Program Calling Context

Motivation

- Calling context enhances program understanding and dynamic analyses by providing a rich representation of program location
- Collecting calling context can be expensive
- The resulting representation of contexts can be bulky
Overview

- Probabilistic Calling Context [2]
- Precise Calling Context Encoding [3]

Probabilistic Calling Context[2]

Based on Mike Bond's slides[1]
Why Context Sensitivity?

- Static program location not enough

at com.mckoi.db.jdbcserver.JDBCInterface.executeQuery():213
Why Context Sensitivity?

• Static program location not enough

at com.mckoi.db.jdbcserver.JDBCInterface.execQuery():213
at com.mckoi.db.jdbc.MConnection.executeQuery():348
at com.mckoi.db.jdbc.MStatement.executeQuery():110
at com.mckoi.db.jdbc.MStatement.executeQuery():127
at Test.main():48

• Motivated by
  – Complex programs
  – Small methods
  – Virtual dispatch

Why Context Sensitivity?

• Static program location not enough

at com.mckoi.db.jdbcserver.JDBCInterface.execQuery():213
at com.mckoi.db.jdbc.MConnection.executeQuery():348
at com.mckoi.db.jdbc.MStatement.executeQuery():110
at com.mckoi.db.jdbc.MStatement.executeQuery():127
at Test.main():48

• Motivated by
  – Complex programs
  – Small methods
  – Virtual dispatch

C/Fortran method

Java/C# method
Context Is Nontrivial

<table>
<thead>
<tr>
<th>Program</th>
<th>API calls</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Call sites</td>
<td>Distinct contexts</td>
<td></td>
</tr>
<tr>
<td>antlr</td>
<td>4,184</td>
<td>128,627</td>
<td></td>
</tr>
<tr>
<td>bloat</td>
<td>3,306</td>
<td>600,947</td>
<td></td>
</tr>
<tr>
<td>chart</td>
<td>2,335</td>
<td>202,603</td>
<td></td>
</tr>
<tr>
<td>eclipse</td>
<td>9,611</td>
<td>226,020</td>
<td></td>
</tr>
<tr>
<td>fop</td>
<td>2,225</td>
<td>37,710</td>
<td></td>
</tr>
<tr>
<td>hsqldb</td>
<td>947</td>
<td>16,050</td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>1,830</td>
<td>628,048</td>
<td></td>
</tr>
<tr>
<td>luindex</td>
<td>654</td>
<td>102,556</td>
<td></td>
</tr>
<tr>
<td>lusearch</td>
<td>507</td>
<td>905</td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>1,890</td>
<td>847,108</td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>1,530</td>
<td>17,905</td>
<td></td>
</tr>
</tbody>
</table>

Example: Residual Testing

At production time, are there any untested behaviors, such as unexercised calling context?

```java
class SimpleWindow {
    close() {
        ...
    }
}

class EditorWindow {
    close() {
        ...
    }
}
```
Example: Residual Testing

At production time, are there any untested behaviors, such as unexercised calling context?

```java
inputHandler() {
    ...
    case CLICK_EXIT:
        w.checkUnsaved();
        w.close();
        ...
}
```

```java
class EditorWindow {
    close() {
        ...
    }
}
```

```java
class SimpleWindow {
    close() {
        ...
    }
}
```

```java
autoUpdate() {
    ...
    for all windows w
        w.close();
    ...
}
```

Bug!
Example: Residual Testing

At production time, are there any untested behaviors, such as unexercised calling context?

```java
class SimpleWindow {
    close() {
        ...
    }
}

class EditorWindow {
    close() {
        ...
    }
}
```

```java
inputHandler() {
    ...
    case CLICK_EXIT:
        w.checkUnsaved();
        w.close();
        ...
    }
}
```

```java
autoUpdate() {
    for all windows w
        w.close();
    ...
}
```

New behavior indicates bugs

Context sensitivity helps find new behavior

Bug!

