Automated Adaptive Bug Isolation using Dyninst

Piramanayagam Arumuga Nainar,
Prof. Ben Liblit
University of Wisconsin-Madison
Cooperative Bug Isolation (CBI)

++branch_17[p != 0];
if (p) ...
else ...

Program Source

Predicates
Sampler
Compiler

Shipping Application

Top bugs with likely causes

Statistical Debugging

Counts & 😊/😊
Issues

• Problem with static instrumentation
  – Predicates are fixed for entire lifetime
  – Three problems
    1. Worst case assumption
    2. Cannot stop counting predicates
      – After collecting enough data
    3. Cannot add predicates we missed

• Current infrastructure supports only C programs
CBI for Binaries

Program Source

Compile

Compile

Shipping Application

Executable

Predicates

Binary editor

New Executable

Top bugs with likely causes

Statistical Debugging

Counts & 😊/😊

developer

user
Adaptive Bug Isolation

• Strategy:
  – Adaptively add/remove predicates
    • Based on feedback reports
• Retain existing statistical analysis
  – Goal is to guide CBI to its best bug predictor
    • Reduce the number of predicates instrumented
Adaptive Bug Isolation (contd.)
Dyninst Instrumentor - Features

- Counters in shared segment
- Removes snippets
  - After they execute once
- Call graph, CFG, dominator graphs
- Snippets are feather weight
  - Don’t save/restore FPRs
- More…
  - Better overheads
  - Expose data dependencies
Technique

• Control Dependence Graph (CDG)

• Algorithm:
  – For each *suspect* branch edge:
    • Enable all predicates in
      – basic blocks control dependent on that edge

• How to identify suspect edges?
  – Pessimistic - all edges are suspects
Simple strategy: BFS

• All branch predicates are suspicious
Can we do better?

- Assign scores to each predicate
- Edges with high scores are suspect
  - Many options
    - Top 10
    - Top 10%
    - Score $> \text{threshold}$
  - For our experiments, only the topmost predicate
  - Other predicates: may be revisited in future
- Key property: If no bug is found, no predicate is left unexplored
Scores – heuristics 1,2,3

- Many possibilities. We evaluate five
- For a branch predicate p,
  - \( F(p) \) = no. of failed runs in which p was true
  - \( S(p) \) = no. of successful runs in which p was true
1. Failure count: \( F(p) \)
2. Failure probability: \( F(p) / (F(p) + S(p)) \)
3. T-Test

<table>
<thead>
<tr>
<th></th>
<th>Succ</th>
<th>Fail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p ) true</td>
<td>30%</td>
<td>70%</td>
</tr>
<tr>
<td>( p ) false</td>
<td>50%</td>
<td>50%</td>
</tr>
</tbody>
</table>

\( Pr \) (\( p \) being true affects program outcome in a statistically significant manner)
Scores - heuristic 4

4. *Importance* \((p)\)
   - CBI’s ranking heuristic [PLDI ’05]
   - Harmonic mean of two values
   - For a branch predicate ‘\(p\)’:
     - Sensitivity
       - \(\log (F(p)) / \log \) (total failures observed)
     - Increase
       - \(\Pr \) (Failure) at \(P_2\) – \(\Pr \) (Failure) at \(P_1\)
Scores - heuristic 5

5. Maximum possible *Importance* score
   - Problem: sometimes, *Importance* (p) mirrors p’s properties and says nothing about the branch’s targets
     - score (p) = Maximum possible Importance score in p’s targets

Edge label a/b means
- Predicate was true in ‘a’ successful runs
- Predicate was true in ‘b’ failed runs
Optimal heuristic

• Oracle
  – points in the direction of the target (the top bug predictor)
  – Used for evaluation of the results
  – Shortest path in CDG
Evaluation

• Binary Instrumentor: using DynInst
• Heuristics:
  – 5 global ranking heuristics
  – simplest approach: BFS
  – optimal approach: Oracle
• Bug benchmarks
  – siemens test suite
• Goal: identify the best predicate efficiently
  – Best predicate: as per the PLDI ’05 algo.
  – efficiency: no. of predicates examined
Evaluation (cont.)
Conclusion

• Use binary instrumentation to
  – Skip bug free regions
    ➔ more data from interesting sites
• Fairly general
  – Can be applied to any CBI-like tool
• Backward search – in progress
Questions?
Binary Instrumentor

- Using DynInst
- Large slowdowns
- Reduce no. of branch predicates

Gathering true/false values instead of counts

1. No increment. Just store 1 (true)
2. Self-removing instrumentation
   - Removes itself after executing
   - Applies only to dynamic instrumentation

Better performance: 2-3 times slowdown for go
- But not enough

If program crashes between $p_1$ and $p_2$

\[
\begin{align*}
  c_1 &= c_2 + c_3 + 1 \\
  c_1 &= c_2 + c_3
\end{align*}
\]
Branch predicate inference

- $c_1$ can be inferred if
  - $P_1$ dominates $P_2$
  - $P_2$ or $P_5$ have an instrumentation site
    (in general, any block equivalent to $P_2$)
Can we do better?

- Choose one branch over the other

<table>
<thead>
<tr>
<th></th>
<th>Succ</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>then</em> path</td>
<td>30%</td>
<td>70%</td>
</tr>
<tr>
<td><em>else</em> path</td>
<td>50%</td>
<td>50%</td>
</tr>
</tbody>
</table>

Program fails more often in the *then* path

Pr (there is a significant difference in the two directions) > 95%:
- Only *then* path is interesting
- else both *then* and *else* paths are interesting

Use *T-Test* (paired)
Simple strategy: BFS

• All branch predicates are suspicious