (which is initialized to the original branch but subsequently changed via branch-removal operations). This allows us to apply multiple branch removals in an arbitrary order because our analysis only needs the original branch and the *final* target branch for each checkin. Regardless of the order of branch removals, the final target branch for a checkin on a victim branch is always well-defined; checkins are moved to the first non-victim parent branch of their original branch. Thus, branch removal is associative and commutative.

5.4 Measuring Liveness and Isolation

We now describe the two measures that we use to quantify the benefit and cost of branches: *delay* and *provided isolation*. We present an intuitive description here. A more formal definition is available for the interested reader in the online appendix [12].

Delay. Recall that transit time is the time that it takes for an edit to reach the root from the branch that it was checked into. Once an alternative history has been created through branch removal operations from the original history, the transit time for some edits may have changed (for example, see checkins *e* and *f* in Figure 7). The *delay* that branches within a scenario incur is the difference in total transit time (the sum of transit times for all edits) for all edits between the original history H0 and an alternative history H1.

$$\text{Delay} = \text{TotalTransitTime(H0)} - \text{TotalTransitTime(H1)}$$

Isolation. We quantify the isolation that a branch provides by determining the number of conflicts that are avoided because of the existence of the branch. If there was concurrent activity to the same file in a branch and its parent or in a branch and its children, then the branch provided a level of development isolation and was beneficial. However, if development in a file on a branch had no potential conflicting changes in its children or parent, then this isolation was likely not needed. We calculate this by examining the checkins on the parent in the alternative history H1 and counting the number of conflicts. A conflict is a pair of subsequent edit checkins on a branch in the alternative history H1 that occurred on different branches in the original history H0 (for example, edits *a* in conflict with *b* in Figure 7.b). These are indicative of checkins that may be incompatible; even if the algorithm used by the SCM to merge textual changes runs without error, a developer must still validate (e.g., through builds and test runs) that the merged file does not contain any problems. Thus each conflict introduced by the removal of a branch represents a non-trivial amount of additional work for a developer. We compare the number of conflicts in H1 against the number of conflicts in H0 during integrations.

$$\text{Isolation} = \text{Conflicts(H1)} - \text{Conflicts(H0)}$$

While we cannot know what exactly would have actually happened had a branch not existed, our alternative history effectively quantifies the isolation provided and delay introduced by a given branch. Even if developers coordinated their changes to avoid conflicts, this *would* be additional effort.

5.5 Normalization

Some branches have an order of magnitude more changes than others. Thus total delay and total isolation may be misleading, especially when comparing different branches. As an example, branches with many edits will have more influence on total delay just because of the high number of edits. Therefore, depending on the question that we are interested in answering, delay and isolation may need to be normalized: For scenarios related to comparisons and decisions on individual branches (or subtrees), we normalize the delay and isolation measures. For scenarios related branch structure as a whole, we do not normalize. More specifically, for the scenarios presented in this paper, we normalize in the following ways.

Normalized delay. The removal of branches can only affect the transit time of edits on the victim branches and on their children (recursively). We call the edits on these branches the affected edits (regardless of if their transit time is changed). Therefore, when normalizing delay, we divide the sum of the differences in transit time by the number of affected edits:

$$\text{NormalizedDelay} = \frac{\text{Delay}}{\text{NumberAffectedEdits(H0,H1)}}$$

Put simply, the normalized delay for a branch is the average change in transit times for edits that occur on or below the branches that have been removed. We say that the branches incur this delay per edit for edits on and below them.

Normalized isolation. Here we normalize by the maximum number of possible conflicts that can be introduced. All edit checkins on the removed branches end up in the victims' parent branches (there may be multiple victims if multiple branch removal steps are taken from H0 to create H1). Thus, the maximum number of conflicts occurs when there is perfect interleaving of edits that were created on different branches in H0:

$$\text{PossibleConflicts} = \text{NumberOfEditsOnVictims} + \text{NumberOfEditsOnParents} - 1$$

We normalize isolation by dividing the number of conflicts that the branch avoids by the maximum number of possible conflicts.

$$\text{NormalizedIsolation} = \frac{\text{Isolation}}{\text{PossibleConflicts}}$$

Intuitively, the normalized isolation indicates how many conflicts per edit checkin a branch prevents.

