

# **Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis**

---

Guoqing Xu, Atanas Rountev, Manu Sridharan

**Ohio State University**

**IBM T. J. Watson Research Center**

# Points-to Analysis

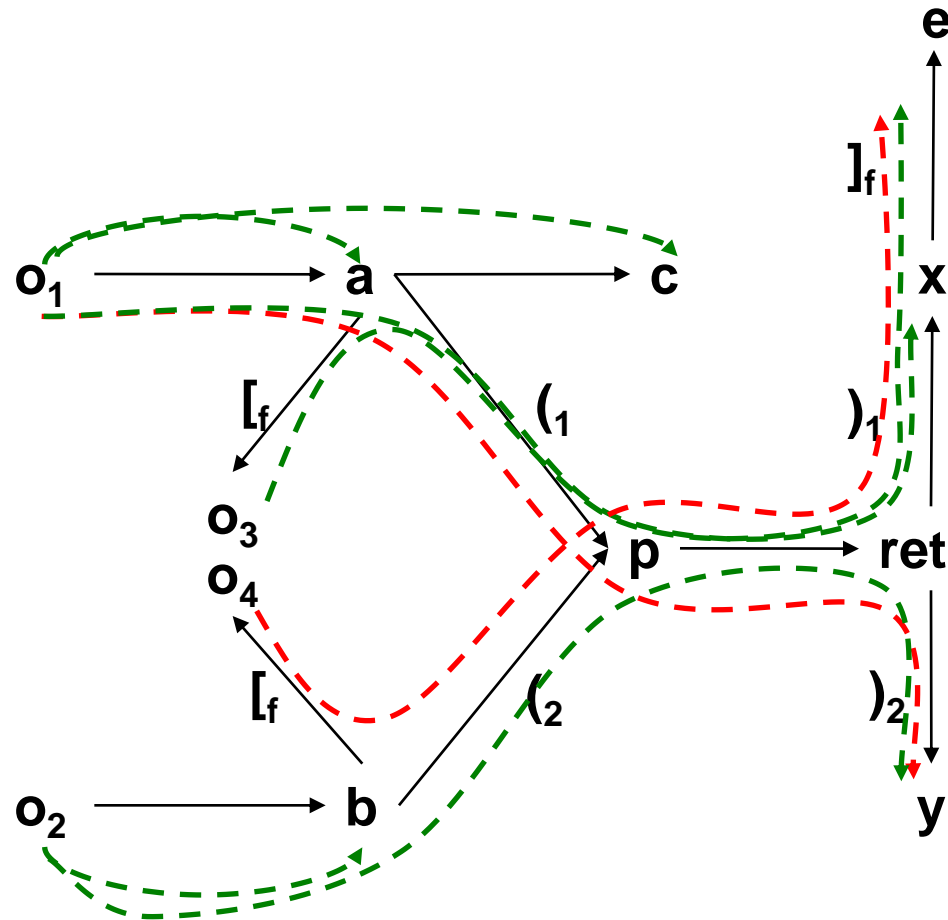
- Many static analysis tools need highly precise whole-program points-to solution
  - (e.g., data race detector, slicer)
- Context-free-language(CFL) reachability formulation of points-to/alias analysis [*Sridharan-Bodik PLDI'06*]
  - High precision
  - Does not scale well for whole program analysis
  - A lot of redundant computation
- Our approach targets CFL-reachability-based points-to analysis
  - Pre-analyze the program to reduce the redundancy

# Example of CFL-Reachability Formulation [PLDI'06]

```
a = new A(); // o1  
b = new A(); // o2  
c = a;
```

```
id(p){ return p;}  
x = id(a); // call 1  
y = id(b); // call 2
```

```
a.f = new C(); //o3  
b.f = new C(); //o4  
e = x.f;
```

