Scalable Isolation of Failure-Inducing Changes via Version Comparison

Mohammadreza Ghanavati, Artur Andrzejak, Zhen Dong
Institute of Computer Science
Heidelberg University, Germany
{mohammadreza.ghanavati, artur.andrzejak, zhen.dong}@informatik.uni-heidelberg.de

***Abstract*—Despite of indisputable progress, automated debugging methods still face difficulties in terms of scalability and runtime efficiency. To reach large-scale projects, we propose an approach which reports small sets of suspicious code changes. Its essential strength is that size of these reports is proportional to the amount of *changes* between code commits, and not the total project size. In our method we combine version comparison and information on failed tests with static and dynamic analysis.**

We evaluate our method on real bugs from Apache Hadoop, an open source project with over 2 million LOC[1]. In 2 out of 4 cases, the set of suspects produced by our approach contains exactly the location of the defective code (and no false positives). Another defect could be pinpointed by small approach extensions. Moreover, the time overhead of our approach is moderate, namely 3-4 times the duration of a failed software test.

***Index Terms*—Automated debugging, version comparison, failure-inducing changes, thin slicing, large-scale projects, software tests**

I. Introduction

Debugging is an expensive and time-consuming task in the software development process. According to studies, half of the programming time of the developers dedicates to investigate and correct bugs. The total cost of testing and debugging of the software development can easily range from 50 to 75 percent of the total development cost [22]. For these reasons, automated debugging has attracted a great deal of interest.

Techniques of automated debugging attempt to find the causes of a program failure without or with only minimal human involvement. In practical terms, after analysis of data obtained from testing results and code instrumentation, a programmer is supplied by a ranked list of suspicious code locations. By examining these locations she can possibly identify the code fragment bearing the true defect.

Despite of indisputable recent advances ([6], [8], [3], [1], [15], [4]), automated debugging is still facing significant challenges preventing its widespread adoption [13]. One of the essential ones is that while excelling at fault localization, they usually do a poor job in facilitating fault understanding. However, even knowing a (potential) fault location still requires the developer to find out what could happen there - a cognitively demanding task.

The second weakness of automated debugging is its limited scalability in terms of program size. Here even pinpointing the 0.1% of code which might contain the defect is not precise enough. Such level of specificity means that on a project with 100k Lines of Code (LOC), a developer still needs to examine 100 suspicious lines to find the defect. This is beyond the typical acceptance levels of programmers (most users do not inspect search results after the 1st page [13]), greatly reducing the utility of such techniques in large-scale projects.

We attempt to approach the latter problem by narrowing our focus to scenarios where software is developed through a series of minor modifications, with each intermediate version being tested thoroughly. This is a typical way how medium-size and large-size projects are developed, for example in a setup of continuous integration testing [22]. Consequently, this assumption is not a serious limitation, especially since our technique targets large-size projects.

Our basic idea is to use *version comparison* to localize newly introduced defects in the latest development version. Version comparison contrasts the behavior of two software versions under the same unit and/or integration tests. In more detail, using static and dynamic analysis we compare the sets of statements executed in the latest (faulty) version against those executed in a previous (base) version. We assume here that the faulty new version has caused a unit/integration test to fail, while the same test succeeded on the base version.

The key advantages of our approach is its precision and efficiency. Assuming that only the recently changed code contains the defect (this is not always but usually true), we can reduce the set of suspicious statements to a few LOC, see Section III. The good news here is that size of this set depends primarily on the *size of changes* (i.e. amount of differences between commits) and not on the total project size. Assuming frequent commits and test runs, our technique is thus likely to scale and present the developers a small set of suspicious statements - even for very large projects.

Secondly, we show in Section III-C that the overhead introduced by our approach is moderate. The essential additional execution cost comes from the requirement to execute a (crashing) test on each of the instrumented base and the instrumented faulty version. Due to very sparse instrumentation, the execution time of the instrumented versions is almost identical to the original versions. As shown in Table IV, the total time of executing our approach is about 2.5-3.5 times the duration of the failed test.

