Estimating Path Execution Frequency Statically

Ray Buse
Wes Weimer

THE ROAD NOT TAKEN
The Big Idea

- Developers often have a expectations about common and uncommon cases in programs
- The structure of code they write can sometimes reveal these expectations
public V function(K k, V v)
{
    if ( v == null )
        throw new Exception();

    if ( c == x )
        r();

    i = k.h();

    t[i] = new E(k, v);
    c++;

    return v;
}
Example

```java
public V function(K k, V v) {
    if (v == null)
        throw new Exception();
    if (c == x)
        restructure();
    i = k.h();
    t[i] = new E(k, v);
    c++;
    return v;
}
```

Exception

Invocation that changes a lot of the object state

Some computation
```java
public V function(K k, V v) {
    if (v == null)
        throw new Exception();

    if (c == x)
        restructure();

    i = k.h();

    t[i] = new E(k, v);
    c++;

    return v;
}
```
```java
public V function(K k, V v) {
    if (v == null) {
        throw new Exception();
    }
    if (c == x) {
        restructure();
    }
    i = k.h();
    t[i] = new E(k, v);
    c++;
    return v;
}
```
public V function(K k, V v) {
    if (v == null)
        throw new Exception();
    if (c == x)
        restructure();
    i = k.h();
    t[i] = new E(k, v);
    c++;
    return v;
}
public V put(K key, V value) {
    if (value == null) 
        throw new Exception();

    if (count >= threshold) 
        rehash();

    index = key.hashCode() % length;

    table[index] = new Entry(key, value);
    count++;

    return value;
}
Intuition

How a path modifies *program state* may correlate with its runtime execution frequency

- Paths that change a lot of *state* are rare
  - Exceptions, initialization code, recovery code, etc.
- Common paths tend to change a small amount of *state*
More Intuition

- Number of branches
- Number of method invocations
- Path length
- Percentage of statements in a method executed
- ...

...
Hypothesis

We can *accurately* predict the runtime frequency of program *paths* by analyzing their static *surface features*.

Goals:
- Know what programs are *likely* to do without having to run them (Produce a *static profile*)
- Understand the *factors* that are predictive of execution frequency
Our Path

- Intuition
- Candidates for *static profiles*
- Our approach
  - a descriptive model of path frequency
- Some Experimental Results
Applications for Static Profiles

- Indicative (dynamic) profiles are often hard to get

Profile information can improve many analyses
- Profile guided optimization
- Complexity/Runtime estimation
- Anomaly detection
- Significance of difference between program versions
- Prioritizing output from other static analyses
Approach

- **Model** path with a set of features that may correlate with runtime path frequency
- **Learn** from programs for which we have indicative workloads
- **Predict** which paths are most or least likely in other programs
Experimental Components

- Path Frequency Counter
  - Input: Program, Input
  - Output: List of paths + frequency count for each

- Descriptive Path Model

- Classifier
Our Definition of Path

- Statically enumerating full program paths doesn't *scale*
- Choosing only intra-method paths doesn't give us enough *information*
- Compromise: Acyclic Intra-Class Paths
  - Follow execution from public method entry point until return from class
  - Don’t follow back edges
Experimental Components

- Path Frequency Counter
  - Input: Program, Input
  - Output: List of paths + frequency count for each

- Descriptive Path Model
  - Input: Path
  - Output: Feature Vector describing the path

- Classifier
<table>
<thead>
<tr>
<th>Count</th>
<th>Coverage</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
<td>pointer comparisons</td>
</tr>
<tr>
<td>•</td>
<td><strong>new</strong></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td><strong>this</strong></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>all variables</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>assignments</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>dereferences</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>fields</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>fields written</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>statements in invoked method</td>
</tr>
<tr>
<td>•</td>
<td><strong>goto</strong></td>
<td>stmts</td>
</tr>
<tr>
<td>•</td>
<td><strong>if</strong></td>
<td>stmts</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>local invocations</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>local variables</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>non-local invocations</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>parameters</td>
</tr>
<tr>
<td>•</td>
<td><strong>return</strong></td>
<td>stmts</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>statements</td>
</tr>
<tr>
<td>•</td>
<td><strong>throw</strong></td>
<td>stmts</td>
</tr>
</tbody>
</table>
Experimental Components

