Selective Bisection Debugging

Ripon Saha*[1] and Milos Gligoric[2]

[1] Fujitsu Laboratories of America, Sunnyvale, CA-94085, USA
rsaha@us.fujitsu.com
[2] The University of Texas at Austin, Austin, TX-78712, USA
gligoric@utexas.edu

Abstract. Bisection debugging, which is based on binary search over software version history, is widely used in practice to identify the bug introducing commit. However, this process can be expensive because it requires costly compilation and test execution at many commits. We introduce a novel technique—selective bisection (consisting of test selection and commit selection)—to reduce the number of bisection steps, the number of compiler invocations, and the number of executed tests. We evaluated selective bisection on 10 popular open-source projects by performing 25,690 debugging sessions and measuring: (1) savings in number of compiler invocations obtained by commit selection, (2) savings in number of executed tests obtained by test selection, and (3) savings in overall debugging time by selective bisection. Our results show that, in 65% of debugging sessions, commit selection saves between 14% and 71% compiler invocations. Test selection saves 74% of testing effort on average (ranging from 42% to 95%) compared to when developers do not use any test selection. Finally, we demonstrate that one can save substantial time using selective bisection for large projects.

1 Introduction

In large software systems, where many developers work together making hundreds of commits per day [14, 36, 37], coping with regression bugs is one of the most challenging problems. According to Linux Kernel developers, 80% of the release cycle time is dedicated to fixing regression bugs [3]. Identifying the *bug introducing commit* is very important to isolate and understand regression bugs. *Bisection debugging* is a well known technique that performs a binary search over software version history to identify the bug introducing commit. The popular version control systems, such as Git and Mercurial, have in-built commands (`git bisect` and `hg bisect`) to help developers perform bisection debugging [7, 15]. Since these commands are integrated with the version control systems, they are frequently used by developers. For example, a well known Linux developer, Ingo Molnar says about his use of Git bisect [3]:

"I most actively use it during the merge window (when a lot of trees get merged upstream and when the influx of bugs is the highest) - and yes, there

* Most of this work was completed when Ripon Saha was a Ph.D. Student at The University of Texas at Austin.

have been cases that I used it multiple times a day. My average is roughly once a day."

Although bisection debugging can isolate the bug introducing commit quickly in terms of number of bisection steps, the whole process could be still expensive if the program takes a long time (even a couple of minutes) to compile and the tests take a long time to run, which is the case for most large systems [2, 8].

This paper presents a novel technique called *selective bisection* that frequently reduces the cost of bisection debugging by minimizing (1) the number of compiler invocations using *commit selection* and (2) the number of tests to execute using *test selection*. Test selection [10, 14, 19, 20, 23, 30, 31, 35–39, 42] is a well known technique to select tests that are affected by a particular change. Therefore, test selection is used to speed up regression testing [39]. Our key insight is that a significant amount of compilation and testing effort could be reduced during debugging by selecting only relevant commits and running only those tests that are relevant to the buggy changes. To this end, we have proposed a commit selection technique and leveraged an existing test selection technique for bisection debugging. In order to evaluate the effectiveness of our idea, we performed 25,690 bisection debugging sessions in 10 open-source projects. Our empirical evaluation shows that commit selection saves between 14% and 71% compiler invocations across all projects in 65% of debugging sessions. Additionally, test selection saves 74% of testing effort on average (ranging from 42% to 95%) compared to when developers do not use any test selection. This paper makes the following key contributions:

- ⋆ To the best of our knowledge, we are the first to introduce the notion of commit selection and test selection in bisection debugging.
- ⋆ We present a commit selection approach to save the number of compilations. We also present various testing strategies used in bisection debugging, and show how an existing test selection technique can be integrated with various strategies to substantially reduce the testing effort.
- ⋆ We present an extensive evaluation to demonstrate the effectiveness of commit selection and test selection.

2 Background

This section introduces the basic terminology used in this paper and briefly describes bisection debugging and test selection.

2.1 Commit and Version

In this paper, by a *commit* we mean a set of changes that developers include one at a time in a version control system. By a *version*, we mean the snapshot of the code base at a given commit.

2.2 Bisection Debugging

In large projects, commits happen so frequently that a bug introducing commit, even after a few days of introduction, can be hundreds of commits away from

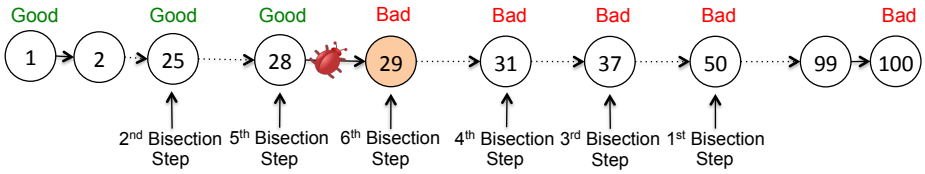

Fig. 1. Process of Bisection Debugging

the current version [2]. Certainly, going through all the commits in software history from the latest known good version is not feasible. Bisection debugging performs a binary search through the commit history to help developers quickly identify the bug introducing commit. `git bisect` is a popular tool for bisection debugging [7]. Given a range of commits where the version (ν_{good}) at the first commit is good and the version (ν_{bad}) at the last commit is bad, `git bisect` checks out the version (ν_{bisect}) at the middle commit. Then ν_{bisect} is tested (manually or automatically) to determine whether it is good or bad, and marked as ν_{good} or ν_{bad} for the next step depending on the test results. The whole process is repeated until a ν_{good} followed by a ν_{bad} is found. Figure 1 illustrates a debugging scenario where a developer starts bisection debugging with 100 commits and finally discover that 29th commit introduced the bug.