Finally, while not evaluated in this work, our approach is

[1]On September 14, 2013, Ohloh (http://www.ohloh.net/p/Hadoop) was reporting that Apache Hadoop has 2,280,391 lines of code.

potentially capable to contribute to failure understanding. To this aim, one can present the developer the contents of the relevant variables as well as execution paths in both base and faulty versions. While this requires future work, it is a promising and low-overhead way to supporting a developer in making sense what has changed during the execution of suspicious code.

In summary, this paper makes the following contributions:

- We propose and implement an approach to isolate test-failing defects which is based on comparing subsequent versions of software under development. It combines static analysis (backward slicing) and information on changed code to indicate which (small) code sections should be be instrumented. A subsequent dynamic analysis (code coverage of a failing and passing version) reveals the statements which are likely to contain the defects (Section II).
- We evaluate our approach on real defects from a large-scale software project, namely Apache Hadoop (Section III). The results show that it can pinpoint the defective statements with high precision.
- We also compare the execution overhead of the images instrumented according to our approach against alternative instrumentation schemes. Our results show that the overhead of the dynamic analysis proposed by us is negligible (Section III-C).

This paper is organized as follows: The approach and its implementation is described in detail in Section II. The result of the experiments and threats to validity are presented in Section III. Section IV considers the related work, and finally Section V concludes the paper.

II. The Version Comparison Approach

This section describes the details of our approach. As indicated in Section I, we assume that software under development evolves as a series of versions, each one checked by executing one or more test suites. We trigger our automated debugging approach on the event that some test T has failed while executing the latest software version. We denote this latest version as v_f and call it a *failing* version. We also retrieve from repository some previous software version v_p (called *passing* version) which passed T successfully; usually it is a version directly preceding v_f.

A. Approach overview

The steps of our approach are explained below and illustrated in Algorithm 1.

As first we retrieve the set of changes between v_p and v_f, i.e. $Dif = \Delta(v_p, v_f)$ and call it a *difference set*. This is done with tools provided with popular software configuration management systems like SVN, Git, CSV.

As a consequence of the failure of T on v_f, the JVM (or operating system) provides a stack trace which is analyzed by our approach. We call the code location referenced by the top-level entry within this trace (yet not devoted to exception handling) a *failure manifestation site* or just *failure site* f_{site}.

Algorithm 1 Steps of the proposed approach

```
Step 1: Find the differences of versions $v_p$ and $v_f$:
    $Dif = \Delta(v_p, v_f)$
Step 2: Retrieve failure site $f_{site}$ from the stack trace
Step 3: Compute backward slices for each version:
    $BSlice_p = \text{BackwardSlice}(v_p, f_{site})$,
    $BSlice_f = \text{BackwardSlice}(v_f, f_{site})$
Step 4: Compute the intersections $IS_x = BSlice_x \cap Dif$ and instrument versions:
    $v_{p,inst} = \text{Instrument}(v_p, IS_p)$,
    $v_{f,inst} = \text{Instrument}(v_f, IS_f)$
Step 5: Execute test $T$ on each $v_{p,inst}$ and $v_{f,inst}$ to get code coverage profiles:
    $cov_p = \text{Run}(v_{p,inst})$,
    $cov_f = \text{Run}(v_{f,inst})$
Step 6: Get list of suspects by applying filtering lemmas:
    $SuspectSet = \text{FilteringLemmas}(cov_p, cov_f)$
```

We use f_{site} as seed to compute (for each version v_p, v_f) the *backward slice* $BSlice_p$, $BSlice_f$ [18], [16]. Essentially, it is the set of code statements which could have affected variable values at the failure site.