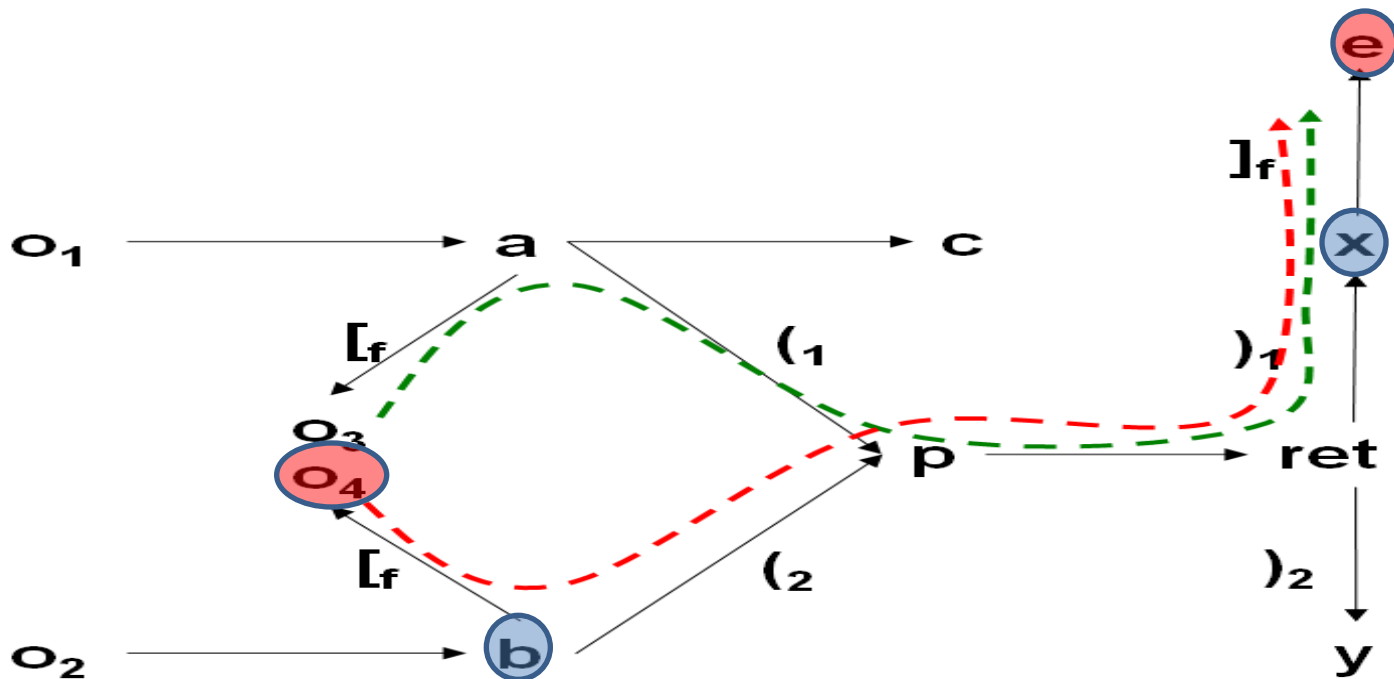


$o \in \text{pts}(v)$  if  $o$  *flowsTo*  $v$



# Our Approach

- Must-not-alias analysis
  - Use an imprecise but cheap off-line analysis to find  $x$  and  $b$  are not aliases under any possible calling context
  - Quickly conclude that  $e$  cannot point to  $o_4$  in the points-to analysis, if our analysis reports  $(x, b)$  *must not* alias

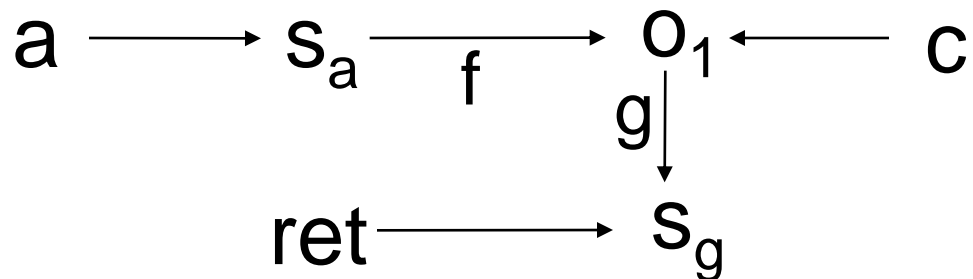


# Program Representation

- Intraprocedural Symbolic Points-To Graph (SPG)
  - Introduce *symbolic node*  $s$  for
    - formal parameter
    - field dereference  $a.f$  in a heap load
    - a call site that returns a reference-typed value
  - Compute intraprocedural *flowsTo* path
  - Points-to edge  $a \longrightarrow o \in \text{SPG}$  if  $o$  *flowsTo*  $a$  is found
  - Points-to edge  $o_1 \xrightarrow{f} o_2 \in \text{SPG}$  if  $o_1$  *flowsTo*  $a$ ,  $o_2$  *flowsTo*  $b$ , and  $a.f = b$  are found

```

B m(A a){
  C c = new C();// o1
  a.f = c;
  return c.g;
}
  