Although bisection debugging is based on the binary search, the number of steps in bisection debugging to isolate a bug introducing commit and that of binary search to search a value in a list is not the same. In a list of n values, a binary search can terminate even at the first step if the value is found there. Therefore, the best case performance of binary search is $\mathcal{O}(1)$. On the other hand, in the worst case when the desired value is not in the list, the complexity is $\mathcal{O}(\log_2 n)$. However, the number of steps in bisection debugging is always $\lfloor \log_2 n + 1 \rfloor$, since we do not know which commit actually introduced the bug. Even if the version at first bisection step introduced the bug, we cannot terminate the search until we find two consecutive versions such that a good version is followed by a bad version.

2.3 Test Selection

Given a set of changes in a project, a test selection technique [10, 14, 19, 20, 23, 30, 31, 35–37, 39, 42] seeks to select a subset of tests that are sufficient to test the new version of the program. A test selection technique is *safe* if it selects all tests affected by changes. Among many test selection techniques, we have chosen Ekstazi [9,10] for our work, since our goal was to improve bisection debugging of projects written in JVM languages (e.g., Java, Scala, etc.). Additionally, Ekstazi is publicly available. Ekstazi collects *coverage* for each test class, i.e., dynamically accessed files. It then selects, at a new version, all tests that depend on at least one modified file. Ekstazi collects both the executable and data files (e.g., property files) that are used during the execution of the test class; in the reminder of the text, we denote a test class with test. Ekstazi is considered safe under

certain assumptions [29, 32], e.g., that test cases are deterministic and that the environments used in testing remains unchanged. Prior work showed Ekstazi's effectiveness (selects a small number of tests and provides speedup compared to running all the tests) on a number of open-source projects [4, 10]. Recently, Ekstazi has been adopted by several open-source projects and companies.

In the default configuration, which we use in Section 4, Ekstazi collects a checksum for each dependency used by tests; the checksum is used later to find the tests that should be run after code changes. Ekstazi computes the checksum of executable files (i.e., classfiles) by ignoring the content that is commonly not observed by tests (e.g., debug information). Ekstazi smoothly integrates with popular testing frameworks (e.g., JUnit) and build systems (e.g., Maven) [5], which simplified our study.

3 Selective Bisection

This section describes selective bisection that comprises of two techniques: commit selection and test selection.

3.1 Commit Selection

In bisection debugging, at each bisection step, first the current version is compiled and then tests are executed. However, compiling a large project is costly [12, 40]. *Commit selection* predicts if a certain commit in a bisection step is likely irrelevant to failing tests. If it predicts that a commit is irrelevant to the failing tests, it skips compiling that version, and moves to the next bisection step. Predicting if a given commit ν_{bisect} is relevant or irrelevant to the failing tests is always performed with respect to a reference version ν_{ref}, for which we already have the test results. Our key insight is that for a given pair of versions (ν_{ref} and ν_{bisect}), if we have the test coverage for ν_{ref} and we know all the source code changes between ν_{ref} and ν_{bisect}, we can predict whether the failing tests are affected due to changes between ν_{ref} and ν_{bisect} without compiling ν_{bisect}. If failing tests are not affected by the changes, the test results of ν_{ref} and ν_{bisect} are the same, i.e., if ν_{ref} is good, ν_{bisect} is good; if ν_{ref} is bad, ν_{bisect} is also bad.

Note that this is a prediction, not a determination, because it detects differences between source files, but test selection technique that we used collects tests coverage on compiled code (i.e., classfiles). For example, if two classes are defined in a single source file, they are compiled to two classfiles (and a test can depend on either of those classfiles or both of them), however, by looking at differences between source files, we can detect changes only in the classfile that matches the name of the source file. However, we can make the whole approach correct by automatically detecting inconsistencies due to any inaccurate prediction, and then switching back to traditional bisection debugging. On the other hand, if prediction is accurate, we may save substantial amount of compile time.

Prediction. This section describes our technique to predict if the failing tests are affected by the change between ν_{ref} and ν_{bisect}. The following steps describe the way we predict if a test should execute:

Table 1. An Example of Commit Selection for Jackrabbit for the Failing Test `OakSolrNodeStateConfigurationTest`. Result: Number of Compilations With and Without Commit Selection is 2 and 7 Respectively. Savings: 71%.