Subsequently, we compute (for each version v_p, v_f) the intersection IS_x of the backward slice $BSlice_x$ and the difference set Dif as $IS_x = Dif \cap BSlice_x$ ($x \in \{p, f\}$). This intersection gives us statements and method names which are likely to contain the defect. In the next step, we instrument the function calls within this intersection IS_x for both passing v_p and failing v_f version.

In the next step we re-execute the test T on the both instrumented versions of v_p and v_f and record coverage information. The results are *coverage profiles* cov_p and cov_f, i.e. reports which code has been executed in the respective version.

The last step involves comparison and filtering of the coverage profiles using the following lemmas.

Filtering lemmas

The following statements are included in the set of suspects:

1) All statements *added* to or changed in v_f which are in the *failing coverage profile* cov_f.
2) All statements *deleted* from v_p which are in the *passing coverage profile* cov_p.

Finally, the resulting list of suspicious statements (suspects) together with their locations is reported to the developer as the potential root causes of a test failure.

```
org.apache.hadoop.ipc.StandbyException: Operation category
READ is not supported at the BackupNode
at org...$BNHAContext.checkOperation(BackupNode.java:443)
at org...FSNamesystem.checkOperation(FSNamesystem.java:759)
at org...system.getServerDefaults(FSNamesystem.java:1019)
  ...
  ...
```

Figure 1. Stack trace of bug reported by Issue HDFS-3856 (bold: failure site)

We exemplify our approach on a real defect from the Apache Hadoop project, see Section III-A1.

B. Further aspects

In the following we discuss some secondary aspects of our approach.

As mentioned in Section II-A, we examine the stack trace of the test execution on v_f to retrieve the failure site. However, usually we cannot take the top-most entry of the stack trace as this frequently points to logger code (or some exception-handling code). Therefore we use a heuristic and check the stack trace entries (from the topmost one) whether they point to "interesting" code, i.e. non-library and functional code. Figure 1 shows an example of the stack trace for issue Hadoop-3856 (Section III-A1), where the 2nd topmost entry points to assumed failure site.

The other aspect which requires explanation is slicing. For a given program P and statement s with a variable v at the program location ℓ, a *backward slice* computed from s contains all of the statements in P which can affect the value of v [18]. It is obvious that if ℓ is the failure site, a backward slice includes all of the statements which might be the root cause of the failure at ℓ.

However using traditional full slicing [18] results in a too large size of slice specially in real large-scale applications. To address this issue, we use thin slicing [16] which only contains statements that directly affect the value of the seed statement.

C. Implementation

Our approach is implemented on the top of WALA [17] which is a static analyzer developed by IBM. Slicing and instrumentation are the features of WALA which making it useful for our approach. Implementation of our approach is designed in two parts: static analysis and dynamic analysis. Finding version differences and computing backward slice are lied in the static analysis section. Slicing is implemented in WALA. For each application, WALA builds the corresponding call graph. Next it computes backward slice using this call graph and the failure manifestation point as a seed statement. For compatibility with our needs, we have modified some parts of WALA source code.

Dynamic profiling in our approach is implemented by Shrike which is a third-party library for instrumenting Java byte-code which is connected to WALA. It instruments the output of the static analysis by using the predefined instrumenting schema. Due to flexibility of instrumentation, we can easily exclude instrumenting of some parts of the code which is not necessary, e.g. Java libraries.

Table I
OVERVIEW OF THE TEST CASES USED IN THIS WORK

Bug issue	Broken by issue	Failing test case
HDFS-3856	HADOOP-8689	TestHDFSServerPorts
HDFS-4887	HDFS-4840	TestNNThroughputBenchmark
HDFS-4282	HADOOP-9103	TestEditLog.testFuzzSequences
Yarn-960	Yarn-701	TestBinaryTokens, TestMRCredentials

III. EXPERIMENTAL EVALUATION

Our evaluation tries to answer the following research questions:

RQ1. How accurate is our approach to locate failure-inducing changes?

