Faster Mutation Analysis via Equivalence Modulo States

Bo Wang, Yingfei Xiong, Yangqingwei Shi,
Lu Zhang, Dan Hao

Peking University
July 12, 2017
Mutation Analysis

- Mutation analysis is a fundamental software analysis technique

  - **Mutation Testing** [DeMillo & Lipton, 1970]
  - Mutation-Based **Test Geneartion** [Fraser & Zeller, 2012]
  - Determining **Mutant Utility** [Just et al., 2017]
  - Mutation-based **Fault Localization** [Papadakis & Traon, 2012]
  - Generate-Validate **Program Repair** [Weimer et al., 2013]
  - Testing **Software Product Lines** [Devroey et al., 2014]
Scalability: A Key Limiting Issue

- The testing time of a single program is amplified $N$ times
  - $N$ is the number of mutants
  - $N$ can be usually large
  - $N$ is related to the size of the program

- Plain mutation analysis scales to only programs less than 10k lines of code
Redundant Computations

- Many computation steps in mutation analysis are equivalent
- Reusing them could possibly enhance scalability
Example

test:
p();
assert(...);

```
p():
1: a=x();
2: a=a/2;
3: y(a);
```
Existing work 1: Mutation Schemata [Untch, Offutt, Harrold, 1993]

- Procedures x() and y() are the same in the three mutants, but they are compile three times.
- Redundancy in Compilation
Existing work 1: Mutation Schemata [Untch, Offutt, Harrold, 1993]

- Generate one big program that compiles once
- Mutants are selected dynamically through input parameters
Existing work 2: Split-Stream Execution

[King, Offutt, 1991][Tokumoto et al., 2016][Gopinath, Jensen, Groce, 2016]

- The computations before the first mutated statement are redundant

```
0: a=x();
1: a=a-2;
2: y(a);

0: a=x();
1: a=a+2;
2: y(a);

0: a=x();
1: a=a*2;
2: y(a);
```
Existing work 2: Split-Stream Execution

- Start with one process
- Fork processes when mutated statements are encountered
Redundancy After the First Mutated Statement

1: a=x();
2: a=a-2;
3: y(a);

1: a=x();
2: a=a+2;
3: y(a);

1: a=x();
2: a=a*2;
3: y(a);
Our Contribution

• Equivalence Modulo States
  • Two statements are *equivalent modulo the current state* if executing them leads to the same state from the current state
  • Statements
    • \( a = a \times 2 \)
    • \( a = a + 2 \)
  • are equivalent modulo
    • State 2 where \( a == 2 \)
Mutation Analysis via Equivalence Modulo States

• Start with a process representing all mutants
• At each state, group next statements into equivalence classes modulo the current state
• Fork processes and execute each group in one process
Challenges

- **Objective:** Overheads $\ll$ Benefits
  - Challenge 1: How to efficiently determine equivalences between statements?
  - Challenge 2: How to efficiently fork executions?
  - Challenge 3: How to efficiently classify the mutants?
Challenge 1: Determine Statement Equivalence

- Performance trial executions of statements and record their changes to states
  - State: a==2
  - a=a+2 → {a → 4}
  - a=a*2 → {a → 4}

- Compare their changes to determine equivalence

- Does not work on statements making many changes
  - f(x, y), f(y, x)
Challenge 1: Determine Statement Equivalence

- Record abstract changes that can be efficiently compared
- Ensuring $c(s_1) \neq c(s_2) \implies a(s_1) \neq a(s_2)$
  - $s_1, s_2$: Statements
  - $c(s)$: Concrete changes made by $s$
  - $a(s)$: Abstract changes made by $s$

- Abstract changes of method call: values of arguments
  - State: $x = 2$, $y = 2$
  - $f(x, y) \implies <2,2>$
  - $f(y, x) \implies <2,2>$
Challenge 2: Fork Execution

• Memory: the POSIX system call “fork()”
  • Implements the copy-on-write mechanism
  • Integrated with POSIX virtual memory management

