Scalable Isolation of Failure-Inducing Changes via Version Comparison

Mohammadreza Ghanavati, Artur Andrzejak, Zhen Dong

Heidelberg University
artur@uni-hd.de
http://pvs.ifi.uni-heidelberg.de/
Debugging in Large-Scale Software

Root Cause

- Manual
- Tedious
- Time Consuming
- Expensive

Bug Symptom

Automated Debugging
Selected Works on Automated Debugging

Automated Debugging

Slicing-based Debugging
- Static Slicing (81)
- Dynamic slicing (88, 93)
- Whyline (08)

Statistical Debugging
- Tarantula (01)
- CBI (03)
- Holmes (08)

Delta Debugging
- DD (99, 01)
- HDD (05)

Spectrum-based Debugging
- Renieris (03)
- Wong (10)

Formula-based Debugging
- Darwin (09)
- Error Invariants (11)
More Works on Automated Debugging

Slicing-based Debugging
- Static Slicing (81)
- Dynamic slicing (88, 93)
- Whyline (08)
  - Wong: crosstab analysis-based method (08)
  - Holmes (08)

Statistical Debugging
- Tarantula (01)
- CBI (03)
  - Wong: crosstab analysis-based method (08)

Delta Debugging
- DD (99,01)
  - Renieris (03)

Spectrum-based Debugging
- HDD (05)
  - Wong (10)

Machine Learning-based Methods
- Brun: for Latent Code Errors (04)
  - Briand: Decision tree algorithm (07)
  - Wong: back-propagation neural network (09)
    - Error Invariants (11)

Formula-based Debugging
- Darwin (09)
  - BugAssist (11)
    - Nesse: N-grams (09)
  - Error Invariants (11)

Data-Mining-based Methods
- Cellier: FCA (08)

Model-based Methods
- DeMillo: modeling failure (97)

Sources:
1. A. Orso: **Automated Debugging: Are We There Yet?**, Dagstuhl Seminar Feb. 2013
Challenges of Automated Debugging

• In most AD approaches, result is a (ranked) set of suspicious statements to be examined by the developer.

• “Usability” problems reported in Parnin and Orso (ISSTA ’11):
  1. “Specificity” is reported as a relative fraction of total LOC.
  2. No further help with bug understanding (no explanations).

• For (1): suspect set is typically ~ 0.1 - 1% of total LOC.
  – OK for small projects: 1k LOC => 10 LOC.
  – Infeasible for large ones: 1 mio LOC => 10k LOC !!!

• Can we keep the set of suspects small and independent of project size?
Exploiting SW Development Processes

• Development process in large SW projects:
  – Incremental SW development with many intermediate versions (subjected to tests)

• Observation: even in large projects, fraction of code changed between commits is roughly constant
  – Assuming frequent & regular commits
  – Ignore topic merges (i.e. a feature branch merged into main branch)
Exploiting SW Development Processes

• Idea: investigate only the changes of the code
  – “Constant” size of suspect set
  – Behavior of previous version provides “prior knowledge” – source of additional information

• Drawback: not all defects can be discovered
  – But: better a practical technique for some bugs than an impractical one for all
Successful at: Memory Leak Isolation

**Identical** (but arbitrary) workload (unit / integ. tests etc.)

Software version $i$  \[\rightleftharpoons\]  Software version $i+1$

- $\text{new}_1$: allocated \[\rightarrow\] released
- $\text{new}_k$: allocated \[\rightarrow\] released
- $\text{new}_n$: allocated \[\rightarrow\] released

Problem!

