Evaluation of Version Control Merge Tools

Benedikt Schesch
b.schesch@googlemail.com
ETH Zürich

Ryan Featherman
ryan.featherman3@gmail.com
Microsoft

Kenneth J. Yang Ben R. Roberts Michael D. Ernst
{kjy5,brober3,mernst}@cs.washington.edu
University of Washington

ABSTRACT

A version control system, such as Git, requires a way to integrate changes from different developers or branches. Given a merge scenario, a merge tool either outputs a clean integration of the changes, or it outputs a conflict for manual resolution. A clean integration is correct if it preserves intended program behavior, and is incorrect otherwise (e.g., if it causes a test failure). Manual resolution consumes valuable developer time, and correcting a defect introduced by an incorrect merge is even more costly.

New merge tools have been proposed, but they have not yet been evaluated against one another. Prior evaluations do not properly distinguish between correct and incorrect merges, are not evaluated on a realistic set of merge scenarios, and/or do not compare to state-of-the-art tools. We have performed a more realistic evaluation. The results differ significantly from previous claims, setting the record straight and enabling better future research. Our novel experimental methodology combines running test suites, examining merges on deleted branches, and accounting for the cost of incorrect merges.

Based on these evaluations, we created a merge tool that outperforms all previous tools under most assumptions. It handles the most common merge scenarios in practice.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**.

KEYWORDS

software merging, version control, structured merge, mining software repositories, git merge, Spork, IntelliMerge, git-hires-merge

1 INTRODUCTION

A merge occurs when developers using a version control system (VCS), such as Git, integrate changes from different branches — for example, after two developers concurrently edit their own copies of the code. Merge tools, commonly utilized by VCSs, are designed to automatically integrate changes. The input to a merge tool is a pair of revisions or commits.

A merge tool can produce three kinds of results. In an *unhandled* merge, the tool is unable to integrate some changes and reports conflicts. The programmer must handle the conflicts. A *clean* merge has no unhandled changes (that is, no conflicts); it can be further divided into the other two results, *correct* and *incorrect* merges. In a correct merge, the tool produces the desired output: a runnable program version that properly integrates both changes. In an incorrect merge, the tool still produces a single merged program version without conflicts. However, that program version fails to correctly integrate both changes. An incorrect merge could result in a compilation error or in a runnable program that fails its tests.

Merging is a well-known pain point for developers, so many new merge tools have been proposed [2–9, 12, 24–27, 30, 32, 33, 47, 49, 51, 54]. These papers present and evaluate many exciting and intriguing ideas, such as structured merging. Unfortunately, the state-of-the-art tools Hi-res Merge [1], IntelliMerge [46], and Spork [30] have not been compared to one another. To the best of our knowledge, all previous evaluations of merge tools suffer from at least one of these three problems: they do not evaluate *merge correctness*, they are not evaluated on *representative merges*, or they do not compare with *state-of-the-art tools*.

We address these shortcomings and provide a more comprehensive evaluation of merge tools. To detect incorrect merges at scale, we utilize automated testing. To indicate how well a merge tool works under a realistic mix of development scenarios, we collected merges from all branches, including ones that have been deleted from the VCS. We compared a broad set of merge tools (section 2.3), including some that have never been evaluated before and new ones that we created.

The contributions of this paper include:

- A novel experimental methodology for evaluating merge tools. It validates merges via testing, includes non-main-branch merges, and quantitatively accounts for the cost of incorrect merges.
- Open-source experimental infrastructure that implements our methodology.
- Comparison of both industrial and research merge tools.
- New merge tools that outperform existing ones.

Here are some of the novel findings from our research:

Accounting for the **cost of incorrect merges** changes results, compared to previous experiments. The best merge tool depends on the relative cost k of incorrect merges to unhandled merges (fig. 9). Spork is the best merge tool if incorrect merges cost no more than unhandled merges ($k = 1$). But by $k = \sim 2$, Spork is the *worst* tool other than IntelliMerge. Future research should determine the value of k and should develop heuristics to reward tools for producing unhandled merges that are easy to manually resolve, or producing incorrect merges that are easy to debug.

Some **previous proposals** underperformed their claims. IntelliMerge [46] is rarely applicable. When it is applicable, it produces more incorrect merges than Git Merge does. Merging edits on adjacent lines, as recommended by [39], improves performance if $k < 6$ and degrades performance if $k > 6$.

Non-main branches have more challenging merges than main branches do. In order to reflect expected real-world performance, evaluations should include both types of merges.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10.
https://doi.org/10.1145/3691620.3695075

A diff algorithm's **readability** does not materially affect its success for merging. Handling whitespace is important, however: about 1/8 of conflicts are due to whitespace alone (fig. 7).

Current merge tools work conflict-by-conflict (except when calling out to a refactoring-detection tool). Accounting for the **context** of a conflict (e.g., other code in the same file) leads to better merges.

A merge algorithm that merely augments Git's handling of **import statements** usually outperforms Spork, IntelliMerge, and other tools. This suggests that finding simple solutions to common problems is a more effective approach to building a merge tool than complex algorithms that handle relatively uncommon cases. The research community should reward the former as well as the latter.

2 MERGE ALGORITHMS

2.1 Terminology

The input to a VCS merge is a pair of commits or branch heads, called the *parent commits*. The VCS stores the result in a merge commit, which is a commit with two parents.

A commit represents a single state of the file system that the VCS manages. To perform a merge, a VCS calls out to a three-way merge tool [36], passing the parent commits and their base commit. The base commit is the nearest common ancestor in the VCS. Use of a three-way merge tool simplifies the task of merging two file system states to the task of integrating two sets of changes: the changes from the merge base to parent 1, and the changes from the merge base to parent 2.

To the best of our knowledge, every three-way merge algorithm has two phases: alignment and resolution. The biggest differences are the program representation and the change representation.

The *alignment* or matching phase, identifies the unchanged sections in all three versions, thus determining the relative position of changes. The alignment phase is performed by a tool such as diff. A line-based diff consists of alternating common (unchanged) code sequences and *hunks*. A hunk is a set of contiguous added, removed, and/or changed lines between versions of a file. For generality to non-line-based tools, this paper uses the term "change" rather than "hunk". The common code sequences are typically left implicit in the diff representation.

The *resolution* phase of three-way merging uses the following algorithm. For each change C in a 3-way diff, let C1 be the difference between the base and parent 1 and let C2 be the difference between the base and parent 2. C1 and C2 are at the same location in the source code.

- If C1 is the same as C2, use it; equivalently, if parent 1 is the same as parent 2, use it.
- If C1 is empty, use C2; equivalently, if the base is the same as parent 1, use parent 2.
- If C2 is empty, use C1; equivalently, if the base is the same as parent 2, use parent 1.
- If C1 differs from C2, report a conflict; equivalently, if the base, parent 1, and parent 2 all differ, report a conflict.

Alternative merging schemes have been proposed that utilize a different representation of a program than its lines. For example, the Abstract Syntax Tree (AST) is a parsed representation of a program that represents program constructs with parent–child relationships. The line-based representation of two changes might be a conflict,

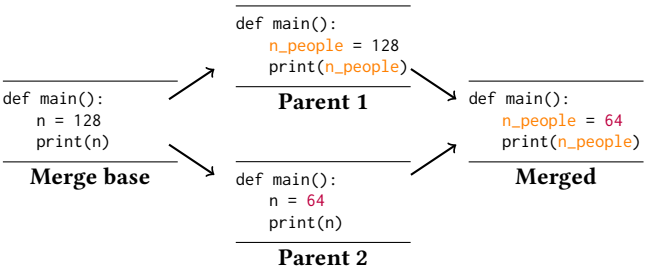

Figure 1: Mergeable changes that line-based merge reports as a conflict.