- **Path Frequency Counter**
  - Input: Program, Input
  - Output: List of paths + frequency count for each

- **Descriptive Path Model**
  - Input: Path
  - Output: Feature Vector describing the path

- **Classifier**
  - Input: Feature Vector
  - Output: Frequency Estimate
Learn a logistic function to estimate the runtime frequency of a path.

\[ f(z) = \frac{1}{1 + e^{-z}} \]

\[ z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots + \beta_k x_k, \]
Model Evaluation

- Use the model to rank all static paths in the program
- Measure how much of total program runtime is spent:
  - On the top X paths for each method
  - On the top X% of all paths
- Also, compare to static branch predictors
- Cross validation on Spec JVM98 Benchmarks
  - When evaluating on one, train on the others
## Spec JVM 98 Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>LOC</th>
<th>Methods</th>
<th>Paths</th>
<th>Paths/Method</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>check VM features</td>
<td>1627</td>
<td>107</td>
<td>1269</td>
<td>11.9</td>
<td>4.2s</td>
</tr>
<tr>
<td>compress</td>
<td>compression</td>
<td>778</td>
<td>44</td>
<td>491</td>
<td>11.2</td>
<td>2.91s</td>
</tr>
<tr>
<td>db</td>
<td>data management</td>
<td>779</td>
<td>34</td>
<td>807</td>
<td>23.7</td>
<td>2.8s</td>
</tr>
<tr>
<td>jack</td>
<td>parser generator</td>
<td>7329</td>
<td>304</td>
<td>8692</td>
<td>28.6</td>
<td>16.9s</td>
</tr>
<tr>
<td>javac</td>
<td>compiler</td>
<td>56645</td>
<td>1183</td>
<td>13136</td>
<td>11.1</td>
<td>21.4s</td>
</tr>
<tr>
<td>jess</td>
<td>expert system shell</td>
<td>8885</td>
<td>44</td>
<td>147</td>
<td>3.3</td>
<td>3.12s</td>
</tr>
<tr>
<td>mtrt</td>
<td>ray tracer</td>
<td>3295</td>
<td>174</td>
<td>1573</td>
<td>9.04</td>
<td>6.17s</td>
</tr>
<tr>
<td>Total</td>
<td>or Average</td>
<td>79338</td>
<td>1620</td>
<td>26131</td>
<td>12.6</td>
<td>59s</td>
</tr>
</tbody>
</table>
Evaluation: Top Paths

Choose 5% of all paths and get 50% of runtime behavior.

Choose 1 path per method and get 94% of runtime behavior.

Number of paths selected per method vs. Percent of all paths selected.
At each branching node...

- Partition the path set entering the node into two sets corresponding to the paths that conform to each side of the branch.
- Record the prediction for that branch to be the side with the highest frequency path available.

Given where we’ve been, which branch represents the highest frequency path?
We are even a reasonable choice for static branch prediction.

Branch Taken; Forward Not Taken

A set of heuristics

Always choose the higher frequency path
Model Analysis: Feature Power

Exceptions are predictive but rare

Many features “tie”

More assignment statements → lower frequency

Path length matters most
Conclusion

A formal model that statically predicts relative dynamic path execution frequencies

A generic tool (built using that model) that takes only the program source code (or bytecode) as input and produces

- for each method, an ordered list of paths through that method

The promise of helping other program analyses and transformations
Questions? Comments?
Evaluation by Benchmark

1.0 = perfect

0.67 = return all or return nothing

F-measure

Spec JVM Benchmark

intra-method

inter-method