Profile-Guided Program Simplification for Effective Testing and Analysis

Lingxiao Jiang  Zhendong Su
A profile is a set of information about an execution, either succeeded or failed.

Profiles from users are more interesting.
Profile-Collecting Infrastructure

- To reduce users' concern
- 🔄 The Cooperative Bug Isolation Project
- Low-overhead
- Sparse information
- Statistical analysis on profiles from many users for bug isolation
- Low costs on profile triage at the developers' side

Lack of many details that developers desire
Bridge the Information Gap

- An insight: Many in-house development tools, as already used by developers for testing and debugging, can provide the desired information, if the program is "simple" enough.
Bridge the Information Gap

- Our realization: profile-guided program simplification

Program Instrumentation → Execution Profiles

Profile-Collecting Infrastructure

In-house Testing & Analysis

Profile Analyzer and Program Simplifier
A question: why not use in-house techniques on the original programs directly
Testing & Analysis - Incapabilities

- Dynamic techniques, e.g., random testing
  - Incomplete coverage $\rightarrow$ false negatives
- Static techniques, e.g., BLAST
  - Undecidable problems $\rightarrow$ false positives

- Also, implementation limitations

- Developers are used as an expensive remedy:
  - Manual filtering
  - Manual annotations/specifications
Example of Incapability

- Failure occurs when \((I*J)\) is within the range of \([991, 1000]\)

```c
I = 0;
J = 1;
......
I = library_func(J);
J = library_func(I);
if ( I * J > 990 &&
    I * J <= 1000 )
    assert( FALSE );
```

Library functions are not easy to track/control

Non-linear multi-variable constraints are difficult to solve
Example Revisited: Benefit of Program Simplification

Failure occurs when \((I \times J)\) is within the range of \([991, 1000]\)

\[
\begin{align*}
I &= 0; \\
J &= 1; \\
\ldots \ldots \\
I &= \text{library_func}(J); \\
J &= \text{library_func}(I); \\
\text{if} \ (I \times J > 990 \ \&\& \ I \times J \leq 1000) \\
&\quad \quad \text{assert( FALSE );}
\end{align*}
\]

\[
\begin{align*}
I &= 0; \\
J &= 1; \\
\ldots \ldots \\
I &= \text{library_func}(J); \\
J &= \text{library_func}(I); \\
\text{if} \ (\text{TRUE}) \\
&\quad \quad \text{assert( FALSE );}
\end{align*}
\]

This is possible based on statistical analysis on large amount of execution profiles.
Approach Overview

1. Program Instrumentation
   → Execution Profiles
      Profile-Collecting Infrastructure

2. In-house Testing and Analysis

3. Profile Analyzer and Program Simplifier

   Fully automatable and focus naturally on failures experienced more often by users.
Approach Overview

Program Instrumentation

- Execution Profiles
- Profile-Collecting Infrastructure

Profile Analyzer and Program Simplifier

- Statistical Debugging
- Branch Prediction

Program Reduction

- Simplified Programs with Failure Paths Preserved
- Error-Related Code Locations

In-house Testing and Analysis
For each program execution, CBI collects:

- Outcome: exit status of the execution
- List of predicate counters: e.g., the numbers of times branching conditions are observed to be true

\[(p_{11}, \ldots, p_{n1}, \text{label}_1),\]
\[(p_{1i}, \ldots, p_{ni}, \text{label}_i),\]
\[\ldots\]
\[(p_{1m}, \ldots, p_{nm}, \text{label}_m)\]
Statistical Debugging

- A black-box view

\[(p_{11}, \ldots, p_{n1}, \text{label}_1)\]
\[(p_{1i}, \ldots, p_{ni}, \text{label}_i)\]
\[
\ldots
\]
\[(p_{1m}, \ldots, p_{nm}, \text{label}_m)\]

Statistical Debugging

\[p_i\]
\[p_{ij}\]
\[p_{ik}\]

Bug Predictors: the most failure-related code fragments
Branch Prediction

- For each conditional $C$, suppose its *then branch* be $C_t$ and its *else branch* be $C_f$.
- Assign a truth value for every $C_i$, e.g.,
  - True for bug predictors
  - False for $C_i$ never taken in failed executions
  - Unknown for $C_i$ taken in failed and succeeded executions
- Decide branching directions

<table>
<thead>
<tr>
<th>Direction for $C'$</th>
<th>Truth Value of $C_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Truth</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Truth Value of $C_f$</th>
<th>Both</th>
<th>True</th>
<th>False</th>
<th>Both</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>Both</td>
<td>True</td>
<td>Neither</td>
<td>True</td>
</tr>
<tr>
<td>Unknown</td>
<td>Both</td>
<td>False</td>
<td>Both</td>
<td></td>
</tr>
</tbody>
</table>
Branch Reduction

