Practical Virtual Method Resolution for Java

Vijay Sundaresan
Laurie Hendren, Chrislain Razafimahefa,
Raja Vallée-Rai, Patrick Lam,
Etienne Gagnon and Charles Godin
(Winghong Felix Kwok)

Sable Research Group, McGill University
Montreal, Canada
www.sable.mcgill.ca
Outline

- What is Virtual Method Resolution? Why?
- The Soot Framework
- Simple Existing Techniques (CHA and RTA)
- The Quest: better accuracy with only one iteration
- Solution: *Variable Type Analysis*
- Experimental Results
- Related Work and Conclusions
Virtual Method Resolution
Which methods might be called at run-time?

```java
public class A {
    public void m(int p1, ..., int pn) {
        System.out.println(p1+...+pn);
    }
}

public class B {
    public void m(int p1, ..., int pn) {
        ...
    }
}

public class C {
    public void m(int p1, ..., int pn) {
        ...
    }
}
```
Benefits of resolving virtual method calls

```java
public class A {
    public void m(int p1, ..., int pn) {
        System.out.println(p1+...+pn);
    }
}
```

**Devirtualize**

```java
public class A {
    public static void m(a1, a2, ..., an) {
        System.out.println(a1+...+an);
    }
}
```

**Inline**

```java
{ int a1, a2, ..., an;

    System.out.println(a1+...+an);

}
```
A Conservative Call Graph

Entry Points

Method

Call edge

Potentially Polymorphic Call Site
Improving the call graph

Reachable Method

Unreachable Method

Necessary call edge

Potentially polymorphic call site

Call edge that may be eliminated
Pruned Call Graph

Entry Points

Good Call Graph

Minimize:
- Number of reachable methods
- Number of call edges
- Number of potentially polymorphic call sites

Reachable Method

Necessary call edge

Potentially Polymorphic Call Site
Implmented using the Soot framework
(see www.sable.mcgill.ca/soot and OOPSLA posters)

source

Java SML Scheme Eiffel source source source source
javac MLJ KAWA SmallEiffel

class files

SOOT

Produce Jimple 3-address IR

Analyze and Optimize

Generate Bytecode

Optimized class files

Interpreter JIT Adaptive Engine Ahead-of-Time Compiler
The Jimple Typed 3-address Representation

- there is no expression stack;
- each statement has a simple three-address form;
- variables are split by U/D - D/U webs; and
- each variable has a declared type that has been inferred from the bytecode (Gagnon and Hendren, SAS 2000).
Existing Simple Methods for Virtual Method Call Resolution

What do we know about the type of receiver o???
Using the declared type: Class Hierarchy Analysis

```java
public class A {
    public void m (int p1, ..., int pn) {
        System.out.println(p1+...+pn);
    }
}

public class B {
    public void m (int p1, ..., int pn) {
    ...
    }
}

public class C {
    public void m (int p1, ..., int pn) {
    ...
    }
}

Object
  D
  C
  A
  B

C o;
o.m(a1,a2,...,an)
...
Class Hierarchy Analysis (CHA) (Example 2)

```java
public class A {
    public void m (int p1, ..., int pn) {
        D o;
        o.m(a1,a2,...,an);
    }
}

public class B {
    public void m (int p1, ..., int pn) {
        ... }
}

public class C {
    public void m (int p1, ..., int pn) {
        ... }
}

public class D {
    o.m(a1,a2,...,an);
}
```

Object Diagram:
- Object
  - D
  - C
  - A
  - B

Sable Research Group
Using the types of allocated objects: Rapid Type Analysis (RTA)
Bacon and Sweeney (1996)

{ int a1, a2, ..., an;
  D o;
  o.m(a1,a2,...,an)
... 
}

public class A {
  public void m (int p1, ..., int pn) {
    System.out.println(p1+...+pn);
  }
  ... 
}

public class B {
  public void m (int p1, ..., int pn) {
    ... 
  }
  ... 
}

public class C {
  public void m (int p1, ..., int pn) {
    ... 
  }
  ... 
}

Objects Allocated
{ Object, A, C }
Quest: Improve upon RTA, restrict the analysis to one iteration

- RTA assumes that all allocated objects can reach a receiver.
- Want to provide a more accurate approximation;
- by tracking assignments from allocation sites, to method invocations.

