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

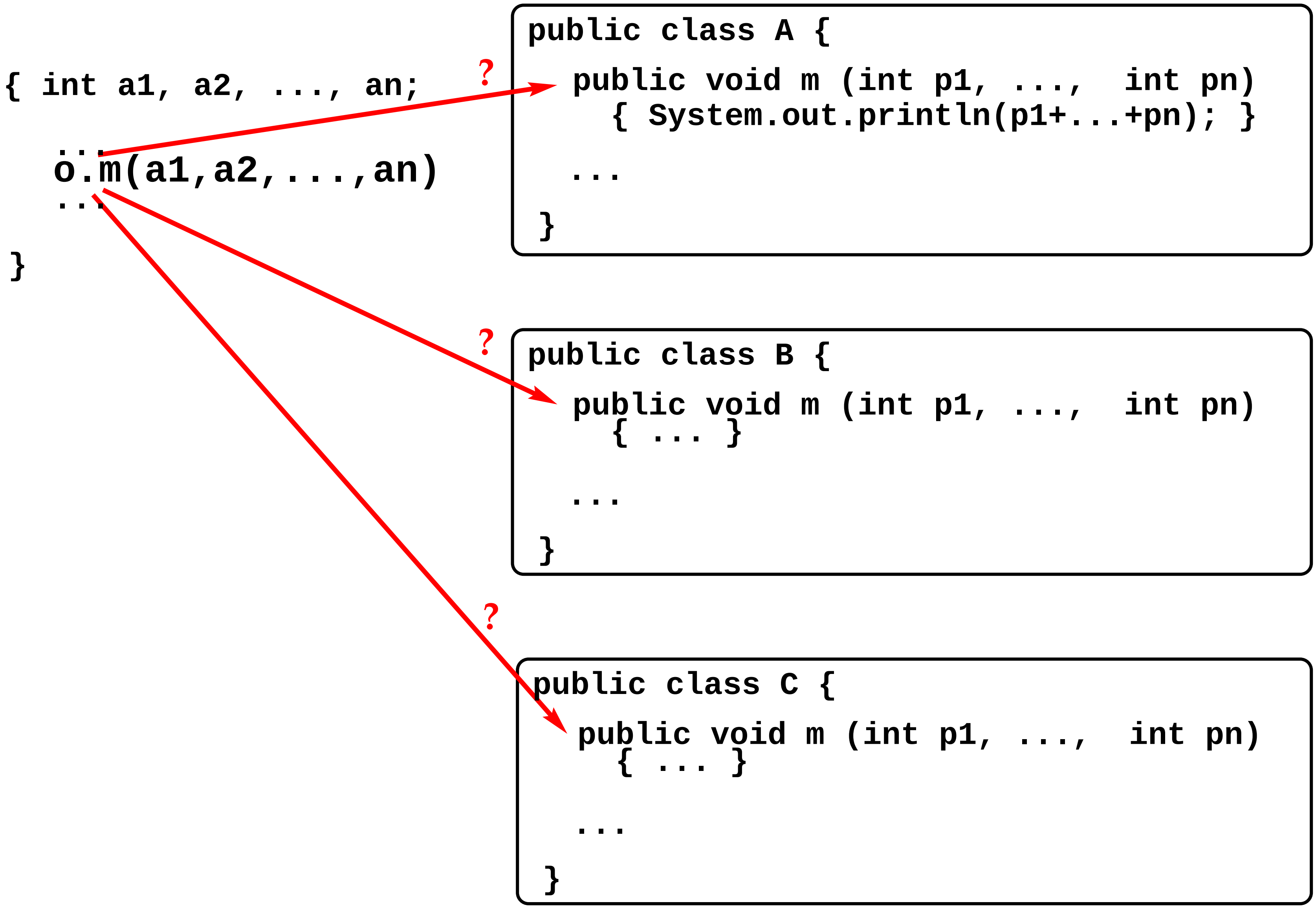
Montreal, Canada

`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?



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

    ...
   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)
    { ... }
  ...
 }
```

Benefits of resolving virtual method calls

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

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

Devirtualize

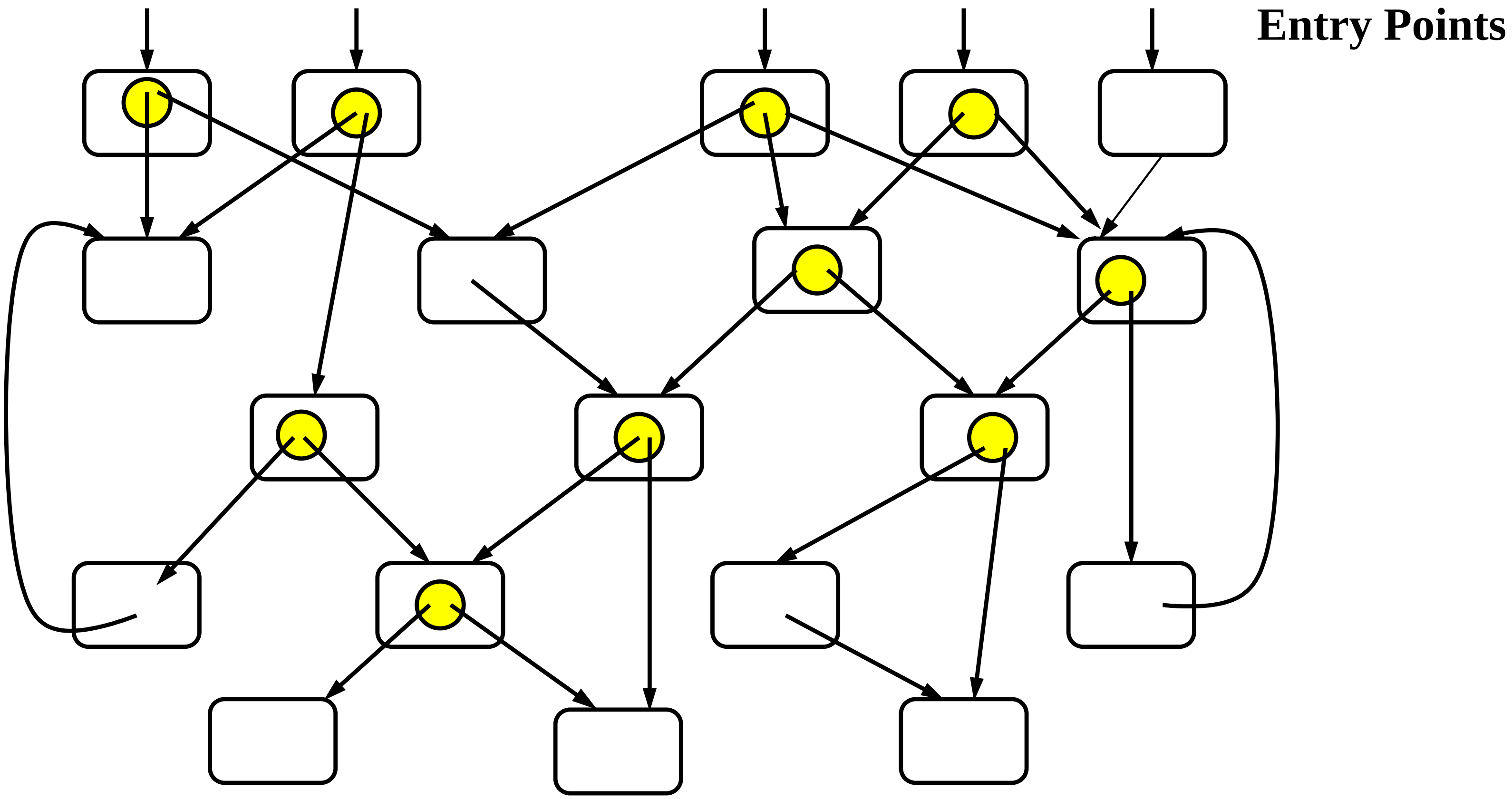
```
{ int a1, a2, ..., an;
    ...
    m(a1,a2,...,an);
    ...
}
```

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

Inline