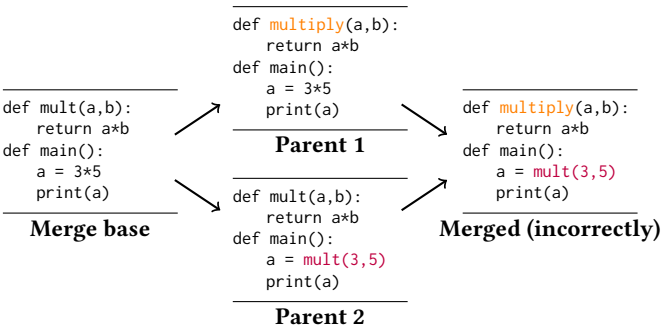

Figure 2: Conflicting changes that line-based merge cleanly, but incorrectly, merges. Most previous evaluations count this as a successful merge.

but the tree-based representation might not be a conflict because the two changes occur in different places in the tree even though they appear on the same line in the source code. Tree-based [3, 5–7, 12, 25, 26, 30, 32, 32, 33, 47, 51, 52, 54] and graph-based [9, 24, 27] merge algorithms can correctly merge edits that line-based tools consider a conflict.

2.2 Weaknesses of Merge Algorithms

Every merge algorithm suffers from two complementary problems. (1) It might fail to merge semantically independent changes to the same code construct (that is, it might leave a conflict for the user to resolve), (2) It might incorrectly merge changes in different constructs that are semantically related. Figures 1 and 2 give examples.

Figure 1 shows changes that line-based Git Merge reports as a conflict, but a more sophisticated algorithm could resolve. Parent 1 renames variable `n` to `n_people`. Parent 2 changes the value of `n`. Since these changes occur on the same line, Git Merge reports a merge conflict. In fact, the two changes (rename a variable and change a value) are semantically independent and can be performed independently, in either order. Another example of an undesirable conflict is when Parent 1 changes the indentation of a line, while Parent 2 makes a code change to that line.

Figure 2 shows an example of an incorrect merge that Git Merge would perform. Parent 1 renames the function `mult` to `multiply`, while Parent 2 adds an invocation of `mult`. These changes are on different lines, and so Git Merge integrates them cleanly. In the

Name	Arguments
ort	(default)
ort-ignorespace	--ignore-space-change
recursive-histogram	-s recursive -X diff-algorithm=histogram
recursive-minimal	-s recursive -X diff-algorithm=minimal
recursive-myers	-s recursive
recursive-patience	-s recursive -X diff-algorithm=patience
resolve	-s resolve

Figure 3: Command-line arguments to git merge**.**

merged version, the invocation to mult on line 4 uses the old name and fails because the program no longer contains a mult routine.

2.3 Evaluated Tools

2.3.1 Previous tools. **Git Merge** is *line-based*: it treats a software artifact as a sequence of lines and resolves merge conflicts based on textual equivalence of lines. Line-based merging is the default in most version control systems, including Git, the most widely used [48].

Git Merge permits customization of both the alignment and resolution stages. Git Merge uses the term "strategy" (the -s command-line argument) for the resolution algorithm, and it uses command-line arguments, which vary by strategy, for the alignment algorithm. We ran Git Merge in the configurations shown in fig. 3.

Git's resolution algorithms (in Git parlance, the strategies) handle criss-cross merges differently. A criss-cross merge is one in which there is no unique base commit — that is, the parents have no unique nearest common ancestor. Without loss of generality, assume there are two nearest common ancestors, B1 and B2.

- resolve: arbitrarily chooses B1 or B2 as the base. It produces a confusing conflict when one parent renames a file and the other changes the file.
- recursive: recursively merges B1 and B2 to create a new commit B. B is not added to the repository, but it is used as the base for the merge of parent 1 and parent 2. In addition, the recursive strategy handles file renaming.
- ort: a reimplementation of the recursive strategy, with the same concepts and high-level algorithm. Ort improves correctness, performance, and code structure, compared to the recursive strategy. Ort was added to Git in 2021 and became Git's default merge strategy in 2023.

An alignment algorithm (in Git parlance, a diff algorithm) tries to find longest common subsequences between parent 1 and parent 2; the *changes* are what appear between those common subsequences. Git's recursive resolution algorithm permits customizing the diff algorithm. The options are:

- myers: a basic greedy algorithm. It is the default.
- minimal: a slower algorithm that tries to produce diffs that are as small as possible.
- patience: an algorithm designed to improve readability and avoid spurious matches. It focuses on low-frequency high-content lines, and looks for longest matches that include them. Then, the patience diff is recursively invoked on the unmatched text that is before and after the match.
- histogram: similar to patience, but constructs a histogram of element occurrences to use with a heuristic when unique common elements cannot be used to match up a sequence.

The ort strategy always uses the histogram diff algorithm.

Hires merge [1], for "high-resolution" merge, ignores whitespace and automatically resolves non-overlapping changes at the *character* level, rather than by lines as Git Merge does. It calls git merge-file, and it does more work only if that leads to a merge conflict. The implementation puts every character on a separate line, merges them using standard tools, and then removes the extra newline characters. (This implementation is reminiscent of a technique for version control of trees by translating to a syntax that line-based version control systems can merge [4].) If this merge strategy fails, a conflict is reported.

IntelliMerge [46] is *graph-based* and exclusively targets Java files. It is a refactoring-aware merge algorithm. Refactorings, such as the rename operations in figs. 1 and 2, can often lead to conflicts or erroneous merges in other merge algorithms. IntelliMerge converts the base and parent versions of a source file into program element graphs (PEGs). A PEG's nodes are classes, methods, fields, etc. The edges define relationships, such as a class implements an interface or a class defines a class member. IntelliMerge performs a matching or alignment step to detect refactorings and code edits, merges the PEGs, and then serializes the merged PEGs back into text files.

Spork [30] is *tree-based* and exclusively targets Java files. It uses a tree-based merge algorithm called 3DM-Merge that operates on ASTs (abstract syntax trees). It uses GumTree [19] for its alignment phase. It uses domain knowledge to resolve specific conflicts during the merge process, such as the fact that import statements and method declarations can be ordered arbitrarily. It is the most starred AST-based merge tool on GitHub.

2.3.2 Previously-proposed algorithms. **Adjacent** is our implementation of a proposal by Nguyen et al. [39]. It resolves non-overlapping changes on adjacent lines — for example, when Parent 1 modifies line n but not $n + 1$ and Parent 2 modifies line $n + 1$ but not n. In 51,007 conflicting merges, programmers resolved 24–85% of adjacent-line conflicts by applying both changes, as opposed to taking just one of the changes, neither of the changes, or making a different change [39]. They conclude "Git should merge two adjacent-line [sic] instead of considering them as a conflict." We implemented this recommendation in order to evaluate it.

By contrast to Adjacent, Git Merge reports a conflict if adjacent lines are changed. This strategy conservatively avoids some incorrect merges, at the cost of creating unhandled merges for the programmer to address. We reviewed the literature that discusses the fact that Git Merge gives a conflict for adjacent-line edits. Every discussion we found treats this as a design decision. However, this behavior falls out of Git Merge's implementation of alignment, which requires a line to be in all three versions (base, parent1, and parent2) — no alignment occurs in Git when adjacent lines are edited.

2.3.3 New tools. **Imports** resolves conflicts among Java import statements. Both IntelliMerge and Spork implement special-case handling of import statements. Neither paper [30, 46] mentions this fact nor evaluates how important it is to their results. We implemented the Imports merge tool in order to answer this important

question. In other words, the Imports tool enables an *ablation study* that compares the importance of resolving import statements to the importance of the non-import-statement parts of a merge tool.

To our surprise, Imports performed *better* than IntelliMerge and Spork. This suggests that — if their import handling is implemented well — the other parts of those tools are a net negative. This also suggests that tool builders and researchers should first focus on doing simple things well before adding complexity that may be unneeded or undesirable. Unfortunately, simple but effective approaches are often rejected by the academic community.

Our Imports tool first runs Git Merge. Then, it re-merges any changes that involve only import statements. The Imports tool may re-introduce an import statement that was removed by either parent, if the imported symbol is used in the merged code. We found that simply unioning the import lines in each git conflict performed much worse. This illustrates that a merge tool should consider the context (such as the rest of the file), not just the text of the conflict itself.

Version-Numbers first runs Git Merge, then resolves remaining conflicts among version numbers. When version numbers differ in all three versions of a program, and those in the two edited versions are both larger than that in the base version, it chooses the largest one. It requires version numbers to contain at least one period (“.”). We implemented this tool based on our observations (section 7) of common merge tool failures. It is useful not just in Java files, but also in buildfiles such as Maven pom.xml files.

IVn combines the Imports and Version Numbers functionality. It shows what is achievable without complex algorithms and implementations. Some authors of this paper use IVn for our daily work; we find that it saves us work.