G=Good, B=Bad, C=Current, LAG=Last Actual Good, LAB=Last Actual Bad

Step	G	B	C	Affected(G,C)	Affected(B,C)	Compile	LAG	LAB
1	1	100	50	Yes	No	No	1	100
2	1	50	25	No	Yes	No	1	100
3	25	50	37	Yes	No	No	1	100
4	25	37	31	Yes	No	No	1	100
5	25	31	28	Yes	Yes	Yes&Run	1	28
6	25	28	26	No	Yes	No	1	28
7	26	28	27	No	Yes	No	1	28
8	27	28	-	-	-	Yes&Run	27	28

1. For a given version ν_{bisect}, and a reference version ν_{ref}, we extract the source code differences between (ν_{bisect} and ν_{ref})=Δ_{ref} only for Java source files.
2. We extract the file names of each added, deleted, and modified Java files from Δ_{ref}, and store them in a list, $F_{ref} = \{F_1, F_2, .., F_k\}$.
3. We extract the coverage information from ν_{ref}, which contains the information of all tests and the name of source code classes that each test class executes. We convert class names to file names. For most classes, class name is the same as the file name. However, if a class is an inner class then file name does not directly match the class name. For example, for an inner class B in file A, the fully qualified name would be $A\$B$. We will discard the later part of the fully qualified name to get the file name A.
4. Then we search if the failing tests access any of the files in F_{ref}. If yes, then we conclude that we have to run that test for ν_{bisect}.

Commit selection in action. A developer generally starts bisection debugging with a known good version ν_{good} and a known bad version ν_{bad}.

1. We instrument both versions using Ekstazi to collect test coverage matrix at file level for a negligible cost [10].
2. In each bisection step, we predict if failing tests are affected by the changes between i) ν_{bisect} and ν_{good}, and ii) ν_{bisect} and ν_{bad}.
   (a) If both reference versions tell that the failing tests are affected at ν_{bisect}, then we compile ν_{bisect}, run the tests, and mark ν_{bisect} as ν_{good} or ν_{bad} depending on the results.
   (b) If one of the reference versions (ν_{good} or ν_{bad}) tells that the failing tests are not affected, we would simply transfer the corresponding test result of ν_{ref} (ν_{good} or ν_{bad}) to ν_{bisect}, and thus would not compile ν_{bisect}.
3. We keep track of versions where the test results are updated using prediction, and where the results are updated after actual test run. Therefore, we always know the last *actual* good version ($\nu_{lastActualGood}$) and the *actual* bad version ($\nu_{lastActualBad}$) where the good or bad was decided after running tests.

Detecting inconsistencies and switching back. After we get a buggy version ($\nu_{probableBuggy}$) at the end of a debugging session, we check the consistency of our result. For a valid bug introducing version, the version before the

bug introducing changes ($\nu_{probableBuggy-1}$) should be good. Therefore, we test $\nu_{probableBuggy-1}$. If that is a good version, then we conclude that $\nu_{probableBuggy}$ is actually buggy. Otherwise, we perform traditional bisect (without prediction) between $\nu_{lastActualGood}$ and $\nu_{lastActualBad}$. Table 1 illustrate commit selection on an open-source project, Jackrabbit, when its 28th commit is buggy.

3.2 Test Selection

During a bisection debugging session, at each step, (some) tests are executed to determine if the current version is good or bad. To this end, developers may follow one of the three testing strategies:[3]

1. **All Tests** (AT): In the safest form, developers run all the tests to determine if a given version is good or bad.
2. **Failing Tests** (FT): In the most optimistic form, developers only run the failing tests to determine if a given version is good or bad.
3. **All Tests After Failing Tests** (ST): A middle ground may be that developers first run the failing tests. If they keep failing, the version is marked as bad. However, if the failing tests pass, developers run other tests to make sure that the version is good indeed.

At a first glance, the second strategy may be tempting but it may not always give the expected result, since even if the failing test passes, it is not guaranteed that the version is good since other affected tests may fail. To get developers' feedback, we asked a question on Stack Overflow[4]. As we expected, we got mixed answers. One developer prefers the second strategy but another developer disagrees. By combining the first and the second strategy, we introduced ST, which we believe is the optimal strategy that gives the expected result.

Our key insight is that developers could safely use a test selection tool such as Ekstazi to select only the tests that are affected by the changes between ν_{bad}/ν_{good} and ν_{bisect} in each debugging step. In traditional test selection, generally a subset of tests are selected with respect to a single (often the previous) version. However, for bisection debugging, we can take advantage of both ν_{good} and ν_{bad} to reduce the number of tests further. The idea is that only intersection of two selected subsets for ν_{bisect} with respect to ν_{good} and ν_{bad} are enough to test ν_{bisect} safely [11].

Let us discuss a debugging scenario with a hypothetical example, as presented in Figure 2. Assume that Alice is debugging with seven versions of a program (ν_1 and ν_7) to identify a bug introducing commit (which are changes introduced in ν_2 in our example). Let us assume that there are n source files $\{C_1, C_2, \ldots, C_n\}$ and m tests $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m\}$ in the project. In Figure 2, the label between two consecutive versions represents the program difference between

[3] It should be noted that we have not found any study on the use of bisection debugging; the identified methodologies are inspired by reading blogs and posts on GitHub related to bisection debugging [7, 8].