For each conditional $IF(B_p, B_t, B_f)$ and its prediction $P$, if

- $P$ == Neither, replace it with $exit(0)$
- $P$ == True, replace it with $IF(TRUE, B_t, B_f)$
- $P$ == False, replace it with $IF(FALSE, B_t, B_f)$
- $P$ == Both, keep it
Almost Ready

For dynamic testing & analysis tools, the simplified code may exhibit additional failures

```c
if ( p != NULL )
{
    p->next = ...
    ......
    state = ...
    assert ( state != 0 );
}

if ( TRUE )
{
    p->next = ...
    ......
    state = ...
    assert ( state != 0 );
}
```

Segfault
Branch Validation – Assumes

- Require supports of testing or analysis tools
- May still be faster than on original programs, but better not to add it arbitrarily

```c
if ( p != NULL )
{
    p->next = ...
    ......
    state = ...
    assert ( state != 0 );
}
```

```c
if ( TRUE )
{
    assume ( p != NULL );
    p->next = ...
    ......
    state = ...
    assert ( state != 0 );
}
```
The current branch validation strategy: 

*none* or *all*

- Simplify code with *no assume first*
- If segfaults, add in *all assumes* and apply back-end tools again
Experimental Setup

- Subject programs
  - Siemens Test Suite from the Aristotle Analysis System
  - Gzip from the Software-artifact Infrastructure Repository (SIR)

- CBI profiling and statistical bug localization [ASE 2007]

- Testing & analysis tools
  - CUTE, a testing engine that combines both concrete and symbolic execution, [PLDI 2005, FSE 2005]
  - BLAST, a software model checker based on predicate abstraction, [POPL 2002, SPIN 2003]
## Results on CUTE

<table>
<thead>
<tr>
<th>Program</th>
<th># of Faults Found in</th>
<th># of Conditionals</th>
<th>Reduction</th>
<th># of Iterations Taken for</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Simplified</td>
<td>Total</td>
<td>Reduced</td>
</tr>
<tr>
<td>PRINT_TOKENS</td>
<td>1</td>
<td>1</td>
<td>231</td>
<td>46</td>
</tr>
<tr>
<td>PRINT_TOKENS2</td>
<td>9</td>
<td>9</td>
<td>809</td>
<td>41</td>
</tr>
<tr>
<td>REPLACE</td>
<td>27</td>
<td>28</td>
<td>2683</td>
<td>881</td>
</tr>
<tr>
<td>SCHEDULE</td>
<td>8</td>
<td>8</td>
<td>262</td>
<td>104</td>
</tr>
<tr>
<td>SCHEDULE2</td>
<td>6</td>
<td>6</td>
<td>371</td>
<td>123</td>
</tr>
<tr>
<td>TCAS</td>
<td>27</td>
<td>41</td>
<td>1348</td>
<td>926</td>
</tr>
<tr>
<td>TOT_INFO</td>
<td>4</td>
<td>6</td>
<td>1011</td>
<td>296</td>
</tr>
<tr>
<td>GZIP</td>
<td>3</td>
<td>6</td>
<td>5237</td>
<td>3635</td>
</tr>
<tr>
<td>Total</td>
<td>85</td>
<td>105</td>
<td>11952</td>
<td>6052</td>
</tr>
</tbody>
</table>

**Relative Improvements:** +20%
**Extra Bugs Detected: Examples**

**Complement Implementation Limitations**

```c
if ( (float)input >= 1.0 )
{
    ......
    assert( FALSE );
}
```

```c
if ( TRUE )
{
    ......
    assert( FALSE );
}
```
Simplify Path Constraints

```c
int longNumArray[] = { 13, 25, 48, ...
... , /* a lot of numbers */ ...
255, ... };

if ( longNumArray[ input ] == 255 )
{
    ....
    assert( FALSE );
}
```

```c
int longNumArray[] = { 13, 25, 48, ...
... , /* a lot of numbers */ ...
255, ... };

if ( TRUE )
{
    ....
    assert( FALSE );
}
```