```
{ int a1, a2, ..., an;
    ...
    System.out.println(a1+...+an);
    ...
}
```

A Conservative Call Graph

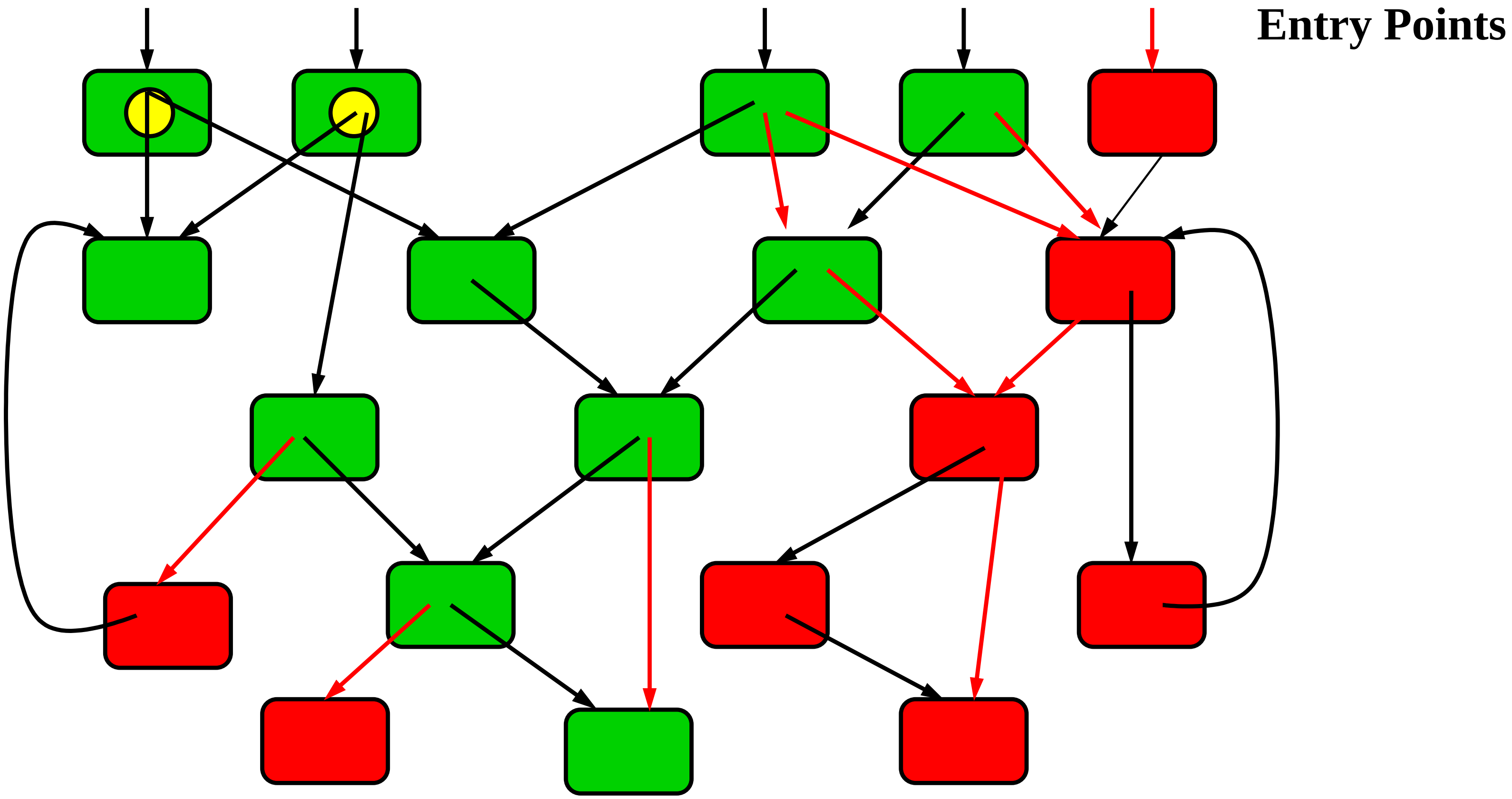


▭ **Method**

→ **Call edge**

● **Potentially Polymorphic Call Site**

Improving the call graph



🟩 **Reachable Method**

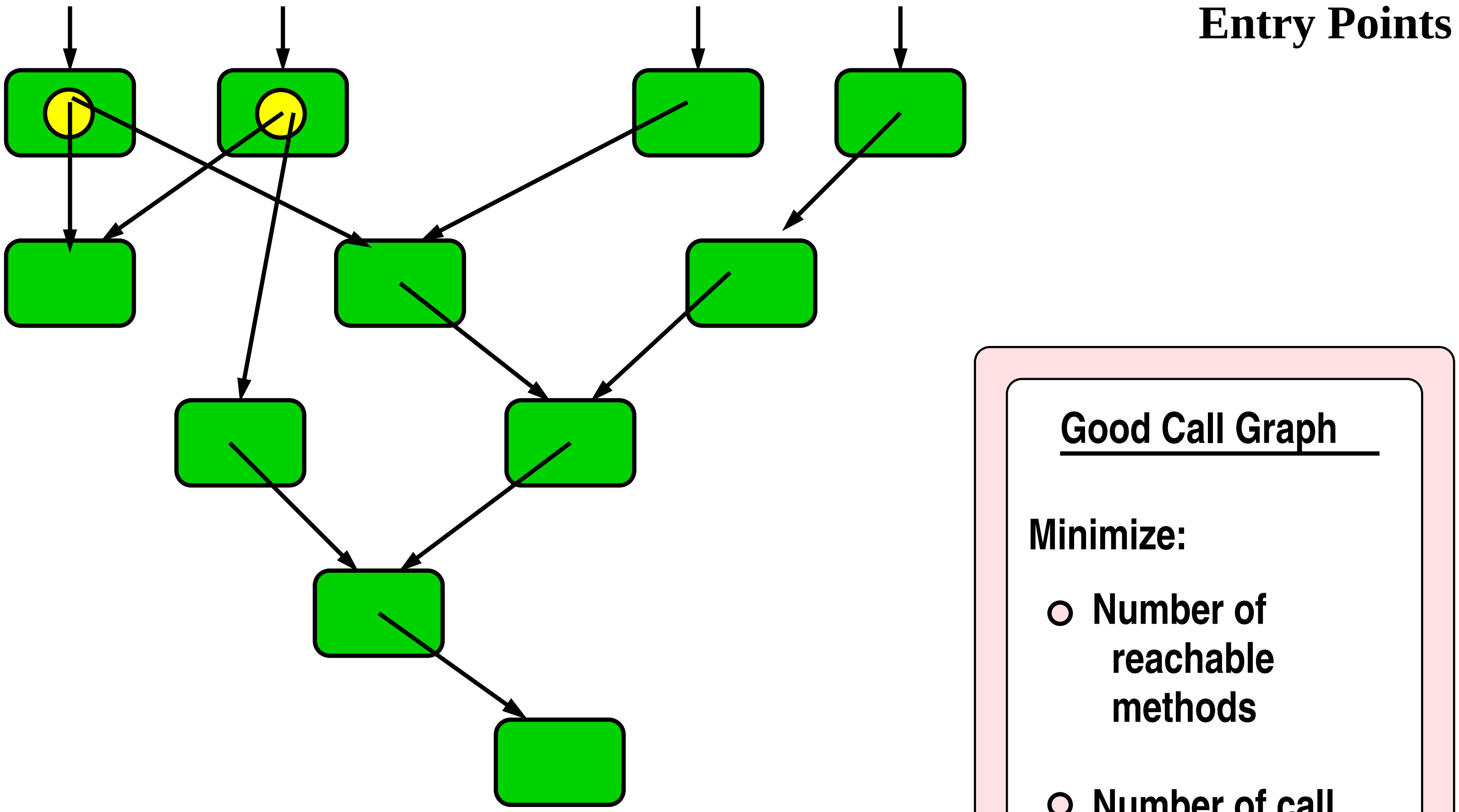
🟥 **Unreachable Method**

→ **Necessary call edge**

🟡 **Potentially polymorphic call site**

→ **Call edge that may be eliminated**

Pruned Call Graph



🟩 **Reachable Method**

→ **Necessary call edge**

🟡 **Potentially Polymorphic Call Site**

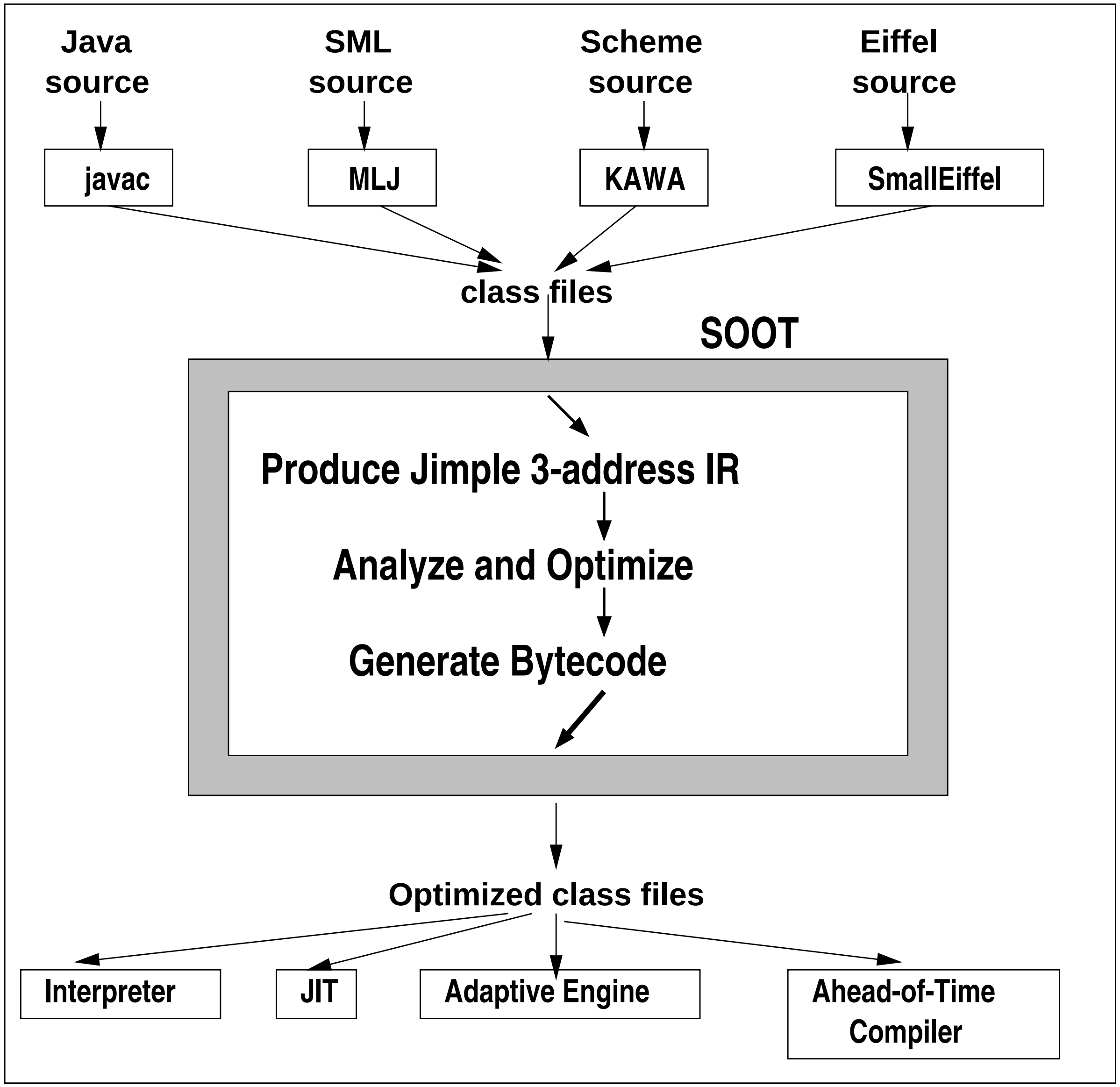
Good Call Graph

Minimize:

- ○ Number of reachable methods
- ○ Number of call edges
- ○ Number of potentially polymorphic call sites

Implemented using the Soot framework

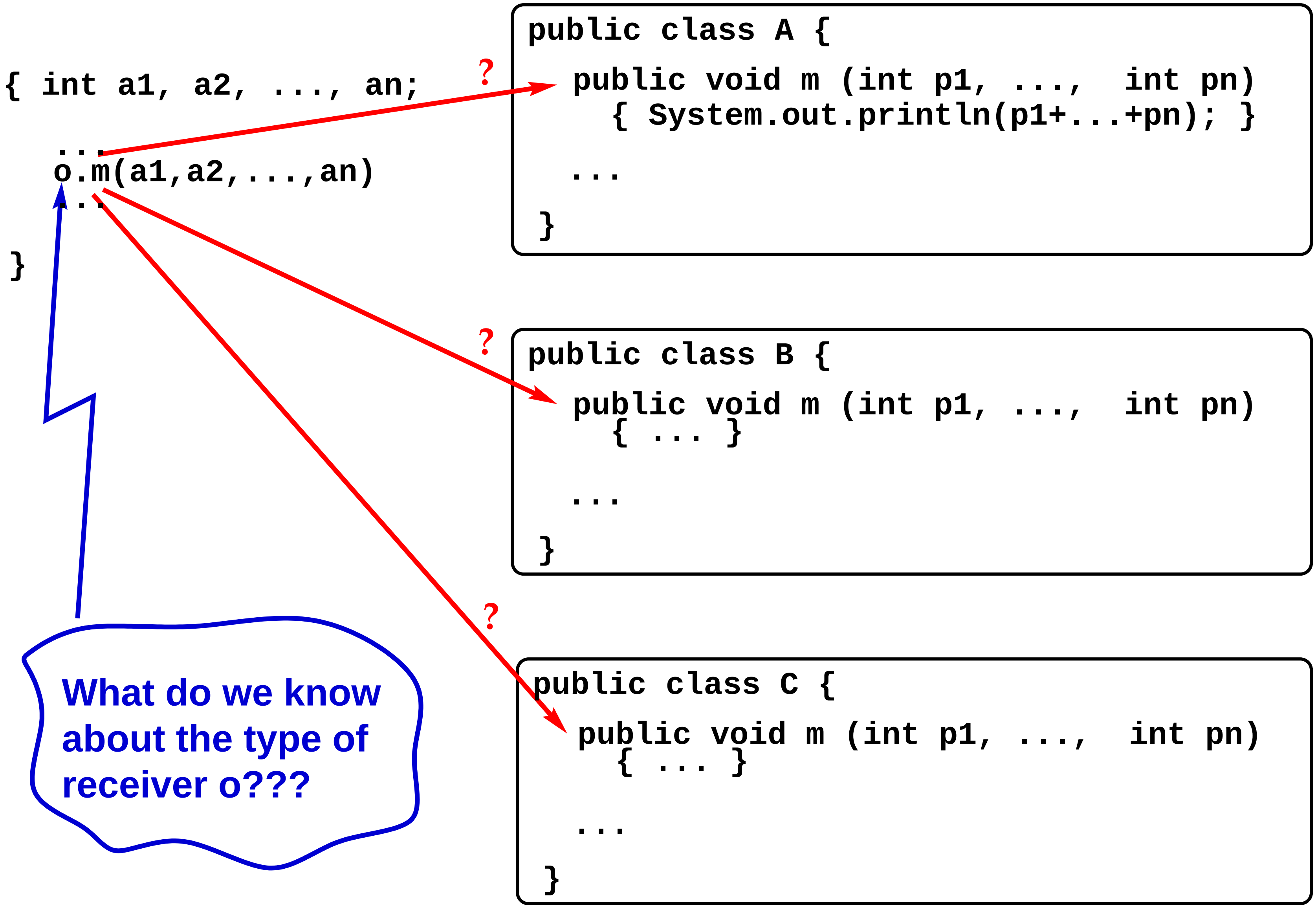
(see `www.sable.mcgill.ca/soot` and OOPSLA posters)



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



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

   ...
   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)
    { ... }

  ...
 }
```

