ICSE 2009
Vancouver, BC

THE ROAD NOT TAKEN

Ray Buse
Wes Weimer

Estimating Path Execution Frequency Statically

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

Example


```
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


```
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

Path 1


```
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;
}
```

Path 2


```
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;
}
```

Path 3

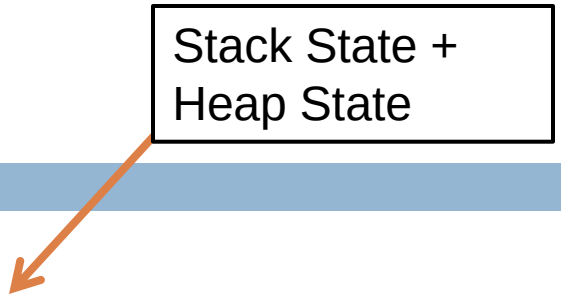

```
public V function(K k , V v)
{
    if ( v == null )
      throw new Exception();

    if ( c == x )
      restructure();

    i = k.h();

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

    return v;
}
```

HashTable: put


```
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;
}
```

*simplified from java.util.HashTable jdk6.0

Intuition



How a path modifies *program state* may correlate with it's 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
- 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

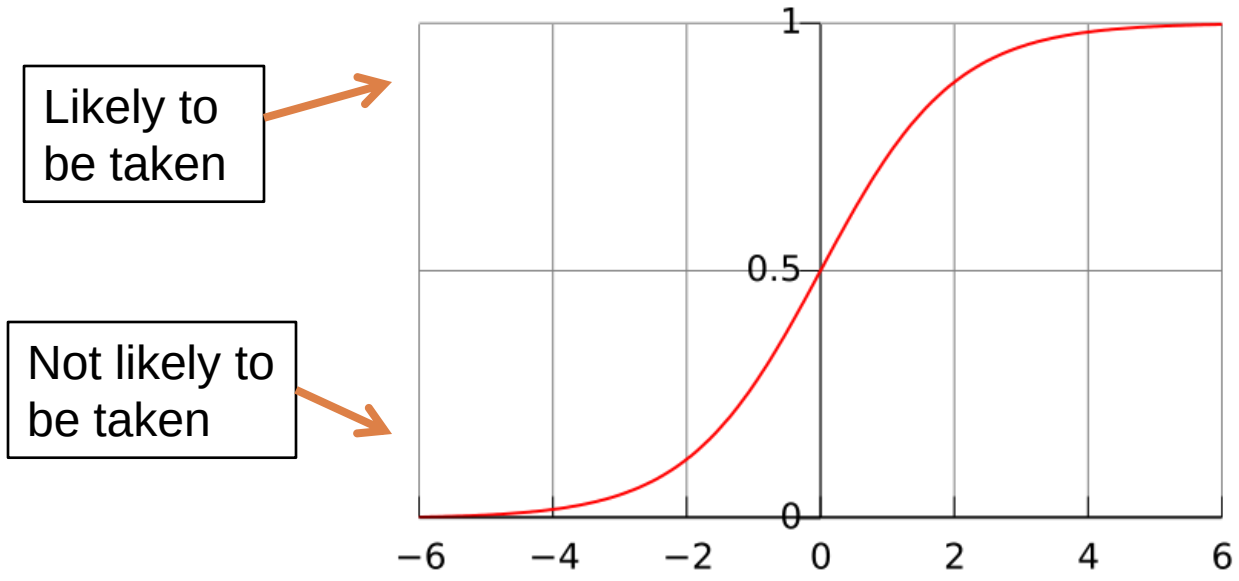
Count	Coverage	Feature
•		pointer comparisons
•		`new`
•		`this`
•		all variables
•		assignments
•		dereferences
•	•	fields
•	•	fields written
•	•	statements in invoked method
•		`goto` stmts
•		`if` stmts
•		local invocations
•	•	local variables
•		non-local invocations
•	•	parameters
•		`return` stmts
•		statements
•		`throw` stmts

