Fault Localization Using Value Replacement

Dennis Jeffrey
Neelam Gupta
Rajiv Gupta

ISSTA '08

Presented by Amy Siu 12/9/08
What is fault localization?

- Software building is a human-intensive process
  - Prone to error

- Software debugging consists of two phases
  - Locating the error
  - Fixing the error
Why is fault localization difficult?

- The point of failure may appear anywhere after the faulty code statement
- The faulty code statement is not always obvious

- **Manual** inspection requires
  - Human effort
  - Code familiarity
  - Domain knowledge
- **Automatic** localization is still an open research problem
  - Computationally intensive
Problem statement

- Investigate an alternative method to localize faulty code statements
  - Automated
  - Computationally less intensive
  - Able to locate faulty code statement even if the point of failure occurs after that statement
State of the art - dynamic analysis

- **Dynamic program slicing**
  - Computationally expensive and not scalable

- **Delta debugging**
  - Cause effect chains have less granularity

- **Nearest neighbor**
  - Poorer performance compared to the rest of the state-of-the-art

- **Statistical technique**
  - Most similar approach to proposed technique
State of the art - *Tarantula*

- *Tarantula* by Jones and Harrold '05 is the closest statistical technique in the state-of-the-art to the proposed technique

- **Baseline** for this paper

- Evaluated over the same Siemens benchmark suite

How does *Tarantula* work?

- **Intuition:** statements executed primarily by failing runs are more likely to be faulty

- **Keep track** of statements in successful and failing runs

- **Rank** statements based on statistics
Definitions

- **Value mapping**
  - Variables: concrete value, e.g. \( x = 10 \)
  - Predicate statements: branch outcome, e.g. “else” branch

- **IVMP** (Interesting Value Mapping Pair)
  - A pair of value mappings
  - **Original** value mapping exists in failing run with **wrong output**
  - **Alternate** value mapping causes the output to become **correct**

- **Value profile**
  - All value mappings for a program
  - Each mapping may contribute to original, alternate, or both types of IVMPs
IVMP algorithm

Step 1: initialize value profile

Step 2: search for IVMPs

Running time = $O(t \times m)$

- $t = \#$ statements
- $m = \max. \#$ alt. mappings

Figure 1: General algorithm for computing IVMPs with respect to a failing run and its test suite.
IVMP Example 1

IVMP at a faulty statement

```
argc := ...;
1: if (argc < 3) /* 3 should actually be 4 */
2:   print ("Too few");
3: else
4:   print ("Okay");
```

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Input Values</th>
<th>Actual Output</th>
<th>Expected Output</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>argc = 2</td>
<td>Too few</td>
<td>Too few</td>
<td>PASS</td>
</tr>
<tr>
<td>B</td>
<td>argc = 3</td>
<td>Okay</td>
<td>Too few</td>
<td>FAIL</td>
</tr>
<tr>
<td>C</td>
<td>argc = 4</td>
<td>Okay</td>
<td>Okay</td>
<td>PASS</td>
</tr>
</tbody>
</table>

Figure 2: Code fragment and test suite based on schedule faulty version v9.
IVMP directly linked to a faulty statement

```c
AltLayVal := ...;
1:  Pos_RA_Alt_Thresh[0] = 400;
2:  Pos_RA_Alt_Thresh[1] = 550; /* Should be 500 */
3:  Pos_RA_Alt_Thresh[2] = 640;
4:  Pos_RA_Alt_Thresh[3] = 740;
... 
5:  if (Pos_RA_Alt_Thresh[AltLayVal] < 525)
6:     print (0);
7:  else
8:     print (1);
```

<table>
<thead>
<tr>
<th>Test Case</th>
<th>AltLayVal Values</th>
<th>Actual Output</th>
<th>Expected Output</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>AltLayVal = 0</td>
<td>0</td>
<td>0</td>
<td>PASS</td>
</tr>
<tr>
<td>B</td>
<td>AltLayVal = 1</td>
<td>1</td>
<td>0</td>
<td>FAIL</td>
</tr>
<tr>
<td>C</td>
<td>AltLayVal = 2</td>
<td>1</td>
<td>1</td>
<td>PASS</td>
</tr>
</tbody>
</table>

Figure 3: Code fragment and test suite based on tcas faulty version v7.
IVMP Example 3

IVMP in the presence of erroneously omitted statements

```c
int foo(int x, int y)
1: /* if (y < 0) return x; */
2: if (y == 0) return 0;
3: return x + 1;
```

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Input Values</th>
<th>Actual Output</th>
<th>Expected Output</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(x,y) = (1,-1)</td>
<td>2</td>
<td>1</td>
<td>FAIL</td>
</tr>
<tr>
<td>B</td>
<td>(x,y) = (2,2)</td>
<td>3</td>
<td>3</td>
<td>PASS</td>
</tr>
<tr>
<td>C</td>
<td>(x,y) = (0,1)</td>
<td>1</td>
<td>1</td>
<td>PASS</td>
</tr>
</tbody>
</table>

Figure 4: Code fragment and test suite inspired by schedule2 faulty version v1.
Dependence cause vs. Compensation cause

- **Dependence cause**
  - Different statements in the same *definition-use chain*
  - Applying IVMP to *any* statement corrects the error
  - But only *one* statement is the root cause

- **Compensation cause**
  - *Unrelated* statements
  - Applying IVMP to *any* statement also corrects the error
  - The paper does not further discuss details
Ranking statements using IVMPs