[4] `http://goo.gl/oHyX2g`

Table 2. A Hypothetical Test Coverage Matrix

	C1	C2	C3	C4	C5	C6	⋯	Cn
$\mathcal{T}_1$	✓			✓			⋯	
$\mathcal{T}_2$		✓		✓			⋯	
$\mathcal{T}_3$	✓		✓				⋯	
$\mathcal{T}_4$				✓			⋯	
$\mathcal{T}_5$					✓		⋯	
$\mathcal{T}_6$			✓				⋯	✓
⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯
$\mathcal{T}_m$							⋯	✓

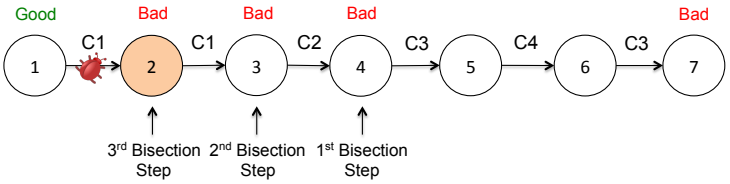

Fig. 2. Some Hypothetical Change Scenarios in a Git Repository

those two versions in terms of Java files. For instance, $\delta(\nu_1, \nu_2) = C_1$, which means that the file C_1 has been changed between ν_1 to ν_2. The level of granularity in our example is at the file level, which is also the level of granularity of Ekstazi (Section 2). Table 2 represents a hypothetical test coverage matrix. In this example, we assume that the matrix remains constant, but our approach works even if the matrix changes due to changes in code.

Savings when running all tests (AT). Now let us simulate the number of tests Alice has to run without test selection during this debugging session. First, Alice marked ν_1 as `good` and ν_7 as `bad`. At this step bisection debugging moves to ν_4. Alice runs all m tests to test ν_4. In this case, ν_4 would be buggy since the bug was introduced in ν_2. So Alice marks ν_4 as bad. The next bisection step is at ν_3. Alice runs the tests again, marks ν_3 as buggy. Finally Alice does the same thing for ν_2, and identifies that ν_2 is the bug introducing commit. So Alice executed all tests 3 times, which is $3 \times m$ tests in total.

Now we simulate the same scenario when Alice integrates Ekstazi in the project and then starts debugging. In the first step at ν_4, Ekstazi first uses ν_1 and then ν_7 as a reference version to select tests for ν_4. When ν_7 is the reference version, $\delta(\nu_4, \nu_7) = \{C_3, C_4\}$. From the coverage matrix in Table 2 (which is generated by Ekstazi) we see that only $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$, $\mathcal{T}_4$, and $\mathcal{T}_6$ are affected by the changes. When ν_1 is the reference version, the affected tests are $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ since the change set is $\{C_1, C_2\}$. Therefore, Alice runs only the tests in the intersection, i.e., $\mathcal{T}_1, \mathcal{T}_2, and\ \mathcal{T}_3$, to test ν_4. The results for the other tests can be transferred from the corresponding reference versions. In the second bisection step, Alice need not run any tests in ν_3 since no test is affected by both changes in C_1 and C_2. Finally, Alice runs $\mathcal{T}_1$ and $\mathcal{T}_3$ to test ν_2 due to the change in C_1. Therefore, in total Alice runs only five tests using test selection instead of $3 \times m$ tests.

Savings when running only failing tests (FT). Now we consider the scenario where Alice plans to run only failing tests. Even in this scenario, Ekstazi can save testing effort by not running the failing tests at all, if they are not affected by the changes. Since in this scenario Alice would run only the failing tests, she knows the failing tests in advance. To simplify the discussion, let us assume that Alice got only one failing test, $\mathcal{T}_1$. Now while testing at the first bisection step ν_4, we can see that the changes between ν_4 and ν_7 are $\{C_3, C_4\}$. From the coverage matrix we observe that test $\mathcal{T}_1$ is affected by the change in C_3 and C_4. Therefore, Ekstazi would run $\mathcal{T}_1$. Since at ν_4 the result of $\mathcal{T}_1$ should be a failure, ν_4 would be bad. Now for the second bisection step, ν_3, Ekstazi sees that only C_2 changed. From the coverage matrix we observe that $\mathcal{T}_1$ is not affected by this change. Therefore, Ekstazi would not run the test at all, and would transfer the results of $\mathcal{T}_1$, which is a "failure" from ν_4 to ν_3. Therefore, ν_3 would be bad. Further, Ekstazi would run $\mathcal{T}_1$ for ν_2 since it is affected. So Ekstazi would save running the failing test one out of three times for this example.
Savings when running all tests after failing tests (ST). Since ST is a combination of AT and FT, we do not describe it step by step.

4 Empirical Evaluation

To investigate the effectiveness of selective bisection, we performed an empirical evaluation in terms of three research questions.

RQ1: How much compilation effort is saved through commit selection?

RQ2: How much testing effort is saved through test selection for different testing strategies (AT, FT, and ST)?

RQ3: How much overall time is saved through selective bisection?

4.1 Projects

We used 10 open-source projects in our evaluation. We followed several criteria, similar to prior studies on regression testing [10, 34], to select these projects. Specifically, the projects i) use Git as a version control system, ii) use Maven as a build system, iii) have at least 100 commits, and iv) build without any error. These projects are from diverse application domains and have been widely used in software testing research. The first requirement is necessary since we are investigating git bisect as bisection debugging. The requirements of Maven and JUnit tests were set to make our experiments fully automatic. Finally, the requirement of 100 commits helps ensure that the projects are non trivial. For each selected project, Table 3 shows its name, the start and end version SHA (which we consider to be a starting pair of a good and a bad version in each debugging session), the code size in terms of lines of code (LOC), the number of source files and test classes, and their build time and test execution time (on the latest version). From the table, we can see that the sizes of the projects vary from small (Codec) to fairly large (Jetty) in terms of LOC. Their build times (without test execution) vary from few seconds to several minutes, and