Two-Phase Dynamic Analyses

Training

Behavior observed

Production

New or anomalous behavior detected
Probabilistic Calling Context

- Adds context sensitivity to dynamic analyses
- Maintains value representing context
  - Unique with high probability
  - New value \( \rightarrow \) new context \( \rightarrow \) walk stack
- High accuracy: <0.1% false negatives
- Low overhead: 3% overhead, 0-8% for clients

\[ f(V, cs) \]

- \( V \) is PCC value
- \( cs \) is call site ID
PCC Function

\[ f(V, cs) \]

- \( V \) is PCC value
- \( cs \) is call site ID

\[ V \leftarrow f(V, cs_1) \]

\[ V \leftarrow V_{\text{saved}} \]

\[ V \leftarrow f(V, cs_2) \]

\[ V \leftarrow V_{\text{saved}} \]

\[ 3V + cs \pmod{2^{32}} \]

- \( V \) is PCC value
- \( cs \) is call site ID
**PCC Function**

\[ f(V, cs) \equiv 3V + cs \ (mod \ 2^{32}) \]

- Cheap to compute
- Desirable properties:
  - Non-commutative
  - Composition efficient to compute

**Differentiating Similar Contexts**

\[ V \leftarrow 3V + cs_1 \]
\[ V \leftarrow 3V + cs_2 \]

... → A() → B() → ...

... → B() → A() → ...

\[ V \leftarrow 3V + cs_2 \]
\[ V \leftarrow 3V + cs_1 \]
Differentiating Similar Contexts

- Non-commutative
  \[ f(f(V, cs_1), cs_2) \neq f(f(V, cs_2), cs_1) \]

Efficiency at Inlined Calls
Efficiency at Inlined Calls

\[ V \leftarrow 3V + cs_1 \]
\[ V \leftarrow 3V + cs_2 \]

Efficiency at Inlined Calls

\[ V \leftarrow 3V + 3cs_1 \]
\[ V \leftarrow 3V + 3cs_2 \]
Efficiency at Inlined Calls

- Composition efficient to compute
  \[ f^n (V, cs) = 3^n V + \sum_i 3^i cs_i \]

Outline

- Introduction
- Previous approaches
- Maintaining the PCC value
- Evaluation
  - Methodology
  - Evaluating potential clients
  - Accuracy
  - Performance
Methodology

• Implementation in Jikes RVM 2.4.6
  – Available on Jikes RVM Research Archive
• Deterministic calling context profiling
  – Maintains CCT node at each call & return
• Benchmarks: DaCapo, SPEC JBB2000, SPEC JVM98
• Platform: 3.6 GHz Pentium 4 w/Linux

How Clients Use PCC

Record values
Training
Behavior observed

New value → new context → walk stack
Production
New or anomalous behavior detected
Evaluating Potential Clients

Memory overhead: proportional to contexts

Record values: Training  →  Global hash table
Behavior observed

Check values (no new values): Production
New or anomalous behavior detected

Record values: Training  →  Global hash table
Behavior observed

Check values (no new values): Production
New or anomalous behavior detected
Evaluating Potential Clients

- Anomaly-based intrusion detection
  - Check PCC value at system calls (Network, I/O, OS)
- Residual testing
  - Check PCC value at Java API calls (calls to java.*)
- Upper bound
  - Check PCC value at all calls

Ideal Accuracy

- PCC maps context to value
  - New PCC value $\rightarrow$ new context
  - Familiar PCC value $\rightarrow$ probably familiar context
## Ideal Accuracy

- PCC maps context to value
  - New PCC value → new context
  - Familiar PCC value → probably familiar context

<table>
<thead>
<tr>
<th>Distinct contexts</th>
<th>Expected conflicts (false negatives)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32-bit values</td>
</tr>
<tr>
<td>100,000</td>
<td>1 (0.0%)</td>
</tr>
<tr>
<td>1,000,000</td>
<td>116 (0.0%)</td>
</tr>
<tr>
<td>10,000,000</td>
<td>11,632 (0.1%)</td>
</tr>
<tr>
<td>100,000,000</td>
<td>1,155,170 (1.2%)</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>107,882,641 (10.8%)</td>
</tr>
<tr>
<td>10,000,000,000</td>
<td>6,123,623,065 (61.2%)</td>
</tr>
</tbody>
</table>
## Ideal Accuracy

- PCC maps context to value
  - New PCC value → new context
  - Familiar PCC value → probably familiar context

<table>
<thead>
<tr>
<th>Distinct contexts</th>
<th>Expected conflicts (false negatives)</th>
<th>32-bit values</th>
<th>64-bit values</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
<td>1 (0.0%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>1,000,000</td>
<td></td>
<td>116 (0.0%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>10,000,000</td>
<td></td>
<td>11,632 (0.1%)</td>
<td></td>
</tr>
<tr>
<td>100,000,000</td>
<td></td>
<td>1,155,170 (1.2%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td></td>
<td>107,882,641 (10.8%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>10,000,000,000</td>
<td></td>
<td>6,123,623,065 (61.2%)</td>
<td>3 (0.0%)</td>
</tr>
</tbody>
</table>
### PCC’s Accuracy