The new tools are available at https://github.com/plume-lib/merging, which gives instructions for using tham as git merge drivers, git mergetools, or to clean up conflicts at an arbitrary time..

3 RESEARCH QUESTIONS

RQ1 Which Git Merge configurations save the most developer time? Are Git's defaults the best for merging?

RQ2 Which merge tools save the most developer time? Do these conclusions differ from those claimed in previous work?

RQ3 Are there differences in merge tool performance on merges from different sources (main branch, other branches)? Is performance on the main branch indicative of overall performance?

RQ4 What are the most common causes of merge tool failures (conflicts and incorrect merges)?

4 MEASUREMENTS

4.1 Correctness of Merge Resolutions

A correct merge resolution integrates both parent changes without introducing any defects that violate the program's specification. If a merge tool reports any conflicts, we label the (entire) merge resolution as ***unhandled***. Otherwise, our methodology uses the project's test suite as a proxy for correctness [10, 45]. Our experimental infrastructure uses the tests in the repository, including any files that have just been merged. If the merge is clean and the test suite passes, the merge resolution is ***correct***. If the merge is clean and compilation or testing fails, the merge resolution is ***incorrect***. We also use the ***incorrect*** label for uncaught exceptions and timeouts.

We reclassified some merges from “unhandled” to “incorrect”, when the output both a conflict *and* an incorrectly-merged hunk. For every “unhandled” merge, we ran a fixup tool that never makes a mistake in merging. If that tool merges all the remaining hunks, but the resulting code fails its tests, then the original merge must have contained at least one incorrectly-merged hunk, and our experimental infrastructure reclassifies it as “incorrect”. The fixup tool is IVn. We manually inspected hundreds of IVn executions. In every case, the preceding merge tool was responsible for the test failure and IVn was not.

4.2 Merge Tool Performance: Developer Time

The best merge tool is the one that saves developers the most time. Let the average cost to a developer of manually merging a conflict be *UnhandledCost*, and likewise for *CorrectCost* and *IncorrectCost*.

Suppose that a developer encounters 100 merge scenarios, and merge tool T yields 85 correct merges, 10 conflicts, and 5 incorrect merges. Then the total cost to the developer is $85 \times CorrectCost + 10 \times UnhandledCost + 5 \times IncorrectCost$.

We expect that $IncorrectCost \gg UnhandledCost \gg CorrectCost \approx 0$. Incorrect merges may require a developer to debug a failure, in order to locate a defect caused by the incorrect merge, when the tests are next run. Unhandled merges are easier because they come with location and diff information. The best merge tool will prioritize reducing incorrect merges, while also minimizing the number of unhandled merges as a second priority. Like us, [12] speculates that *IncorrectCost* (which they call “false negatives”) is high. Future work should measure these costs.

Setting *CorrectCost* to zero simplifies the equation to $Cost(T) = 10 \times UnhandledCost + 5 \times IncorrectCost$. Let k, the *cost factor* for incorrect merges, be $IncorrectCost/UnhandledCost(T)$. This further simplifies the example to $Cost(T) = 10 \times UnhandledCost + 5 \times k \times UnhandledCost = (10 + 5 \times k) \times UnhandledCost$.

Generalizing beyond the 100 commits, the developer cost when using tool T is $Cost(T) = (numUnhandled(T) + numIncorrect(T) \times k) \times UnhandledCost$. If the developer used no merge tool, the developer cost would be $ManualCost = numMerges \times UnhandledCost$. A merge tool T is valuable iff it reduces the developer's effort: $Cost(T) < ManualCost$. The benefit of using tool T is

$$\begin{aligned} EffortReduction(T) &= \frac{ManualCost - Cost(T)}{Manual\_Cost} \\ &= 1 - \frac{Cost(T)}{Manual\_Cost} \\ &= 1 - \frac{numUnhandled(T) + numIncorrect(T) \times k}{NumMerges} \end{aligned}$$

Larger values of *EffortReduction* are better. A tool that produces only correct merges gets a score of 1, while a tool that only produces unhandled merges (equivalent to a purely manual resolution strategy) gets a score of 0. A tool that costs a developer more than manual resolution gets a negative score. The score depends on the value of k, which future work should experimentally measure.

			Merge size				
			∩	total			
Phase	Repos	Merges	#f	#f	hunks	lines	imp
Java repos	42092	-	-	-	-	-	-
Head passes	4072	294714	1	11	28	299	73%
Java diff	1653	21860	3	32	96	1190	94%
Parents pass	1120	6045	3	24	73	823	93%

Figure 4: Repositories and merges at each phase of collection for our data sets. "∩ #f" gives the median number of files changed by both parent 1 *and* parent 2. The "total" lines are medians for the changes in parent 1 *or* parent 2. "Imp" is the percentage of merges that involve Java import statements.

5 METHODOLOGY

5.1 Data Set

We used GitHub repositories from two datasets of high-quality code: GitHub's Greatest Hits [21] and Reaper [38]. GitHub's Greatest Hits contains GitHub's 17,000 most popular and depended-upon repositories, as ranked by a combination of popularity (star count) and dependency extent. Reaper is a subset of the GHTorrent [22] dataset. Reaper consists of only "engineered software projects … that leverage sound software engineering practices", as determined by community, continuous integration, documentation, history, issues, license, and unit testing.

We selected all Java projects from GitHub's Greatest Hits. We selected all Java projects from Reaper that have a *unit_test* score (the ratio of test lines of code to source lines of code) of at least 0.25 and have at least 10 stars. There are 42092 unique Java repositories.

We retained repositories that utilize the Maven or Gradle build automation tools and whose head (latest) commit on the main branch passes its tests (under JDK 8, 11, or 17) within 30 minutes.[1] This yielded a total of 4072 projects containing at least one merge. We collected 294714 merges from these projects.

We retained only merges that contain a diff within a .java file, and both of whose parents pass tests within a timeout of 30 minutes. This left us with 6045 merges.

Requiring both parents to pass their tests makes it highly likely that a test failure in the merged result can be attributed to the merge tool. This was always the case, in our manual examination of hundreds of merges.

Our methodology depends on running test suites. If some programs have poor test coverage, then the fact that all tests passed might just mean that the tests didn't cover the code that was merged. To assess this threat to validity, we used JaCoCo [16] to compute the coverage of 100 randomly-chosen projects. The average code coverage was 53% and the 25/50/75 percentile values were 28/54/77%.

Figure 4 shows how the filtering affected the set of merges. Merge size in the "parents pass" row is smaller than in the "Java diff" row. However, merges where the parents pass tests are still larger than merges overall (the "head passes" row). The most important metric is the number of files that were edited by both parents, "∩ #f". Only these files can be involved in a conflict. This is unaffected by requiring that the parents pass tests.

[1]We used an Intel i9-13900KF @ 5.8GHz.

5.2 Flaky tests

Flaky tests — those that sometimes pass and sometimes fail — are prevalent in software projects [29, 31, 34, 37, 43], so an experiment that runs tests must account for them. We wished to avoid nondeterministic, unrepeatable results. However, it is essential to utilize the entire test suite, all of which provides information to developers. Therefore, throughout our experiments we ran every test 5 times, counting the test as a success if any run succeeded.

There were 44985 merges for which tests passed. On average, each of these merges had to be tested 1.01 times before the tests passed. 256 of the merges first passed on the 2nd test run, and 35 of the merges first passed on the 5th test run.

5.3 Merge Inputs

We collected merge commits from two sources: the main branch and other branches.

The main, or mainline, branch is often named `main`, `master`, or `trunk`. Other branches include feature branches, release branches, etc. Some other-branch merges are also accessible from the main branch, so we removed any duplicates and only labeled a merge as an other-branch merge if it does not appear as a main branch merge.

Main branches are often meant to represent clean snapshots of a project. They might encounter different activity and therefore different merges than other branches, which could contain sloppier or finer-grained changes that nonetheless need to be merged. In 2017, among the 19 most popular Java projects on GitHub, only 13 had any merges in Java files on the main branch [4].

Feature branches are deleted after being merged, so they are not available from the git repository. We obtained the information from GitHub, which retains information about deleted branches.

