Learning to Accelerate Compiler Testing

Junjie Chen
Dan Hao, Yingfei Xiong, Hongyu Zhang
2017.12.13
Compiler Testing

Ensure Correctness

Compilers

Operating Systems

Apps

Safe-critical Systems

Compiler Testing
Compiler Testing Efficiency

Efficiency Problem

Consuming an extremely long period of time to find a small number of bugs\[1\]\[2\]

Accelerating compiler testing is necessary!

---

Test Prioritization

Only a subset of test programs triggering compiler bugs

Intuitively

compiler testing can be accelerated by running these test programs earlier

Two challenges:
• How to identify which test programs are able to trigger compiler bugs
• How to distinguish which test programs trigger different compiler bugs
Learning to Test – Challenge 1

➢ **Idea**: learns the characteristics of bug-revealing test programs to prioritize new test programs

    ➢ Model the relationship between the characteristics of the test programs and the discovery of compiler bugs
    ➢ Model the relationship between the characteristics of the test programs and the execution time
    ➢ Use the models to help us prioritize new test programs

Based on this idea, we develop **LET** (short for learning to test), a learning-to-test approach to *accelerating compiler testing*. 
LET Approach Overview

- **Learning process**: Identifying features, Training a capability model, Training a time model
- **Scheduling process**: Ranking new test programs
Identifying features

• **Existence features**, are concerned with whether certain types of elements exist in the target program
  • \( \text{EXIST} = \text{STMT} \cup \text{EXPR} \cup \text{VAR} \cup \text{OP} \)
    • \( \text{STMT} \) is the set of all statement types in C language
    • \( \text{EXPR} \) is the set of all expression types in C language
    • \( \text{VAR} \) is the set of all variable types in C language
    • \( \text{OP} \) is the set of all operation types in C language

• **Usage features**, are concerned with how the elements in a program are used
  • To save the feature collection time, we directly use the usage features collected by CSmith for our offline training.
    • Address features, e.g., the number of times the address of a struct
    • Pointer dereference features, e.g., the depth of pointer dereference
    • ......
Training a capability model

Predict the probability of a test program triggering bugs

Feature selection
Normalization
Building the capability model

Filter useless features: >> information gain ratio = 0
Normalize each value of these features into the interval [0, 1] >> min-max normalization
Use Sequential Minimal Optimization (abbreviated as SMO) algorithm (Puk kernel)
Training a time model

Predict the execution time of a test program

Same features

Gaussian processes

Execution time on previous version

Time Model (Regression model)
Scheduling

Prioritize new test programs based on the descendent order of their bug-revealing probabilities in unit time.
Empirical Study

Cross versions: GCC-4.3.0 → GCC-4.4.3, LLVM-2.6 → LLVM-2.7
Cross compilers: Open64-5.0 → GCC-4.4.3 & LLVM-2.6
GCC-4.3.0 → LLVM-2.6

<table>
<thead>
<tr>
<th>Subject</th>
<th>LOC</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC-4.3.0</td>
<td>3,343,377</td>
<td>Learning</td>
</tr>
<tr>
<td>GCC-4.4.3</td>
<td>4,727,209</td>
<td>Scheduling</td>
</tr>
<tr>
<td>LLVM-2.6</td>
<td>684,114</td>
<td>Learning &amp; Scheduling</td>
</tr>
<tr>
<td>LLVM-2.7</td>
<td>795,152</td>
<td>Scheduling</td>
</tr>
<tr>
<td>Open64-5.0</td>
<td>6,078,400</td>
<td>Learning</td>
</tr>
</tbody>
</table>
Compiler Testing Techniques

**DOL[1]**: determines whether a compiler contains bugs by comparing the results produced by the same test program with different optimization levels.

**EMI[2]**: generates some equivalent variants for any given test program and determines whether a compiler contains bugs by comparing the results produced by the original test program and its variants.

Compared Approaches

**RO**: randomly selects an execution order of new test programs, is taken as the baseline

**TB-G[1]**: regards each test program as text and transforms each test program into a text-vector by extracting corresponding tokens from text, and then prioritizes test programs based on the distance between the text-vector and the origin vector (0, 0, ..., 0)

Overall Effectiveness

1. LET does accelerate compiler testing
2. LET perform much better and more stable than TB-G in accelerating compiler testing
Effectiveness for different techniques and on different scenarios

<table>
<thead>
<tr>
<th>Summary</th>
<th>Techniques</th>
<th></th>
<th>Scenarios</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DOL</td>
<td>EMI</td>
<td>Cross-compiler</td>
</tr>
<tr>
<td>Mean(%)</td>
<td>30.91</td>
<td>50.81</td>
<td>47.85</td>
<td>27.69</td>
</tr>
<tr>
<td>p-value</td>
<td>0.000(+)</td>
<td>0.000(+)</td>
<td>0.000(+)</td>
<td>0.000(+)</td>
</tr>
</tbody>
</table>

1. LET achieves great effectiveness for accelerating different compiler testing techniques.

2. LET accelerates compiler testing no matter which compiler or version is used to train the models.
How to identify duplications?

It is scarcely possible to know which test programs trigger the same bug before testing.