<table>
<thead>
<tr>
<th>Program</th>
<th>System calls</th>
<th>Java API calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>211,490</td>
<td>1,567</td>
</tr>
<tr>
<td>bloat</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>chart</td>
<td>63</td>
<td>62</td>
</tr>
<tr>
<td>eclipse</td>
<td>14,110</td>
<td>197</td>
</tr>
<tr>
<td>fop</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>hsqldb</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>jython</td>
<td>5,929</td>
<td>4,289</td>
</tr>
<tr>
<td>luindex</td>
<td>2,615</td>
<td>14</td>
</tr>
<tr>
<td>lusearch</td>
<td>141</td>
<td>11</td>
</tr>
<tr>
<td>pmd</td>
<td>1,045</td>
<td>25</td>
</tr>
<tr>
<td>xalan</td>
<td>137,895</td>
<td>59</td>
</tr>
</tbody>
</table>
### PCC’s Accuracy

<table>
<thead>
<tr>
<th>Program</th>
<th>All calls</th>
<th>Dynamic</th>
<th>Distinct</th>
<th>Conf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>490,363,211</td>
<td>1,006,578</td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>bloat</td>
<td>6,276,446,059</td>
<td>1,980,205</td>
<td>453</td>
<td></td>
</tr>
<tr>
<td>chart</td>
<td>908,459,469</td>
<td>845,432</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>eclipse</td>
<td>1,266,810,504</td>
<td>4,815,901</td>
<td>2,652</td>
<td></td>
</tr>
<tr>
<td>fop</td>
<td>44,200,446</td>
<td>174,955</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>hsqldb</td>
<td>877,680,667</td>
<td>110,795</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>5,326,949,158</td>
<td>3,859,545</td>
<td>1,738</td>
<td></td>
</tr>
<tr>
<td>luindex</td>
<td>740,053,104</td>
<td>374,201</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>lusearch</td>
<td>1,439,034,336</td>
<td>6,039</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>2,726,876,957</td>
<td>8,043,096</td>
<td>7,653</td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>10,083,858,546</td>
<td>163,205</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

### PCC’s Execution Time Overhead

- **DaCapo**: 3%
- **SPEC**: 0%
- **All**: 0%

Overhead (%) range from 0% to 60%.
Summary

- **PCC maintains calling context value**
  - New value indicates new behavior
- **Low overhead**
  - Maintaining PCC value adds 3%
  - Checking PCC value 0-8%
  - Memory overhead proportional to contexts
- **High accuracy**
  - Less than 0.1% false negative rate
- **PCC adds context sensitivity to clients that detect anomalous behavior**
Precise Calling Context Encoding [3]

Based on Slides[5]

Motivation

• Calling context is important for profiling, debugging, and event logging
• The prior work encodes each calling context into a hash value, but cannot decode the context out of the value
• Is it possible to encode calling contexts into numbers from which the contexts can be restored?
Background

- Efficient Path Profiling [4]
  - Naively, where do you want to instrument to get the path profile?
  - Is there any way to reduce instrumentation edges?

The Basic Idea

- If one node has only one outgoing edge, you don’t need to instrument the edge
- If different paths can be encoded to different numbers, you can restore the whole paths purely based on the number values
- A naïve example: Given the CFG, how do you map the values to paths?
Edge Value Assignment

• Assign each edge a value such that:
  – Sum of values along DAG path is unique, non-negative integer
  – Sum lies in range 0 \(\ldots\) num_paths - 1
• Simple linear-time algorithm
Edge Value Assignment

• Algorithm

foreach vertex v in reverse topological order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v -> w {
      Val(e) = NumPaths(v);
      NumPaths(v) = NumPaths(v) + NumPaths(w);
    }
  }
}

How to decode a path from a number?

• Algorithm

Given a path id Idx, set current node cur = A
while (cur != F) {
  take the path whose value val(e) is
  (1) smaller than Idx, and
  (2) the maximum value among all the
  values smaller than Idx
  Idx = Idx - val(e);
  cur = tgt(e);
}


Is the Ball-Larus algorithm applicable to Call Graph encoding?

