Dynamic Analysis

Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
Jan 2016

Outline

- Applications of dynamic analysis
  - Limitations of static analysis
  - Trade-offs
- Profiling techniques
- Finding invariants
  - Equality
  - Affine
- Dynamic type inferencing

Limitations of Static Analysis

- Reduced precision: Over-approximations
- Cannot perform input-dependent analysis

Applications

- Bug finding (testing)
- Data race detection
- Identifying security vulnerabilities
- Improved precision of static analysis
- Input-dependent analysis

Static versus Dynamic

- Sound
- Imprecise
- Input-oblivious
- Incomplete
- Precise
- Input-dependent

Choosing between static and dynamic analysis often requires a trade-off between soundness and precision. Current trend is to combine the two techniques to get better precision at improved scalability.

Profiling

- Profiling is a method of collecting information of interest during program execution.
- The information is often useful to find hot-spots in the program.
- Examples
  - Number of times an instruction is executed
  - Number of page faults
  - Number of cache hits
  - Total memory used
  - ...

Outline

- Applications of dynamic analysis
  - Limitations of static analysis
  - Trade-offs
- Profiling techniques
- Finding invariants
  - Equality
  - Affine
- Dynamic type inferencing

Limitations of Static Analysis

- Reduced precision: Over-approximations
- Cannot perform input-dependent analysis

Applications

- Bug finding (testing)
- Data race detection
- Identifying security vulnerabilities
- Improved precision of static analysis
- Input-dependent analysis

Static versus Dynamic

- Sound
- Imprecise
- Input-oblivious
- Incomplete
- Precise
- Input-dependent

Choosing between static and dynamic analysis often requires a trade-off between soundness and precision. Current trend is to combine the two techniques to get better precision at improved scalability.

Profiling

- Profiling is a method of collecting information of interest during program execution.
- The information is often useful to find hot-spots in the program.
- Examples
  - Number of times an instruction is executed
  - Number of page faults
  - Number of cache hits
  - Total memory used
  - ...
Profiling

- **Intrusive**: inserts instructions in the program (source, IR, assembly) **statically**, which get executed at **runtime**
  - File log
  - Memory locations pointed to by a pointer
  - Execution time of a function
- **Non-intrusive**: the program is unaltered; uses external means to profile
  - Hardware counters
  - Program execution time

Path Profiling

Consider a program with an entry node and an exit node. There are several execution paths (traces) that the program takes from entry to exit.

The task is to find the frequency of execution of each path.

<table>
<thead>
<tr>
<th>Path</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDF</td>
<td>90</td>
</tr>
<tr>
<td>ACDEF</td>
<td>60</td>
</tr>
<tr>
<td>ABCDF</td>
<td>0</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>100</td>
</tr>
<tr>
<td>ABDF</td>
<td>20</td>
</tr>
<tr>
<td>ABDEF</td>
<td>0</td>
</tr>
</tbody>
</table>

Edge Profiling

- Path profile is approximated as an edge profile
- The frequency of each edge is calculated – which is used to find the path frequency

Path vs. Edge Profiling

- Path profile is approximated as an edge profile
- The frequency of each edge is calculated – which is used to find the path frequency

Can this instrumentation be optimized?
Can we have better precision?

<table>
<thead>
<tr>
<th>Path</th>
<th>Path Frequency (actual)</th>
<th>Path Frequency (estimated)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDF</td>
<td>90</td>
<td>110</td>
</tr>
<tr>
<td>ACDEF</td>
<td>60</td>
<td>150</td>
</tr>
<tr>
<td>ABCDF</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>ABDF</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>ABDEF</td>
<td>0</td>
<td>20</td>
</tr>
</tbody>
</table>
Efficient Edge Profiling

- Observation: We do not need to instrument every edge.
- How to find a minimal, low-cost set of edges to instrument?
- Use a spanning tree (instrument non-std edges):
  - reduced instrumentation along paths,
  - not all edges carry instrumentation

Classwork: Find counts for uninstrumented edges.

### Path Frequency

<table>
<thead>
<tr>
<th>Path</th>
<th>Frequency</th>
<th>Actual Freq.</th>
<th>Actual Freq. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDF</td>
<td>110</td>
<td>90</td>
<td>110</td>
</tr>
<tr>
<td>ACDEF</td>
<td>150</td>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>ABDF</td>
<td>20</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>ABDEF</td>
<td>20</td>
<td>0</td>
<td>20</td>
</tr>
</tbody>
</table>

But path profiling is expensive.

Efficient Path Profiling

- Path numbering is not unique.

Classwork: Prove that such a path numbering is unique.

### Path x

<table>
<thead>
<tr>
<th>Path</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDF</td>
<td>0</td>
</tr>
<tr>
<td>ACDEF</td>
<td>1</td>
</tr>
<tr>
<td>ABCDF</td>
<td>2</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>3</td>
</tr>
<tr>
<td>ABDF</td>
<td>4</td>
</tr>
<tr>
<td>ABDEF</td>
<td>5</td>
</tr>
</tbody>
</table>

In all the above cases, the path numbering is the same, number of instrumented edges (5) is the same. So, which instrumentation should we choose?

Efficient Path Profiling

1. Assign integer values to edges such that no two paths compute the same path-sum.
2. Use a spanning tree to select edges to instrument and compute the appropriate increment.
3. Select appropriate instrumentation.
4. After collecting the run-time profile, derive the execution paths.

Efficient Path Profiling

1. Assign integer values to edges such that no two paths compute the same path-sum.
2. Use a spanning tree to select edges to instrument and compute the appropriate increment.
3. Select appropriate instrumentation.
4. After collecting the run-time profile, derive the execution paths.

**NumPaths(node) = 0**

**NumPaths(leaf) = 1**

In reverse topological order

For each edge v → w {
  
  Val(v → w) = NumPaths(v)
  
  NumPaths(v) ↔ NumPaths(w)
}

Efficient Path Profiling

Classwork: Find counts for uninstrumented edges.
Efficient Path Profiling

1. Assign integer values to edges such that no two paths compute the same path-sum.
2. Use a spanning tree to select edges to instrument and compute the appropriate increment.
3. Select appropriate instrumentation.
4. After collecting the run-time profile, derive the execution paths.

• Find a spanning tree.
• Find chord (non-ST) edges.
• For each chord, find fundamental cycle.

Classwork

• Find the instrumentation for the following CFG

Path Regeneration
Path id → Path mapping?

<table>
<thead>
<tr>
<th>Path</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACF</td>
<td>0</td>
</tr>
<tr>
<td>ACDEF</td>
<td>1</td>
</tr>
<tr>
<td>ABCDF</td>
<td>2</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>3</td>
</tr>
<tr>
<td>ABDEF</td>
<td>4</td>
</tr>
<tr>
<td>ABDEF</td>
<td>5</td>
</tr>
</tbody>
</table>

Prelude: Allocate and initialize the array of counters

Postlude: Write the array to permanent storage

Main:
• Initialize path register r in the entry vertex
• Increment path memory counter in the exit vertex
• Optimizations