What do we know about the type of receiver o???

Using the declared type: Class Hierarchy Analysis

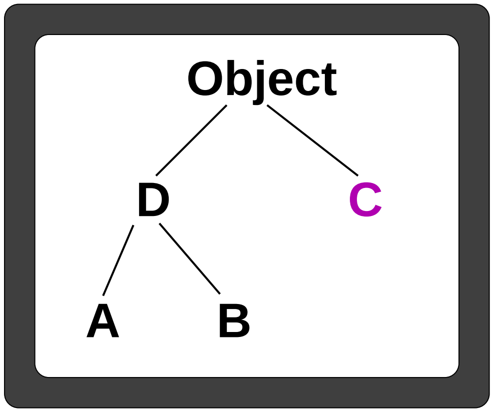
Dean, Grove and Chambers (1995), Fernandez (1995)

```
{ int a1, a2, ..., an;
    C o;
    ...
    o.m(a1,a2,...,an)
    ...

}
```

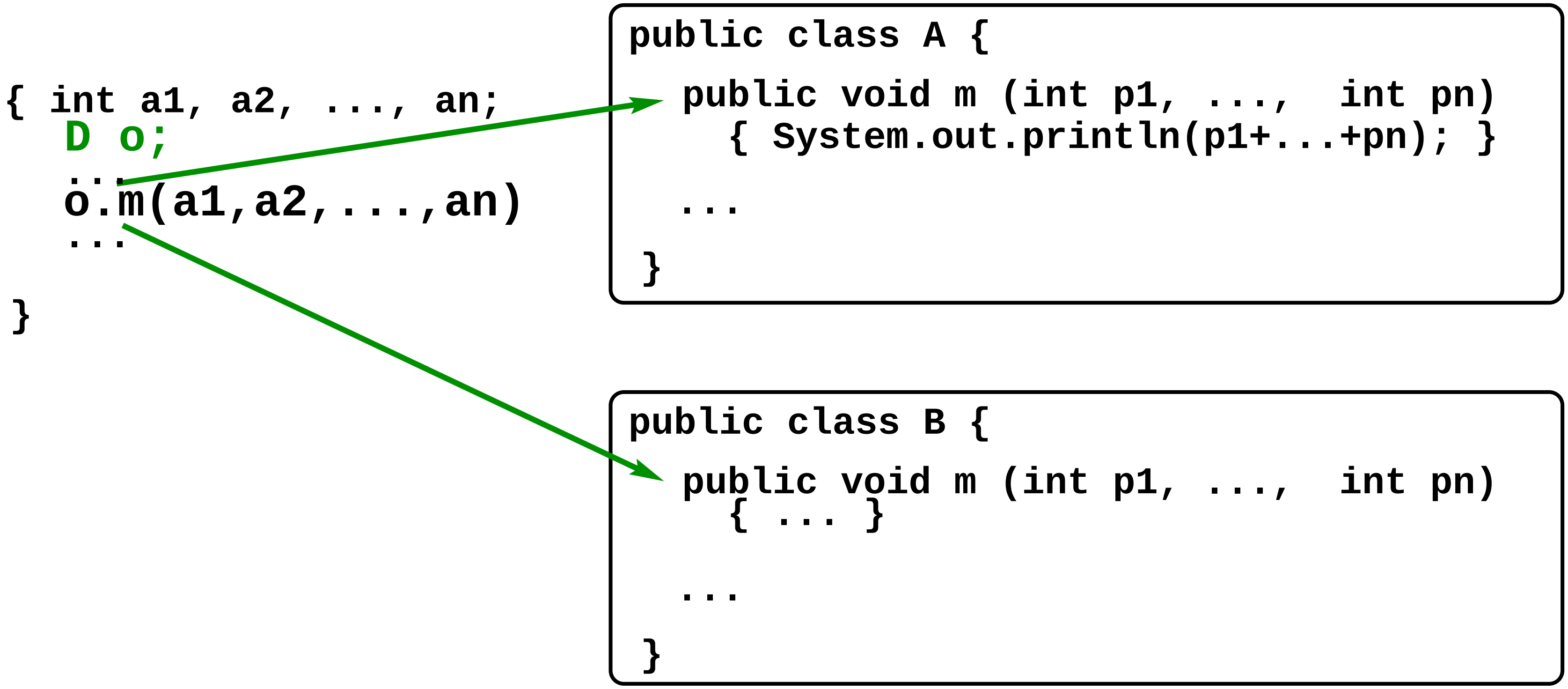
```
public class A {
   public void m (int p1, ...,  int pn)
     { System.out.println(p1+...+pn); }

   ...
 }
```

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

   ...
 }
```

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

   ...
 }
```



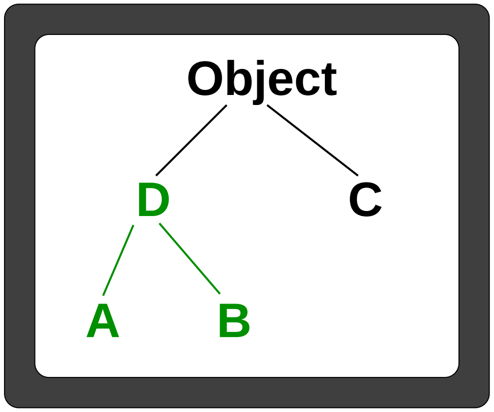
Class Hierarchy Analysis (CHA) (Example 2)



```
{ 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); }

  ...
 }
```