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

Classifier: Logistic Regression

- Learn a logistic function to estimate the runtime frequency of a path



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

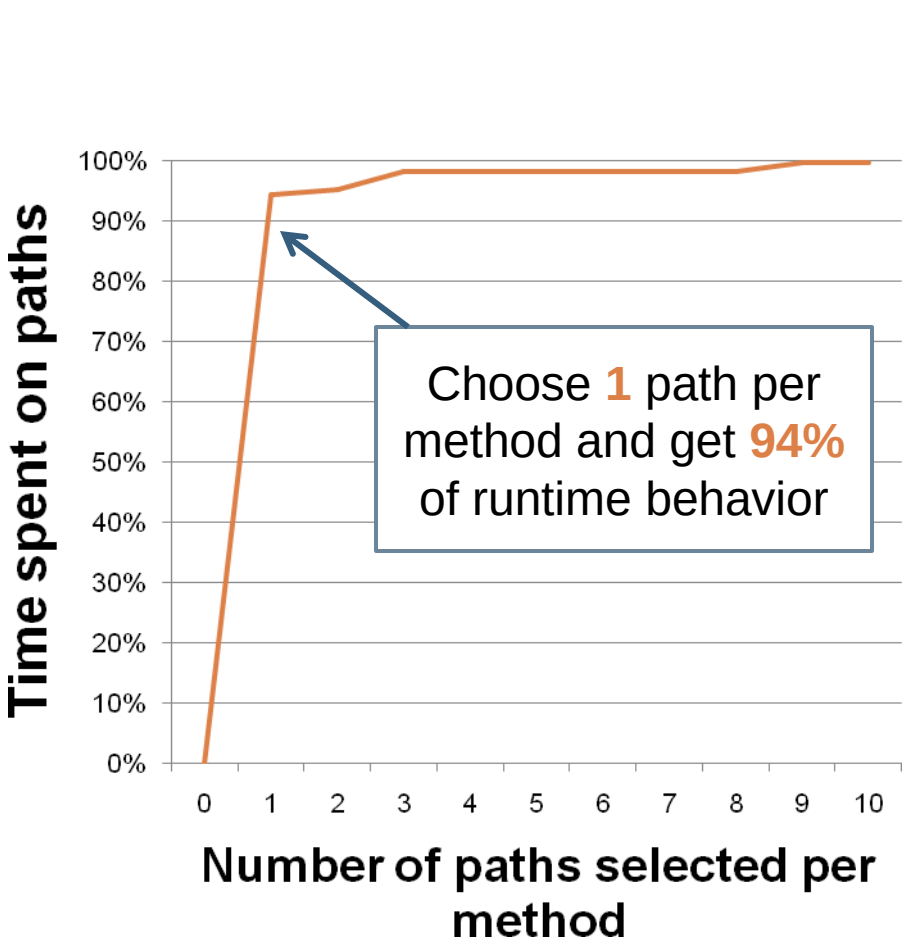
Input path {x_1, x_2 … x_n}

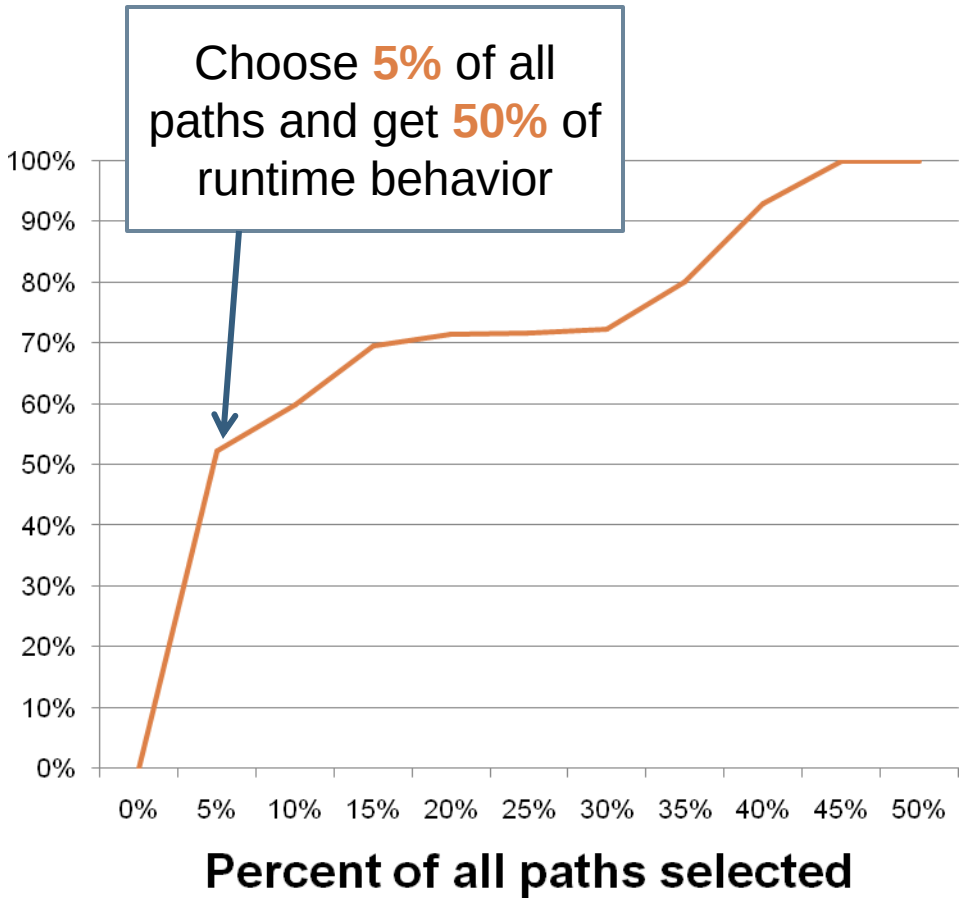
$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots + \beta_k x_k,$$