6. DECISION SUPPORT SCENARIOS

Having described our methodology, we now illustrate these scenarios by using our analysis on Windows 7 development history.

6.1 Branch Health

Our branch assessment metrics can provide awareness of branch health to project stakeholders such as developers, managers, and build engineers. In the same way that test results or code coverage metrics can alert project members to potentially problematic parts of the software, the measures of isolation and liveness can be used to alert project members to parts of the branch structure that are unnecessarily impeding progress. Over the past year, we have provided branch health reports to Windows, Windows Mobile, and Bing. For each branch the reports contain standard measures such as number of edits, integrations, edit/conflict ratio as well as delay and isolation based on the scenarios listed in Section 5.3. Our analysis helped the product groups identify what specific parts of the branch structure were responsible for low liveness.

Note that high-delay-low-isolation branches do not necessarily have to be removed from the branch hierarchy. As with most, if not with all metrics, the actions to be taken depend highly on the context [13]. For example, branches might exhibit a high delay because of a temporary code freeze or because they are integral parts of the quality assurance and serve as quality gates; these branches should likely not be removed. Other than removing a branch, a team can also decide to integrate more frequently to the parent branch to decrease delay.

6.2 Separating the Sheep from the Goats[2]

Figure 8 contains a scatterplot showing the normalized delay and isolation of all branches during the complete development cycle of Windows 7. The graph shows the results for recursive branch removal (removing a branch and all of its descendants). Isolation

[2] Separating the sheep from the goats is an English idiom and an allusion to a biblical metaphor (Matthew 25:32-34) in which sheep provide value and are blessed while goats do not and are cursed.

is shown along the x-axis (farther right is better as it means more conflicts are prevented by a branch) and delay is shown on the y-axis (lower is better). We consider isolation to be the *benefit* that a branch provides at the *cost* of delay.

High-benefit-low-cost branches (sheep) are colored green and on the bottom and right. Low-benefit-high-cost branches (goats) are colored red and are on the top and left. Medium-benefit-medium-cost branches (hybrids) provide isolation but also incur delays and are near the *x*-*y* line. Our categorization is based on ranking each branch in terms of the isolation that it provides and the delay that it incurs. We give each branch two ranks, one for delay (from least delay to most) and one for isolation (from most isolation to least). Then branches are sorted by the sum of their two ranks. The first 25% of branches are labeled sheep in graph in Figure 8; the last 25% have highest delay and lowest isolation and are labeled goats; the middle 50% are hybrid branches. This method of ranking is simply *one* way to combine isolation and delay and is not intended to be definitive; for example other rankings could weight one measure more than the other. We include our two dimensional thresholds in the graph for ease of reading.

The graph shows a number of extreme branches. Approximately 7% of the branches don't avoid any conflicts at all (points on the *y* axis). These branches do not provide any benefit. In contrast, 27% of the branches provide isolation while incurring only little or no delay (branches that lie near the *x* axis). These branches are the ideal as they provide benefit at almost no cost.

Such identification of sheep and goat branches is most useful if it can inform decisions about future branch structures and if it can be used at any point in the development cycle. To assess whether what-if analysis can inform decisions *during* development, we evaluated the effect of making decisions based on what-if analysis prior to the end of a product development cycle. Like many industrial projects, Windows development occurs in iterations around milestones. We performed our what-if analysis to evaluate branches based on development of Windows 7 that occurred prior to the end of the first milestone (M1). That is, we labeled each branch as a sheep, a goat, or a hybrid based on the delay and isolation for that branch in the first milestone. We then evaluated the effect of removing goat branches (those that provided the least isolation while causing the most delay) for the remaining milestones after M1 to determine if taking action based these metrics would be effective.

