Reference Analysis - 2

- More flow-insensitive, context-insensitive points-to analyses
- How to deal with dynamic class loading and reflection in static points-to analysis?
- Kinds of context-sensitivity
  - k-CFA versus object sensitivity (examples)
- Object-sensitive points-to for Java
- Dimensions of precision in static analysis

More FI-CI Points-to Analyses

- Liang et. al, Paste'01
  - Empirical comparison of flow- and context-insensitive analyses with different choices for representations
    - Steensgaard- and Andersen-based analyses for Java
    - Static call graph (CHA, RTA) with on-the-fly
    - Experiments with instance fields and abstract class fields
      - » Per object per field points-to set
      - » Per class per field points-to set (1 abstract object)
    - Use stubs for library methods
  - Found Andersen (inclusion) analyses significantly more precise than Steensgaard (unification) on call graph construction and escape analysis

D. Liang, M. Pennings, M.J. Harrold, "Extending and Evaluating Flow-insensitive and Context-insensitive Points-to Analysis for Java", PASTE'01

More FI-CI Points-to Analyses

- **Whaley and Lam, SAS'02**
  - **Used flow-sensitive analysis within methods, but context-insensitive overall in a variant of field-sensitive Andersen for Java**
    - **Results are interesting, but inconclusive on virtual call resolution and escape analysis clients**

Handling Dynamic Class Loading

- **All static points-to analyses require the whole program (including libraries or models of them)**
- **Dynamic class loading, reflection, native libraries present problems**
  - **Effects at runtime must be correctly modeled at compile-time or the analysis will be *unsafe***
- **New algorithm incrementally accounts for classes loaded and performs analysis updates *online at runtime***
  - **Generates constraints at runtime and propagates them when a client needs valid points-to results**

M.Hirzel, A. Diwan, M. Hind, "Pointer Analysis in the Presence of Dynamic Class Loading", ECOOP 2004

Hirzel et.al Algorithm

- Andersen's analysis with field-sensitive object representation, objects represented by their creation sites, and static call graph (CHA)
- Two stages (can be iterated when get new constraints)
  - Constraint generation
  - Constraint propagation with type filtering (producing points-to sets through fixed-point iteration)
- Use CHA call graph (generated online) to get call edges
  - Process constraints from an edge only after have seen both source and target

Hirzel et.al Algorithm

- Uses deferred evaluation to handle unresolved references
  - From native code, reflection, JIT compilation of a method, type resolution, class loading, VM startup
- Handle reflection through instrumenting the JVM to add constraints dynamically
  - Need to re-propagate at runtime as new constraints are added
  - Use JVM to catch reflection and add appropriate constraints when it occurs
  - Native code with returned heap value assumed to return any allocated object
  - Initial prototype assumed that any exception throw could hit any catch

Hirzel et.al Algorithm

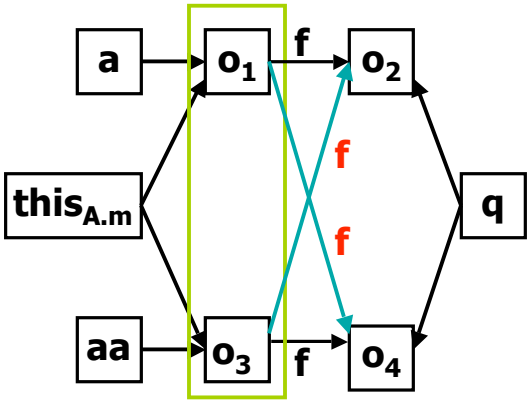
- Showed efficacy through use in new connectivity-based GC algorithm
  - Used Jikes RVM 2.2.1 on Specjvm98 benchmarks with good results; claimed need long-running programs for the incremental computation cost to be amortized.
- Validation:
  - Need to make sure points-to solution is updated before do a GC.
  - Then GC verifies the points-to solution by making sure the dynamically observed points-to's are in the solution.