5.4 Evaluation of Merge Correctness

If a tool produces a clean merge, our experimental infrastructure checks whether the merge passes its test suite (section 4.1), using a timeout of 45 minutes per run. This timeout is larger than the 30 minutes of section 5.1 in case both Parent 1 and Parent 2 added tests to the base version, in which case the merged tests may take longer to run than the tests on either Parent 1 or Parent 2.

5.5 Tool Implementations

We wrote wrapper scripts that invoke each merge tool on a set of merge inputs.

5.5.1 Git Merge Wrapper. For Git Merge, the wrapper checks out the first parent branch, then invokes `git merge` with the second parent branch as an input. Extra inputs are used for specifying specific strategies.

5.5.2 Hires-Merge Wrapper. The wrapper configures Hires-Merge as a Git merge driver, by writing to the `.gitattributes` file after cloning a repository. Then, `git merge` can be invoked as usual.

5.5.3 Spork Wrapper. The wrapper configures Spork as a Git merge driver for Java files, by writing to the `.gitattributes` file after cloning a repository. This makes Git use Spork to resolve conflicts

Tool	Merges					
	Correct		Unhandled		Incorrect	
	#	%	#	%	#	%
Gitmerge-ort	2748	46%	3078	51%	157	3%
Gitmerge-ort-ignorespace	2889	48%	2905	49%	189	3%
Gitmerge-recursive-histogram	2748	46%	3078	51%	157	3%
Gitmerge-recursive-minimal	2748	46%	3078	51%	157	3%
Gitmerge-recursive-myers	2748	46%	3078	51%	157	3%
Gitmerge-recursive-patience	2751	46%	3074	51%	158	3%
Gitmerge-resolve	2703	45%	3124	52%	156	3%

Figure 5: Performance of different Git Merge configurations. Figure 6 visualizes this data.

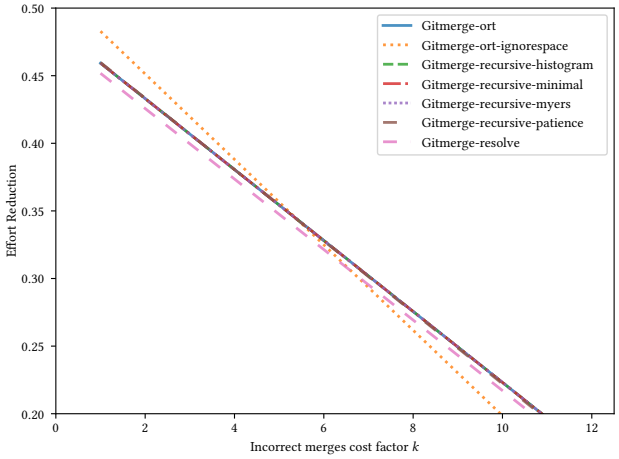

Figure 6: Effort reduction as a function of k, the relative cost of incorrect merges. This graph visualizes the data of fig. 8. The best merge tool is Gitmerge-ort or Gitmerge-ort-ignorespace, depending on k.

between Java files, and use its default merge strategy (ort) for all other files. Then, git merge can be invoked as usual.

5.5.4 IntelliMerge Wrapper. IntelliMerge only outputs merge results for Java files. Our wrapper first runs IntelliMerge to merge Java files, and stores them in a temporary location. It then runs Git Merge to generate merge results for all files in-place. Finally, it overwrites all the Java files with the IntelliMerge versions. File copying has a negligible effect on run time.

One complication is conflict detection — IntelliMerge's exit code only indicates whether the tool completed without exception, rather than whether the merge was clean as merge tools are expected to do. Therefore, our wrapper ignores exit codes; it determines whether a merge is successful by searching for conflict markers (e.g., "<<<<<<<") that appear in conflicted files.

6 RESULTS

6.1 Git Merge Configurations (RQ1)

Figures 5 and 6 show the performance of Git Merge configurations. The differences are relatively small, but these small differences are important. For example, section 7.2.2 gives an example of an incorrect merge that compiles and passes some tests, but it contains a race condition that was not in either parent. A merge may create a defect that is not detected by the test suite. The resulting bug might be detected long afterward (making it more expensive to fix) or might be deployed to production (which is even more expensive). Thus, bad merge resolutions can have very significant implications. [41] statistically justifies special attention to merge failures.

6.1.1 Resolution Phase. Git Merge supports three resolution strategies. Ort, the newest, is tied for the best, with now fewer correct merges and no more incorrect merges than any other strategy. The ort and recursive strategies, behave the same on this dataset, when both use the Myers alignment algorithm. This surprised us, given the hate that the recursive strategy received on forums.

6.1.2 Alignment Phase. The recursive strategy permits selecting a diff algorithm to use for the alignment phase. Patience is a popular diff algorithm because it is considered to create the most human-readable diffs [13]. Although it has the most correct *and* incorrect merges, it is nearly indistinguishable from the other alignment algorithms. The "minimal" diff algorithm creates the smallest possible diffs, at the cost of run time, but it performs identically (on our dataset) to myers, a simple greedy algorithm. Differences that are important to software developers when viewing diffs do not seem to matter much to line-based merge tools. However, we found that setting git to use the zdiff3 conflict style (rather than diff3) hindered downstream tools. zdiff3 moves lines common to both parents out of a hunk and into the surrounding common text. This is considered better for human inspection because the hunk is shorter, but it gives an incorrect view of the base text.

6.1.3 Ignoring Space Changes. In our dataset, ignoring whitespace decreases unhandled merges by 5%, but increases incorrect merges by 10% (section 6.1.3). Figures 6 and 9 shows that Git Merge Ignorespace outperforms Git Merge if $k < 5$; Git Merge is better if $k > 5$.

This supports the folk wisdom that spacing conflicts can cause issues for merge tools, and it supports the inclusion of the --ignore-space-change argument to Git.

The absolute difference between Git Merge and Git Merge Ignorespace is small. Ignoring whitespace changes could be catastrophic in other languages, such as Python and YAML, where the amount of white space (indentation) is semantically significant (see example in section 7.3.1).

6.1.4 Differences in Merge Output. Figure 7 shows, for each pair of configurations, the number of times they produced clean merges with *distinct* contents. Apart from Gitmerge-ort-ignorespace, all the configurations produce the same output in most situations. With regard to syntax (fig. 7), Gitmerge-ort-ignorespace is quite different from other git configurations. However, with regard to semantics (fig. 5), Gitmerge-ort-ignorespace is relatively similar to other git configurations. Perhaps most of the textual differences were formatting differences rather than semantic changes.

We selected Git Merge and Git Merge Ignorespace to compare with other merge tools (section 6.2). Henceforth, just "Git Merge" means ort, which is Git's default configuration.

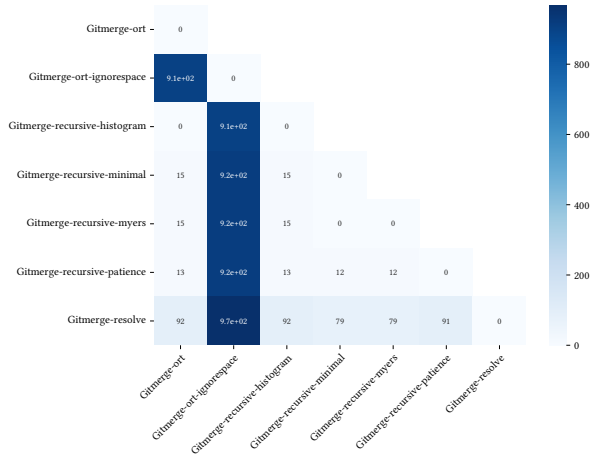

Figure 7: Number of clean merges that are different between each pair of Git Merge configurations. Each Git Merge configuration produces around 7000 clean merges.

Tool	Merges					
	Correct		Unhandled		Incorrect	
	#	%	#	%	#	%
Gitmerge-ort	2748	46%	3078	51%	157	3%
Gitmerge-ort-ignorespace	2889	48%	2905	49%	189	3%
Hires-Merge	3040	51%	2721	45%	222	4%
Spork	3260	54%	2080	35%	643	11%
IntelliMerge	1434	24%	1582	26%	2967	50%
Adjacent	3073	51%	2692	45%	218	4%
Imports	2904	49%	2910	49%	169	3%
Version Numbers	2782	46%	3044	51%	157	3%
IVn	3011	50%	2803	47%	169	3%
IVn-ignorespace	3116	52%	2665	45%	202	3%

Figure 8: Performance of merge tools (visualized in fig. 9).