```java
public class B {
    c.m();   // X.m()
}

public class A {
    public void f (C c) {
        c.m();   // X.m()
    }
}

public class B {
    public void f (C c) {
        c.m();   // X.m()
    }
}

a = new X();
...
b = a;
...
o.f(b);
```
Solution: Variable Type Analysis (VTA)

```java
public class A {
   public void f (C c) {
      c.m();   // X.m()
   }
}
public class B {
   public void f (C c) {
      c.m();   // X.m()
   }
}
```

Initial Type Propagation Graph

Final Type Propagation Graph
Three Steps in VTA

1. Form initial conservative call graph (CHA, RTA, VTA).

2. Build type propagation graph.

3. Solve type propagation graph in one iteration.
Building the Type Propagation Graph

Assuming, statement is in class C, method m:

\[ a = b; \]
\[ a[i] = b; \]
\[ a = b[i]; \]

Assuming field f is declared in class A:

\[ o.f = b; \]

If either left or right side is Object or Array type:

\[ a = b; \]
Building the Type Propagation Graph - method calls

Assuming the initial call graph of:

```
class X {
    D f (A a) {
        ....;
        return(r);
    }
}

class Y {
    D f (A a) {
        ....;
        return(r);
    }
}
```

```java
C.m.o
X.f.this
C.m.p
X.f.a
C.m.q
X.f.return
Y.f.this
Y.f.a
Y.f.return
q = o.f(p);
```
Propagating Types

1. For each statement of the form \texttt{x = new A();}, initialize the node for \texttt{x} with the type \texttt{A}.

2. Collapse strongly connected components, forming a DAG.

3. Propagate types on resulting DAG in one topological sweep.
Building the Type Propagation Graph

(a) Program

```
A a1, a2, a3;
B b1, b2, b3;
C c;

a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;
```

(b) Nodes and Edges
Propagating Types