Here we want to evaluate whether the approach can find the true defect location (sensitivity), and how many false positives are contained in the final report (specificity). We show in Section III-A the results for 4 test cases used in this study (Table I). They demonstrate that in 2 of our 4 test cases we could find the correct root cause of the failure without false positives. Analysis of another case indicates that by small extensions of our method the defect location could be narrowed to 20 LOC. We also discuss the real bug fixes of each issue as stated in the Hadoop bug log.

RQ2. Are there any alternatives to our approach which are simpler yet have comparable accuracy?

This question targets the practicability of implementing our approach, and tries to answer whether a more simple variant of the method could yield similar results. Brief analysis in Section III-B indicates that all of the steps shown in Algorithm 1 are necessary to achieve this level of specificity.

RQ3. What is the time overhead of our approach, and how does it compare to alternatives?

We have collected for each of the test cases execution times and code size in various phases of our approach (Section III-C). This data is used to evaluate the overhead by two performance metrics, runtime slowdown and size overhead (Table III) and to compare them against alternative approaches. We also show the total time of our method (Table IV).

A. Experiment setup and case studies

All our experiments were run on a 2.9 GHz Intel Dual Core laptop with 8 GB physical memory (4 GB was allocated for the JVM), running Linux.

In this section we try to answer question RQ1. To this aim we use the Apache Hadoop as a real-world, complex and large-scale project. It deploys test cases and frequent versioning which fits our requirements. We use *real* bugs from Hadoop issue tracking[2] system between 15th August 2012 and 27th July 2013. The selection of defects was done according to the following criteria:

1) The failure should be caused by a Hadoop component and manifest via a Hadoop test case. In other words, we do not consider library issues or other artifacts.

[2]http://hadoop.apache.org/issue_tracking.html

In class: org.apache.hadoop.hdfs.server.namenode.NameNode

```
@@ -511,9 +511,7 @@ public class NameNode {
}
private void startTrashEmptier(Configuration conf) throws
IOException {
-  long trashInterval = conf.getLong(
-      CommonConfigurationKeys.FS_TRASH_INTERVAL_KEY,
-      CommonConfigurationKeys.FS_TRASH_INTERVAL_DEFAULT);
+  long trashInterval=namesystem.getServerDefaults()
+                        .getTrashInterval();
   if (trashInterval == 0) {
      return;
   } else if (trashInterval < 0) {
  ...
```

Figure 2. Excerpt of code changes for issue Hadoop-8689 (simplified)

2) The bug should be well documented, and it should be clear which update or patch caused the test to fail. We require this information to validate the result of our approach.
3) There should exist a passing version, i.e. a (previous) version which passed the test which failed in a subsequent (failed) version.

1) Issue HDFS-3856: The first issue is an attempted solution to a new feature request HADOOP-8689. We explain it in more detail as a showcase for the approach. Figure 2 shows a subset of changes making up the (erroneous) solution. A part of this patch provides separate trash cleanup intervals (`fs.trash.interval`) for client side versus the server side. However, this change causes the test `TestHDFSServerPorts` to fail due to new call `getServerDefaults()` in the `startTrashEmptier()` function[3]. The failure of this test is reported in Hadoop bug HDFS-3856.

In Step 1 of our approach (Algorithm 1) we retrieve a passing v_p as well as a failing v_f code version (passing or failing for test `TestHDFSServerPorts`). As shown in Table II, the difference set Dif between v_p and v_f is very large, amounting to more than 109 kLOC.

Step 2 needs the failure stack trace of the test `TestHDFSServerPorts` to identify the failure site. As shown in Figure 1, code location `FSNamesystem.java:759` is the assumed failure site.

In Steps 3 and 4 we obtain the respective backward slices for v_p and v_f (922 and 759 JVM-bytecode statements), instrument the intersections IS_p and IS_f of Dif and backward slices (544 and 445 statements). In the subsequent Step 5 we obtain the code coverage of executing `TestHDFSServerPorts` on each of the two instrumented versions $v_{p,inst}$ and $v_{f,inst}$ (sizes 189 and 166 statements).