Imprecision of Context-insensitive Analysis

- Does not distinguish contexts for instance methods and constructors
  - States of distinct objects are merged
- Common OOPL features and idioms result in imprecision
  - Encapsulation
    - Set() method conflates all instances with same field
  - Inheritance
    - Initialized fields in superclass constructor conflates points-to sets of subclass objects created
  - Containers, maps and iterators
    - Same creation site results in apparent unioning of all contents

Example: Imprecision

```
class Y extends X { … }
class A {
  X f;
  void m(X q) {
      this.f=q ; }
  }
A a = new A() ;
a.m(new X()) ;
A aa = new A() ;
aa.m(new Y()) ;
```



Context Sensitivity

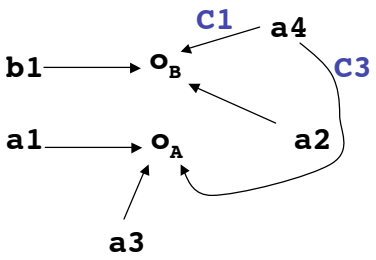
- **Keeping calling contexts distinct during the analysis**
- **Classically two approaches (Sharir, Pneuli 1981)**
  - *Call string* **- distinguish analysis result by (truncated) call stack on which it is obtained**
    - **e.g., k-CFA**
  - *Functional* **- distinguish analysis result by (partial) program state at call**
    - **e.g., receiver identity, argument types**

M. Sharir, A. Pneuli, "Two Approaches to Interprocedural Dataflow Analysis". Ch 7 in Program Flow Analysis, Edited by S. Muchnick, N. Jones, Prentice-Hall 1981

1-CFA Analysis

- **Calling context is tail of *call string* (1-CFA is last call site)**

```
static void main(){
      B b1 = new B();//O_B
      A a1 = new A();//O_A
      A a2,a3;
C1:   a2 = f(b1);
C2:   a2.foo();
C3:   a3 = f(a1);
C4:   a3.foo();
}
public static A f(A a4){return a4;}
```



Points-to Graph

at C2, main calls B.foo()
at C4, main calls A.foo()

1-CFA Characteristics

- **Call-string approach to context sensitivity**
- **Only analyzes methods *reachable* from main()**
- **Keeps track of individual reference variables and fields**
- **Groups objects by their creation site**
- **Incorporates reference value flow in assignments and method calls**
- **Differentiates points-to relations for different calling contexts**

Object-sensitive Points-to Analysis

- Object sensitivity
  - *Functional* context sensitivity for flow-insensitive points-to analysis of OO languages
- Object-sensitive Andersen's analysis
  - Object sensitivity also applicable to other analyses
- Parameterization framework
  - Cost vs. precision tradeoff
- Empirical evaluation
  - Vs. field-sensitive context-sensitive analysis

Milanova, A. Rountev, B. G. Ryder,
"Practical Points-to Analyses for Java", ISSTA'02;
"Parameterized Object Sensitivity for Points-to Analysis for Java", TOSEM, Jan 2005

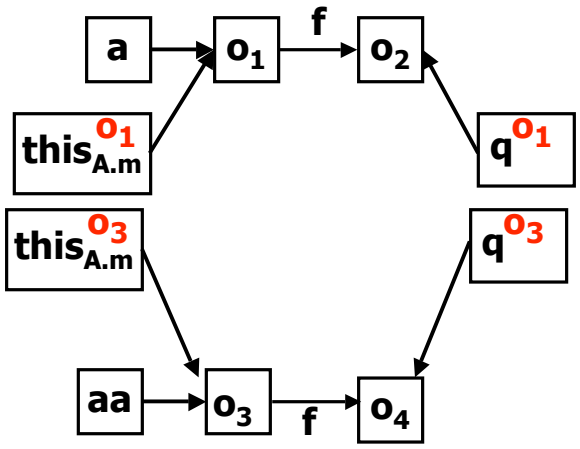
Object-sensitive Analysis

- Instance methods and constructors analyzed for different contexts
- Receiver objects used as calling context
- Multiple copies of local reference variables

this.f=q $\xrightarrow{o_1}$ $this^{o_1}_{A.m}.f=q^{o_1}$

Example: Object-sensitive Analysis

```
class A {
  X f;
  void m(X q) {
this^o3_A.m.f=q^o3; }
  }
A a = new A();
a.m(new X());
A aa = new A();
aa.m(new Y());
```