Line 2 is more likely to be faulty than lines 4 and 6
Suspiciousness

Proposed metric: **suspiciousness**

\[
suspiciousness(s) := \left| \{ f : f \in F \land s \in \text{STM}_{IVMP}(f) \} \right|
\]

Tie-breaker metric: suspiciousness as per **Tarantula**

\[
suspiciousness_{\text{tarantula}}(s) = \frac{\frac{\text{failed}(s)}{\text{totalFailed}}}{\frac{\text{passed}(s)}{\text{totalPassed}}} + \frac{\text{failed}(s)}{\text{totalFailed}}
\]
Ordering failing runs algorithm

Step 1: find IVMPs

Step 2: use IVMPs to rank statements

# Re-executions = O(f*t*m)
- f = # failing runs
- t = # statements
- m = max. # alt. mappings

But can limit statement instances and alternative mappings – use shortest failing runs first
Summary of proposed technique

1. Gather successful and failing runs
2. Establish value profile
3. Search for IVMPs
4. Rank statements using suspiciousness
5. Break ties with Tarantula's suspiciousness
Experiment 1 – implementation

**Valgrind** infrastructure
- Synthetic, simulated CPU
- Machine-level instructions
- Value mappings manipulated at the machine instruction level

**Machine profile**
- Dell PowerEdge 1900 server
- 2 Intel Xeon quad-core processors at 3.00GHz
- 16 GB RAM
- No parallel processing
Experiment 1 – subject programs

**Siemens benchmark** suite
- All faults are seeded
- At least 5 successful and 5 failing runs

<table>
<thead>
<tr>
<th>Prog. Name</th>
<th>LOC</th>
<th># Ver.</th>
<th>Avg. Suite (Pool) Sizes</th>
<th>Program Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>138</td>
<td>41</td>
<td>17 (1608)</td>
<td>altitude separation</td>
</tr>
<tr>
<td>totinfo</td>
<td>346</td>
<td>23</td>
<td>15 (1052)</td>
<td>statistic computation</td>
</tr>
<tr>
<td>sched</td>
<td>299</td>
<td>9</td>
<td>20 (2650)</td>
<td>priority scheduler</td>
</tr>
<tr>
<td>sched2</td>
<td>297</td>
<td>9</td>
<td>17 (2710)</td>
<td>priority scheduler</td>
</tr>
<tr>
<td>ptok</td>
<td>402</td>
<td>7</td>
<td>17 (4130)</td>
<td>lexical analyzer</td>
</tr>
<tr>
<td>ptok2</td>
<td>483</td>
<td>9</td>
<td>23 (4115)</td>
<td>lexical analyzer</td>
</tr>
<tr>
<td>replace</td>
<td>516</td>
<td>31</td>
<td>29 (5542)</td>
<td>pattern substituter</td>
</tr>
</tbody>
</table>

Table 1: The Siemens benchmark programs. From left to right: program name, # lines of code, # faulty versions, average suite size (and test case pool size), and description of program functionality.
Experiment 1 – compared approaches

5 approaches compared in the experiment

- **IVMP**
- **Tarantula**
- **Tarantula-Pool** – use entire test case pool to get larger test suite
- **IVMP-1** – use only 1 failing run to search IVMPs with
- **IVMP-2** – use 2 failing runs to search IVMPs with

Assign a **score** to ranked statements

\[
\text{score}(S) = \frac{\text{totalStmtsEx} - r}{\text{totalStmtsEx}} \times 100%
\]

- Higher score ➔ more statements executed by failing runs are ignored before faulty statement is found
Results – effectiveness

- **IVMP** ranks faulty statements higher than *Tarantula*
- **Larger test suite pool** help rank faulty statement higher
- **More failing runs** help rank faulty statements higher
Results – efficiency (I)

- Compare variations within IVMP search algorithm
- **Reduced IVMP search** technique drastically reduces # re-executions
Results – efficiency (II)

- Almost 90% of faulty versions have all IVMPs searched under 100 minutes
- Maximum time of 840 minutes due to unusual case – long failing runs cannot limit IVMP search
Results – efficiency (III)

- **Size of value profile** increases logarithmically to test suite size
- **Unusual case** – difficult to pinpoint exact floating point values
Experiment 2 – larger programs

Second experiment ran on **5 larger subjects programs**

- Similar IVMP search time to experiment 1
- Search time depends on length of **shortest failing trace**, not program size
- Proposed technique is **scalable**
Discussion

**Scalability**
- Further enhance scalability by limiting IVMP search
- Combine other techniques such as program slicing

**Multiple simultaneous faults**
- Difficult to find IVMPs that influence each other, or have different effects on program output
- Diminishes effectiveness of proposed approach

**Address values**
- Ignored by proposed approach
Conclusions

Proposed IVMP approach is

- **More effective** than the best technique in the state-of-the-art, *Tarantula*
- **Scalable**
Limitations and future work – noted by authors

Limitations

- Only find faults that can be detected via value replacement
- Multiple simultaneous faults
- Address values

Future work

- (No explicit future work section in the paper)
- Combine proposed technique with program slicing to limit IVMP search
Limitations and future work – class discussion

Limitations

- Indirectly linked faulty statements
- Extraneous statements causing a fault – no example, unclear how that works
- “Fuzzy” values generate huge value profile *a la* floating point example
- Dependent on existing runs – both successful and failing ones – to generate rankings

Future work

- Adapt proposed technique in practical environment without machine instruction-level simulator
- Try new technique on even larger programs
- How to use proposed technique when there are no existing test runs to extract value profile from