Finally, we can apply the filtering lemmas in Step 6. They yield the final report which is shown in Figure 3. In the future, we will present to the developer not only this report but also the differences of code coverage to support failure understanding.

[3]This explanation is taken from the comments on issue HDFS-3856 (Hadoop bug repository).

Suspicious failure-inducing changes:

In class: org.apache.hadoop.hdfs.server.namenode.NameNode:514

```
long trashInterval =
   namesystem.getServerDefaults().getTrashInterval();
```

Figure 3. Final report for using our approach for Hadoop-3856

We also compared our suspect against the fix of this problem, which is committed in SVN revision 1377934. We found that indeed our hypothesis was correct. In the fix, the previously added faulty change is replaced with a new statement (shown in bold):

```
@@ -511,13 +511,13 @@
public class NameNode {
}
private void startTrashEmptier(Configuration conf)
             throws IOException {
-  long trashInterval = namesystem.getServerDefaults()
-                         .getTrashInterval();
+  long trashInterval = conf.getLong(FS_TRASH_INTERVAL_KEY
+                         , FS_TRASH_INTERVAL_DEFAULT);
   if (trashInterval == 0) {
   ...
```

2) Issue HDFS-4887: The following case shows a limitation of our method but simultaneously points a way to extend it.

The considered issue is a failure in `TestNNThroughputBenchmark` test. It is reported that a fix to bug HDFS-4840 is responsible for the failure. The patch for fixing HDFS-4840 has introduced the following new (faulty) code to the `BlockManager` class in HDFS package of Hadoop. The defect here comes from stopping the `ReplicationMonitor` while `NameNode` is still running.

```
if (!namesystem.isRunning()) {
  LOG.info("Stopping ReplicationMonitor.");
  if (!(t instanceof InterruptedException))
    LOG.info("ReplicationMonitor received an exception"
                 + " while shutting down.", t);
   break;
}
LOG.fatal("ReplicationMonitor thread received
                 Runtime exception. ", t);
terminate(1, t);
```

Our approach applied to this case gives (after the filtering lemmas) an empty list of suspects, which shows a limitation of our method. The difficulty is caused by the small size of the set of instrumented statements (9 statements per version). Consequently, the coverage profiles have only 2 statements each (see Table II, middle rows). Both coverage profiles Cov_p and Cov_f contain the same statements, namely a `while`-statement (line 3092 of `BlockManager`-class) and `thread.sleep()`-statement (line 3096 of same class).

However, after investigating additional information provided by instrumentation (not used in this work) we discovered that the values of the conditional expression in the `while`-statement are different in $v_{p,inst}$ and $v_{f,inst}$. It is an easy extension of the current approach to consider such information. By extending the filtering lemmas we can then include in the final report the conditional statements with diverging condition values.

Even assuming that the `while`-statement would be included in the final report, it is not the precise root cause of the failure. However, this statement is within the method `run()` in the inner class of `ReplicationMonitor` in the `BlockManager` class. The method `run()` (about 20 LOC) indeed contains the defect. Thus, by small extension we can at least indicate the method with the actual defect.

An investigation of the actual fix which is committed in SVN revision 1501841 shows that (because the defect is outside the changes), merely deleting the added changes from the new version does not solve the problem. In fact, to fix this bug, a check statement (shown in bold in following) was added to the conditional branch in the code:

```
@@ -3129,6 +3138,9 @@
if (!namesystem.isRunning()) {
...
  break;
} else if (!checkNSRunning &&
                      t instanceof InterruptedException) {
  LOG.info("Stopping ReplicationMonitor for testing.");
  break;
}
LOG.fatal("ReplicationMonitor thread received
                      Runtime exception. ", t);
terminate(1, t);
```

3) Issue HDFS-4282: Bug issue HDFS-4282 reports the failing of `TestEditLog.testFuzzSequence` test. Comments in the issue show that this test is broken by HADOOP-9103. Due to errors in decoding Unicode characters, in HADOOP-9103 a patch is created which has modified code of UTF8 class in Hadoop Common project. However, this patch caused a failure of `TESTEditLog` test.