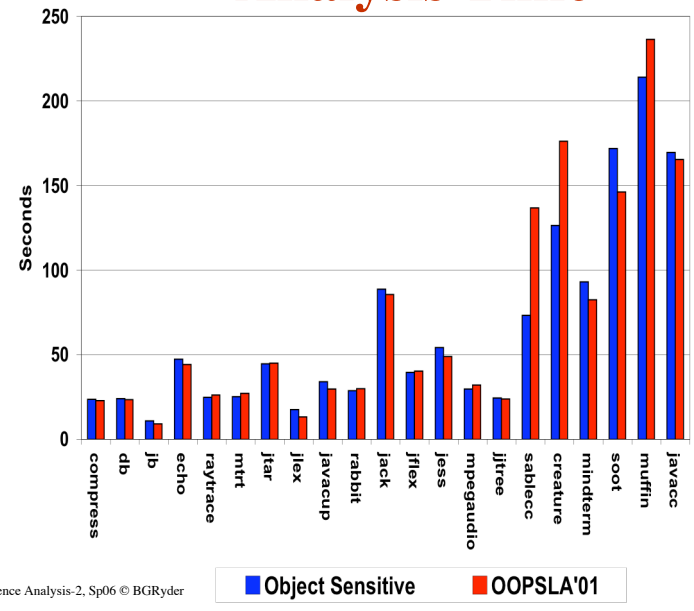
Implementation

- Implemented one instance of parameterization framework
  - `this`, formals and return variables (effectively) replicated
  - Optimized constraint-based analysis using previous technique
  - Comparison with field-sensitive (context-insensitive) analysis

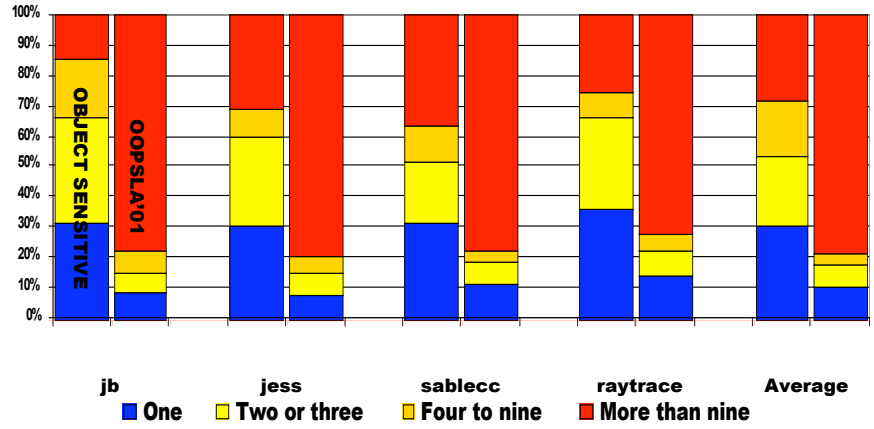
Empirical Results

- 23 Java programs: 14 – 677 user classes
  - Added the necessary library classes
  - Machine: 360 MHz, 512Mb, SUN Ultra-60
- Object-sensitive vs. field-sensitive points-to
- Found comparable cost with better precision
  - Modification side-effect analysis
  - Virtual call resolution