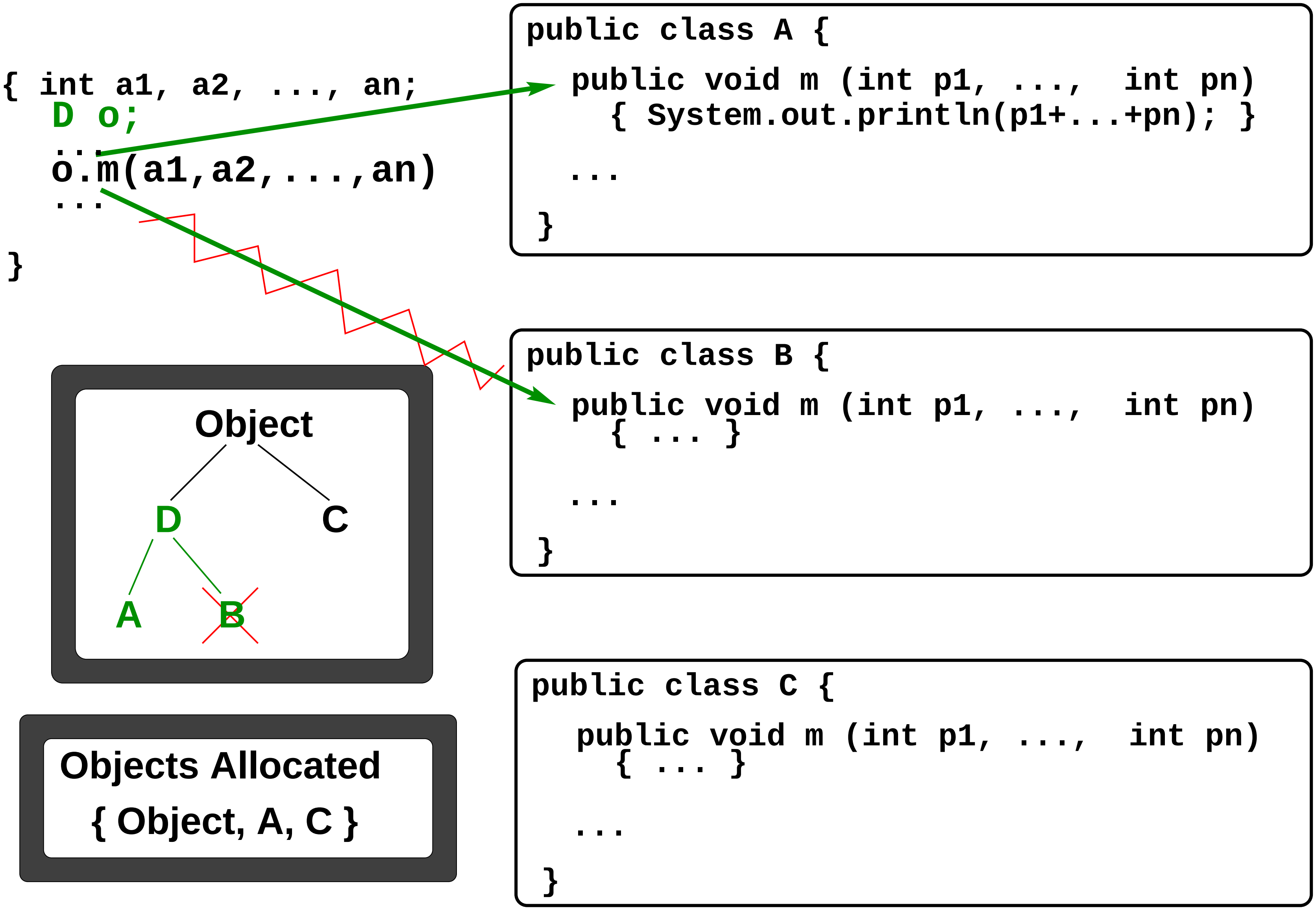
```
public class B {
  public void m (int p1, ...,  int pn)
    { ... }

  ...
 }
```



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

  ...
 }
```

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)
    { ... }
  ...
}
```

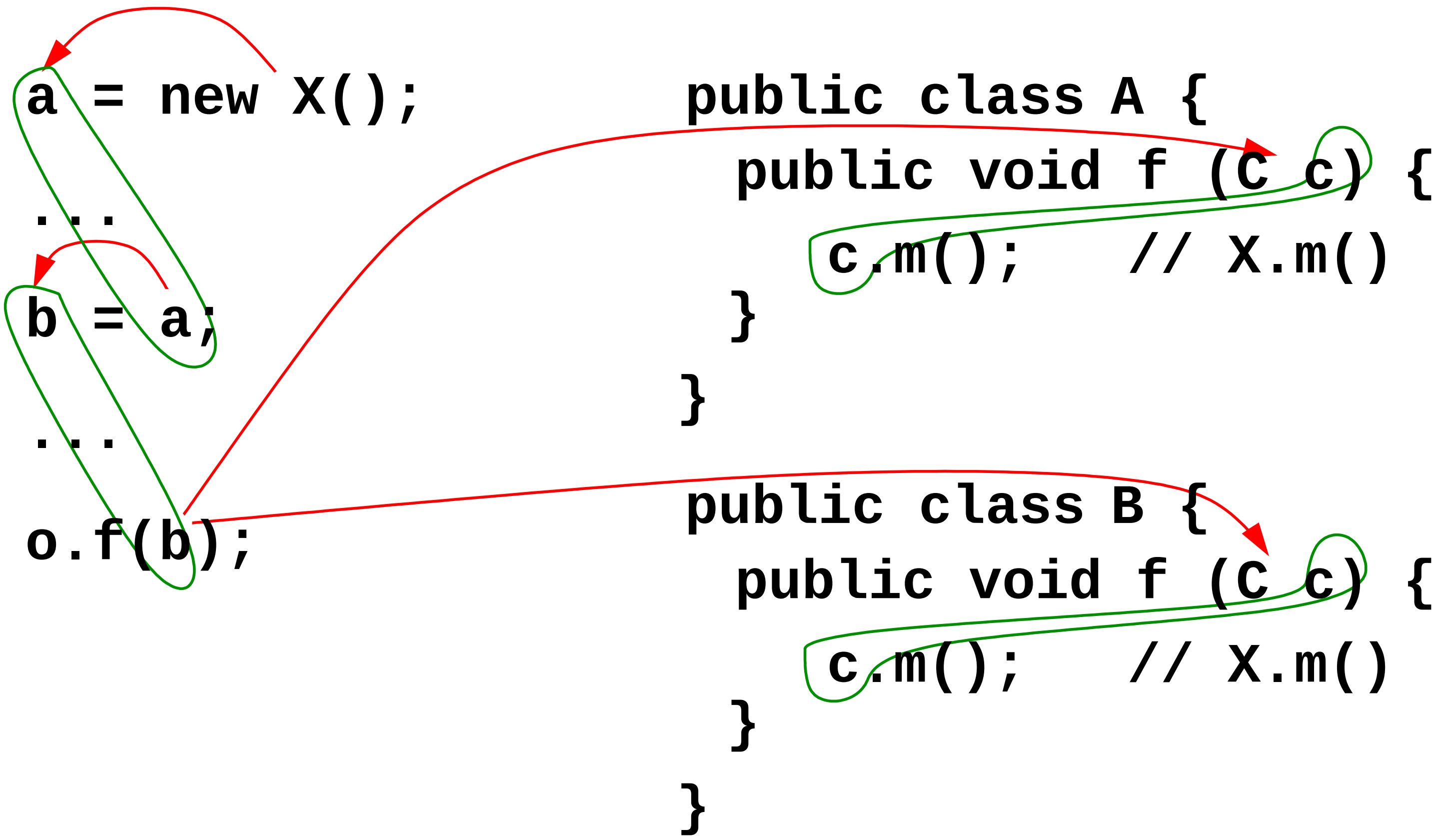
```
public class C {
  public void m (int p1, ...,  int pn)
    { ... }
  ...
}
```

Objects Allocated

{ Object, A, C }

Quest: Improve upon RTA, restrict the analysis to one iteration

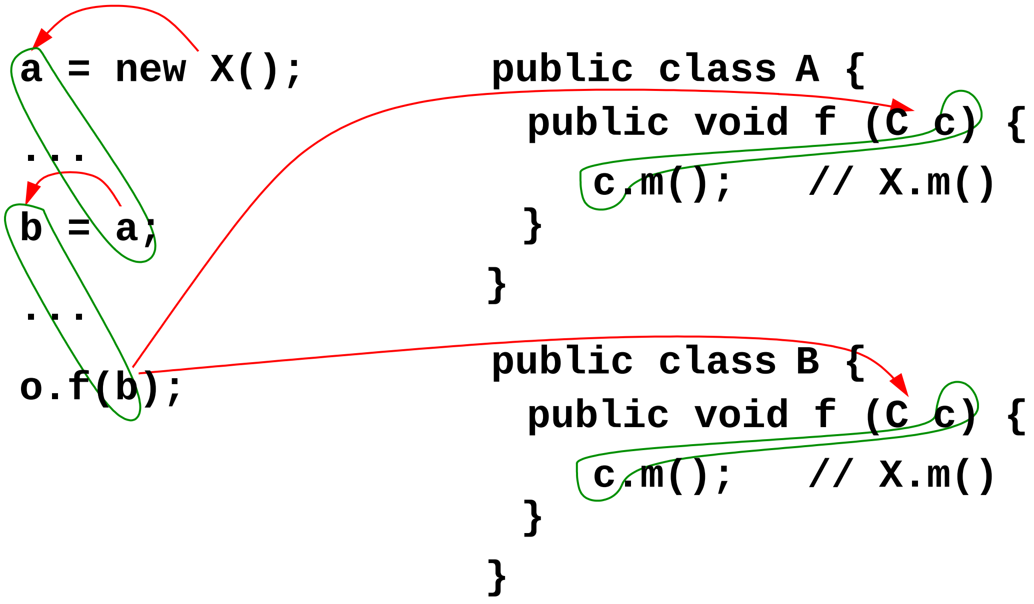
- RTA assumes that all allocated objects can reach a receiver.
- Want to provide a more accurate approximation;
- by tracking assignments from allocation sites, to method invocations.



```
a = new X();

...

b = a;

...

o.f(b);
```