```

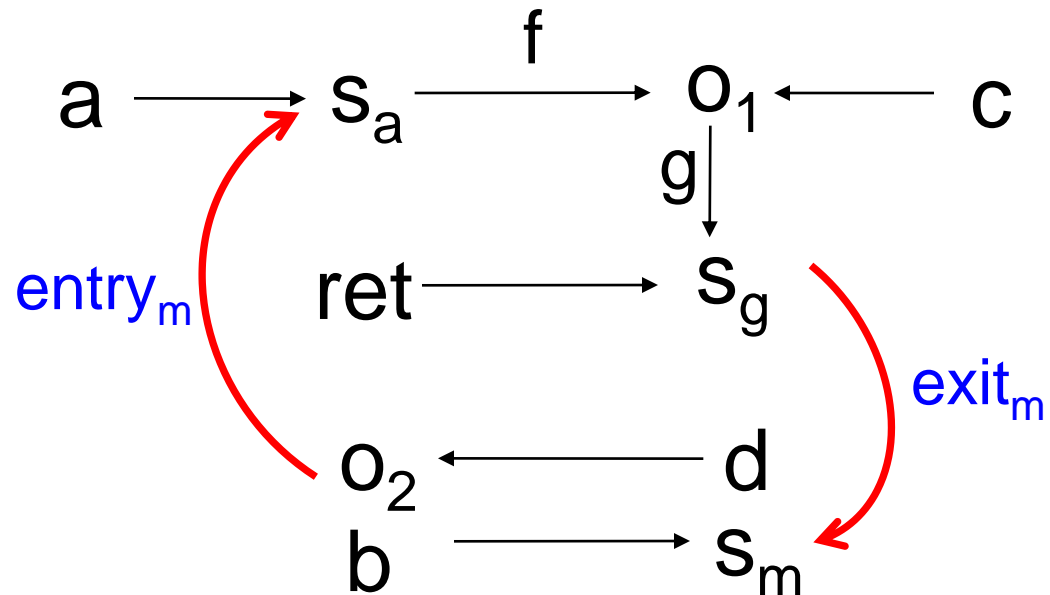


# Interprocedural Symbolic Points-To Graph

- Connect intraprocedural symbolic points-to graphs with **entry** and **exit** edges

```
B m(A a){  
  C c = new C(); // o1  
  a.f = c;  
  return c.g;  
}
```

```
A d = new A(); // o2  
B b = m(d); // call m
```