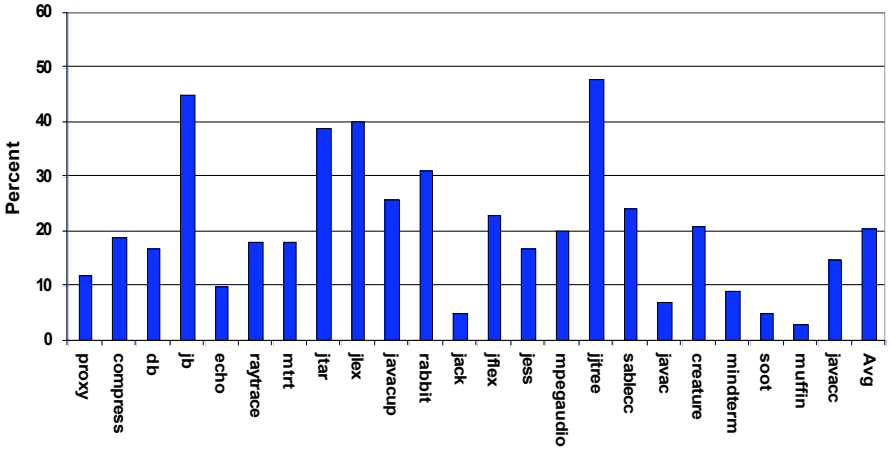
Analysis Time



Side-effect Analysis: Modified Objects Per Statement



Improvement in Resolved Calls



Milanova et.al, TOSEM 1/05

More Experiments

- Explored variants of object-sensitive analysis on faster implementation
  - Keep context-sensitive info for this, formals (+/-)return param
  - Used context-sensitive object naming (remember call on which object was created)
  - Compared to 1-CFA and context-insensitive analysis
- Obtained better precision than Andersen at approx same cost (for % stmts with #modified objs per assignment, on average)

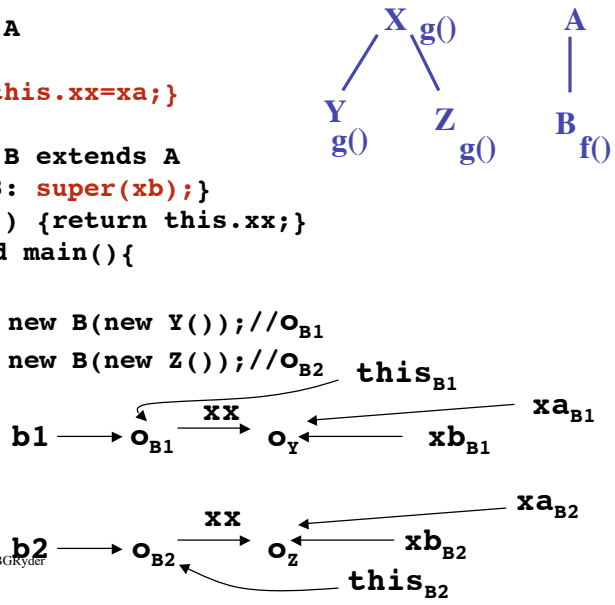
	And	1-CFA	ObjSens
1-3objs	18%	23%	54%
4-9objs	4%	5%	18%
>=10obj	78%	72%	28%

Comparison of 1-CFA w Object-sensitive

- Two algorithms are incomparable
  - (see following examples)
  - Some evidence suggests that ObjSens is more practical (Lhotak & Hendren, CC'06)
- Newer implementations use BDDs for scalability
- Context-sensitive object naming found to be useful (Lhotak & Hendren, CC'06)

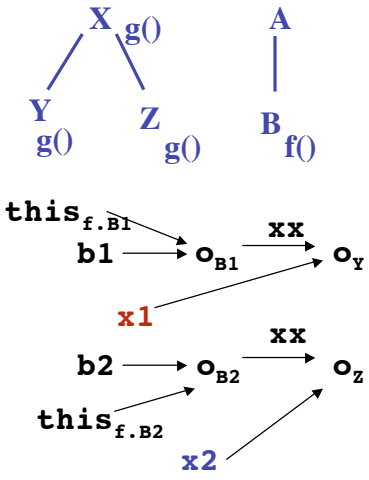
Object-Sensitive Points-to EG

```
public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
    X x1,x2;
  C1: B b1 = new B(new Y());//oB1
  C2: B b2 = new B(new Z());//oB2
```



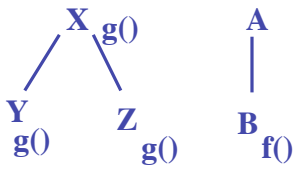
Object-sensitive Points-to EG

```
public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
   X x1,x2;
  C1: B b1 = new B(new Y());//oB1
  C2: B b2 = new B(new Z());//oB2
    x1=b1.f();
C4: x1.g();
    x2=b2.f();
C5: x2.g();
}
```