Table 1 shows the results. By removing branches labeled goats in M1, each edit saved, on average, 8.9 days of delay from M1 to the end of development and experienced 0.04 additional conflicts each. In contrast, removing sheep branches at M1 would only save 2.3 days of delay per edit while incurring 0.22 additional conflicts (more conflicts **and** less saved time than goats). To put these values in perspective, we compared this to the optimal choice of branches to remove if we had perfect foresight and removed the branches that actually performed worst post-M1. In that case, 9.7 days of delay per edit would be saved at the cost of 0.035 conflicts each. Thus, making decisions on which branches are the most costly in M1 achieves 92% of the maximum possible cost savings while incurring only 17% more conflicts than the least possible.

We also examined the correlation between branch categorization based on M1 development and branch categorization after M1. We found that the category remained the same for 85.3% of the branches and a Kendall's τ correlation of branch category before and after M1 was 0.86 ($p \ll 0.01$). Sheep branches tend to remain sheep, goat branches remain goats, etc.

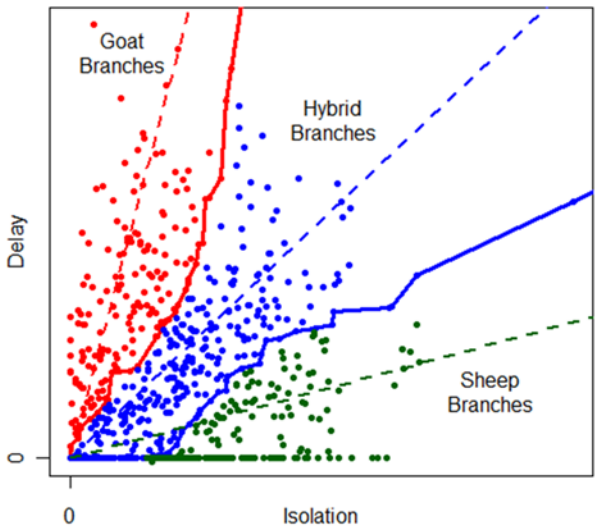


Figure 8. Usefulness of Branches in terms of provided isolation and delay based on *recursive branch removal*. Top and left shows goat branches (low benefit, high cost) and bottom and right show sheep branches (high benefit, low cost) while those in the center are hybrid branches that exhibit a tradeoff between delay and isolation.

Removal Strategy	Delay Saved	Conflicts Added
Sheep	2.3 Days	0.218
Goats	8.9 Days	0.042
Optimal	9.7 Days	0.035

Table 1. The number of days saved and conflicts added per edit if sheep or goats, classified from data in the first milestone, are removed for later milestones.

Both of these results indicate that what-if analysis based on development data mid-cycle is reliable. Put more pragmatically, decisions about branch practices such as which to remove or which to focus resources on can be made during development with high confidence based on measurement earlier in the development cycle.

6.3 Quantifying the liveness–isolation tradeoff

Our choice of division of branches into sheep, goats, and hybrids based on a 25%-50%-25% split may seem arbitrary, and indeed it is to some degree. The divisions into the interquartile range and the resulting visualization were inspired by standard boxplot analysis [14]. The divisions simply represent the tradeoff between providing isolation and reducing delay. To quantify this tradeoff, we computed regression lines (shown in Figure 8) for each group. For confidentiality reasons, we are unable to disclose the actual absolute measurements. However, since a regression line defines proportions, we normalize to one conflict and use the generic term delay "unit". We found that the average tradeoff was 1 prevented conflict per edit at the cost of 3 delay units for sheep, 11 delay units for hybrid, and 46 delay units for goats.

Thus, if one is willing to deal with one conflict per edit in order to save 46 units of delay per edit (these are actually ratios, so this is the same as one conflict every two edits to save 23 delay units), then goat branches should be eliminated. In contrast, removing a sheep branch will only save 3 delay units per additional conflict.

In practice, development teams can define the tradeoffs that they are willing to accept and make decisions accordingly. Thresholds are, in fact, not required in order to use what-if analysis results. For example, branches may be ranked according to some combination of isolation and delay and those with the lowest ranks could be removed.