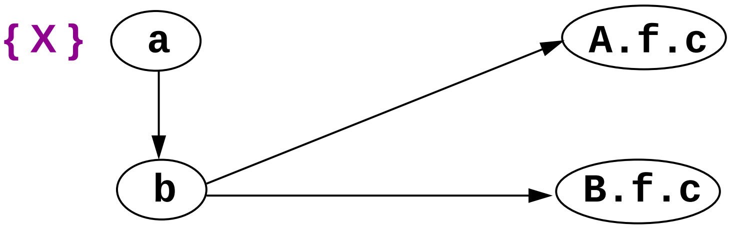
```
public class A {
  public void f (C c) {
     c.m();   // X.m()
  }
}

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

Solution: Variable Type Analysis (VTA)

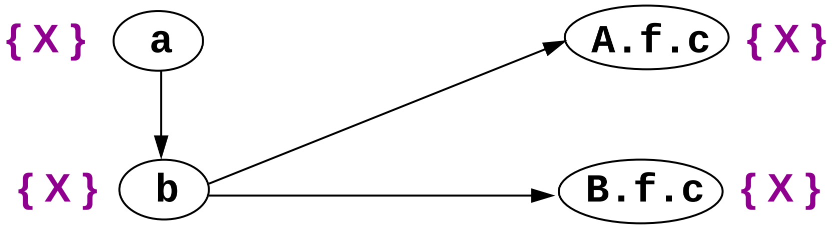


```
a = new X();
...
b = a;
...
o.f(b);
```

```
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

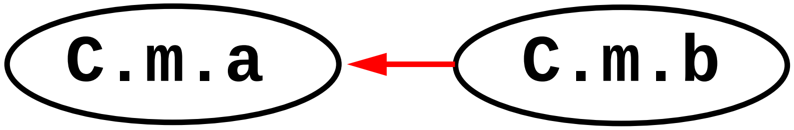
Three Steps in VTA

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

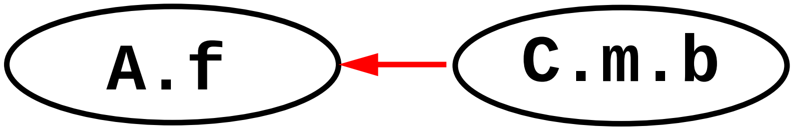
Assuming, statement is in class C, method m;

```
a = b;

a[i] = b;

a = b[i];
```



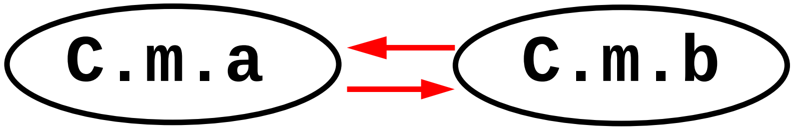
Assuming field f is declared in class A:

```
o.f = b;
```



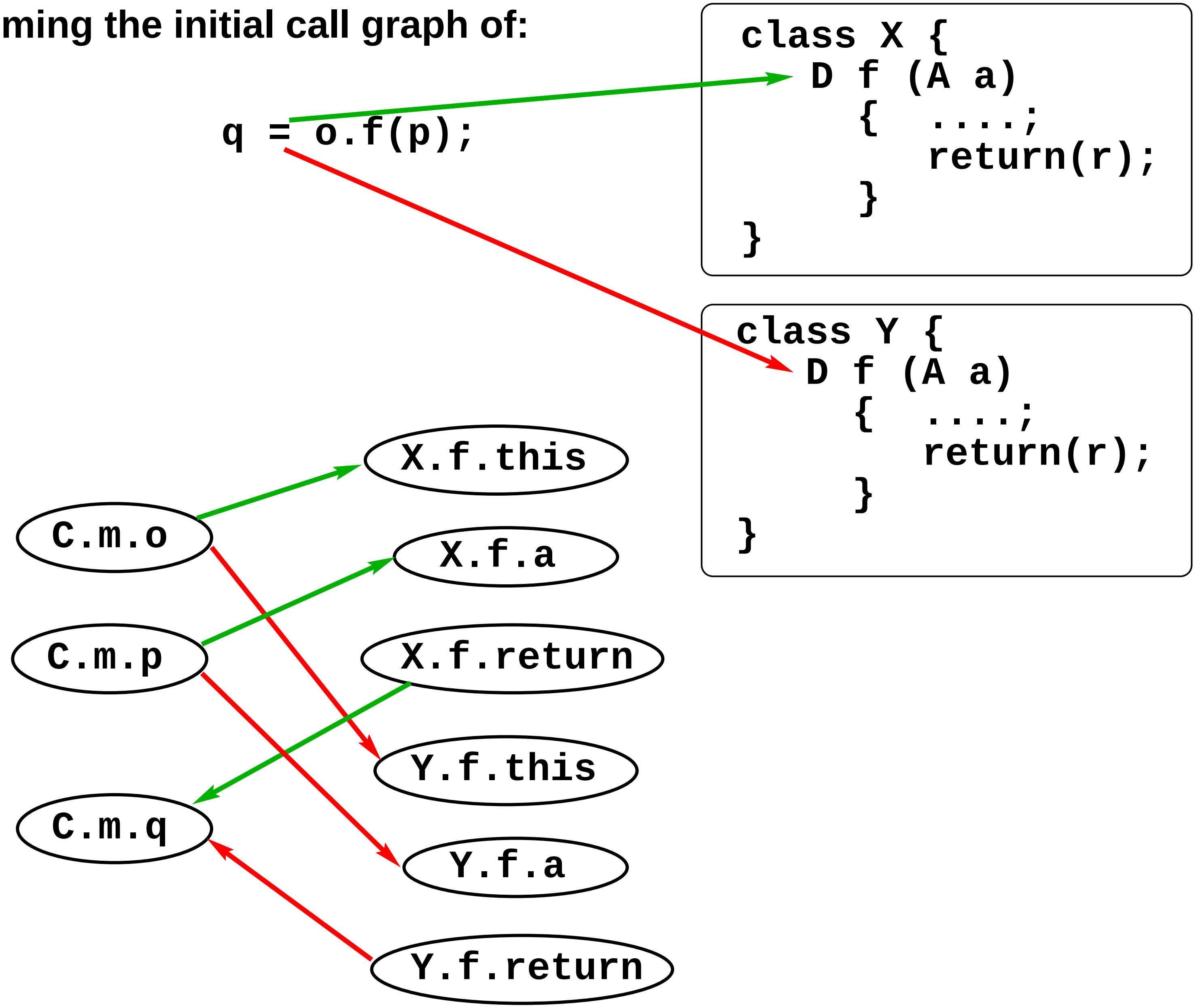
If either left or right side is Object or Array type:

```
a = b;
```