Table 3. Projects Used in the Evaluation

Project	Start SHA	End SHA	KLOC	#Source files	#Test classes	Time [mm:ss] build	test
CCompiler	14a9e6fe	a8a53e83	239.2	630	262	01:08	02:20
Codec	5af6d236	535bd812	17.6	67	48	00:29	00:18
Collections	45a0337e	c87eeaa4	60.3	357	160	00:24	00:42
Lang	9e575c4d	17a6d163	69.0	159	134	00:30	00:33
Math	ff4ec1a3	471e6b07	174.8	841	479	00:30	02:35
Net	17ecff74	4450add7	26.9	224	42	00:26	01:13
GraphHopper	d1a0fd81	c0a328f8	43.6	254	100	00:35	00:45
Guava	67695cce	e9a23fe5	274.2	1,372	364	01:09	08:15
Jackrabbit	4a309b76	222b4cda	253.1	1,654	539	02:12	44:18
Jetty	f645e186	f630a841	301.1	1,929	550	07:05	30:29
$\sum$	N/A	N/A	1,459.8	7,487	2,678	14:28	91:28
Average	N/A	N/A	146.0	749	268	01:27	09:08

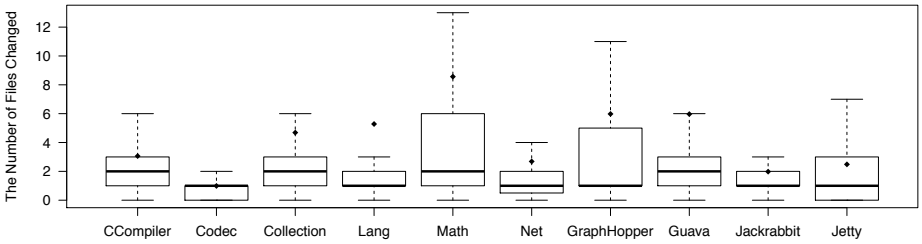

Fig. 3. Distribution of Number of Files Changed in Each Commit

test execution times vary from few seconds to 44 minutes (Jackrabbit). The last two rows show the total and average values computed across all projects.

We also present the distribution of changes per commit in Figure 3 in terms of the number of files added, deleted or modified. We presented the changes at file level, since all of our analysis is at that level. From the figure we observe that in all projects the number of files changed was one or two (median values). This statistics further motivated us to propose selective bisection debugging since it is highly likely that a small number of tests would be selected due to small changes.

4.2 Experimental Setup

For an extensive evaluation, we designed our experiment based on simulation to replicate the steps in bisection debugging. More specifically, in order to answer **RQ1** we took 100 versions $[\nu_i, \nu_{i+99}]$ for each project, and considered that any of the intermediate versions $[\nu_{i+1}, \nu_{i+98}]$ can be the one with a bug introducing commit. Then for each intermediate version ν_j where $(i+1) \leq j \leq (i+98)$, we set the test results w.r.t. the bug introducing commit. For example, if for a given debugging session we assume that ν_5 is the bug introducing commit, all the commits after (and including) ν_5 would be bad, and all the commits before ν_5 would be good. Then for each bisection step, we have used Ekstazi to select tests for the version under test based on the real changes between that version and good/bad version and test coverage information.

To answer **RQ2**, in addition to assuming a buggy version, we also had to assume the fault-revealing tests. For a change set between two versions, literally any test, which is affected by the changes, can be the failing test. For example, when we assume that ν_j is buggy, we first check which tests are affected by the changes between ν_{j-1} and ν_j. If there are m tests that are affected by the bug introducing change, any subset of them can be failing. If we assume that there is only one failing test, we have to simulate our experiment m times for the buggy version ν_j. For two failing tests, we have to simulate for $\binom{m}{2}$ times, and so on. In order to keep the experiment cost affordable, we assumed there is only one failing test due to the bug. Even for a single test failure, we have simulated our experiment 25,690 times in 10 projects.

4.3 Results

This section presents the experimental results for our research questions.

RQ1: Savings due to commit selection. We present the savings in compilation effort in terms of the proportion of compilations skipped. More specifically, for a given bisection debugging session, if we need to compile nc times without commit selection and mc times with commit selection, the savings is computed as $\frac{nc-mc}{nc}$. It should be noted that a negative saving indicates the case where commit selection actually increases the cost. This may happen when commit selection predicts an irrelevant commit inaccurately, and our technique moves some steps back to perform traditional bisection (Section 3). Therefore, to present the complete results, we provide the number of simulations where we decreased or increased the number of compilations in Table 4, and complete distribution of savings, both positive and negative, in Figure 4.

From the results, we observe that commit selection frequently reduces compilation cost. The "Total" row in Table 4 shows that commit selection helped saving compilation cost in 65% of simulations, whereas it increased the cost in only 6% of simulations. Interestingly, most of the cost increase came from Jetty. For other projects, the cost increase happened in only 1% of simulations. Commit selection did not change any effort in compilation in 29% of simulations. From Figure 4, we observe that the improvement (in 65% of simulations) varied between 14% and 71% across projects. Although cost increase can be also high in some cases (up to 67% for Jetty; this cannot be seen in Figure 4 as we do not show the outliers), our results show that this happens rarely.