The only possible direction is to find a proper approximation to distinguish them.

- the most widely-used methods to measure the test effectiveness
- has been widely adopted by researchers and practitioners to improve software testing/debugging process
How to acquire test coverage?

Instrumenting and executing

- Hugeness and complexity of compilers
- Test programs generated on the fly

The process of collecting coverage is very time-consuming
The volume of the coverage information is also very large, which incurs overhead in storage and maintenance

In compiler testing, the coverage information of these test programs is not available in advance.

Direction: acquiring test coverage statically, without executing the test programs
Approach

COP (short for COverage Prediction)

Two innovation points:

- the first time to predict test coverage statically for compilers, without executing the test programs.
- it utilizes clustering to distinguish test programs triggering different compiler bugs based on the predicted coverage information, so as to accelerate compiler testing.
COP

Predicting coverage

Clustering test programs

Prioritizing test programs

Identifying features: language features, operation features, structure features

Labeling: the covered fraction of each compiler module (removing imbalance cases)

Building a Prediction Model: normalization + training using the gradient boosting for regression

Predicting and Aligning Coverage: removing newly added and out-of-date compiler modules

Multi-output regression problem
COP always achieves great effectiveness for different compiler testing techniques on different application scenarios.

### TABLE 5
Effectiveness on Different Compiler Testing Techniques and Different Application Scenarios

<table>
<thead>
<tr>
<th>Summary</th>
<th>Techniques</th>
<th>Scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Same-version</td>
</tr>
<tr>
<td></td>
<td>DOL</td>
<td>EMI</td>
</tr>
<tr>
<td>Acc. (%)</td>
<td>94.44</td>
<td>93.55</td>
</tr>
<tr>
<td>Mean (%)</td>
<td>46.12</td>
<td>54.90</td>
</tr>
<tr>
<td>p-value</td>
<td>0.000(†)</td>
<td>0.000(†)</td>
</tr>
</tbody>
</table>
Comparison with LET

![Box plots comparing COP and LET speedup distributions](image)

**TABLE 6**

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>GCC-4.4.3 → GCC-4.4.3</th>
<th>GCC-4.3 → GCC-4.4.3</th>
<th>LLVM-2.6 → GCC-4.4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOL Mean(%)</td>
<td>17.61</td>
<td>20.26</td>
<td>14.26</td>
</tr>
<tr>
<td>p-value</td>
<td>0.006(+)</td>
<td>0.001(+)</td>
<td>0.064</td>
</tr>
<tr>
<td>EMI Mean(%)</td>
<td>22.45</td>
<td>43.81</td>
<td>55.27</td>
</tr>
<tr>
<td>p-value</td>
<td>0.014(+)</td>
<td>0.000(++)</td>
<td>0.022(++)</td>
</tr>
</tbody>
</table>

Fig. 1. Speedup distribution of acceleration approaches, COP and LET
## Coverage Prediction Effectiveness

### TABLE 2
Effectiveness of COP for predicting coverage in the same-version application scenario

<table>
<thead>
<tr>
<th>Subjects</th>
<th>EV</th>
<th>MAE</th>
<th>R²</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>COP</td>
<td>RG</td>
<td>COP</td>
</tr>
<tr>
<td>GCC</td>
<td>4.3.0→4.3.0</td>
<td>0.899</td>
<td>-1.423</td>
</tr>
<tr>
<td></td>
<td>4.4.0→4.4.0</td>
<td>0.884</td>
<td>-1.672</td>
</tr>
<tr>
<td></td>
<td>4.5.0→4.5.0</td>
<td>0.897</td>
<td>-1.600</td>
</tr>
<tr>
<td></td>
<td>4.6.0→4.6.0</td>
<td>0.891</td>
<td>-1.686</td>
</tr>
<tr>
<td>LLVM</td>
<td>2.6→2.6</td>
<td>0.799</td>
<td>-1.134</td>
</tr>
<tr>
<td></td>
<td>2.7→2.7</td>
<td>0.790</td>
<td>-1.179</td>
</tr>
<tr>
<td></td>
<td>2.8→2.8</td>
<td>0.822</td>
<td>-1.172</td>
</tr>
</tbody>
</table>

### TABLE 3
Effectiveness of COP for predicting coverage in the cross-version application scenario

<table>
<thead>
<tr>
<th>Subjects</th>
<th>EV</th>
<th>MAE</th>
<th>R²</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>COP</td>
<td>RG</td>
<td>COP</td>
</tr>
<tr>
<td>GCC</td>
<td>4.3.0→4.4.0</td>
<td>0.761</td>
<td>-1.529</td>
</tr>
<tr>
<td></td>
<td>4.4.0→4.5.0</td>
<td>0.749</td>
<td>-1.708</td>
</tr>
<tr>
<td></td>
<td>4.5.0→4.6.0</td>
<td>0.874</td>
<td>-1.786</td>
</tr>
<tr>
<td>LLVM</td>
<td>2.6→2.7</td>
<td>0.476</td>
<td>-1.092</td>
</tr>
<tr>
<td></td>
<td>2.7→2.8</td>
<td>0.535</td>
<td>-1.283</td>
</tr>
</tbody>
</table>
Thank You!