Building the Type Propagation Graph - method calls

Assuming the initial call graph of:



```
q = o.f(p);
```

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

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

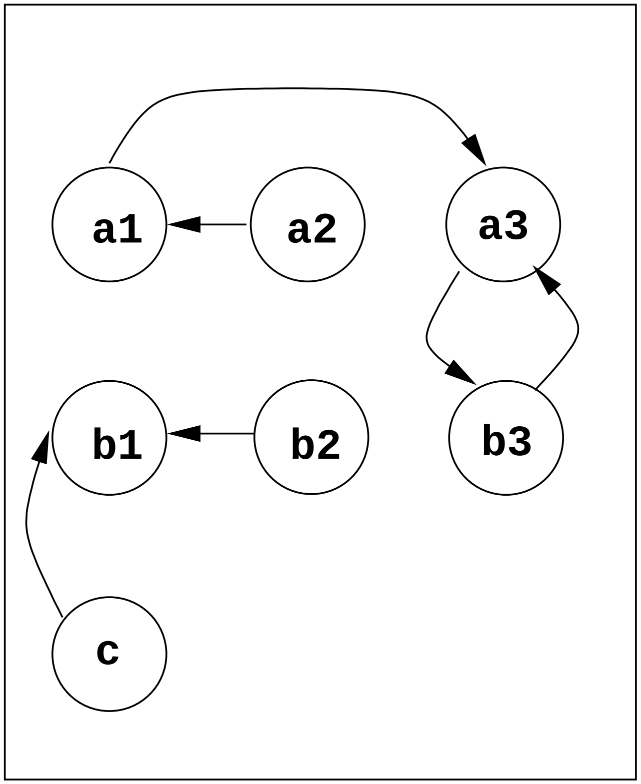
Propagating Types

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

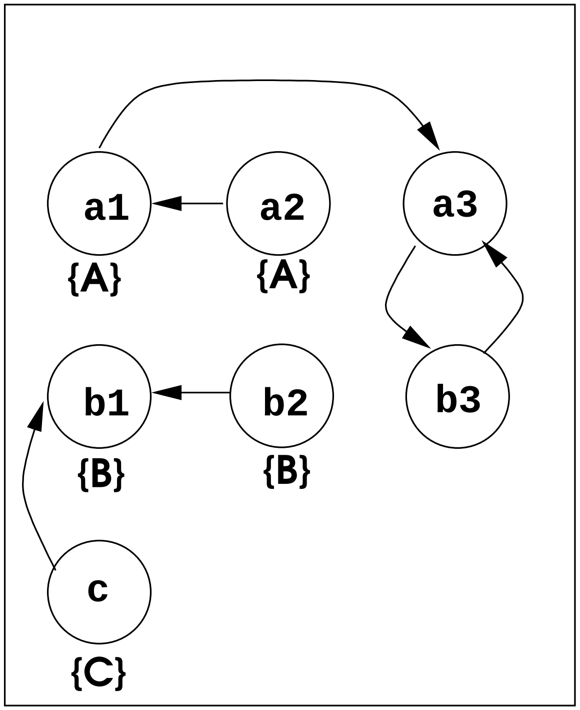
(a) Program



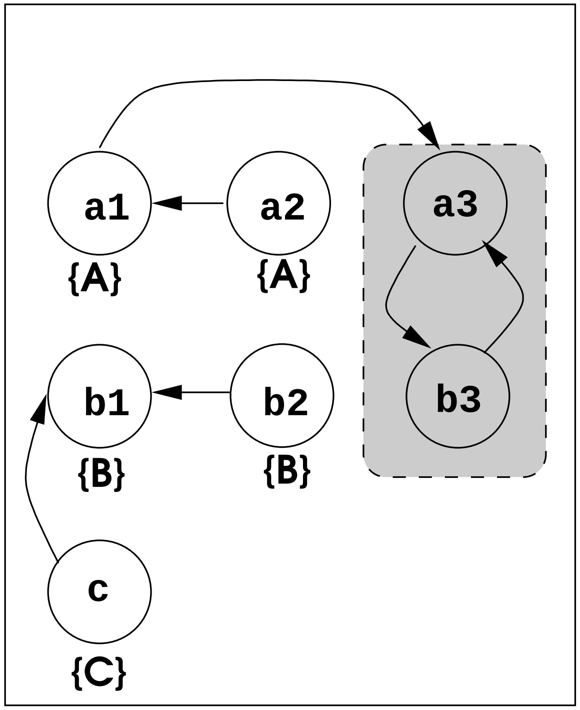
(b) Nodes and Edges

Propagating Types

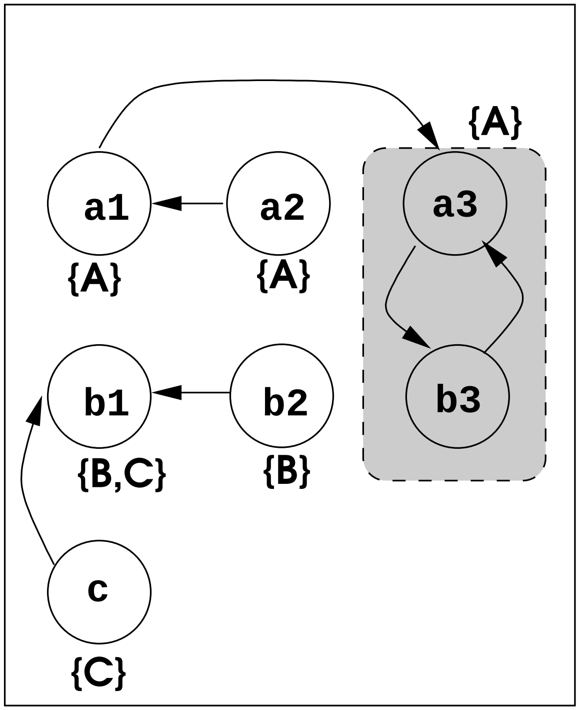
```
a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();
```



(c) Initial Types



(d) Strongly-connected components

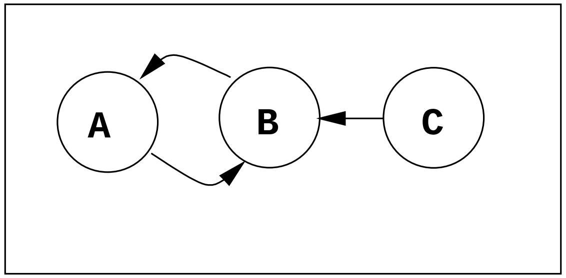


(e) final solution

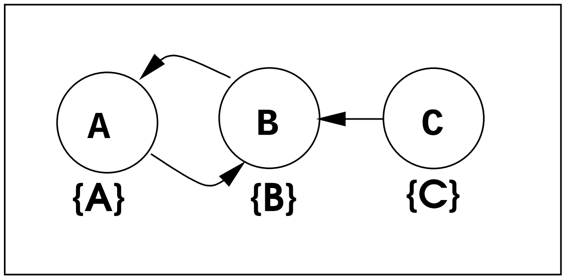
A Coarser Approximation: Declared Type Analysis

```
A 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;
```

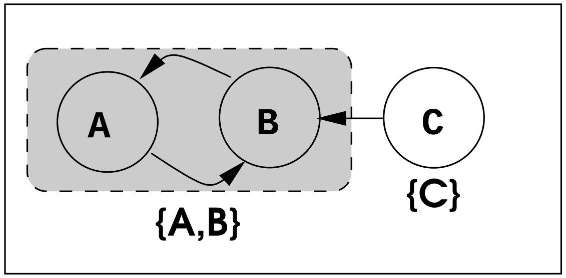
(a) Program



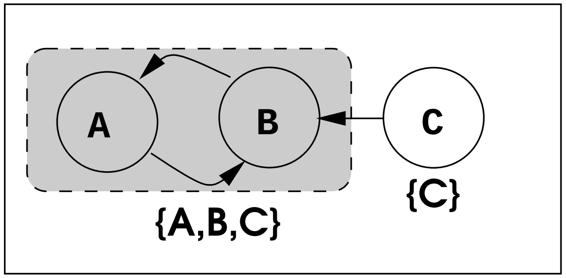
(b) Nodes and Edges



(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

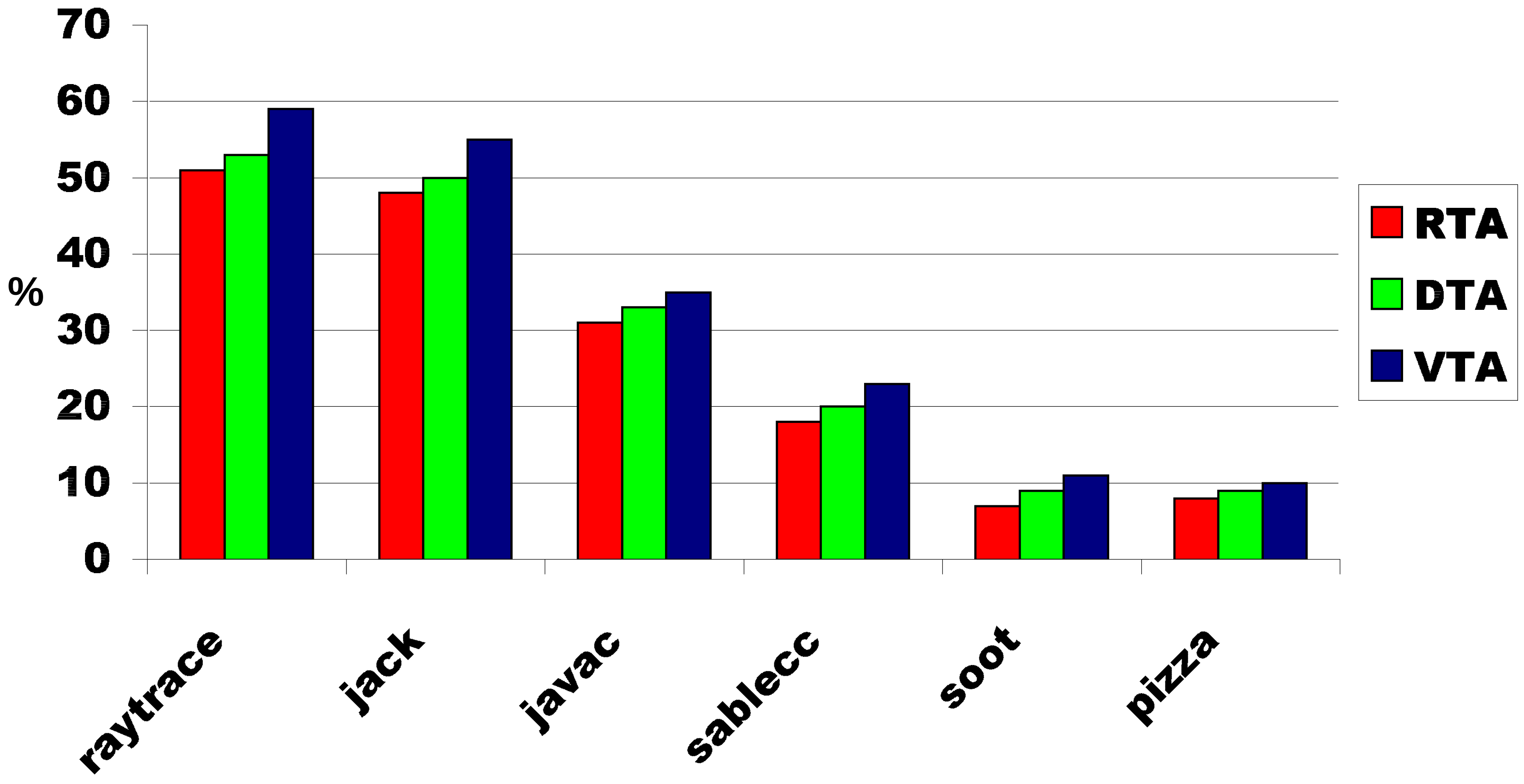
Benchmark Characteristics

	Total	Benchmark Only		
	# Stmts.	# Stmts	# Classes	# Interfaces
raytrace	49239	5347	34	1
jack	55107	11215	62	5
javac	69585	25304	177	5
sablecc	68575	24621	298	13
soot	63506	33396	497	34
pizza	73130	42805	207	11