RQ2: Savings due to test selection. We present the savings in testing effort in terms of the proportion of tests that one can skip with test selection. More specifically, for a given bisection debugging, we compute the savings as:

$$PT_{ts} = \frac{\sum_{for-each-bisection-step} n(T_{ts})}{\sum_{for-each-bisection-step} n(T)} \tag{1}$$

$$savings = 1 - PT_{ts} \tag{2}$$

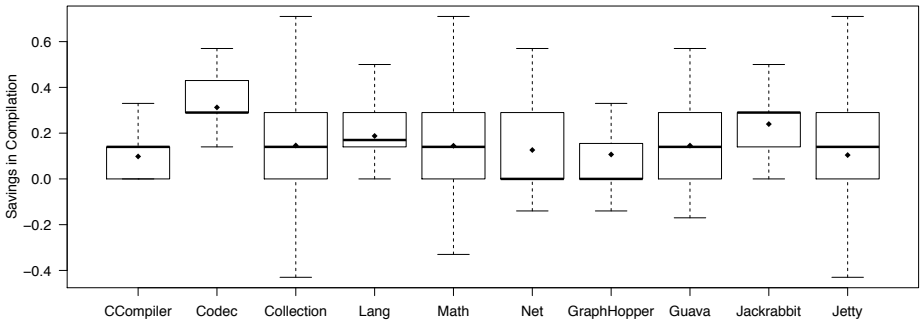

Fig. 4. Savings in the Number of Compiler Invocations via Commit Selection

Table 4. Number of Simulations When the Compilation Cost Decreased, Remained the Same, or Increased due to Commit Selection

Projects	**#Simulations**	**#Decreased**	**#Same**	**#Increased**	**%Decreased**
CCompiler	6,171	3,615	2,556	0	59
Codec	91	87	4	0	96
Collection	975	657	212	106	67
Lang	678	548	125	5	81
Math	5,275	3,411	1,856	8	65
Net	204	96	104	4	47
GraphHopper	1,247	613	612	22	49
Guava	1,347	997	346	4	74
Jackrabbit	3,436	3,034	313	89	88
Jetty	6,266	3,667	1,404	1195	59
All Projects	25,690	16,725	7,532	1,433	65

where $n(T_{ts})$ is the number of tests to run selected by Ekstazi and $n(T)$ is the number of tests to run without Ekstazi. Therefore, the savings may vary between 0 and 1, which can be translated to percentages as well.

Figure 5 presents the distribution of savings, computed by Equation 2, for AT, FT, and ST strategies. Our results show that, regardless of a testing strategy, test selection is very effective to reduce the number of tests in bisection debugging. For AT (the first/orange box in each group), the median savings varied from 43% (CCompiler) to 95% (Codec). Considering that there are hundreds of tests in projects and a number of steps in bisection debugging, the savings are significant. For example, Jetty has 550 test classes. In our simulation for 100 commits, it takes 6 or 7 steps to complete a bisection debugging. Therefore, we may need to run 3,850 tests to test Jetty in a single debugging session if all tests are executed. From Figure 5, we see that the median savings in Jetty is 63%. So test selection may skip running 2,425 tests to debug Jetty. For majority of the projects, the savings in AT is 70% or more. Even in FT (the second/gray box in each group), where only the failing tests are run, test selection can reduce the number of executed tests by 29%-71% (median values). Like AT, we also observe a similar savings in ST, which we consider to be the optimal strategy.

Fig. 5. Savings in the Number of Tests during Bisection Debugging via Test Selection

The median savings (the third/white box in each group) varied from 42% to 91% across projects. For majority of projects, the median savings are more than 70%. Finally, the fourth/blue box in each group shows the savings for ST when we also apply commit selection. Results show that the savings of ST is further increased (up to 7%) by commit selection. Interestingly, although in some cases commit selection increases the number of compilation, it does not increase the number of tests to run.

RQ3: Overall time savings. Since ST is the optimal strategy with respect to the testing effort and correctness, we calculate the end-to-end time savings for ST. It should be also noted that we considered the median compilation and testing savings for each project. Table 5 presents the total time required for traditional bisection and selective bisection including the time required for program instrumentation and test coverage collection by Ekstazi. From the results, we observe that, on average, we achieved 24% to 60% of time savings across projects. Even for small projects like Codec, we achieved more than 2 minutes of savings, which is 44% of time using traditional bisection. The savings can be as big as 1 hour and 23 minutes (Jackrabbit). It also should be noted that this saving is for ST, which is already considered optimal for traditional bisection. If developers follow AT, the savings would be even more.

5 Discussion

Effect of simulation based evaluation. Our evaluation is based on simulation. However, in the context of measuring savings using selective bisection debugging, there is no difference between a real bug reproducing experiment and a simulation. For example, in our evaluation, when we assume that ν_i is buggy and $\mathcal{T}_j$ is the failing test, we followed exactly the same steps to isolate the bug what any bisection debugging technique (e.g., git bisect) would take if there is a real bug in ν_i and $\mathcal{T}_j$ is the real failing test. And since, we have used real tests in each version and the real commits, we got exact number of tests in

Table 5. Time Savings (in Seconds) Using Selective Bisection for *ST*

Projects	Traditional bisection	CS+TS	Ekstazi overhead	Selective bisection	Total time savings	Savings [%]
CCompiler	1,029	672	81	753	276	27
Codec	276	150	4	154	122	44
Collection	329	187	6	193	136	41
Lang	337	200	5	205	132	39
Math	831	387	152	539	292	35
Net	403	231	5	236	167	41
GraphHopper	423	295	6	301	122	29
Guava	1,880	683	75	758	1,122	60
Jackrabbit	8,873	3,299	544	3,843	5,030	57
Jetty	8,462	5,515	908	6,423	2,039	24

simulation that developers need to run with and without test selection. Furthermore, due to simulation our advantage is that we were able to conduct massive number of experiments for many combination of buggy version and failing tests, that would have never been possible with real bugs.