Object-sensitive Points-to EG

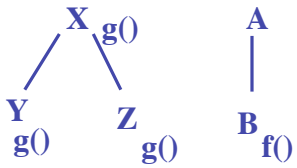
```
public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
   X x1,x2;
  C1: B b1 = new B(new Y());//oB1
  C2: B b2 = new B(new Z());//oB2
     x1=b1.f();
C4:  x1.g();
     x2=b2.f();
C5:  x2.g();
}
```

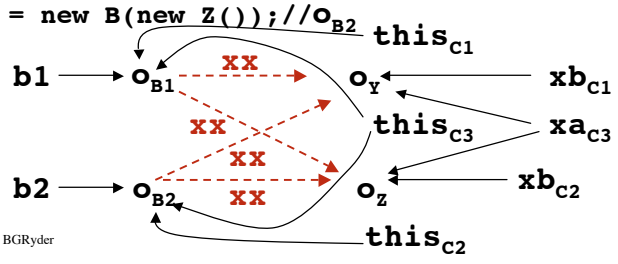


ObjSens finds
C4 calls Y.g() and
C5 calls Z.g()

1-CFA on Object-sensitive EG

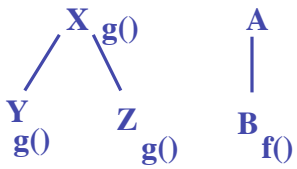
```
public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
    X x1,x2;
  C1: B b1 = new B(new Y());//oB1
  C2: B b2 = new B(new Z());//oB2
```

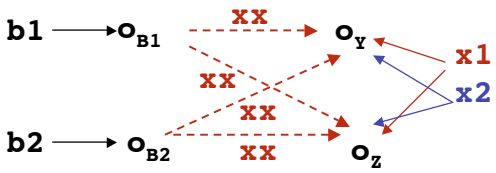




1-CFA on Object-sensitive EG

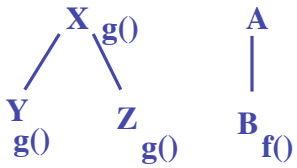
```
public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){C3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
   X x1,x2;
  C1: B b1 = new B(new Y());//oB1
  C2: B b2 = new B(new Z());//oB2
     x1=b1.f();
C4: x1.g();
     x2=b2.f();
C5: x2.g();
}
```





1-CFA on Object-sensitive EG

```
public class A
{ X xx;
  A (X xa){ this.xx=xa;}
}
public class B extends A
{ B (X xb){c3: super(xb);}
  public X f() {return this.xx;}
  static void main(){
   X x1,x2;
  C1: B b1 = new B(new Y());//oB1
  C2: B b2 = new B(new Z());//oB2
     x1=b1.f();
C4: x1.g();
     x2=b2.f();
C5: x2.g();
}
```

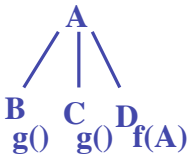


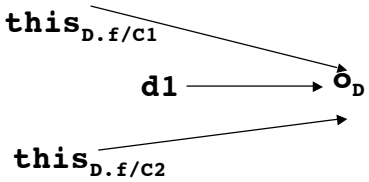
1-CFA finds
C4 calls Y.g(), Z.g() and
C5 calls Y.g(), Z.g()

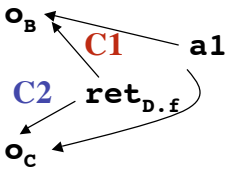
ObjSens is better than 1-CFA here!