6.4 Depth Analysis

Managers have considered limiting the maximum depth of the branch structure due to a belief that liveness would be improved if there are fewer branch levels. Until now, this belief has not been empirically confirmed or refuted.

We used what-if analysis to investigate branch depth by looking at the total isolation and total delay when restricting the branch structure to different maximum depth levels. A depth level of *n* means that there are at most *n* levels of branches below the root (which has level 0). Branches closer to the root are called *shallow* branches, while branches further away from the root are called *deep* branches. In this scenario we are not comparing branches to each other, but rather taking a global view on the branch structure as a whole. Therefore we use total delay and total isolation and show the percentage decrease in transit time and the percent of edits that cause conflicts.

Our findings are shown in Table 2 and are two-fold. First, the branches at very deep levels don't actually incur very much delay. In fact, limiting the depth to four levels of branching saves less than 0.1% of the total transit time. Most of the delay can be attributed to the branches closer to the root. A policy of maximum branch depth would have to make the branch structure quite shallow for a non-trivial effect on delay; however, this would come at a rather high cost of severely reduced isolation.

- For an 8.9% speedup, Windows would have had to deal with 30.3% of the edits creating conflicts (maximum depth of 1).
- If the branch structure had only a single branch, that is the root (maximum depth of 0), the transit time would reduce by 100% to 0 for all edits, but then 40.4% of edits would incur conflicts. Having only a single branch is not reasonable for other reasons than just conflicts: build breaks would stall the entire project, preventing thousands of people from being able to work.

These findings suggest that deep branches actually do not impede liveness. They may not be needed, as they also do not provide much isolation, but removing them would have only a trivial effect, as they integrate their changes to parent branches on a frequent basis. In contrast to conventional wisdom that the holdup is a deep branch structure, our results show that in the case of Windows, the key to increasing liveness may actually lie in finding ways to move changes through shallow branches more quickly.

Max Depth	Transit Time Decrease	Isolation (edits in conflict)
0	100%	41.0%
1	8.9%	30.3%
2	3.4%	10.5%
3	1.4%	2.3%
4	< 0.1%	0.2%

Table 2. The decrease in transit time and percent of edits in conflict if the branch structure is limited to a maximum depth. Depth of 1 represents 1 level of branching below the root, etc.

7. DISCUSSION

In this section we discuss future areas of research in the area of SCM branches. We also present assumptions in our methodology, potential threats to result validity as well as our mitigation steps, and common misconceptions.

7.1 Branch Refactoring and Optimization

In this paper, we have introduced a technique to empirically characterize delay and isolation for individual branches. This supports data-driven decision making on branches, for example to identify candidates for deletion.

We believe that this is just the first step towards a new discipline, **branch refactoring**, which is the process of improving and refining branch structures as a software project evolves. For this paper we focused on the refactoring "Remove useless branch". However, as projects evolve there will be other opportunities for refactoring such as "Create new branch", "Split branch", "Merge related branches", and "Bypass branch". A related area is **branch optimization**, which is concerned with distributing files and people across branches based on empirical evidence.

Both branch refactoring and branch optimization offer opportunities for new research and tool development:

- Assemble a branch refactoring catalogue with empirically validated guidelines of when to apply a refactoring.
- Develop techniques to distribute files, people, and teams across branches.
- Build a recommender system to identify branch refactoring opportunities.
- Train prediction models to predict which branches will turn from sheep to goats.
- Empirical studies on relationship between branch structures and code quality.