`*(longNumArray + input) == 255`
### Results on CUTE

<table>
<thead>
<tr>
<th>Program</th>
<th># of Faults Found in</th>
<th># of Conditionals</th>
<th>Reduction</th>
<th># of Iterations Taken for</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Simplified</td>
<td>Total</td>
<td>Reduced</td>
</tr>
<tr>
<td>PRINT_TOKENS</td>
<td>1</td>
<td>1</td>
<td>231</td>
<td>46</td>
</tr>
<tr>
<td>PRINT_TOKENS2</td>
<td>9</td>
<td>9</td>
<td>809</td>
<td>41</td>
</tr>
<tr>
<td>REPLACE</td>
<td>27</td>
<td>28</td>
<td>2683</td>
<td>881</td>
</tr>
<tr>
<td>SCHEDULE</td>
<td>8</td>
<td>8</td>
<td>262</td>
<td>104</td>
</tr>
<tr>
<td>SCHEDULE2</td>
<td>6</td>
<td>6</td>
<td>371</td>
<td>123</td>
</tr>
<tr>
<td>TCAS</td>
<td>27</td>
<td>41</td>
<td>1348</td>
<td>926</td>
</tr>
<tr>
<td>TOT_INFO</td>
<td>4</td>
<td>6</td>
<td>1011</td>
<td>296</td>
</tr>
<tr>
<td>GZIP</td>
<td>3</td>
<td>6</td>
<td>5237</td>
<td>3635</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>85</td>
<td>105</td>
<td>11952</td>
<td>6052</td>
</tr>
<tr>
<td><strong>Relative Improvements:</strong></td>
<td>+20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
May Be Slower Without Assumes

- Assertions may not be violated with invalid inputs

Suppose \((\text{sum} == 0)\) iff input is in the range of \([99900, 99998]\)

```c
if ( input > 99899 && input < 99999 ) {
    \text{for (i=0; i<input; i++)}
    \text{sum += foo(i);}
    \text{assert ( sum != 0 );}
}

if ( \text{TRUE} ) {
    \text{assume (input > 99899 && input < 99999);}  
    \text{......}
    \text{for (i=0; i<input; i++)}
    \text{sum += foo(i);}  
    \text{assert ( sum != 0 );}
}
```

When CUTE is applied: three iterations vs. tens of thousands of iterations
# Results on BLAST

<table>
<thead>
<tr>
<th>Program Name</th>
<th># of Faults Found in</th>
<th>Reduction Rate</th>
<th># of Calls to Theorem Prover &amp; Reachability Analysis for</th>
<th>Relative Improvements</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINT_TOKENS</td>
<td>6/1 7</td>
<td>19.9%</td>
<td>0 5695 0 4147</td>
<td>0.0% 27.2%</td>
</tr>
<tr>
<td>PRINT_TOKENS2</td>
<td>10 10</td>
<td>5.1%</td>
<td>14424226 1481178 14309606 1474237</td>
<td>0.8% 0.5%</td>
</tr>
<tr>
<td>REPLACE</td>
<td>2/30 18/13</td>
<td>32.8%</td>
<td>0 662 0 265</td>
<td>0.0% 60.0%</td>
</tr>
<tr>
<td>SCHEDULE</td>
<td>4/1 7/1</td>
<td>39.7%</td>
<td>136238 11896 0 2428</td>
<td>100% 79.6%</td>
</tr>
<tr>
<td>SCHEDULE2</td>
<td>6/1 6/1</td>
<td>33.2%</td>
<td>39638 13799 29100 7447</td>
<td>26.6% 46.0%</td>
</tr>
<tr>
<td>TCAS</td>
<td>41 41</td>
<td>68.7%</td>
<td>0 111514 0 13734</td>
<td>0.0% 87.7%</td>
</tr>
<tr>
<td>TOT_INFO</td>
<td>19/2 20/1</td>
<td>29.3%</td>
<td>0 39429 0 28693</td>
<td>0.0% 27.2%</td>
</tr>
<tr>
<td>GZIP</td>
<td>7 7</td>
<td>69.4%</td>
<td>0 11164 0 4452</td>
<td>0.0% 60.1%</td>
</tr>
<tr>
<td>Total</td>
<td>95/35 116/16</td>
<td>50.6%</td>
<td>14600102 1675337 14338706 1535403</td>
<td>1.8% 8.4%</td>
</tr>
</tbody>
</table>

**Relative Improvements:** +21/-19
Discussion

- Not a direct testing or debugging tool itself
- Need different interpretation of bug reports

```c
I = 0;
J = 1;
......
I = library_func(J);
J = library_func(I);
if ( TRUE )
   assert( FALSE );
```

Error Trace

```
I * J > 990 &&
I * J <= 1000
```

Reduced Path Constraints
Discussion

- Not a direct testing or debugging tool itself
- Need different interpretation of bug reports
- May not be faster without selective assumes
- Reply on the accuracy of statistical debugging
- Evaluation for larger-scale programs
Take-away Message

Profile-Guided Program Simplification:
a way to combine many benefits of two existing research areas
and help bridge the information gap between what developers
desire and what users may provide.
Thank you!

Questions?

jiangl@cs.ucdavis.edu