# Must-Not-Alias Analysis

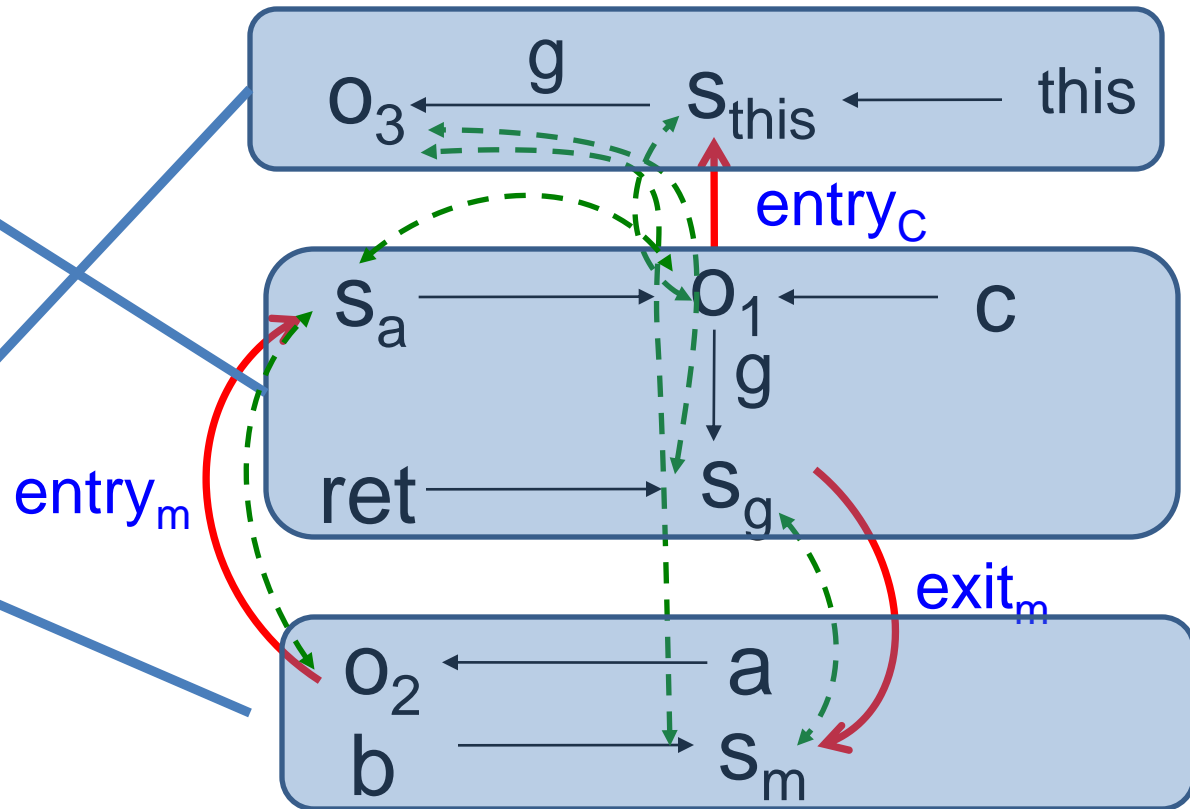
- Context-insensitive *memAlias* formulation
  - Treat a pair of points to edges  $\xrightarrow{f}$  and  $\xleftarrow{f}$  as balanced parentheses
  - $memAlias \rightarrow \bar{f} memAlias f$ 
    - | *memAlias memAlias*
    - | entry
    - |  $\overline{\text{entry}}$
    - | exit
    - |  $\overline{\text{exit}}$
    - |  $\epsilon$
  - Allocation or symbolic node **m** and **n** are aliases if **m** *memAlias* **n**

# Example

```
B m(A a){  
  C c = new C();// o1  
  a.f = c;  
  return c.g;  
}
```

```
A a = new A(); // o2  
B b = m(a);
```

```
C(){  
  this.g = new B();// o3  
}
```



# Algorithm

- Add pairs of nodes (**a**, **b**) in *memAlias*, if they are reachable from the same node **c**, and the strings between (**c**, **a**) and (**c**, **b**) are the same

– Example:  $a \xleftarrow{f_1} a_0 \xleftarrow{f_2} \dots \xleftarrow{f_n} c \xrightarrow{f_n} \dots \xrightarrow{f_2} b_0 \xrightarrow{f_1} b$

# Algorithm (Cond.)

while a fixed point is not reached do

- Add pairs of nodes  $(a, b)$  in *memAlias*, if  $(a, f) \in \text{memAlias}$ ,  $(g, b) \in \text{memAlias}$ ,  $(f, g) \in \text{memAlias}$   
–  $a \dashleftarrow{\text{green}} f \dashleftarrow{\text{green}} g \dashleftarrow{\text{green}} b$
- Add pairs of nodes  $(d, e)$  in *memAlias*, if there is a pair  $(f, g) \in \text{memAlias}$ ,  $d$  and  $e$  are reachable from  $f$  and  $g$ , respectively, and the two strings between  $(f, d)$  and  $(e, g)$  are the same