1-CFA Example

```
static void main(){
 D d1 = new D();
 if (…)C1: (d1.f(new B())).g();
 else  C2: (d1.f(new C())).g();
}
public class D
{ public A f(A a1){return a1;}
}
```



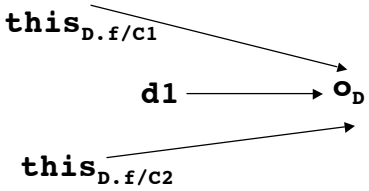


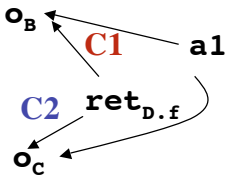


1-CFA Example

```
static void main(){
 D d1 = new D();
 if (…)C1: (d1.f(new B())).g();
 else  C2: (d1.f(new C())).g();
}
public class D
{ public A f(A a1){return a1;}
}
```

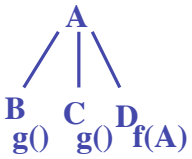
1-CFA distinguishes the two calling contexts of D.f at C1 and C2;
At C1, B.g() called;
At C2, C.g() called;

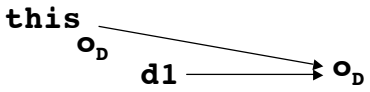


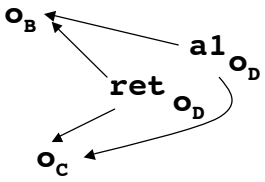


Object-sensitive on 1-CFA EG

```
static void main(){
 D d1 = new D();
 if (…)C1: (d1.f(new B())).g();
 else  C2: (d1.f(new C())).g();
}
public class D
{ public A f(A a1){return a1;}
}
```



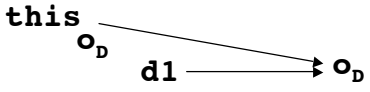


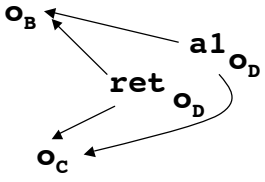


Object-sensitive on 1-CFA EG

```
static void main(){
 D d1 = new D();
 if (…)C1: (d1.f(new B())).g();
 else  C2: (d1.f(new C())).g();
}
public class D
{ public A f(A a1){return a1;}
}
```

ObjSens groups the two calling contexts of D.f with the same receiver at *C1* and *C2*;
Both B.g(),C.g() are called at *C1* and *C2*;





Previous Related Work

- Context-sensitive reference analyses
  - Palsberg and Schwartzbach OOPSLA'91
  - Oxhoj, Palsberg, Schwartzbach ECOOP'92
  - Plevyak and Chien OOPSLA'94
  - Agesen ECOOP'95
  - Chatterjee, Ryder, Landi POPL'99
  - Ruf PLDI'00
  - Grove and Chambers TOPLAS'01
- Most judged too expensive to be practical

Algorithm Design Choices

- Representations
  - Static call graph versus on-the-fly construction
  - Abstract class object, representative of set of object instantiations, context-sensitive object naming
    - Fields or no fields or field-based
  - Abstract reference (by class), or reference representatives per method, or references program-wide by name
- Directionality (interpretation of reference assignments)
  - Symmetric (Unification)
  - Directional (Inclusion)
- Accounting for flow of control
  - Flow sensitivity
  - Context sensitivity

Ryder, Barbara G., "Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages", invited paper in the Proceedings of the *Twelfth International Conference on Compiler Construction*, Warsaw, Poland, April 2003, pp 126-137.

Examples

- Representations
  - Static call graph **VTA** versus on-the-fly construction **RTA**
  - Abstract class object **XTA**, representative of set of object instantiations **field-sensitive,** context-sensitive object naming
    - Fields: **field-sensitive** or no fields or field-based **Spark 0-CFA**
  - Abstract reference (by class) **RTA**, or reference representatives per method **XTA**, or references program-wide by name **VTA, field-sensitive, 1-CFA**
- Directionality
  - Symmetric (Unification) **Hendren'00(variant of VTA), Liang et. al Paste'01**
  - Directional (Inclusion) **field-sensitive, object-sensitive, k-CFA**
- Accounting for flow of control
  - Flow sensitivity, **Chatterjee POPL99**
  - Context sensitivity, **object-sensitive, 1-CFA**

Open Issues

- **Reflection**
- **Dynamic class loading**
- **Java native methods**
- **Exceptions**
- **Incomplete programs**
- **Which benchmarks are best?**