Felix Langner, Artur Andrzejak: Detection and Root Cause Analysis of Memory-Related Software Aging Defects by Automated Tests, MASCOTS 2013
Overview of the Approach

1: Find the differences of versions
2: Retrieve failure site from the stack trace
3: Compute backward slices for each version
4: Compute the intersections of steps 2 & 3
5: Instrument the intersection
6: Execute failed test on each instrumented version obtained from step 5 to get code coverage profiles
7: Get list of suspects with filtering

Suspect Report
Difference set

• Assume 2 versions:
  – Version i as passing version
  – Version i+1 as failing version

• Using version history tools like SVN, GIT or diff command:

1. Code version i
2. Code version i+1
3. Diff tool
4. Difference Set
Stack Trace Analysis

- Stack trace: resulted from running test case on the failing version
- **Failure site** = topmost non-exception-handling entry of the stack
- We compute the **backward slice** for both failing and passing version (with failure site as **seed** statement)

\[
S^B(B) = \{ A | A \rightarrow \ast B \}
\]

A fragment of stack trace

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

```
\[ ... \]
```

\[ ... \]

\[ B \]
Intersecting & Instrumenting

- **Execution of test case T in version i**
- **Execution of test case T in version i+1**
- **Compute backward slice from failure site and intersect with changes**

**Source code**

- **Added changes**
- **Deleted changes**

**Failure site**

**Backward slice**

**Instrument**

**Instrument only:**
- **Added intersected changes (+) in version i+1**
- **Deleted intersected changes (-) in i**
Getting Code Coverage

- **Execute & get code coverage**: re-execution of the test case T (failing for version i+1) on both of instrumented versions
  - Get coverage profile for version i
  - Get coverage profile for version i+1
Comparing the Coverage Information

• The following statements are included in the set of suspects:
  – **All statements added** to or changed in failing version \( i+1 \) which are in the failing coverage profile
  – **All statements deleted** from passing version \( i \) which are in the passing coverage profile

**Final result:** The resulting list of suspicious statements with the locations
Implementation & Evaluation Setup

• Implementation:
  – Using WALA tool for static analysis
    • Building call graph
    • Computing backward slice using thin slicing*
  – Using UNIX diff command for finding differences
  – Using Shrike library for instrumenting (from WALA)

• Application: Apache Hadoop, release 2.0.3-alpha
• Test cases: 4 real bugs from Hadoop bug repository

*Manu Sridharan, Stephen J. Fink, Ratislav Bodik, Thin slicing, in PLDI 2007
### Real Test Cases From Apache Hadoop

<table>
<thead>
<tr>
<th>Bug issue</th>
<th>Broken by Issue</th>
<th>Failing Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS-3856</td>
<td>HADOOP-8689</td>
<td>TestHDFSServerPorts</td>
</tr>
<tr>
<td>HDFS-4887</td>
<td>HDFS-4840</td>
<td>TestNNNThroughputBenchmark</td>
</tr>
<tr>
<td>HDFS-4282</td>
<td>HADOOP-9103</td>
<td>TestEditLog.testFuzzSequences</td>
</tr>
<tr>
<td>Yarn-960</td>
<td>Yarn-701</td>
<td>TestBinaryTokens, TestMRCredentials</td>
</tr>
</tbody>
</table>

We use real bugs from Hadoop issue tracking system between 15\textsuperscript{th} August 2012 and 27\textsuperscript{th} July 2013.

Test Case Requirements

• No consideration of third-party libraries

• Well documented bugs for result validation

• Existence of a passing version for the failing test

Note: test case Yarn-960 did not satisfy first requirement: the failure stack trace contains only third party libraries. We do not consider this test case (no fundamental limitation).
In 2 out of 3 cases no false positives (high specificity)
All of the steps in our approach are needed to reach high specificity
Empty suspect list for one bug, but at least our approach can indicate
the method related to the defect
## Performance Evaluation (1):
### Overheads Comparison