After applying our approach to this test case, the results point that the changes created in HADOOP-9103 are not faulty. However, the only difference in the execution profiles of passing and failing runs is a call of the method `toString()` in UTF8 class of the `io` package of Hadoop Common project. Also here Table II (bottom rows) give the sizes of code over all phases of our approach.

By investigating the execution profiles, we created a hypothesis that a solution to this bug is to add methods related to the `toString()` function. The real fix to this bug (committed in SVN revision 1418214) confirms our hypothesis. A new method `toStringChecked()` (with `IOException`) is added to the `UTF8` class which now can throw an `IOException` for invalid UTF8 characters[4].

4) Issue Yarn-960: Bug reported in issue Yarn-960 manifests in failure of tests `TestbinaryTokens` and `TestMRCredentials`. The reported reasons are changes committed for issue Yarn-701. Unfortunately, when applying our approach to this bug, we can not find the root cause of the error.

As we mentioned in the Section II (Step 2 in Algorithm 1), our approach needs a failure manifestation site in the failing version. However, in the stack trace of the failing executions we can find only code locations in the third-party java libraries and the Junit framework. In our current experimental setting we cannot analyze these libraries (see the scenario description and assumptions in Section III-A). However, our approach fails in this case not due to a fundamental limitation but due to a (current) technical constraint that third-party artifacts like java libraries cannot be analyzed.

[4]https://issues.apache.org/jira/browse/HDFS-4282

Table II
CODE SIZE (#LOC OR #JVM-BYTECODE STATEMENTS) IN DIFFERENT PHASES OF OUR APPROACH; Dif = DIFFERENCE SET (IN #LOC), $BSlice$ = BACKWARD SLICE, IS = INTERSECTION SET, Cov= COVERAGE PROFILE, REPORT = FINAL REPORT (IN #LOC)

Issue	Version	Size				
		Dif (#LOC)	$BSlice$	IS	Cov	Report (#LOC)
HDFS-3856	passing	109207	922	544	189	1
	failing		759	445	166	
HDFS-4887	passing	1030	9	2	2	0
	failing		9	2	2	
HDFS-4282	passing	1325	795	367	88	1
	failing		800	372	89	

B. Complexity of the approach

RQ2 can be partially answered by inspecting Table II. It shows the size of intermediate and final results in LOC (for Dif and Final Report) or JVM-bytecode statements (1-3 such statements typically correspond to 1 LOC). Note that in Dif a replaced line is counted twice - as an added and a removed line.

As an alternative to the Algorithm 1, we could have used the intermediate results for producing the final report. Specifically, this report could be based only on the code changes Dif, or only on the backward slice $BSlice$, or on the intersection set IS, or on the coverage profiles Cov. For issues HDFS-3856 and HDFS-4282 executing all steps is the right way to achieve high specificity. Judging by these cases, our approach cannot be simplified.

However, for issue HDFS-4887, using any of the $BSlice$, IS, or Cov is feasible, and could have led to pointing to the vicinity of the true defect (see Section III-A2). This indicates that we could consider a dynamic workflow, where a result of an intermediate step is used directly for the final report, if its size is below a certain threshold. This is subject to future work.

C. Performance evaluation

To answer RQ3, we first evaluate Table III. In a pair X / Y, X represents the *run-time slowdown*, i.e. ratio of time to execute the instrumented version divided by time to execute the non-instrumented version. Furthermore, Y is the size overhead of instrumenting, i.e. size of the instrumented version divided by size of the original version.

The time and size overheads of the fully instrumented code (Full instr.) are significant. Code size increases by factor 4-5, and execution time up to factor 3.3. Thus, full instrumentation is not efficient. However, instrumenting only the code in the backward slice ($BSlice$ instr.) produces acceptable overheads.