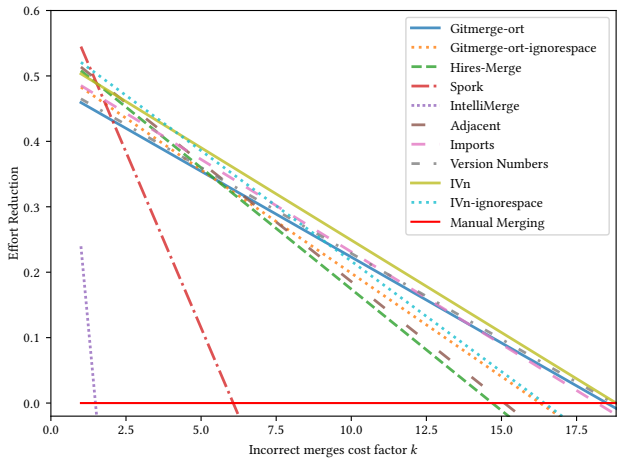

Figure 9: Effort reduction as a function of k, the relative cost of incorrect merges. This graph visualizes the data of fig. 8.

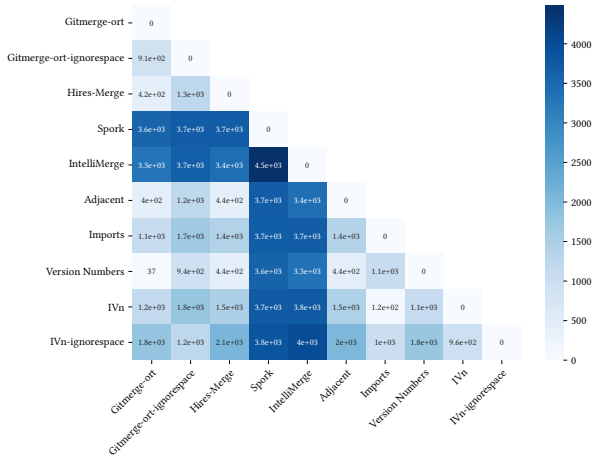

Figure 10: Number of clean merges that are different (i.e., not syntactically identical) between each pair of tools. Each merge tool produces around 7000 clean merges.

6.2 Comparison Among All Merge Tools (RQ2)

Figures 8 and 9 show the experimental results among the merge tools. Figure 10 shows the number of times each pair of tools produced a clean merge that differed. The tree- ("structured") and graph-based merge tools Spork and IntelliMerge are the most different from all the other merge tools — but the single greatest difference was between Spork and IntelliMerge.

IVn or IVn-ignorespace is the best tool except when $k \sim 1$, in which case Spork is best.

"Structured" merge tools (such as Spork and IntelliMerge) produce both more correct merges and more incorrect merges [45], leading to a relatively steep slope in fig. 9. Adjacent and Hires Merge have the next lowest number of unhandled merges.

Whether Spork is better than Git Merge (and the other tools) depends on the relative cost of incorrect merges (see figs. 6 and 9). If an incorrect merge is no worse than an unhandled merge, then Spork is the best tool. If an incorrect merge is at least ~ 2 times as bad as an unhandled merge, then Spork is the *worst* tool other than IntelliMerge. If an incorrect merge is 6 times as bad as an unhandled merge, then using Spork is worse than using no merge tool at all (that is, manually resolving every merge).

6.2.1 Comparison to previous results. The IntelliMerge [46] paper found a significant difference between itself and Git Merge when measuring hunks and lines: "Comparing with GitMerge, IntelliMerge reduces the number of conflict blocks by 58.90% and the lines of conflicting code by 90.98%" [46]. In contrast, our evaluation found IntelliMerge significantly worse. Here are two possible reasons for this discrepancy.

First, IntelliMerge's data set only includes merges that Git Merge was unable to merge cleanly *and* where a refactoring change was part of the conflict. In other words, it was only applied to exactly the merges it is designed to resolve *and* where its competition failed. We applied IntelliMerge to a more representative set of merges.

Tool	Run time (seconds)		
	Mean	Median	Max
Gitmerge-ort	0.04	0.04	1.19
Gitmerge-ort-ignorespace	0.04	0.04	1.22
Hires-Merge	0.23	0.13	17.3
Spork	2.55	1.06	653
IntelliMerge	1.04	0.50	89.3
Adjacent	0.17	0.05	76.9
Imports	0.11	0.07	7.86
Version Numbers	0.07	0.05	2.75
IVn	0.12	0.08	7.97
IVn-ignorespace	0.12	0.08	8.08

Figure 11: Merge tool run time.

Tool	Merges					
	Correct		Unhandled		Incorrect	
	Main	Other	Main	Other	Main	Other
Gitmerge-ort	53%	35%	44%	62%	3%	2%
Gitmerge-ort-ignorespace	56%	38%	41%	59%	3%	3%
Hires-Merge	57%	41%	39%	55%	4%	4%
Spork	62%	44%	28%	45%	11%	11%
IntelliMerge	27%	19%	21%	34%	52%	47%
Adjacent	58%	42%	39%	54%	4%	4%
Imports	56%	38%	41%	59%	3%	3%
Version Numbers	54%	36%	44%	61%	3%	2%
IVn	57%	41%	40%	57%	3%	3%
IVn-ignorespace	59%	42%	38%	54%	3%	4%

Figure 12: Results broken down by merge source: the main branch or other branches. Each percentage indicates the fraction of merges from that source that yielded that outcome.

Second, the success metric of the IntelliMerge paper is reduction in the number of conflict lines and hunks. Perhaps that metric does not correlate with successful merges that lead to correct behavior.

The IntelliMerge paper acknowledges, as a limitation, that it doesn't evaluate the tool's impact on Incorrect Merges, which it refers to as *False Negative Conflicts*.

6.2.2 Run Time. Figure 11 shows the run times of each tool. Each number is the median of 3 runs. Spork and IntelliMerge most often cause noticeable pauses. The IntelliMerge [46] paper reports a median run time of 0.54 seconds, which is very close to our measurement of 0.50 seconds. The IntelliMerge paper did not report the mean, but our data show that it is twice as high. Adjacent's maximum run time is high because of its use of a 3-way dynamic programming algorithm. Perhaps that algorithm could be cut off earlier without materially affecting the output.

6.3 Differences Between Merge Sources (RQ3)

Our dataset contains 3524 (59%) main branch merges and 2459 (41%) other branch merges.

Figure 12 shows merge results by merge source (main branch merges vs. other branch merges). Previous evaluations only use main branch merges.

Merge tools perform better on main branch merges than on other branch merges. An evaluation on only main branch merges is misleading with respect to absolute performance. In real-world usage, merge tools will perform worse than in previous studies.

However, the *relative* performance of tools is similar between main and other branches. The differences do cause different rankings depending on k, but primarily for tools whose Effort Reduction values were already similar. An evaluation on only main branch merges is therefore an acceptable way to measure *relative* performance, with a few caveats.

7 QUALITATIVE ASSESSMENT (RQ4)

We manually examined hundreds of merges in which two tools produced different results.

For each tool X, we created two pools of merges: a pool where X failed and all others succeeded, and a pool where X succeeded and all others failed. We randomly chose merges from each pool, so we saw examples of each tool doing well and doing poorly.

For each selected merge, we compared the base, left, and right versions, the programmer merge, and the results of merge tools. We primarily used the diff and diff3 tools for these comparisons. Every evaluation was performed by one author and reviewed by at least two other authors. Disagreements were resolved by discussion.

The appendix [44] shows our analysis of 75 merges. Here we present a subset of them. Each merge in our dataset has an index such as "123-45". We show edits in diff3 format, which gives the left parent, then the base, then the right parent.

