Program Testing and Analysis:
Path Profiling

Dr. Michael Pradel
Software Lab, TU Darmstadt
What does the following code print?

```javascript
function d(a, b) {
    var sum = 0;
    for (var i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    console.log(sum == (a + b + 67));
}

d(23, 45, 67);
```

true false Something else
Warm-up Quiz

What does the following code print?

```javascript
function d(a, b) {
    var sum = 0;
    for (var i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    console.log(sum == (a + b + 67));
}

d(23, 45, 67);
```

true
false
Something else

Includes all passed arguments, independent of declared parameters
Outline

1. Motivation and Challenges
2. Ball-Larus algorithm for DAGs
3. Generalization and Applications

Mostly based on this paper:

- Efficient path profiling, Ball and Larus, MICRO 1996

Other reading material:

- Whole program paths, Larus, PLDI 1999
- HOLMES: Effective statistical debugging via efficient path profiling, Chilimbi, ICSE 2009
Path Profiling

- **Goal:** Count how often a path through a function is executed

- Interesting for various applications
  - Profile-directed compiler optimizations
  - **Performance tuning:** Which paths are worth optimizing?
  - **Test coverage:** Which paths are not yet tested?
Challenges

- **Runtime overhead**
  - Limit slowdown of program

- **Accuracy**
  - Ideally: Precise profiles (not heuristics, no approximations)

- **Infinitely many paths**
  - Cycles in control flow graph
<table>
<thead>
<tr>
<th>#</th>
<th>Path</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ACDF</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ACDEFG</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ABCDF</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ABCDEFG</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ABDF</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ABDGEF</td>
<td></td>
</tr>
</tbody>
</table>

Running example

B

E
Edge Profiling

Naive approach: Edge profiling

- Instrument each branching point
- Count how often each CFG edge is executed
- Estimate most frequent path: Always follow most frequent edge
Example: Edge profiling

\[ \begin{align*}
A & \quad 120 \quad 150 \\
B & \quad 100 \quad C \\
& \quad 20 \quad 250 \\
D & \quad 160 \quad 110 \\
E & \quad 160 \quad F
\end{align*} \]

Frequency of execution

Quiz: What is the most frequent path?

AC DEF

Really? Two possible path profiles:

<table>
<thead>
<tr>
<th>Path</th>
<th>Profile 1</th>
<th>Profile 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC DEF</td>
<td>90</td>
<td>110</td>
</tr>
<tr>
<td>A C D E F</td>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>A B C D F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A B C D E F</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>A B D E F</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>A B D E F</td>
<td>0</td>
<td>20</td>
</tr>
</tbody>
</table>
Edge Profiling

Naive approach: Edge profiling

- Instrument each branching point
- Count how often each CFG edge is executed
- Estimate most frequent path: Always follow most frequent edge
Edge Profiling

Naive approach: Edge profiling

■ Instrument each branching point
■ Count how often each CFG edge is executed
■ Estimate most frequent path: Always follow most frequent edge

Fails to uniquely identify most frequent path
Ball-Larus Algorithm

- Assign a **number to each path**
- Compute path number of **incrementing a counter** at branching points
- **Properties of path encoding**
  - Precise: A single **unique encoding for each path**
  - Minimal: Instruments subset of edges with minimal cost
Example: Path Encoding

- A: r = 0
- B: r = 2
- C: r = 4
- D: r += 1
- E: count [r]++

Instrumentation:
- State "r"
- Array of counters "count"

<table>
<thead>
<tr>
<th>Path</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDF</td>
<td>0</td>
</tr>
<tr>
<td>ACDEDF</td>
<td>1</td>
</tr>
<tr>
<td>ABCDF</td>
<td>2</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>3</td>
</tr>
<tr>
<td>ABCDF</td>
<td>4</td>
</tr>
<tr>
<td>ABCDEF</td>
<td>5</td>
</tr>
</tbody>
</table>
Algorithm for DAGs

Assumptions

- Control flow graph is a directed acyclic graph (DAG)
- $n$ paths (numbered 0 to $n - 1$)
- Graph has unique entry and exit nodes
- Artificial back edge from exit to entry
Assumptions

A

unique entry node

B

C

D

E

F

graph is acyclic

artificial backedge

unique exit node
Algorithm: Overview

- **Step 1**: Assign integers to edges
  - Goal: Sum along a path yields unique number for path
  - Enough to achieve "precise" goal

- **Step 2**: Assign increment operations to edges
  - Goal: Minimize additions along edges
  - Instrument subset of all edges
  - Assumes to know/estimate how frequent edges are executed
Representing Paths with Sums