• Other resources: files, network accesses, databases
  • Solution 1: implement the copy-on-write mechanism
  • Solution 2: map them into memory
Experiments – Mutation Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOR</td>
<td>Replace arithmetic operator</td>
<td>$a + b \rightarrow a - b$</td>
</tr>
<tr>
<td>LOR</td>
<td>Replace logic operator</td>
<td>$a &amp; &amp; b \rightarrow a \mid b$</td>
</tr>
<tr>
<td>ROR</td>
<td>Replace relational operator</td>
<td>$a == b \rightarrow a &gt;= b$</td>
</tr>
<tr>
<td>LVR</td>
<td>Replace literal value</td>
<td>$T \rightarrow T + 1$</td>
</tr>
<tr>
<td>COR</td>
<td>Replace bit operator</td>
<td>$a &amp;&amp; b \rightarrow a \mid\mid b$</td>
</tr>
<tr>
<td>SOR</td>
<td>Replace shift operator</td>
<td>$a &gt;&gt; b \rightarrow a &lt;&lt; b$</td>
</tr>
<tr>
<td>STDC</td>
<td>Delete a call</td>
<td>$\text{foo}() \rightarrow \text{nop}$</td>
</tr>
<tr>
<td>STDS</td>
<td>Delete a store</td>
<td>$a = 5 \rightarrow \text{nop}$</td>
</tr>
<tr>
<td>UOI</td>
<td>Insert a unary operation</td>
<td>$b = a \rightarrow a + +; \ b = a$</td>
</tr>
<tr>
<td>ROV</td>
<td>Replace the operation value</td>
<td>$\text{foo}(a, b) \rightarrow \text{foo}(b, a)$</td>
</tr>
<tr>
<td>ABV</td>
<td>Take absolute value</td>
<td>$\text{foo}(a, b) \rightarrow \text{foo}(\text{abs}(a), b)$</td>
</tr>
</tbody>
</table>

- Defined on LLVM IR
- Mimicking Javalanche and Major
# Experiments - Dataset

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Tests</th>
<th>Mutants</th>
<th>Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>flex</td>
<td>10334</td>
<td>42</td>
<td>56916</td>
<td>5119</td>
</tr>
<tr>
<td>gzip</td>
<td>4331</td>
<td>214</td>
<td>37326</td>
<td>3058</td>
</tr>
<tr>
<td>grep</td>
<td>10102</td>
<td>75</td>
<td>58571</td>
<td>4373</td>
</tr>
<tr>
<td>printtokens</td>
<td>475</td>
<td>4130</td>
<td>1862</td>
<td>199</td>
</tr>
<tr>
<td>printtokens2</td>
<td>401</td>
<td>4115</td>
<td>2501</td>
<td>207</td>
</tr>
<tr>
<td>replace</td>
<td>512</td>
<td>5542</td>
<td>3000</td>
<td>220</td>
</tr>
<tr>
<td>schedule</td>
<td>292</td>
<td>2650</td>
<td>493</td>
<td>55</td>
</tr>
<tr>
<td>schedule2</td>
<td>297</td>
<td>2710</td>
<td>1077</td>
<td>121</td>
</tr>
<tr>
<td>tcas</td>
<td>135</td>
<td>1608</td>
<td>937</td>
<td>73</td>
</tr>
<tr>
<td>totinfo</td>
<td>346</td>
<td>1052</td>
<td>756</td>
<td>63</td>
</tr>
<tr>
<td>vim 7.4</td>
<td>477257</td>
<td>98</td>
<td>173683</td>
<td>14124</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>504482</strong></td>
<td><strong>20736</strong></td>
<td><strong>337122</strong></td>
<td><strong>27612</strong></td>
</tr>
</tbody>
</table>
Experiments - Results

2.56X speedup over SSE, and 8.95X speedup over MS
Experiments - Results

Our Approach
Split-Stream Execution
Mutation Schemata
Discussion: Why worked?

• Overheads: the overhead for each instruction is *small*
  • Not related to the size of the program, effectively $O(1)$

• Benefits: equivalences between statements modulo the current state are *common* in mutation analysis

  \[
  a \geq b \\
  a > b \Rightarrow a > b + 1 \\
  a > c \\
  c > b
  \]

• See paper for a detailed study on overheads/benefits
Discussion: Eliminating More Redundancies

- Translating to model checking problem
  - [Kästner et al., 2012]
  - [Kim, Khurshid, and Batory, 2012]

- Record multiple states as a meta state at variable level
  - [Kästner et al., 2012]
  - [Meinicke, 2014]

- Overheads yet need to be controlled
Conclusion

• Mutation analysis is useful
• Scalability is the a key challenge
• Eliminating redundancy is a promising way to address scalability
• Overhead and benefit must be balanced
• Equivalence modulo states could achieve 2.56X speedup over SSE
Acknowledgments

• We acknowledge Rene Just and Micheal Ernst for fruitful discussion helping scope the paper

• and ISSTA Program Committee for the recognition
• and you for listening!