Model Evaluation

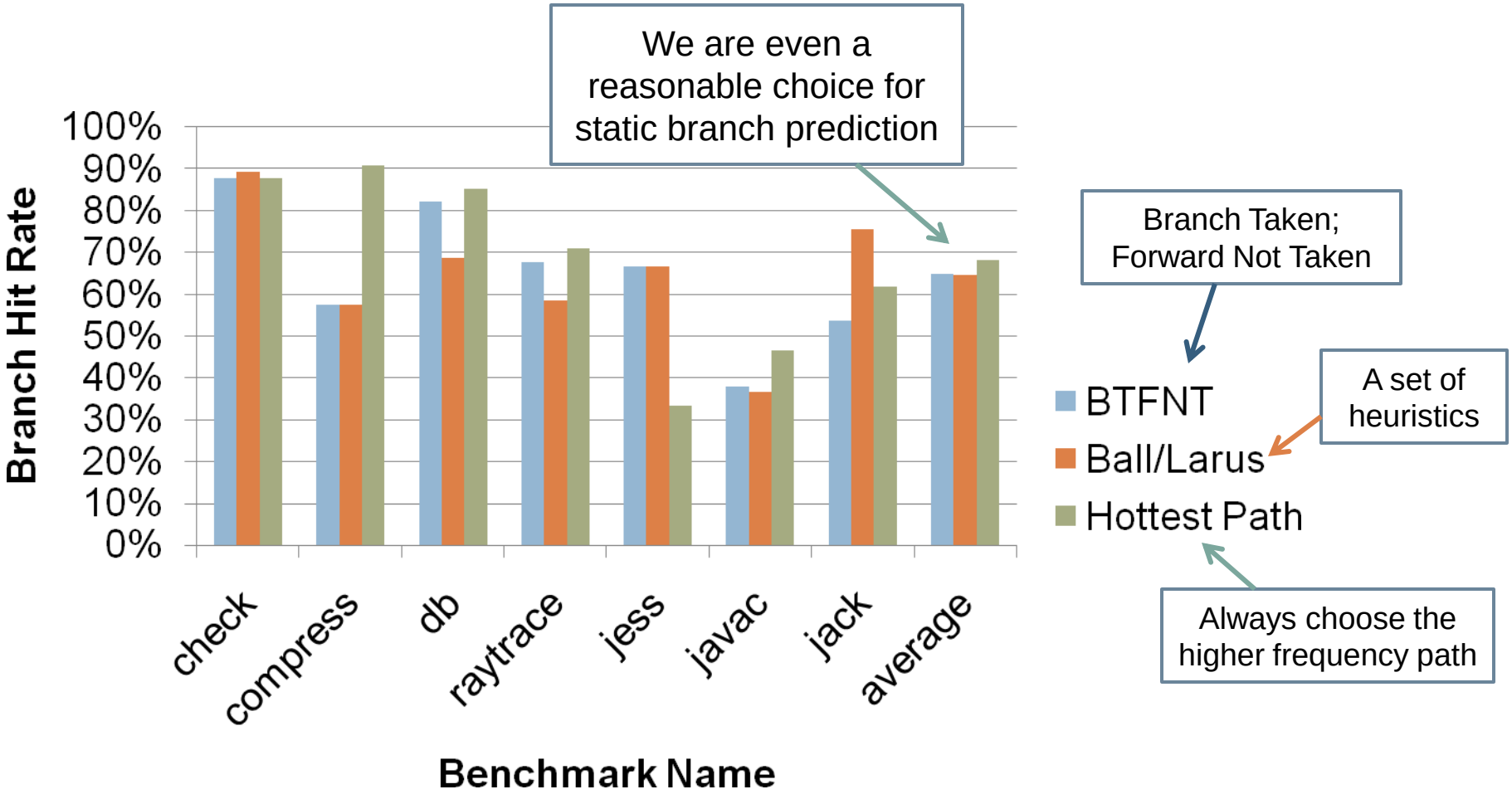
- Use the model to rank all static paths in the program
- Measure how much of total program runtime is spent:
  - On the top X paths for each method
  - On the top X% of all paths
- Also, compare to static branch predictors
- Cross validation on Spec JVM98 Benchmarks
  - When evaluating on one, train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