Effect of number of commits. In our simulation, we have isolated each bug by considering only 100 commits. Our rational is that developers use bisection debugging like git bisect for a reasonable number of commits since linear search for a bug introducing commit is not effective [2]. Therefore, we believe that the evaluation with 100 commits shows the effectiveness of our technique but in a wider range of commits, our technique would provide additional savings.

6 Threats to Validity

External. The projects used in our evaluation may not be the representative of the general projects population. To mitigate the threat, we performed our experiments on a diverse set of projects in terms of size, number of tests, and applications. However, we do not generalize our results to other projects. Furthermore, we performed our experiments with 100 commits in each project. For a different set or length of commits, we may have different savings. We discussed the rationale and effect of this choice in the previous section. We have used only projects that are written in the Java programming language. In the future, it will be interesting to explore if the results differ for projects written in other programming languages.

We have used Ekstazi as the test selection tool. A different tool, which tracks test dependencies on methods or statements, would likely produce different results (and select even small number of tests). Future work should evaluate various test selection techniques with selective bisection.

Internal. The implementation of Ekstazi or our scripts for bisection debugging may have bugs which may impact our conclusion. However, Ekstazi is, to the best of our knowledge, the only available tool for regression test selection (for Java). Furthermore, it has been adopted in a number of open-source projects,

which increases our confidence in its correctness. We have also performed many small experiments and code reviews to validate our scripts. Therefore, the effect of this threat should be minimal.

We have assumed that there is only one bug introducing commit in the versions under investigation. Furthermore, we assume that the monotonicity property holds, i.e., once a change introduces a bug, any subsequent version manifests the bug as well.

Construct. Our experiment is based on simulation. We have already discussed the effect of this threat in detail.

7 Related Work

Automated debugging has been an active research area over the past few decades. To date, researchers proposed many approaches for localizing and isolating the root causes of bugs automatically. Test selection is also an active research area for a long time. Therefore, related work in these fields are enormous. In this section, we focus on the representative work in each area.

Bug Localization. Researchers proposed many automated bug localization approaches to aid debugging. Existing techniques can be broadly categorized into two categories: dynamic [1] and static [16]. Spectrum based bug localization [1, 17, 21] and dynamic slicing [43] are some of the well known techniques in this category. Spectrum based techniques generally monitor the program execution of passing and failing tests. Then based on the execution traces, these tools present developers a ranked list of suspicious lines. Pastore et al.'s [27] and Zuddas et al.'s [46] techniques do not only provide the suspicious lines but also provide explanations to help developers understand the bug.

Static approaches, on the other hand, do not require any program tests or execution traces. In most cases, they need only program source code and/or bug reports. The static approaches can be also divided into two categories: i) program analysis based approaches, and ii) information retrieval based approaches. FindBug [16] and Error Prone [6] are two popular bug localization tools based on static program analysis that can detect bugs by identifying buggy patterns that frequently happen in practice. On the other hand, IR based approaches utilize the contents of bug reports. In these approaches [22, 25, 28, 33, 45], each bug report is treated as a *query*, and all the source files in the project comprise the *document collection*. IR techniques then rank the documents by predicted relevance, returning a ranked list of candidate source code files that may contain the bug. Recently, researchers also combined the spectrum based and IR-based bug localization to get advantages from both approaches [18]. Unlike the foregoing techniques, selective bisection localizes the bug introducing commit rather than suspicious lines.

Isolating Buggy Changes. Ness and Ngo [24] first proposed a linear approach to isolate buggy changes. In their approach, when a bug is discovered in a particular version, they consider a set of ordered changes to investigate, and apply one after another until they find first buggy version. Gross first introduced the notion

of bisection debugging [13]. However, his context was a little different from `git bisect`. Given two versions of a program, Gross applied binary search by partitioning program source code until the bug is isolated in minimum changed lines. Delta debugging is a well known technique to isolate a buggy change between two versions [41].

Recently Ziftci and Ramavajjala [2] proposed an algorithm to rank all the commits based on a suspiciousness score to find the bug introducing commit as early as possible. They calculated the suspicious score using metrics such as the change size, and the distance between changed files and the project under investigation. However, all these prior approaches run all tests once a version is selected for testing. In this paper, we use dynamic test dependencies to select commits and then we reduce the number of executed tests at each version.

Test Selection. To date, researchers have proposed quite a few approaches for test section [10,14,19,20,23,30,31,35–37,39,42]. These approaches vary in terms of strategies such as static program analysis based [44] vs. dynamic test coverage based [10], and/or granularity of tests e.g., class [10,26] vs. method [29]. In this paper, we have used Ekstazi, which is a dynamic tool and works at the class granularity. In this work, our objective is not to introduce any new test selection technique. Rather, we introduce an application of test selection in automated debugging to improve debugging effectiveness.