Table III
OVERHEADS OF OUR APPROACH COMPARED TO FULL INSTRUMENTATION (FULL INSTR.) AND INSTRUMENTING ONLY THE CODE IN $BSlice$ ($BSlice$ INSTR.); IN "X/Y", X IS THE RUN-TIME SLOWDOWN (A FACTOR) AND Y SIZE OVERHEAD OF INSTRUMENTING (A FACTOR); P = PASSING VERSION, F = FAILING VERSION, "-" = INSTRUMENTATION NOT POSSIBLE

Issue	Ver.	Original version		Full instr.	$BSlice$ instr.	Our approach
		Run time (s)	Size (MB)			
HDFS-3856	p	6	4.8	- / 4.60	1.20 / 1.04	1.00 / 1.00
	f	6	4.1	- / 5.40	1.00 / 1.02	1.00 / 1.00
HDFS-4887	p	9	4.8	1.60 / 4.60	1.00 / 1.00	1.00 / 1.00
	f	9	4.8	1.60 / 4.60	1.10 / 1.00	1.10 / 1.00
HDFS-4282	p	30	4.4	3.30 / 4.60	1.10 / 1.04	1.03 / 1.02
	f	30	4.4	3.00 / 4.60	1.03 / 1.04	1.00 / 1.02

Table IV
RUNNING TIMES OF A FAILING TEST AND TIMES FOR VARIOUS PHASES OF OUR APPROACH (TIMES IN SECONDS); TOTAL / TEST IS THE RATIO OF TOTAL APPROACH TIME TO TEST TIME

Issue	**App. time for passing + failing version**				**Test time**	**Total / Test**
	Slicing	Instr.	Run	**Total**		
HDFS-3856	4	4	12	20	6	3.3
HDFS-4887	2	2	18	22	9	2.4
HDFS-4282	4	4	64	72	30	2.4

Even so, our approach beats the alternatives, having negligible overheads due to instrumentation.

Even more interesting is Table IV which contrasts the running time of the failed test (Test time) against the total time needed to execute our approach (Total). As shown in the column Total / Test, the total time of our approach requires at most 3.3 times the duration of the failed test, and the latter is just one of many tests executed within a test suite.

D. Threats to Validity

Like in any empirical study, there are some threats to validity in our evaluation. First, we use only four bugs. This is not a sufficient sample size to draw general conclusions, especially in respect to applications. Note that the reason of using a small set of bugs in this study is that finding the real bugs and reproducing them is a time-consuming job.

Secondly, we focus on finding failure-inducing changes which are caused by changes in the faulty version v_f. However, it is be possible that the actual bug has been introduced already in an older version (v_p or before) but manifests only with changes yielding v_f.

IV. RELATED WORK

Recently automated debugging has attracted a great deal of interest. Here, we focus on works which are closely related to our approach.

With increasing size of software, debugging software versions before committing them to the repository is an important step. Approaches in [14] and [2] generate an alternative input (which differs from failing input in the control flow behavior) and then compare their executions to find the root cause of the failure. [2] combines slicing and symbolic execution to locate the faulty code in a modified version of a program. However, these approaches incur large overheads.

One other interesting approach closely related to our work is delta debugging introduced in [20] and extended in [21]. These methods have significant limitations, namely high false positive ratio and large amount of tests. This encouraged others to improve on delta debugging, see [12] and [19]. Delta debugging differs from our version comparison as the former narrows down the search space (to locate suspicious changes) gradually, by applying changes to the application iteratively. This requires lot of test repetitions, creating large time overhead.

Program slicing is another debugging technique which is introduced in [18]. Due to large size of the slice, different approaches ([11], [7], [23], [24]) have been proposed to reduce the size of slice efficiently. Despite of these improvements, the sizes of slices are still fairly large in real-world applications. To address this issue, thin slicing has proposed in [16]. We use this technique for computing the backward slice in our approach (see Section II-B).