end while

# Context-Sensitive Must-Not-Alias Analysis

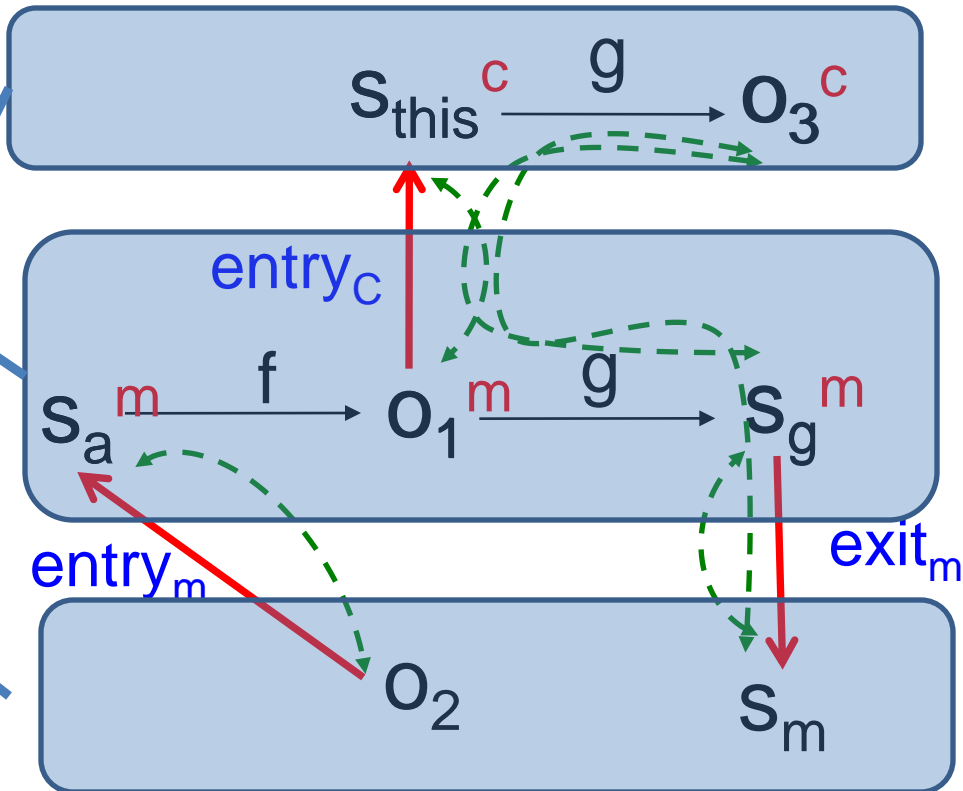
- Context-sensitivity is achieved by
  - Bottom-up traversing the call graph (i.e., summary-based)
  - Cloning objects for 1-level method calls when composing summaries
- Context sensitivity
  - Full context-sensitivity for pointer variables
  - 1-level context-sensitivity for pointer targets
  - Has almost the same precision as the 1-object-sensitive analysis, but much cheaper

# Example

```
B m(A a){  
  C c = new C();// o1  
  a.f = c;  
  return c.g;  
}
```

```
A a = new A(); // o2  
B b = m(a);
```

```
C(){  
  this.g = new B();// o3  
}
```



# Using Must-Not-Alias Information

- Object or symbolic nodes **m** and **n** must not alias if  $(m, n) \notin \text{memAlias}$
- Using must-not-alias information in Sridharan-Bodik analysis
  - Check a pair of load and store  $a.f = o; c = b.f;$
  - Don't check whether **a** and **b** can alias if, for any object or symbolic nodes  $o_a$  and  $o_b$  such that  $a \longrightarrow o_a$  and  $b \longrightarrow o_b \in \text{ISPG}$ ,  $o_a$  and  $o_b$  must not alias