7.1 Hires Merge

7.1.1 Handling Refactorings With Multiple Inline Changes (3183-11). Hires Merge works character-wise. This strategy deals with refactorings quite effectively.

```
<<<<<<< LEFT
HashSet<Range> ranges = new HashSet<>();
||||||| BASE
HashSet<Range> ranges = new HashSet<Range>();
=======
Set<Range> ranges = new HashSet<Range>();
>>>>>>> RIGHT
```

Git Merge gets stuck because the left and right edited the same line. Hires Merge comes up with a correct merge:

```
Set<Range> ranges = new HashSet<>();
```

7.1.2 Hires Merge Incorrectly Identifying Version Numbers (25267-730). Merging character-by-character loses context. In this merge:

```
<<<<<<< LEFT
<version>23.7.0</version>
||||||| BASE
<version>23.6.0</version>
=======
<version>23.6.1</version>
>>>>>>> RIGHT
```

Hires Merge invented a nonexistent version number:

```
<version>23.7.1</version>
```

7.2 Adjacent

7.2.1 Refactoring on Adjacent Lines (1215-3280). Adjacent successfully merged scenarios involving refactoring, particularly when variables were independent.

```
<<<<<<< LEFT
String comments = SourcesHelper.readerToString(reader);
CompilationUnit cu = new JavaParser().setSource(comments).parse();
||||||| BASE
String comments = SourcesHelper.readerToString(reader);
CompilationUnit cu = new InstanceJavaParser(comments).parse();
=======
String comments = readerToString(reader);
CompilationUnit cu = new InstanceJavaParser(comments).parse();
>>>>>>> RIGHT
```

7.2.2 Adjacent Lines are Interdependent (5184-31). The key weakness in the adjacent strategy is its local view, disregarding context. Consider this merge. The left parent changed the variable being synchronized upon.

```
<<<<<<< LEFT
synchronized (cacheMap) {
    List<DNSEntry> entryList = cacheMap.get(dnsEntry.getKey());
    if (entryList != null) {
        entryList.remove(dnsEntry);
||||||| BASE
List<DNSEntry> entryList = this.get(dnsEntry.getKey());
if (entryList != null) {
    synchronized (entryList) {
        entryList.remove(dnsEntry);
=======
List<DNSEntry> entryList = this.get(dnsEntry.getKey());
if (entryList != null) {
    synchronized (entryList) {
        result = entryList.remove(dnsEntry);
>>>>>>> RIGHT
    }
}
/* Remove from DNS cache when no records remain with this key */
if (result && entryList.isEmpty()) {
    this.remove(dnsEntry.getKey());
```

Adjacent readily merges the code, but without moving the outer if-statement inside the synchronized block, leading to code that compiles but contains a race condition.

```
synchronized (cacheMap) {
    List<DNSEntry> entryList = cacheMap.get(dnsEntry.getKey());
    if (entryList != null) {
        result = entryList.remove(dnsEntry);
    }
}
/* Remove from DNS cache when no records remain with this key */
if (result && entryList.isEmpty()) {
    this.remove(dnsEntry.getKey());
}
```

Git Merge left this conflict unhandled, forcing the programmer to do the merge, which is a better outcome.

7.3 Git Merge Ignorespace

7.3.1 Extra Irrelevant Spaces (2955-73). Git Merge Ignorespace is not confused by inconsequential spaces. It merges code like this:

```
<<<<<<< LEFT
 * </p>
||||||| BASE
 *␣
=======
 *
>>>>>>> RIGHT
```

Git Merge Ignorespace caused havoc when merging YAML files (e.g., 14378-60), where indentation matters and there may be multiple occurrences of a key.

7.4 Spork

As explained in the appendix [44], Spork sometimes produced uncompilable code, made gratuitous formatting changes, or omitted method bodies. Spork's maintainers acknowledged our bug reports but have not fixed them. We spent well over a person-month trying to fix the bugs ourselves, but were not able to address them all. Then, we tried to refactor Spork to eliminate its dependence on Spoon (which the Spork maintainers blamed for some of Spork's bugs), but they were so entangled that we were unable to do so. We speculate that a better implementation of the Spork algorithm could be a very effective merge tool.

7.4.1 Overlapping Unique Additions (35091-165). Spork's strategy of parsing code into an AST tree and matching methods by name was quite successful when different branches added different methods at the same location. Examples like this were the bread and butter of Spork's successes.

7.5 Version Numbers

The Version Numbers tool starts with the output of Git Merge. It never underperformed Git Merge.

7.6 Imports

The Imports tool starts with the output of Git Merge and only fixes merges in import statements. It never introduces mistakes, because it parses the entire file looking for uses of imports. It can correct mistakes by re-introducing import statements that Git Merge removed by a clean but incorrect merge.

8 THREATS TO VALIDITY

Construct validity. Testing is an imperfect proxy for correctness. If tests fail, then most likely the merge is incorrect, but if tests pass, the merge might still be incorrect. For instance, the merge might be wrong in files that are not executed by the test suite. Therefore, our measured number of incorrect merges understates the problem of incorrect merges.

Though testing is an imperfect proxy, we believe it is better than the alternatives. One alternative would be an automated proof that the merged program is equivalent to what is in the version control repository or is equivalent to a merge of the branches [24], though verification is too expensive and unscaleable; furthermore, what is committed to the repository may be wrong, as discussed in section 9.3.1. The testing proxy is also used by the program repair community, who discovered very serious errors in papers that did not test repairs [42]. That community calls a patch that passes a test suite "plausible", and reserves "correct" for one that matches the programmer's intent (which is, in general, unknowable).

The values *UnhandledCost* and *IncorrectCost* are averages. A particular merge tool might produce better- or worse-than-average

conflicts and incorrect merges. It seems likely that all tools perform better on merges that a human would find easy and perform worse on merges that a human would find hard.

Internal validity. The difference in performance between merge tools is small. It is possible that a larger experiment would change the results. Even small differences are important because of the high cost of manually reverse-engineering incorrect merges.

Flaky tests can lead to spurious test failures. We mitigated this by running each test suite multiple times.

Our experimental infrastructure ignores merges in which either parent fails its tests. It is conceivable that merge tools have different success rates on such merges.

External validity. Our experiments are on Java programs, because state-of-the-art merge tools are designed for Java. Merge tools (including state-of-the-practice ones) may behave differently in other contexts, such as for Python and YAML where whitespace is semantically significant.

Our experiments use 42092 Java projects from two well-known datasets of high-quality repositories. It is possible that other repositories might have different characteristics.

Our paper presents a metric for determining quality of merge tools. The metric is for use by researchers and merge tool authors, not by end users of merge tools.

Our ranking of tools depends on k, the *average* relative cost of incorrect merges. Our work makes no prediction about any specific merge. The average for k does not capture differences in developer time within these groups. For example, larger diffs may be harder for both tools and developers to manage, though small code changes do not get merged faster than large ones [28]. It is possible that merge tools disproportionately succeed on "easier" merges; nonetheless, automating these cases does reduce developer effort. It is possible that a tool ranked higher by our metrics correctly handles more merges, but simpler ones, than a tool ranked lower by our metrics. Future work should develop heuristics to reward tools for producing unhandled merges that are easy to manually resolve, or producing incorrect merges that are easy to debug.

9 RELATED WORK

9.1 Other merge tools

This section discusses tools that we wished to include in our study, but were unable to.

JDime [32] first runs a faster, less capable algorithm (line-based merging). Only when that algorithm fails does it use a slower, more capable algorithm (tree-based merging). The paper calls this staged approach "auto-tuning". JDime is unsuitable for practical use because it discards comments (as do some other tools [4]), discards file headers (as do some other tools [53]), and arbitrarily reorders methods and fields (as do some other tools [24, 30, 32, 50]), and runs very slowly. Performing a merge with JDime takes "about 15 min" [2], which the paper claims "can be safely neglected." A previous evaluation [30] found that Spork outperforms JDime. We fixed some bugs in JDime, and the JDime authors helpfully addressed a dozen bugs that we reported, but remaining unfixed bugs prevented us from using it. In addition, JDime does not handle the full syntax of Java 8 (released in 2014).

The Spork paper [30] did evaluate against JDime. Maybe JDime's poor performance in those experiments is due its bugs. Or maybe Spork's experiments (containing only 1740 merges) exercised less of JDime. For example, Spork's experiments might have focused on programs written in Java 8 that do not use all of Java 8's functionality. By contrast, our experiments used Java code as recent as Java 17, which was released in 2021.

AutoMerge [53, 54] (sometimes incorrectly called "AutoMerge-PTM" [30]) represents the set of all possible merges via version space algebra. It ranks all the merge possibilities; the developer must choose among them. The AutoMerge tool is not publicly available.

DeepMerge [15] and MergeBERT [49] are neural (that is, deep learning) approaches to merging. Neither tool is publicly available.