In [5], program slicing is used in combination with delta debugging to narrow down the search space of failure. While [5] is focused on the changes in the input, our approach concentrates on the changes in the source code.

Another approach in debugging is spectrum-based techniques which collect the execution information of faulty and correct runs and compare them to find the root cause of the failure [6], [4]. This technique differs from our approach in that we use just one failing and one passing run.

Statistical techniques are used in [9], [10] and [3] for bug isolation. Statistical debugging analyzes a large amount of executions gathered from running the instrumented application to rank the suspicious predicates which are highly relevant to the bug. Contrary to our approach, the requirements on a large amount of executions create significant time and resource overheads.

V. CONCLUSIONS AND FUTURE WORK

We have presented a scalable approach to isolating failure-inducing changes which exploits version differences together with static and dynamic code analysis. Our method has two essential strengths. First, the size of the set of suspicious statements is proportional to the size of recent code changes, making it potentially applicable to very large projects. Secondly, the additional runtime overhead is on the order of executing a test triggering bug search. This allows for integrating our approach in a traditional testing process in order to enhance test outcomes with locations of potential defects. Our preliminary evaluation on a large-scale project (Apache Hadoop) shows that results are promising, and the approach could locate a part of the true defects with high accuracy.

Our approach has also some limitations. The most important one is the property that we can isolate only these defects which cause a failure of a software test. As pointed out by an anonymous reviewer, this ignores the fact that some defects remain latent for long periods, until the are discovered by end users.

In our future work, we will target the following challenges.
Extended evaluation. We will evaluate the approach on more bugs (also artificial defects) and other applications.
More runtime facts. Dynamic analysis will consider not only code coverage, but also other runtime information, like value of conditional expressions.
Adaptivity. We will experiment with modifying the steps of our method depending on the sizes of intermediate results.
Supporting failure understanding. We will study whether contrasting runtime data collected during the passing and failing run can help the programmer to understand the causes of the failure.
Enhancing tools for continuous integration. As a long-term project, we plan to integrate these methods in wide-spread testing tools like Jenkins to support wide-spread adoption of these techniques.

Acknowledgments

We would like to thank the members of PVS group of Heidelberg University, Felix Langner and Lutz Büch for their useful suggestions. This work is supported in part by the grant AN 405/2-1 entitled *Automated, minimal-invasive Identification and Elimination of Defects in Complex Software Systems* financed by the Deutsche Forschungsgemeinschaft (DFG).

References

[1] P. Arumuga Nainar and B. Liblit. Adaptive bug isolation. In *ICSE*, 2010.

[2] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang. Golden implementation driven software debugging. In *FSE*, 2010.

[3] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, 2009.

[4] W. Eric Wong, V. Debroy, and B. Choi. A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw.*, 83(2):188–208, Feb. 2010.

[5] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE*, 2005.

[6] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[7] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, Oct. 1988.

[8] B. Liblit. *Cooperative Bug Isolation: Winning Thesis of the 2005 ACM Doctoral Dissertation Competition*, volume 4440 of *Lecture Notes in Computer Science*. Springer, 2007.

[9] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[10] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[11] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *International Conference on Computers and Applications*, pages 877–882, 1987.

[12] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *ICSE*, 2006.

[13] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011.

[14] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/FSE*, 2009.

[15] J. Roessler, G. Fraser, A. Zeller, and A. Orso. Isolating failure causes through test case generation. In *ISSTA*, 2012.

[16] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.

[17] WALA. http://sourceforge.net/projects/wala/.

[18] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, 1984.

[19] K. Yu, M. Lin, J. Chen, and X. Zhang. Practical isolation of failure-inducing changes for debugging regression faults. In *ASE*, ASE 2012, 2012.

[20] A. Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, 1999.

[21] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.

[22] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[23] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, 2006.

[24] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, 2003.