8 Conclusion

In a large software project, it frequently happens that a bug is detected many commits after it was actually introduced. In this case, bisection debugging such as git bisect is frequently used to isolate the bug introducing commit. However, for large projects even bisection debugging may be expensive. In this paper, we introduced selective bisection debugging, which comprises of commit selection and test selection. We investigated the savings through selective bisection debugging for various testing strategies, where developers execute all tests, only failing tests, and execute passing tests if and only if the failing tests pass. Our evaluation shows that commit selection can save compilation time in 65% of debugging scenarios. The savings (in number of compiler invocations) may vary from 14% to 71%. Test selection can skip up to 95% tests during debugging, where developers follow safe approach, i.e., execute all tests. The saving is also very similar if a developer first execute the failing tests, and then execute the passing tests if and only if failing tests pass. Finally, we demonstrate that the overall time savings can be substantial using selective bisection for large projects. We believe our results will encourage developers to use selective bisection debugging, and researchers to investigate commit selection and test selection in more detail.

Acknowledgments. We thank Ahmet Celik, Julia Lawall, and Darko Marinov for their feedback on a draft of this paper. This research was partially supported by the NSF Grant No. CCF-1566363 and a Google Research Faculty Award.

References

1. R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. Van Gemund. A practical evaluation of spectrum-based fault localization. *JSS*, 82(11):1780–1792, 2009.
2. Finding culprits automatically in failing builds - i.e. who broke the build? GTAC 2013. `https://www.youtube.com/watch?v=SZLuBYlq3OM`.
3. C. Couder. Fighting regressions with git bisect. `https://www.kernel.org/pub/software/scm/git/docs/git-bisect-lk2009.html`.
4. N. Dini, A. Sullivan, M. Gligoric, and G. Rothermel. The effect of test suite type on regression test selection. In *ISSRE*, pages 47–58, 2016.
5. Ekstazi. `http://www.ekstazi.org`.
6. Error Prone. `http://errorprone.info/`.
7. Git bisect. `https://git-scm.com/docs/git-bisect`.
8. OrientDB. `https://github.com/orientechnologies/orientdb/issues/2581`.
9. M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *ICSE Tool Demonstration Track*, pages 713–716, 2015.
10. M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, pages 211–222, 2015.
11. M. Gligoric, R. Majumdar, R. Sharma, L. Eloussi, and D. Marinov. Regression test selection for distributed software histories. In *CAV*, pages 293–309, 2014.
12. M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *OOPSLA*, pages 599–616, 2014.
13. T. R. Gross. Bisection debugging. In *AADEBUG*, pages 185–191, 1997.
14. K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *ICSE*, pages 483–493, 2015.
15. Hg bisect. `https://www.mercurial-scm.org/repo/hg/help/bisect`.
16. D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Note*, 39(12):92–106, 2004.
17. J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.
18. T.-D. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: better together. In *FSE*, pages 579–590, 2015.
19. H. Leung and L. White. A cost model to compare regression test strategies. In *ICSM*, pages 201–208, 1991.
20. H. K. N. Leung and L. White. Insights into regression testing. In *ICSM*, pages 60–69, 1989.
21. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, 2005.
22. S. Lukins, N. Kraft, and L. Etzkorn. Bug localization using latent Dirichlet allocation. *IST*, 52(9):972 – 990, 2010.
23. A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *ICST*, pages 21–30, 2011.
24. B. Ness and V. Ngo. Regression containment through source change isolation. In *COMPSAC*, pages 616–621, 1997.
25. A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *ASE*, pages 263–272, 2011.
26. A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. *ACM SIGSOFT Software Engineering Notes*, 29(6):241–251, 2004.

27. F. Pastore, L. Mariani, and A. Goffi. Radar: A tool for debugging regression problems in C/C++ software. In *ICSE Tool Demonstration Track*, pages 1335–1338, 2013.
28. S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *MSR*, pages 43–52, 2011.
29. G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2):173–210, 1997.
30. G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *ICSM*, pages 358–367, 1993.
31. G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *ICSE*, pages 201–210, 1994.
32. G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *TSE*, 22(8):529–551, 1996.
33. R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.
34. A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *FSE*, pages 246–256, 2014.
35. Streamline testing process with test impact analysis. `http://msdn.microsoft.com/en-us/library/ff576128%28v=vs.100%29.aspx`.
36. Testing at the speed and scale of Google. `http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html`.
37. Tools for continuous integration at Google scale. `http://www.youtube.com/watch?v=b52aXZ2yi08`.
38. S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *ISSTA*, pages 140–150, 2007.
39. S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2):67–120, 2012.
40. Y. Yu, H. Dayani-Fard, and J. Mylopoulos. Removing false code dependencies to speedup software build processes. In *CASCON*, pages 343–352, 2003.
41. A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *FSE*, pages 253–267, 1999.
42. L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *ICSM*, pages 23–32, 2011.
43. X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG*, pages 33–42, 2005.
44. J. Zheng, B. Robinson, L. Williams, and K. Smiley. An initial study of a lightweight process for change identification and regression test selection when source code is not available. In *ISSRE*, pages 225–234, 2005.
45. J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? More accurate information retrieval based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.
46. D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso. Mimic: Locating and understanding bugs by analyzing mimicked executions. In *ASE*, pages 815–826, 2014.