- Yes. The BL algorithm uniquely identifies each path ending at different methods. However, paths ending at different methods can have the same values, if we know ending points.

```
foreach vertex v in topological order {
    if v is a root vertex {
        NumPaths(v) = 1;
    } else {
        NumPaths(v) = 0;
        for each edge e = w -> v {
            Val(e) = NumPaths(v);
            NumPaths(v) = NumPaths(w) + Val(e);
        }
    }
}
```

Edge Value Assignment

- Algorithm
BL vs. New Algorithm

How to decode a context from a number?
Encoding Cyclic CGs

• Introduce dummy edges

Given encoded context \((F, 1)\), does it correspond to path \((A \ C \ D \ F), (A \ B \ D \ F \ C \ D \ F), \text{ etc}?\)

<table>
<thead>
<tr>
<th>Path</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A \ C \ D \ F)</td>
<td>((1, F))</td>
</tr>
<tr>
<td>(A \ B \ D \ F \ C \ D \ F)</td>
<td>((-0, F \text{ to } C, 1, F))</td>
</tr>
</tbody>
</table>

Selective Reduction

• To fit IDs of contexts in 32-bit integers,
  – Profiling is used to identify hot and code call edges
  – Code edges are replaced with dummy edges such as sub-paths starting with these code edges can be separately encoded
Evaluation

- 19 C programs from SPECint 2000 benchmarks, and a set of open source programs

Comparison between BL and the new algorithm

<table>
<thead>
<tr>
<th>program</th>
<th>LOC</th>
<th>CG nodes</th>
<th>CG edges</th>
<th>recursions</th>
<th>fun pointers</th>
<th>our max ID</th>
<th>BL max ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp 2.8.7</td>
<td>6681</td>
<td>68</td>
<td>162</td>
<td>0</td>
<td>5</td>
<td>44</td>
<td>150</td>
</tr>
<tr>
<td>diff 2.8.7</td>
<td>15835</td>
<td>147</td>
<td>465</td>
<td>6</td>
<td>8</td>
<td>645</td>
<td>3160</td>
</tr>
<tr>
<td>ediff 2.8.7</td>
<td>7426</td>
<td>90</td>
<td>291</td>
<td>0</td>
<td>5</td>
<td>242</td>
<td>684</td>
</tr>
<tr>
<td>find 4.4.0</td>
<td>8551</td>
<td>567</td>
<td>1362</td>
<td>28</td>
<td>33</td>
<td>1682</td>
<td>5020</td>
</tr>
<tr>
<td>locate 4.4.0</td>
<td>28393</td>
<td>320</td>
<td>688</td>
<td>3</td>
<td>19</td>
<td>251</td>
<td>1029</td>
</tr>
<tr>
<td>grep 2.5.4</td>
<td>26821</td>
<td>195</td>
<td>665</td>
<td>17</td>
<td>10</td>
<td>17437</td>
<td>44558</td>
</tr>
<tr>
<td>tar 1.16</td>
<td>58301</td>
<td>791</td>
<td>2697</td>
<td>19</td>
<td>46</td>
<td>1865752</td>
<td>403510</td>
</tr>
<tr>
<td>make 3.80</td>
<td>25882</td>
<td>271</td>
<td>1294</td>
<td>61</td>
<td>6</td>
<td>551054</td>
<td>154313</td>
</tr>
<tr>
<td>alpine 2.0</td>
<td>155283</td>
<td>2860</td>
<td>26315</td>
<td>302</td>
<td>1570</td>
<td>1294067205*</td>
<td>4.5e+18</td>
</tr>
<tr>
<td>vim 6.0</td>
<td>194450</td>
<td>2365</td>
<td>15822</td>
<td>1124</td>
<td>27</td>
<td>4291329441*</td>
<td>8.7e+18</td>
</tr>
<tr>
<td>104.gzip</td>
<td>11121</td>
<td>154</td>
<td>426</td>
<td>0</td>
<td>2</td>
<td>536</td>
<td>1283</td>
</tr>
<tr>
<td>176.vpr</td>
<td>29807</td>
<td>327</td>
<td>1328</td>
<td>0</td>
<td>2</td>
<td>1848</td>
<td>13047</td>
</tr>
<tr>
<td>176.gcc</td>
<td>340501</td>
<td>2255</td>
<td>22582</td>
<td>1801</td>
<td>128</td>
<td>4294038599*</td>
<td>2.1e+18</td>
</tr>
<tr>
<td>181.mcf</td>
<td>4820</td>
<td>93</td>
<td>208</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>90</td>
</tr>
<tr>
<td>186.crafty</td>
<td>42203</td>
<td>179</td>
<td>1533</td>
<td>17</td>
<td>0</td>
<td>184589</td>
<td>650779</td>
</tr>
<tr>
<td>197.parser</td>
<td>27371</td>
<td>381</td>
<td>1676</td>
<td>125</td>
<td>0</td>
<td>2734</td>
<td>14066</td>
</tr>
<tr>
<td>255.vortex</td>
<td>102987</td>
<td>980</td>
<td>7697</td>
<td>41</td>
<td>15</td>
<td>4294966803*</td>
<td>1.5e+13</td>
</tr>
<tr>
<td>256.hexj2</td>
<td>8014</td>
<td>133</td>
<td>386</td>
<td>0</td>
<td>0</td>
<td>131</td>
<td>600</td>
</tr>
<tr>
<td>330.twolf</td>
<td>40372</td>
<td>240</td>
<td>1386</td>
<td>9</td>
<td>0</td>
<td>1051</td>
<td>4766</td>
</tr>
</tbody>
</table>