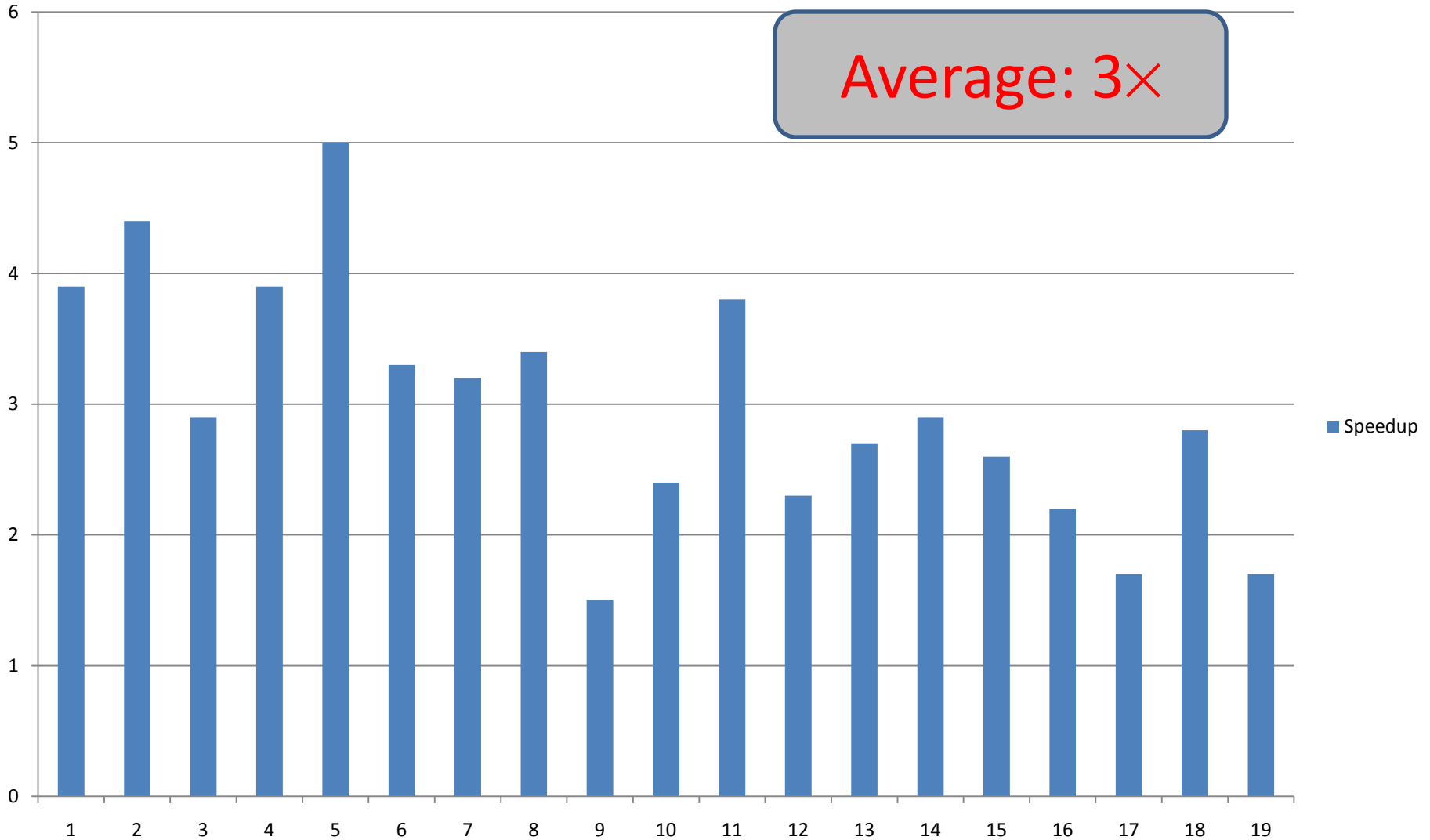
# Experiments

- Benchmarks
  - SpecJVM : 7 programs
  - DaCapo: 4 programs
  - Others: 8 programs
  - Number of methods ranging from 2344 to 8789
- Comparison between Sridharan-Bodik representation and ISPG without var nodes
  - $1.7 \times$  reduction in the number of nodes
  - $5.6 \times$  reduction in the number of edges