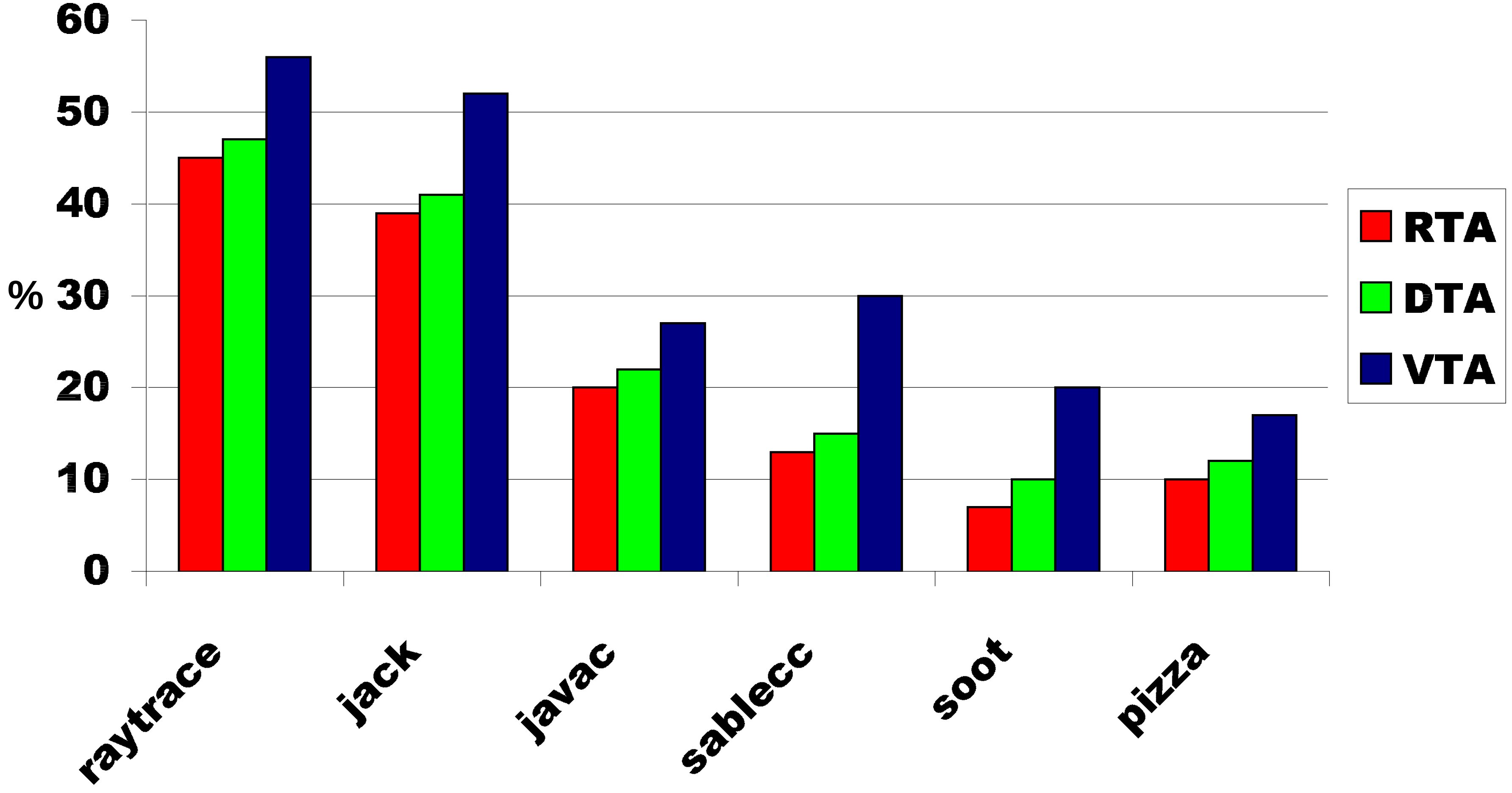
Conservative Call Graph Characteristics (CHA)

	Total	Benchmark Only		
	# Nodes	# Nodes	# Call Sites	*(% Poly.)*
raytrace	1729	207	2049	*(0.6%)*
jack	1857	337	3068	*(12.9%)*
javac	2821	1188	6781	*(12.5%)*
sablecc	3737	1955	6809	*(13.1%)*
soot	2828	2001	10615	*(14.6%)*
pizza	2660	1756	11692	*(4.9%)*

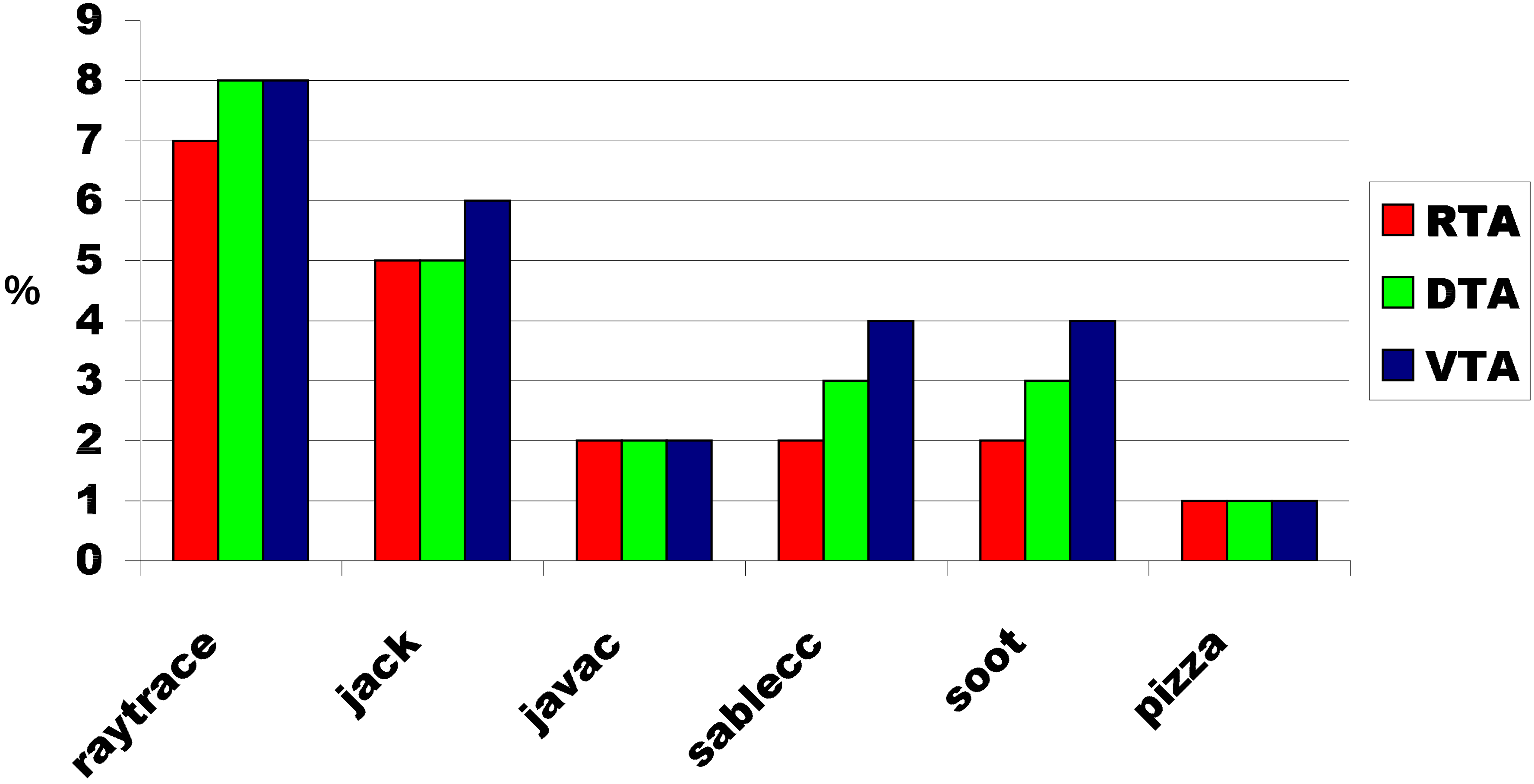
Percentage Methods Removed From Conservative Call Graph (Whole Application)