7.2 Assumptions & Threats to Validity

For our survey we identified the following threats to validity. Our *selection of survey participants* was constrained to only experienced engineers, in our context, engineers who were most active in the SCM. While this skews our results to these engineers, they are also the ones who will benefit most by better branch structures. A related threat is that to some extent our survey operated on a *self-selection principle*: the participation in the survey was voluntary. As a consequence, results might be skewed towards people that are likely to answer the survey, such as engineers with extra spare time—or who care about branch structures. Avoiding the self-selection principle is almost impossible. As pointed out by Singer and Vinson, the decision of responders to participate "could be unduly influenced by the perception of possible benefits or reprisals ensuing from the decision" [15]. Some of our analysis is based on self-reported data (e.g., integration time, Q2). However, software developers are known to underestimate effort [16] and we consider the estimates to be a lower bound. For any empirical study, it is difficult to draw general conclusions because of a large number of contextual variables [17]. For example, different SCMs use different merging tools which may affect developers perceptions of difficulty of integration. In addition, the process used by a development project can have a strong relationship with branch structure and frequency of integrations. However, we are confident our techniques can be applied to other projects, especially given the increased popularity of branching through distributed version control systems [18]. To increase the generality of our results, we hope to partner with academic researchers to repli-

cate this analysis on open source projects such as the Linux kernel.

Some key assumptions underlie our results. First, our measures of delay and isolation assume that a similar sequence of checkins and integrations would occur in a different branch structure. We argue that the changes themselves are necessary to achieve the desired software functionality and that dependencies between edits introduce a partial order that imposes a similar sequence. Further, to minimize the risk of this assumption, each scenario is evaluated in its own alternative history with the rest of the branch structure unchanged rather than evaluating the effect of making multiple changes which would likely effect development behavior more intrusively. Second, we assume that if a branch had not existed, the changes and integrations made on that branch would have instead been made on the parent. Lastly, some branches at shallow depths play a quality gating role, whereby checkins are aggregated, tested, and in some cases corrected before moving to the root. Branches with such roles should be considered carefully when making decisions.

7.3 Common Misconceptions

A common misconception about industrial research at large companies such as Microsoft is that software projects at Microsoft are not representative of other software projects. While projects might be larger in size, most development practices at Microsoft are adapted from the general software engineering community and also used outside Microsoft. For example, branches are frequently used at other companies [19] and in open-source [20].

Another frequent misconception is that empirical research within one company or one project is not good enough, provides little value for the academic community, and does not contribute to scientific development. Historical evidence shows otherwise. Flyvbjerg provides several examples of individual cases that contributed to discovery in physics, economics, and social science [21]. Beveridge observed for social sciences: *"More discoveries have arisen from intense observation than from statistics applied to large groups"* (as quoted in Kuper and Kuper [22], page 95). Please note that this should not be interpreted as a criticism of research that focuses on large samples or entire populations. For the development of an empirical body of knowledge as championed by Basili [17], both types of research are essential. The work presented in this paper has been successfully applied to three Microsoft products (Windows, Bing, and Windows Phone).

8. RELATED WORK

To the best of our knowledge this is the first work that empirically assesses the usefulness of software development branches at an individual level. We were unable to find metrics similar to the ideas of delay and isolation in previous research. We also provide empirical insights into multi-branch software development and qualitative observations from developers on efficiency of branches. Standard simulation techniques [23] rely on distributions and other methods to generate data and evaluate outcomes. In contrast, we use actual data recovered from Windows development and replay these activities on differing branch structures to answer what-if scenarios. This gives us increased confidence in our results as we are not trying to generalize development behavior, but use it as it actually happened.

Two questions in our survey among developers were based on the list of anti-patterns for parallel software development [3]. Several books and articles discuss best practice for software configuration management and branch structures [4,5,6]. However, these practices are mainly based on the authors' experience and less on empirical evidence. With this work we provide a way to empirically assess branches.

Perry et al. observed a high degree and multiple levels of parallel development in the 5ESS system [1]. They also observed a significant correlation between the degree of parallel work and the number of quality problems in a given component. Zimmermann studied workspace updates in GCC, JBoss, JEdit, and Python, and observed that between 3.9% and 20.2% of commits had integrations [24]. Between 22.8% and 46.6% of integrations could not be automatically resolved by CVS and resulted in conflicts. Brun et al. pointed out that besides textual conflicts, there are also compile conflicts (program does not compile) and build conflicts (program fails test suite) when integrating changes [25].