- Associate with each node a value:
  \[ \text{NumPaths}(n) = \text{number of paths from } n \text{ to exit} \]

- Computing \( \text{NumPaths} \)
  - Visit nodes in reverse topological order
  - If \( n \) is leaf node:
    \[ \text{NumPaths}(n) = 1 \]
  - Else:
    \[ \text{NumPaths}(n) = \text{sum of } \text{NumPaths} \text{ of destination of outgoing edges} \]
Example: NumPaths

Reverse topological order:
Successor of \( n \) visited before \( n \)

<table>
<thead>
<tr>
<th>Node ( n )</th>
<th>Num Paths ( (n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
</tr>
</tbody>
</table>
Representing Paths with Sums (2)

For each node in reverse topological order:

- If $n$ is leaf node:
  \[
  \text{NumPaths}(n) = 1
  \]

- Else:
  \[
  \text{NumPaths}(n) = 0
  \]
  For each edge $n \to m$:
  \[
  \begin{align*}
  \text{Val}(n \to m) &= \text{NumPaths}(n) \\
  \text{NumPaths}(n) &= \text{NumPaths}(m)
  \end{align*}
  \]
Example: Values for Edges

A
  \_2\_0\_
  \_\_\_
  B\_0\_
  2\_0\_
  \_\_\_
  D\_9\_
  1\_9\_
  \_\_\_
  E\_0\_
  \_\_\_
  \_\_\_
  T
Quiz: Values for Edges

<table>
<thead>
<tr>
<th>n</th>
<th>NumPath(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

- Sum of all NumPaths? 12
- Sum of path 1346? 3
Algorithm: Overview

- **Step 1:** Assign integers to edges
  - Goal: Sum along a path yields unique number for path
  - Enough to achieve "precise" goal

- **Step 2:** Assign increment operations to edges
  - Goal: Minimize additions along edges
  - Instrument subset of all edges
  - Assumes to know/estimate how frequent edges are executed
Spanning Tree

- **Given**: Graph $G$
- **Spanning tree** $T$:
  - Undirected subgraph of $G$ that is a tree and that contains all nodes of $G$
- **Chord edges**: Edges in $G$ but not in $T$
Example: Spanning Tree

Which of these is a spanning tree of $G$?

Solution: 2, 4, 5
Increments for Edges

Goal: Increment sum at subset of edges

- Choose **spanning tree with maximum edge cost**
  - Cost of individual edges is assumed to be known

- Compute **increments at the chords of the spanning tree**
  - Based on existing event counting algorithm
Example: Increments for Edges

edge cost
most expensive spanning tree

chord edges
non-minimal increments

minimal increments
= path encoding
Instrumentation

■ Basic idea

□ Initialize sum at entry: $r=0$
□ Increment at edges: $r += \ldots$
□ At exit, increment counter for path:
  $\text{count}[r]++$

■ Optimization

□ Initialize with incremented value, if first chord on path: $r=\ldots$
□ Increment sum and counter for path, if last chord on path: $\text{count}[r+\ldots]++$
Regenerating the Path

Knowing the sum $r$, how to determine the path?

- Use edge values from step 1 ("non-minimal increments")
- Start at entry with $R = r$
- At branches, use edge with largest value $v$ that is smaller than $R$ and set $R := v$
Example: Regenerating the Path

\[ r = 4: \text{ABDF} \]

\[ r = 1: \text{ACDE} \]
Generalizing to Cyclic CGFs

- For each backedge $n \rightarrow m$, add dummy edges
  - $Entry \rightarrow m$
  - $n \rightarrow Exit$

- Remove backedges and add DAG-based increments

- In addition, add instrumentation to each backedge
  - `count [r]++; r=0`
Example: Generalizing

<table>
<thead>
<tr>
<th>Path</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF</td>
<td>0</td>
</tr>
<tr>
<td>ABCEF</td>
<td>1</td>
</tr>
<tr>
<td>ABC</td>
<td>2</td>
</tr>
<tr>
<td>BCEF</td>
<td>5</td>
</tr>
<tr>
<td>BCE</td>
<td>6</td>
</tr>
<tr>
<td>BDE</td>
<td>8</td>
</tr>
</tbody>
</table>

dummy edges
backedge increments without backedge
Conclusions

- **Ball-Larus profiling:**
  Beautiful algorithm with various applications

- Path profiles also determine accurate edge and basic block profiles

- Can generalize to other graphs, e.g., to compute call graph profiles