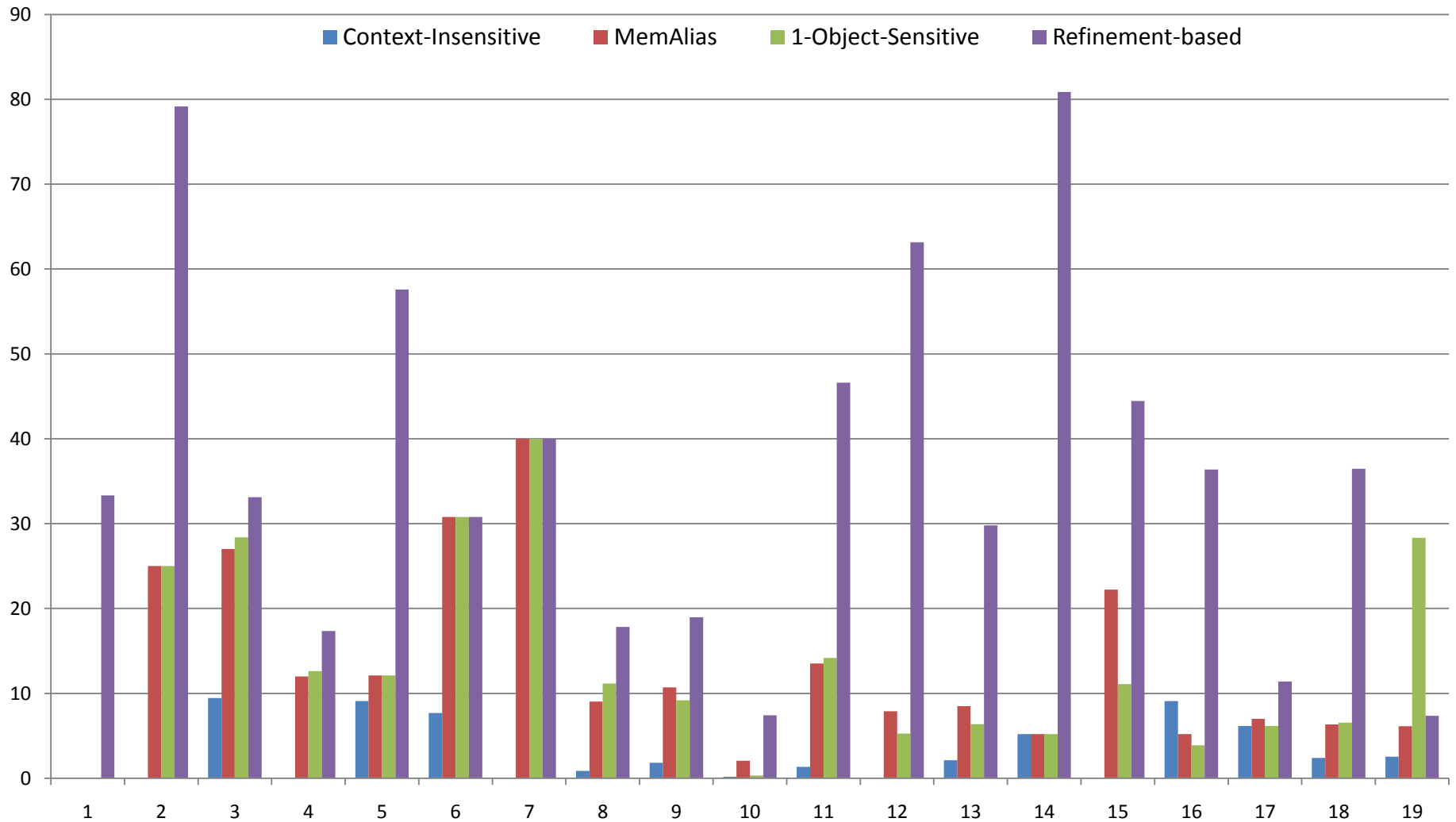
# Running Time Reduction

Speedup

Average: 3x



# Precision (casts proved safe)



CI: 3.2%, MA: 8.0%, 1-Obj: 10.5%, total: 23.5%

# Conclusions

- Refinement-based points-to analysis is precise but expensive
- A context-sensitive must-not-alias analysis
  - Pre-computes aliasing information
  - *memAlias*-reachability formulation
  - Used to quickly eliminate non-aliasing pairs in the points-to analysis
- Experimental results
  - Alias analysis has short running time
  - Significant time reduction for the points-to analysis
  - Points-to information derived from *memAlias* is almost as precise as 1-object-sensitive analysis

Thank you

# Running Time

- ISPG construction
  - 48 – 245s
  - On average **2.064** s/1000 Jimple statements
- Must-not-alias analysis
  - 9 – 80s
  - On average **0.579** s/1000 Jimple statements
- Modified points-to analysis
  - 185 – 2350s
  - On average **9.65** s/1000 Jimple statements
- Total
  - 282 – 2854s
  - On average **12.294** s/1000 Jimple statements