Independently from us in 2011, Phillips et al. conducted a survey among 140 version control users and asked how branching and merging are used in practice and what defines a successful branching strategy in terms of *user satisfaction* [26]. Premraj et al. surveyed 16 software personnel on their use of branching and merging [19]. Our survey complements both studies as the questions asked are different. We also go beyond user satisfaction and introduce quantitative measurements for cost and benefit of branches.

In our earlier work, we explored how people and files span across multiple branches to understand how socio-technical factors affect parallel development [11]. The earlier work measured similarity between branches—the focus of this paper is instead on a different aspect of branching, namely to measure the cost and benefit of branches. We also introduced an algorithm to identify the changes and bug fixes that are included in a reverse integration [27].

Several tools have been proposed to increase the awareness of changes across different branches or workspaces with the goal to reduce conflicts: Sarma et al. developed Palantír, which shares information about changes to the same files across different workspaces [28,29]. Sarma and colleagues also provided quantitative evidence of the benefits of workspace awareness in software development [30] and compared Palantír with two other awareness tools FASTDash [31] and CollabVS [32]. In a workshop paper, Guimarães and Rito-Silva proposed a system for real-time integration of changes [33]. Brun et al. proposed speculative conflict detection, which searches for unrecognized conflicts across branches and opportunities for straightforward merging [25].

9. CONCLUSION

We have presented a survey on how branches are used at Microsoft and an empirical what-if analysis to assess cost and benefit of branches to aid in a number of branch-related scenarios. Our approach helps to identify and separate, high-benefit-low-cost branches from low-benefit-high-cost branches. Such findings enable informed decisions about branch structures and processes (such as frequency of integrations) regarding branches and allow refining a branch structure as a project progresses. We have only touched on branch removal as one possible branch refactoring operation. There are many other possible operations such as adding, splitting, merging, and restructuring branches. This presents a potential new research area.

With the rise of distributed version control systems such as Git and Mercurial, branching has become more common in open source software development [18] and accessible to a wider research audience. This provides an opportunity for academic research to have immediate impact in industry, where branches are often used to deal with the complexity of software.

10. REFERENCES

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

[2] Zacchiroli, Z. *VCS usage for Debian source packages*. http://upsilon.cc/~zack/stuff/vcs-usage/.

[3] Appleton, B., Berczuk, S., Cabrera, R., and Orenstein, R. Streamed Lines: Branching Patterns for Parallel Software Development. In *The Pattern Languages of Programs Conference* (1998).

[4] Berczuk, S.P., Appleton, B., and Brown, K. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley Professional, 2003.

[5] Aiello, R. and Sachs, L. *Configuration Management Best Practices: Practical Methods that Work in the Real World*. Addison-Wesley Professional, 2010.

[6] Wingerd, L. and Seiwald, C. *High-level Best Practices in Software Configuration Management*. White Paper, Perforce Software, 1996. http://www.perforce.com/perforce/papers/bestpractices.html.

[7] Kitchenham, B.A. and Pfleeger, S.L. Personal Opinion Surveys. In Shull, F. et al., eds., *Guide to Advanced Empirical Software Engineering*. Springer, 2007.

[8] Tyagi, P.K. The Effects of Appeals, Anonymity, and Feedback on Mail Survey Response Patterns from Salespeople. *Journal of The Academy of Marketing Science* (1989).

[9] Punter, T., Ciolkowski, M., Freimut, B.G., and John, I. Conducting on-line surveys in software engineering. In *Proc. of International Symposium on Empirical Software Engineering (ISESE '03)* (2003), 80–88.

[10] Melanchthon, D. and Scheer, O. Grossbaustselle. Making of Windows 7. *c't Magazine*, 23 (2009), 80-86.

[11] Bird, C., Zimmermann, T., and Teterev, A. A Theory of Branches as Goals and Virtual Teams. In *International Workshop on Cooperative and Human Aspects of Software Engineering* (2011).