Many merge tools exist that depend on GUI interaction, rendering them unsuitable for our experiments, which are fully automated with no human interaction. We experimented with these tools and found that they give the same results as Git Merge; we speculate that they use it internally. In other words, their main differentiation from competitors is their GUI, not their merge algorithm. RefMerge [17] uses a different merge algorithm (that of MolhadoRef [14]), but it is implemented as an IntelliJ plugin, so we could not include it in our experiments. FSTMerge [3] depends on the visual kdiff3 tool.

9.2 Comparisons of merge tools

Nugroho et al. [40] compared the alignment algorithms that are built into Git. (Git calls them "diff algorithms.") Their focus is on use in academic research. In 52 papers, all used the default diff algorithm (at the time, Myers). They investigated whether use of a different alignment algorithm might have changed the experimental results. The Myers and Histogram diff output was identical for over 95% of commits. Our experiments show that the 5% of differences do not affect merging.

Ellis et al. [17] compared two operation-based refactoring-aware merge tools (RefMerge and IntelliMerge) against Git Merge. They considered only merge scenarios that contain refactoring-related conflicts. For 2001 such merge scenarios from 20 open-source Java projects on GitHub, IntelliMerge produced a clean merge 3% of the time, and RefMerge produced a clean merge 6% of the time. They did not run tests to verify the merges.

Seibt et al. [45] compared three merge strategies: unstructured, structured, and semi-structured. They used one merge tool: the JDime structured merge tool [32], extended to perform unstructured and semi-structured merge. They evaluated all 7 ($= 2^3 - 1$) combinations of unstructured, semi-structured, and structured merging. They considered 10 repositories compared to our 1120. Notably, they used testing as a proxy just as we did. They reported merge failures and test failures; we additionally offer quantitative guidance regarding which tool programmers should use. Our work complements theirs.

9.3 Weaknesses of previous comparisons

All previous evaluations of merge tools suffer from at least one of these three problems: they do not evaluate *merge correctness*, they are not evaluated on *representative merges*, or they do not compare with *state-of-the-art tools*.

9.3.1 Merge correctness. Most evaluations of merge tools assume that every *clean* merge is a *correct* merge. They falsely assume that no merge tool ever makes a mistake. Thus, they ignore the cost of incorrect merges [12]. In an industrial case study [41], 1% of clean merges were incorrect.

Other evaluations assess correctness by comparing the output of a tool with the resolution in the commit history [15, 46, 49]. This is the wrong metric if a developer uses Git Merge and Git Merge cleanly but incorrectly merges the two branches. Such a situation is not uncommon: in one study [11], over 9% of textually clean merges produced by Git Merge failed to build or failed to pass tests. In other words, a comparison with the VCS history will reward Git Merge while penalizing a correct tool. In one evaluation [46], the authors state that this bias explains why they computed 99.5% precision for Git Merge.

A third approach is manual labeling. For example, [17] evaluated 50 merges where IntelliMerge and a proposed tool produced different resolutions. Especially when performed by the authors of a merge tool, this approach can lead to bias or labeling errors. It is difficult for a maintainer to evaluate correctness, much less an outsider who is unfamiliar with the codebase. A manual approach also scales poorly. A small sample suffers the risk that the results may not generalize to larger samples.

A few previous studies have used test suites as a proxy for merge correctness [10, 45]. However, this approach has not caught on; [49] notes it as a rare and unusual approach. [12] claims to "explore build or test failures", but their experimental methodology does not run tests. The program repair community discovered serious errors in papers that did not test repairs [42].

9.3.2 Representativeness of merges. Some evaluations use synthetic merges [3, 26, 32], which are merges between arbitrary commits in the VCS. Synthetic merges may differ from real-world merges.

Some evaluations [3, 46] consider only merges that are demonstrative of a certain scenario. For example, [46] uses only refactoring-related merge commits when evaluating IntelliMerge, which performs refactoring detection. They did not evaluate how well the tool performs on merges without refactoring, such as whether it mistakenly detects refactorings in them.

Previous evaluations only collect merge commits that lie on a project's main branch. At least, they do not mention considering other branches. [23] explicitly excluded non-main-branch merges from consideration. [50] states that there is no reason to believe that merges not on the main branch differ from those they analyzed. Our section 6.3 shows that they do differ. It is common for projects to maintain a linear VCS history on their main branch, for example by using squash-and-merge for pull requests. The many merges into other branches, such as pulling the main branch into a feature branch, are qualitatively different from the few merges that do appear on the main branch. We also consider merges into long-lived branches, such as release or development branches.

9.3.3 Comparison to state-of-the-art tools. The state-of-the-practice merge tool is Git Merge. The behavior of Git Merge can be customized by command-line arguments, and these arguments affect the quality of its merges. All previous comparisons against Git Merge have only considered its default configuration, without considering these command-line arguments.

To the best of our knowledge, the only publicly-available functional command-line merge tools for Java are Hires-Merge [1], IntelliMerge [46], Spork [30], and our new tools [18]. (Section 9.1 discusses other tools such as JDime, AutoMerge, RefMerge, etc.) Hires-Merge predates IntelliMerge and Spork, but has not yet been experimentally evaluated. The Spork paper adopts IntelliMerge's experimental protocol, but excludes IntelliMerge and Git Merge from its evaluation.[2] Our paper answers the important research question of how these merge tools perform in a head-to-head comparison.

9.4 Other research on merging

[35] found that "attributes such as the number of committers, the number of commits, and the number of changed files seem to have the biggest influence in the occurrence of merge conflicts", in terms of statistical correlation.

[20] found that programmers resolved most hunks by choosing one of the two parent versions.

10 CONCLUSION

Merging allows simultaneous work on multiple tasks during collaborative software development. Multiple merge tools have been proposed with promising evaluations. We showed that these evaluations can be misleading if they fail to include all merges or do not assess merge correctness. To address such shortcomings, we propose a novel evaluation protocol that includes merges from deleted branches, uses a resolution's test suite to identify whether a merge is correct or incorrect, and quantitatively accounts for incorrect merges. We created new merge tools that outperform existing ones. We found that representative merge sources are essential for evaluating the absolute performance of tools, but are less important in comparing them. In several cases, our experimental results answer the questions and update the claims of previous papers, leading to clearer understanding of the strengths and weaknesses of merge tools. We hope our work contributes to a more principled and experimentally driven approach to the development of new merge tools.

Data availability. Our experimental data are publicly available, along with the programs that produced and analyzed them. The code to compute all the results can be found at https://github.com/benedikt-schesch/AST-Merging-Evaluation and the computed data can be found at https://zenodo.org/records/13366866. The new merging tools are available at https://github.com/plume-lib/merging.

REFERENCES

[1] Paul Altin. git-hires-merge. https://github.com/paulaltin/git-hires-merge. Accessed 2023-07-31.

[2] Sven Apel, Olaf Leßenich, and Christian Lengauer. Structured merge with auto-tuning: balancing precision and performance. In *ICSE 2012, Proceedings of the 34th International Conference on Software Engineering*, pages 120–129, Zürich, Switzerland, June 2012.

[3] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: Rethinking merge in revision control systems. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 190–200, Szeged, Hungary, September 2011.

[2]The Spork paper only evaluates "structured merge algorithms" — that is, only tree merge algorithms, not line-based nor graph-based algorithms. By contrast, they classify IntelliMerge as "semi-structured". Inconsistently, their evaluation includes JDime even though JDime contains both an unstructured and a structured pass.

[4] Dimitar Asenov, Balz Guenat, Peter Müller, and Martin Otth. Precise version control of trees with line-based version control systems. In *FASE 2017: Fundamental Approaches to Software Engineering*, pages 152–169, Uppsala, Sweden, April 2017.

[5] Ulf Asklund. Identifying conflicts during structural merge. In *NWPER 1994, Nordic Workshop on Programming Environment Research*, pages 231–242, June 1994.

[6] M. Lamine Ba, Talel Abdessalem, and Pierre Senellart. Uncertain version control in open collaborative editing of tree-structured documents. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng '13, pages 27–36, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Anastasios G. Bakaoukas and Nikolaos G. Bakaoukas. A top-down three-way merge algorithm for HTML/XML documents. In *IntelliSys 2020: Intelligent Systems and Applications: Proceedings of the 2020 Intelligent Systems Conference*, pages 75–96, 2020.