<table>
<thead>
<tr>
<th>Issue</th>
<th>Version</th>
<th>Original version</th>
<th>Full inst.</th>
<th>Bslice inst.</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Run time (s)</td>
<td>Size (MB)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDFS-3856</td>
<td>Passing</td>
<td>6</td>
<td>4.8</td>
<td>- / 4.60</td>
<td>1.20 / 1.04</td>
</tr>
<tr>
<td></td>
<td>Failing</td>
<td>6</td>
<td>4.1</td>
<td>- / 5.40</td>
<td>1.00 / 1.02</td>
</tr>
<tr>
<td>HDFS-4887</td>
<td>Passing</td>
<td>9</td>
<td>4.8</td>
<td>1.60 / 4.60</td>
<td>1.00 / 1.00</td>
</tr>
<tr>
<td></td>
<td>Failing</td>
<td>9</td>
<td>4.8</td>
<td>1.60 / 4.60</td>
<td>1.10 / 1.00</td>
</tr>
<tr>
<td>HDFS-4282</td>
<td>Passing</td>
<td>30</td>
<td>4.4</td>
<td>3.30 / 4.60</td>
<td>1.10 / 1.04</td>
</tr>
<tr>
<td></td>
<td>Failing</td>
<td>30</td>
<td>4.4</td>
<td>3.00 / 4.60</td>
<td>1.03 / 1.04</td>
</tr>
</tbody>
</table>

Our approach has negligible instrumentation overheads
The total time of our approach requires at most 3.3 times the duration of the failed test.
Drawbacks of Our Approach

• Does not work if the actual bug has been introduced already in an older version (or before) but manifests only with recent changes (yielding version $i+1$)

• Evaluation on only four bugs: insufficient sample size to draw general conclusions
Future Work

• **Extended evaluation.** Evaluation of the approach on more bugs (also artificial defects) and other applications

• **More runtime facts.** Extension of dynamic analysis with other runtime information, like value of conditional expressions

• **Adaptivity.** Modification of the steps of our method depending on the sizes of intermediate results
Future Work

• **Supporting failure understanding.** Study of runtime data collection during the passing and failing to improve bug understanding

• **Enhancing tools for continuous integration.** Integration of our methods in wide-spread testing tools like Jenkins to support wide-spread adoption
Summary

• **Objective:** A scalable approach to isolating failure-inducing changes

• **Idea:** Exploiting version differences together with static and dynamic code analysis

• **Strength of the approach:**
  – Suspect set is proportional to the size of recent code changes
  – Runtime overhead is in the order of executing a test triggering bug search

• **Evaluation:**
  – On a large-scale project (*Apache Hadoop*) and real bugs
  – Promising result with high accuracy *(near to zero false positives)*, low complexity, low performance degradation

• **Limitations:**
  – Just crashing errors, no latent errors

Applicable to very large projects
Integration into existing testing processes feasible
Thank you.

QUESTIONS ARE WELCOME!
Additional Slides
Version Comparison: Main Idea

- The main idea is a **differential approach**: compare consecutive development versions.
Detection of Failure-Inducing Changes

- Code version i
- Code version i+1
- Versions differences
- Stack trace analysis
- Intersect
- Instrument
- Execution
- Filtering
- Report
## Complexity Evaluation

### Code Size

<table>
<thead>
<tr>
<th>Issue</th>
<th>Version</th>
<th>Dif (#LOC)</th>
<th>Bslicc</th>
<th>IS</th>
<th>Cov</th>
<th>Report (#LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS-3856</td>
<td>Passing</td>
<td>109207</td>
<td>922</td>
<td>544</td>
<td>189</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Failing</td>
<td>759</td>
<td>445</td>
<td>166</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDFS-4887</td>
<td>Passing</td>
<td>1030</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Failing</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDFS-4282</td>
<td>Passing</td>
<td>1325</td>
<td>795</td>
<td>367</td>
<td>88</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Failing</td>
<td>800</td>
<td>372</td>
<td>89</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- In 2 out of 3 cases no false positives (specificity).
- All of the steps in our approach is necessary to reach this level of specificity.
- Empty suspect list for one bug. But at least our approach can indicate failure-inducing change method.
Performance Evaluation (2)
Comparison of Running Times

<table>
<thead>
<tr>
<th>Issue</th>
<th>App. time for passing + failing version</th>
<th>Test time</th>
<th>Total / Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slicing</td>
<td>Inst.</td>
<td>Run</td>
</tr>
<tr>
<td>HDFS-3856</td>
<td>4</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>HDFS-4887</td>
<td>2</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>HDFS-4282</td>
<td>4</td>
<td>4</td>
<td>64</td>
</tr>
</tbody>
</table>

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.