[12] Bird, C. and Zimmermann, T. *Appendix to Assessing the Value of Branches with What-if Analysis*. Technical Report MSR-TR-2012-33, Microsoft Research, 2012.. http://research.microsoft.com/apps/pubs/?id=161385.

[13] Zeller, A., Zimmermann, T., and Bird, C. Failure is a Four-Letter Word: A Parody in Empirical Research. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering (PROMISE 2011)* (2011).

[14] Dowdy, S., Wearden, S., and Chilko, D. *Statistics for research*. John Wiley & Sons, 2004.

[15] Singer, J. and Vinson, N.G. Ethical issues in empirical studies of software engineering. *IEEE Trans. Software Eng.*, 28, 12 (2002), 1171-1180.

[16] Jørgensen, M. and Grimstad, S. Software Development Estimation Biases: The Role of Interdependence. *IEEE Transactions on Software Engineering, Preprints* (April 2011).

[17] Basili, V.R., Shull, F., and Lanubile, F. Building knowledge through families of experiments. *IEEE Trans. Software Eng.*, 25, 4 (1999), 456-173.

[18] Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., Germán, D.M., and Devanbu, P.T. The promises and perils of mining git. In *MSR '09: Proceedings of the 6th International Working Conference on Mining Software Repositories* (2009), 1-10.

[19] Premraj, R., Tang, A., Linssen, N., Geraats, H., and Vliet, H.v. To branch or not to branch. In *ICSSP '11: Proceedings of the International Conference on Software and Systems Process* (2011), 81-90.

[20] Barr, E., Bird, C., Rigby, P., Hindle, A., German, D., and Devanbu, P. Cohesive and Isolated Development with Branches. In *International Conference on Fundamental Approaches to Software Engineering* (2012).

[21] Flyvbjerg, B. Five misunderstandings about case-study research. *Qualitative inquiry*, 12, 2 (2006), 219-245.

[22] Kuper, A. and Kuper, J. *The Social Science Encyclopedia*. Routledge, 1985.

[23] Müller, M. and Pfahl, D. Simulation Methods. In Shull, F. et al., eds., *Guide to Advanced Empirical Software Engineering*. Springer, 2007.

[24] Zimmermann, T. Mining Workspace Updates in CVS. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories* (2007), 11.

[25] Brun, Y., Holmes, R., Ernst, M.D., and Notkin, D. Proactive detection of collaboration conflicts. In *ESEC/FSE '11: Proceedings of he European Software Engineering Conference and the Symposium on the Foundations of Software Engineering* (2011).

[26] Phillips, S., Sillito, J., and Walker, R. Branching and Merging: An Investigation into Current Version Control Practices. In *CHASE '11: Proceedings of the Workshop on Cooperative and Human Aspects of Software Engineering* (2011).

[27] Tarvo, A., Zimmermann, T., and Czerwonka, J. An Integration Resolution Algorithm for Mining Multiple Branches in Version Control Systems. In *ICSM '11: Proceedings of the 27th IEEE International Conference on Software Maintenance* (2011).

[28] Sarma, A. *Palantir: enhancing configuration management systems with workspace awareness to detect and resolve emerging conflicts*. Doctoral Dissertation, University of California, Irvine.

[29] Sarma, A., Noroozi, Z., and van der Hoek, A. Palantír: Raising Awareness among Configuration Management Workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (2003), 444-454.

[30] Sarma, S., Redmiles, D., and van der Hoek, A. Empirical evidence of the benefits of workspace awareness in software configuration management. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (2008), 113-123.

[31] Biehl, J.T., Czerwinski, M., Smith, G., and Robertson, G.G. FASTDash: a visual dashboard for fostering awareness in software teams. In *CHI '07: Proceedings of the 2007 Conference on Human Factors in Computing Systems* (2007), 1313-1322.

[32] Dewan, P. and Hegde, R. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. In *ECSCW '07: Proceedings of the Tenth European Conference on Computer Supported Cooperative Work* (2007), 159-178.

[33] Guimarães, M.L. and Rito-Silva, A. Towards real-time integration. In *CHASE '10: Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering* (2010), 56-63.