Evaluation: Top Paths

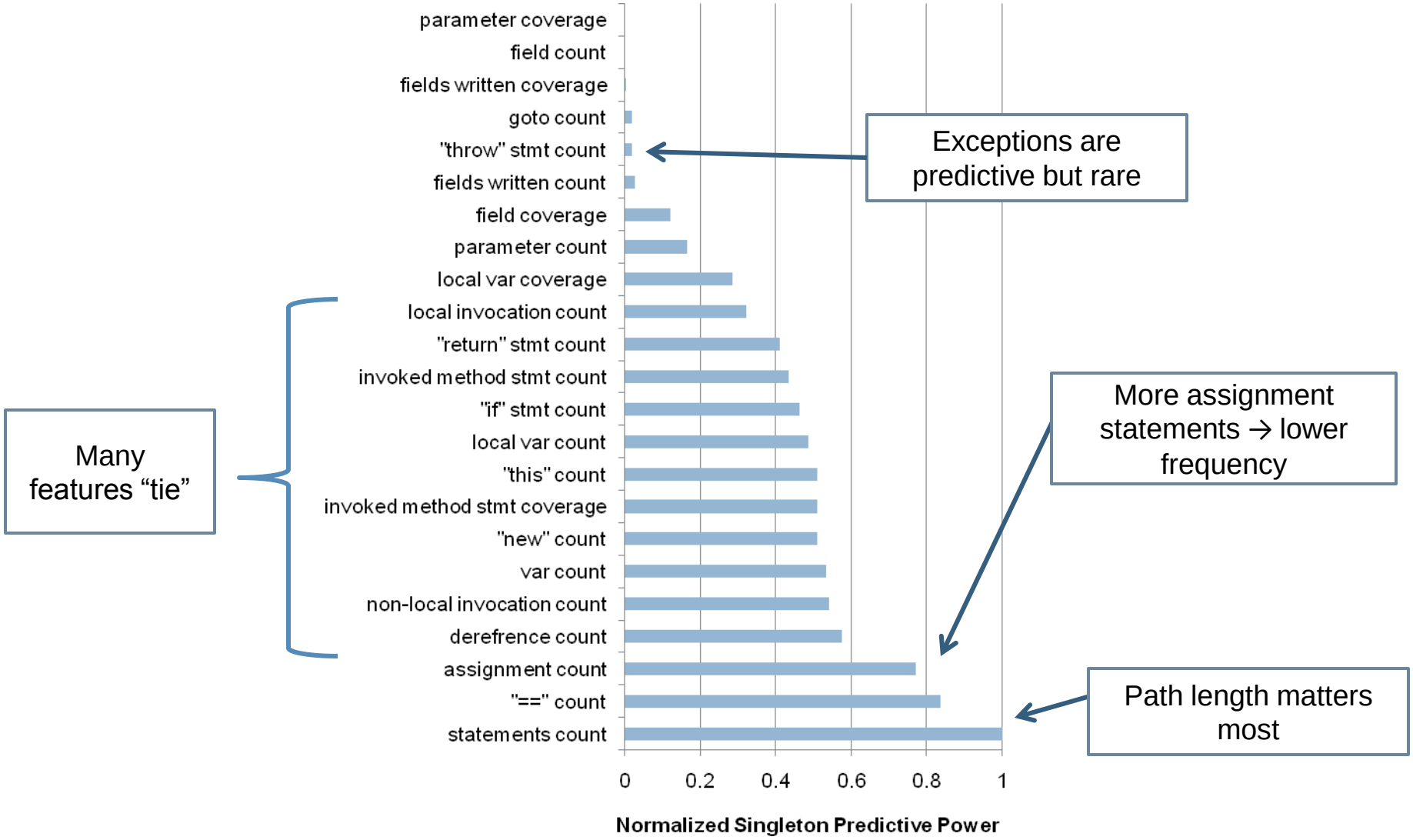




Evaluation: Static Branch Predictor



Model Analysis: Feature Power



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