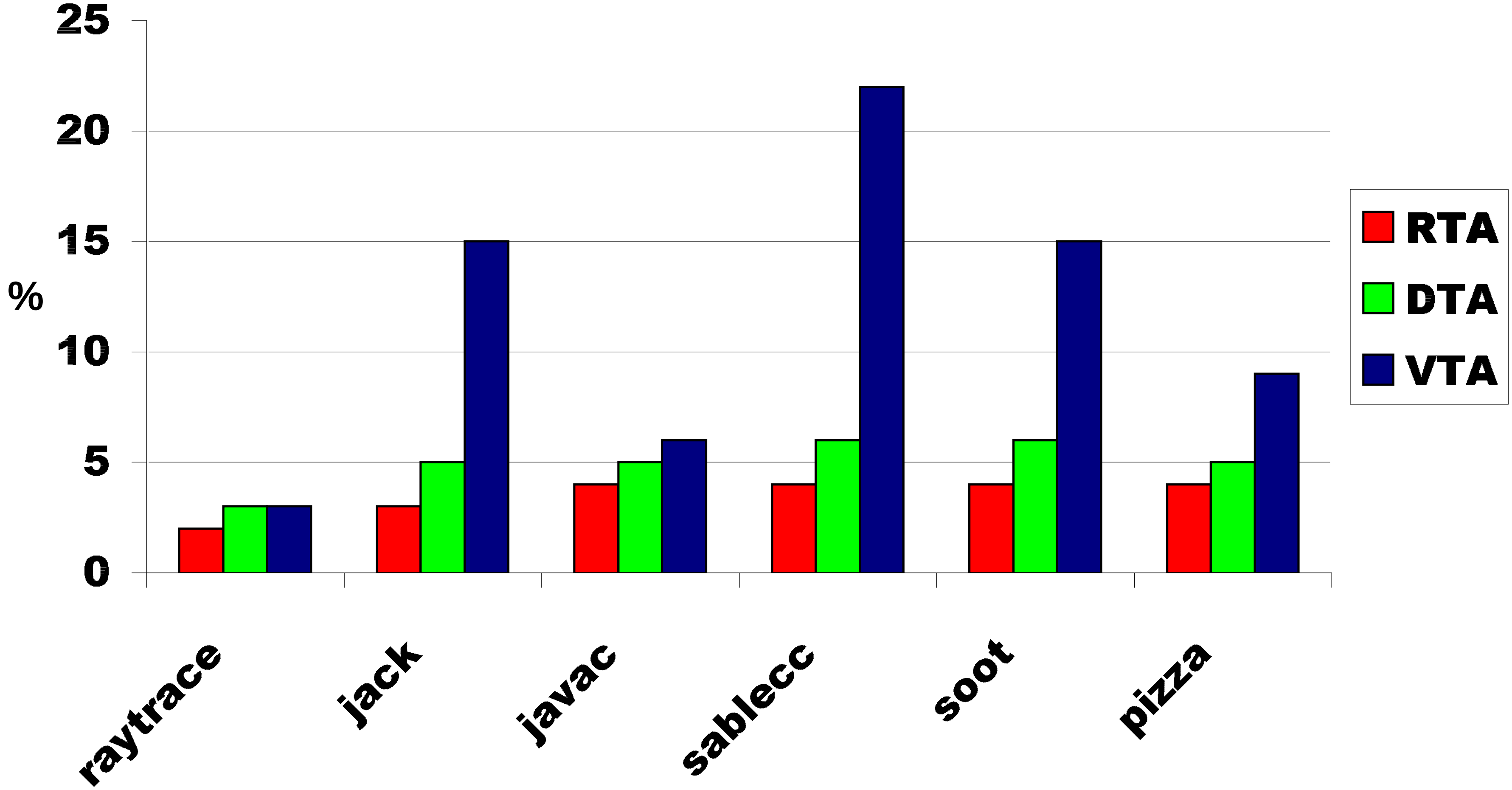
Percentage Edges Removed From Conservative Call Graph (whole application)



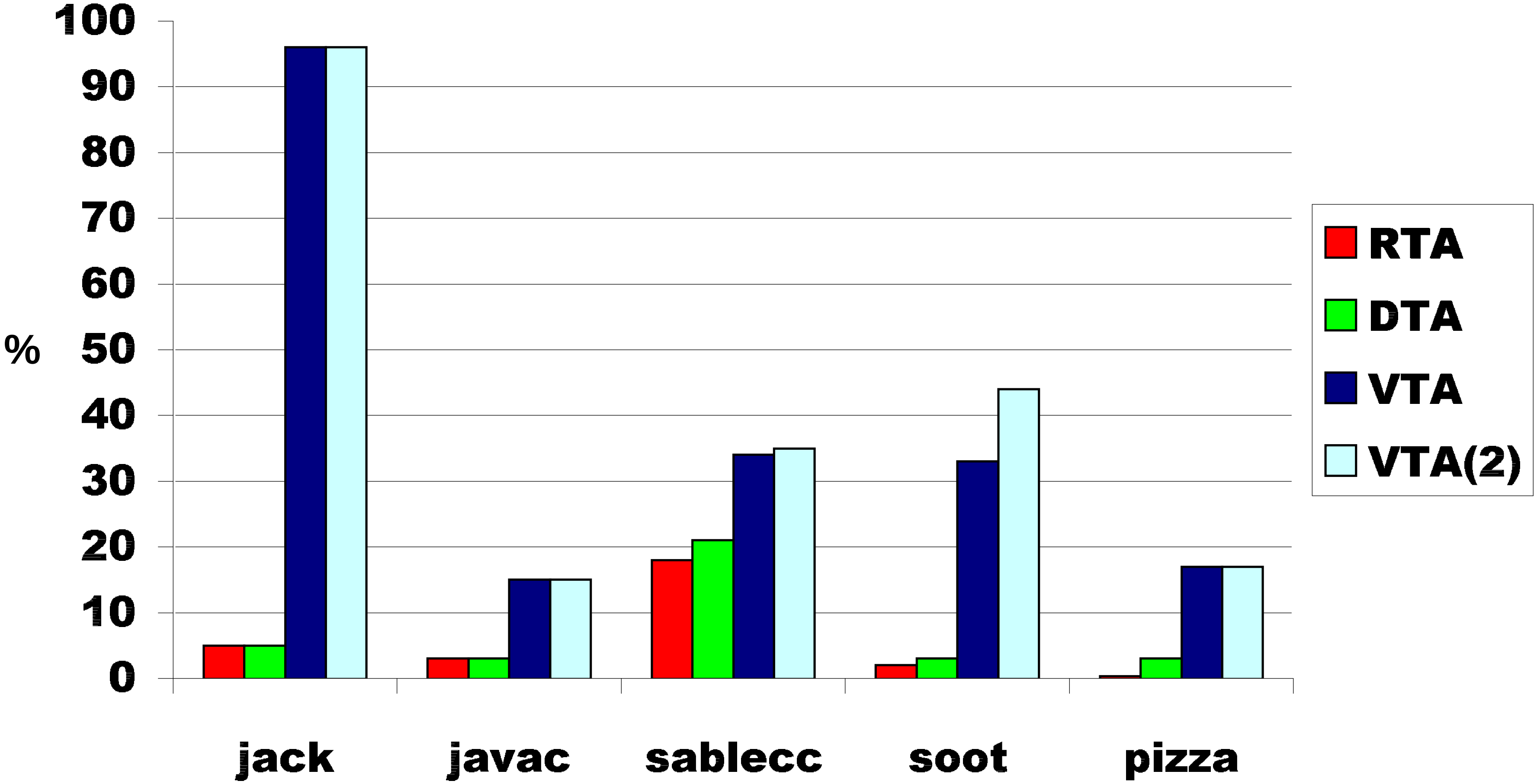
Percentage Methods Removed From Conservative Call Graph
(Benchmark only)



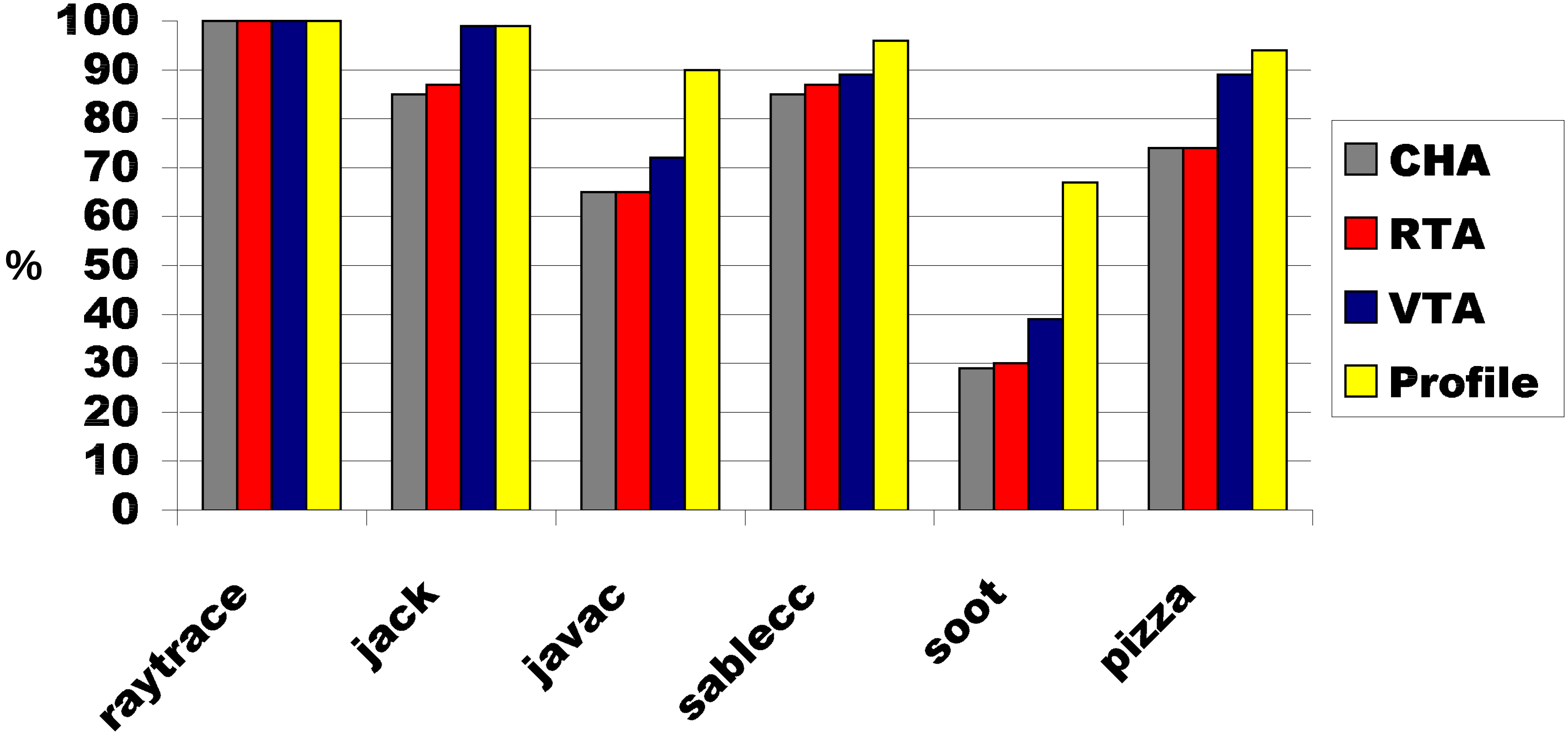
Percentage Edges Removed From Conservative Call Graph
(Benchmark Only)



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)



Sable Research Group

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`