```java
a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();
```

(c) Initial Types

(d) Strongly-connected components

(e) final solution
A Coarser Approximation: Declared Type Analysis

(a) Program

A a1, a2, a3;
B b1, b2, b3;
C c;

a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;

(b) Nodes and Edges

(c) Initial Types

(d) Strongly-connected components

(e) final solution
Tradeoffs to ensure one iteration and reasonably sized graph

- Simple solution to aliasing problem.

- No killing based on casts or declared type during propagation. However, filtering based on declared type is performed after propagation.

- Pessimistic because it starts with a conservative call graph. We can start with a CHA- or RTA-based call graph, or we can run VTA twice, using the first run to compute a better call graph.
Experimental Results

- Measure Benchmarks
  - Amount of code, division between library and user (benchmark) code.
  - Characteristics of conservative call graph (built using CHA)

- Static Improvements in the Conservative Call Graph by applying RTA, DTA and VTA
  - Percent nodes removed.
  - Percent edges removed.
  - Percent of potentially polymorphic call sites resolved/eliminated.

- Dynamic Study of Monomorphic Virtual Calls
## Benchmark Characteristics

<table>
<thead>
<tr>
<th></th>
<th>Total # Stmts.</th>
<th>Benchmark Only # Stmts</th>
<th># Classes</th>
<th># Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>raytrace</td>
<td>49239</td>
<td>5347</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>jack</td>
<td>55107</td>
<td>11215</td>
<td>62</td>
<td>5</td>
</tr>
<tr>
<td>javac</td>
<td>69585</td>
<td>25304</td>
<td>177</td>
<td>5</td>
</tr>
<tr>
<td>sablecc</td>
<td>68575</td>
<td>24621</td>
<td>298</td>
<td>13</td>
</tr>
<tr>
<td>soot</td>
<td>63506</td>
<td>33396</td>
<td>497</td>
<td>34</td>
</tr>
<tr>
<td>pizza</td>
<td>73130</td>
<td>42805</td>
<td>207</td>
<td>11</td>
</tr>
</tbody>
</table>
## Conservative Call Graph Characteristics (CHA)

<table>
<thead>
<tr>
<th></th>
<th>Total # Nodes</th>
<th>Benchmark Only # Nodes</th>
<th># Call Sites</th>
<th>(% Poly.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>raytrace</td>
<td>1729</td>
<td>207</td>
<td>2049</td>
<td>(0.6%)</td>
</tr>
<tr>
<td>jack</td>
<td>1857</td>
<td>337</td>
<td>3068</td>
<td>(12.9%)</td>
</tr>
<tr>
<td>javac</td>
<td>2821</td>
<td>1188</td>
<td>6781</td>
<td>(12.5%)</td>
</tr>
<tr>
<td>sablecc</td>
<td>3737</td>
<td>1955</td>
<td>6809</td>
<td>(13.1%)</td>
</tr>
<tr>
<td>soot</td>
<td>2828</td>
<td>2001</td>
<td>10615</td>
<td>(14.6%)</td>
</tr>
<tr>
<td>pizza</td>
<td>2660</td>
<td>1756</td>
<td>11692</td>
<td>(4.9%)</td>
</tr>
</tbody>
</table>
Percentage Methods Removed From Conservative Call Graph
(Whole Application)
Percentage Edges Removed From Conservative Call Graph
(whole application)

Raytrace  | Jack  | Javac  | Sablecc | Soot  | Pizza
---       | ---   | ---    | ---     | ---   | ---
RTA       | DTA   | VTA    | RTA     | DTA   | VTA
0%        | 10%   | 20%    | 15%     | 5%    | 10%
10%       | 20%   | 30%    | 15%     | 5%    | 10%
20%       | 30%   | 40%    | 25%     | 10%   | 20%
30%       | 40%   | 50%    | 30%     | 15%   | 30%
40%       | 50%   | 60%    | 40%     | 20%   | 40%
50%       | 60%   | 70%    | 50%     | 30%   | 50%
60%       | 70%   | 80%    | 60%     | 40%   | 60%

Sable Research Group
Percentage Methods Removed From Conservative Call Graph
(Benchmark only)
Percentage Edges Removed From Conservative Call Graph (Benchmark Only)
Percentage Potentially Polymorphic Calls Resolved from Conservative Call Graph (Benchmark only)
Percentage Virtual Method Calls that resolve to Exactly One Method at Run-time (Benchmark only)

[Bar chart showing percentage of virtual method calls resolving to exactly one method for different benchmarks: raytrace, jack, javac, sablecc, soot, pizza. Each benchmark is represented by a color-coded bar indicating CHA, RTA, VTA, and Profile.]
Related Work

- Many more expensive techniques.
- Inexpensive techniques include:
  - Diwan, Moss and McKinley (OOPSLA 96);
  - DeFouw, Grove and Chambers (POPL 98) *(merge nodes after visiting n times)*;
  - Tip and Palsberg (OOPSLA 00) *(restrict number of sets to be approximated)*; and
  - Ishizaki, Kawahito, Yasue, Komatsu and Nakatani (OOPSLA 00) *(devirtualization in JITs).*
Conclusions

- Variable Type Analysis (VTA) builds a type propagation graph and solves it in one pass, no iteration.

- VTA resolves (to one method) significantly more potentially polymorphic call sites than RTA.

- VTA is available in the newest release of Soot. Soot is a publically-available framework available from

  www.sable.mcgill.ca/soot