*Selective reduction is applied to 288 nodes in alpine 2.0, 360 in 176.gcc, 33 in 255.vortex, and 877 in vpr.
Observations

- BL usually instruments outgoing edges when there are multiple.
- The new algorithm usually instruments incoming edges when there are multiple.
- It seems that the latter instrumentation is more efficient than the former one.
  - The intuition behind is there are more methods with multi-method callees than methods with multi-method callers.
- No experiment focuses on the instrumentation comparison.

Stack Depth & Encountered Dynamic Contexts

<table>
<thead>
<tr>
<th>programs</th>
<th>Max Depth ours</th>
<th>Max Depth plain</th>
<th>90% Depth ours</th>
<th>90% Depth plain</th>
<th>Dynamic contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp 2.8.7</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>diff 2.8.7</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>sed 2.8.7</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>find 4.4.0</td>
<td>2</td>
<td>12</td>
<td>12</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td>locate 4.4.0</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>grep 2.5.4</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>tar 1.16</td>
<td>3</td>
<td>40</td>
<td>3</td>
<td>1366</td>
<td></td>
</tr>
<tr>
<td>make-3.80</td>
<td>6</td>
<td>82</td>
<td>5</td>
<td>43</td>
<td>1799</td>
</tr>
<tr>
<td>alpine 2.0</td>
<td>11</td>
<td>29</td>
<td>6</td>
<td>18</td>
<td>7575</td>
</tr>
<tr>
<td>vim 6.0</td>
<td>10</td>
<td>31</td>
<td>5</td>
<td>10</td>
<td>3220</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>7</td>
<td>258</td>
</tr>
<tr>
<td>175.vpr</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>6</td>
<td>1553</td>
</tr>
<tr>
<td>176.gcc</td>
<td>19</td>
<td>136</td>
<td>2</td>
<td>15</td>
<td>169390</td>
</tr>
<tr>
<td>181.mcf</td>
<td>14</td>
<td>42</td>
<td>0</td>
<td>2</td>
<td>12920</td>
</tr>
<tr>
<td>186.crafty</td>
<td>34</td>
<td>41</td>
<td>10</td>
<td>23</td>
<td>271083471</td>
</tr>
<tr>
<td>197.parser</td>
<td>36</td>
<td>73</td>
<td>11</td>
<td>28</td>
<td>3623011</td>
</tr>
<tr>
<td>255.vortex</td>
<td>7</td>
<td>43</td>
<td>2</td>
<td>12</td>
<td>205004</td>
</tr>
<tr>
<td>256.tzip2</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>96</td>
</tr>
<tr>
<td>300.twolf</td>
<td>4</td>
<td>11</td>
<td>0</td>
<td>5</td>
<td>971</td>
</tr>
<tr>
<td>Average</td>
<td>8.78</td>
<td>39.22</td>
<td>2.17</td>
<td>13.72</td>
<td>1780292</td>
</tr>
</tbody>
</table>

Table 2: Dynamic context characteristics.
Summary

- Calling context is important for program analysis and debugging
- Both instrumentation overhead and memory overhead (how to encode the context) are important
- Can you think about any approach to further reduce instrumentation effort?

References