[8] Valdis Berzins. Software merge: Semantics of combining changes to programs. *ACM Transactions on Programming Languages and Systems*, 16(6):1875–1903, 1994. November.

[9] David W. Binkley. Multi-procedure program integration. Technical Report 1038, University of Wisconsin – Madison, August 1991.

[10] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 168–178, Szeged, Hungary, September 2011.

[11] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, 39(10):1358–1375, October 2013.

[12] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. Evaluating and improving semistructured merge. In *OOPSLA 2017, Object-Oriented Programming Systems, Languages, and Applications*, pages 59:1–59:27, Vancouver, BC, Canada, October 2017.

[13] Bram Cohen. Patience diff advantages. https://bramcohen.livejournal.com/73318.html, March 2010.

[14] Danny Dig, Tien N. Nguyen, Kashif Manzoor, and Ralph Johnson. MolhadoRef: A refactoring-aware software configuration management tool. In *OOPSLA Companion: Object-Oriented Programming Systems, Languages, and Applications*, pages 732–733, Portland, OR, USA, October 2006.

[15] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu Lahiri. DeepMerge: Learning to merge programs. *IEEE Transactions on Software Engineering*, 49(4):1599–1614, April 2023.

[16] EclEmma team. JaCoCo Java code coverage library. https://www.eclemma.org/jacoco/, March 2024.

[17] Max Ellis, Sarah Nadi, and Danny Dig. Operation-based refactoring-aware merging: An empirical evaluation. *IEEE Transactions on Software Engineering*, 49(4):2698–2721, April 2023.

[18] Michael D. Ernst. Plume-lib merging: merge drivers and merge tools. https://github.com/plume-lib/merging, September 2024.

[19] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ASE 2014: Proceedings of the 29th Annual International Conference on Automated Software Engineering*, pages 313–324, Västerøas, Sweden, September 2014.

[20] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André van der Hoek. On the nature of merge conflicts: A study of 2,731 open source Java projects hosted by GitHub. *IEEE Transactions on Software Engineering*, 46(8):892–915, August 2020.

[21] GitHub. Greatest hits. https://archiveprogram.github.com/greatest-hits/, nov 2020.

[22] Georgios Gousios. The GHTorrent dataset and tool suite. In *MSR 2013: 10th Working Conference on Mining Software Repositories*, pages 233–236, San Francisco, CA, USA, May 2013.

[23] Georgios Gousios and Andy Zaidman. A dataset for pull-based development research. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 368–371, New York, NY, USA, 2014. ACM.

[24] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[25] Tristan Hume. Designing a tree diff algorithm using dynamic programming and A*. https://thume.ca/2017/06/17/tree-diffing/, 2017.

[26] J. J. Hunt and W. F. Tichy. Extensible language-aware merging. In *ICSM 2002: Proceedings of the International Conference on Software Maintenance*, pages 511–520, Montreal, Canada, October 2002.

[27] Maximilian Koegel, Jonas Helming, and Stephan Seyboth. Operation-based conflict detection and resolution. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 43–48, USA, 2009. IEEE Computer Society.

[28] Gunnar Kudrjavets, Nachiappan Nagappan, and Ayushi Rastogi. Do small code changes merge faster? a multi-language empirical investigation. In *MSR 2022: 19th International Conference on Mining Software Repositories*, pages 537–548, Pittsburgh, PA, USA, May 2022.

[29] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. Dependent-test-aware regression testing techniques. In *ISSTA 2020, Proceedings of the 2020 International Symposium on Software Testing and Analysis*, pages 298–311, Los Angeles, CA, USA, July 2020.

[30] Simon Larsén, Jean-Rémy Falleri, Benoit Baudry, and Martin Monperrus. Spork: Structured merge for Java with formatting preservation. *IEEE Transactions on Software Engineering*, 49(01):64–83, January 2023.

[31] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. Assessing transition-based test selection algorithms at Google. In *ICSE-SEIP 2019, International Conference on Software Engineering, Software Engineering in Practice*, pages 101–110, Montreal, Canada, May 2019.

[32] Olaf Leßenich, Sven Apel, and Christian Lengauer. Balancing precision and performance in structured merge. *Automated Software Engineering*, 22(3):367–397, May 2014.

[33] Tancred Lindholm. A three-way merge for XML documents. In *DocEng*, 2004.

[34] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, pages 643–653, Hong Kong, November 2014.

[35] José William Menezes, Bruno Trindade, João Felipe Pimentel, Tayane Moura, Alexandre Plastino, Leonardo Murta, and Catarina Costa. What causes merge conflicts? In *SBES '20: Proceedings of the 34th Brazilian Symposium on Software Engineering*, pages 203–212, Natal, Brazil, oct 2020.

[36] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[37] Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, and Michael D. Ernst. Verifying determinism in sequential programs. In *ICSE 2021, Proceedings of the 43rd International Conference on Software Engineering*, pages 37–49, Madrid, Spain, May 2021.

[38] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating GitHub for engineered software projects. *Journal of Empirical Software Engineering*, 22(6):3219–3253, December 2017.

[39] Hoai Le Nguyen and Claudia-Lavinia Ignat. Parallelism and conflicting changes in Git version control systems. In *IWCES'17 - The Fifteenth International Workshop on Collaborative Editing Systems*, Portland, OR, USA, February 2017.

[40] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. How different are different *diff* algorithms in Git? *Journal of Empirical Software Engineering*, 25(1):790–823, January 2020.

[41] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology*, 10(3):308–337, jul 2001.

[42] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA 2015, Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, Baltimore, MD, USA, July 2015.

[43] Md Tajmilur Rahman and Peter C. Rigby. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *ESEC/FSE 2018: The ACM 26th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 857–862, Lake Buena Vista, FL, USA, November 2018.

[44] Benedikt Schesch, Ryan Featherman, Kenneth J. Yang, Ben R. Roberts, and Michael D. Ernst. Evaluation of version control merge tools (appendix). Technical Report UW-CSE-24-09-01, University of Washington Paul G. Allen School of Computer Science and Engineering, Seattle, WA, USA, September 2024.

[45] Georg Seibt, Florian Heck, Guilherme Cavalcanti, Paulo Borba, and Sven Apel. Leveraging structure in software merge: An empirical study. *IEEE Transactions on Software Engineering*, 48(11):4590–4610, 2022.

[46] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. IntelliMerge: A refactoring-aware software merging technique. In *OOPSLA 2019, Object-Oriented Programming Systems, Languages, and Applications*, pages 170:1–170:28, Athens, Greece, October 2019.

[47] Hala Skaf-Molli, Pascal Molli, Charbel Rahhal, and Hala Naja-Jazzar. Collaborative writing of XML documents. In *ICTTA 2008: 3rd International Conference on Information and Communication Technologies: From Theory to Applications*, pages 1–6, Damascus, Syria, apr 2008.

[48] Stack Overflow. 2022 developer survey. https://survey.stackoverflow.co/2022, May 2022.

[49] Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri. Program merge conflict resolution via neural transformers. In *ESEC/FSE 2022: The ACM 30th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 822–833, Singapore, November 2022.

[50] Alberto Trindade Tavares, Paulo Borba, Guilherme Cavalcanti, and Sérgio Soares. Semistructured merge in JavaScript systems. In *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering*, pages 1014–1025, Melbourne, Australia, September 2020.

[51] Alberto Trindade Tavares, Paulo Borba, Guilherme Cavalcanti, and Sérgio Soares. Semistructured merge in JavaScript systems. In *ASE 2019: Proceedings of the 34th Annual International Conference on Automated Software Engineering*, pages 1014–1025, San Diego, CA, USA, September 2019.

[52] Jean-Yves Vion-Dury. Diffing, patching and merging xml documents: Toward a generic calculus of editing deltas. In *Proceedings of the 10th ACM Symposium on Document Engineering*, DocEng '10, pages 191–194, New York, NY, USA, 2010. Association for Computing Machinery.

[53] Fengmin Zhu and Fei He. Conflict resolution for structured merge via version space algebra. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), October 2018.

[54] Fengmin Zhu, Fei He, and Qianshan Yu. Enhancing precision of structured merge by proper tree matching. In *ICSE Companion 2019, Companion Proceedings of the 41st International Conference on Software Engineering*, pages 286–287, Montreal, Canada, May 2019.