Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java

Ana Milanova Atanas Rountev Barbara G. Ryder
Department of Computer Science
Rutgers University
New Brunswick, NJ 08901
{milanova,rountev,ryder}@cs.rutgers.edu

ABSTRACT

The goal of *points-to analysis* for Java is to determine the set of objects pointed to by a reference variable or a reference objet field. Improving the precision of practical points-to analysis is important because points-to information has a wide variety of client applications in optimizing compilers and software engineering tools. In this paper we present *object sensitivity*, a new approach to context-sensitive flow-insensitive points-to analysis for Java. The key idea of our approach is to analyze a method separately for each of the objects on which this method is invoked. To ensure flexibility and practicality, we propose a parameterization framework which allows analysis designers to control the cost-precision tradeoffs of the object-sensitive analysis.

Side-effect analysis determines the memory locations that can be modified by the execution of a program statement. This information is needed for various compiler optimizations and software engineering tools. We present a new form of side-effect analysis for Java which is based on object-sensitive points-to analysis.

We have implemented one instantiation of our parameterized object-sensitive points-to analysis. We compare this instantiation with a context-insensitive points-to analysis for Java which is based on Andersen's analysis for C [4]. On a set of 23 realistic Java programs, our experiments show that the two analyses have comparable cost. In some cases the object-sensitive analysis is actually faster than the context-insensitive analysis. Our results also show that object sensitivity significantly improves the precision of side-effect analysis, call graph construction, and virtual call resolution.

Our empirical results demonstrate that object-sensitive analyses are capable of achieving significantly better precision than context-insensitive ones, while at the same time remaining efficient and practical. Thus, object-sensitive analysis is a better candidate for a relatively precise, practical, general-purpose points-to analysis for Java.

1. INTRODUCTION

Points-to analysis is a fundamental static analysis used by optimizing Java compilers and software engineering tools to determine the set of objects whose addresses may be stored in reference variables and reference object fields. These *points-to sets* are typically computed by constructing one or more *points-to graphs*, which serve as abstractions of the run-time memory states of the analyzed program. (An example of a points-to graph is shown in Figure 1, which is discussed in Section 2.1)

Optimizing Java compilers can use points-to information to perform various optimizations such as virtual call resolution, removal of unnecessary synchronization, and stack-based object allocation. Points-to analysis is also a prerequisite for a variety of other analyses—for example, *side-effect analysis*, which determines the memory locations that may be modified by the execution of a statement, and *def-use analysis*, which identifies pairs of statements that set the value of a memory location and subsequently use that value. These analyses are necessary to perform compiler optimizations such as code motion and partial redundancy elimination. In addition, such analyses are needed in the context of software engineering tools: for example, def-use analysis is needed for program slicing and data-flow-based testing. Points-to analysis is a crucial prerequisite for employing these analyses and optimizations.

Because of this wide range of applications, it is important to investigate approaches for precise and efficient computation of points-to information. The two major dimensions in the design space of points-to analysis are flow sensitivity and context sensitivity. Intuitively, *flow-sensitive* analyses take into account the flow of control between program points inside a method, and compute separate solutions for these points. *Flow-insensitive* analyses ignore the flow of control between program points, and therefore can be less precise and more efficient than flow-sensitive analyses. *Context-sensitive* analyses distinguish between the different contexts under which a method is invoked, and analyze the method separately for each context. *Context-insensitive* analyses do not separate the different invocation contexts for a method, which improves efficiency at the expense of some possible precision loss.

Recent work [18, 25, 15, 19] has shown that flow- and context-insensitive points-to analysis for Java can be efficient and practical even for large programs, and therefore is a realistic candidate for use in optimizing compilers and software engineering tools. However, context insensitivity inherently compromises the precision of points-to analysis for object-oriented languages such as Java. This imprecision results from fundamental object-oriented features and programming idioms. (Section 2 presents several examples that illustrate this point.) The imprecision decreases the impact of the points-to analysis on client optimizations (e.g.

virtual call resolution) and leads to less precise client analyses (e.g. def-use analysis). To make existing flow- and context-insensitive analyses more useful, it is important to introduce context sensitivity that targets the sources of imprecision that are specific to object-oriented languages. At the same time, the introduction of context sensitivity should not increase analysis cost to the point of compromising the practicality of the analysis.

In this paper we propose *object sensitivity* as a new form of context sensitivity for flow-insensitive points-to analysis for Java. Our approach uses *the receiver object at a method invocation site* to distinguish different calling contexts. Conceptually, every method is replicated for each possible receiver object. The analysis computes separate points-to sets for each replica of a local variable; each of those points-to sets is valid for method invocations with the corresponding receiver object.

We propose a parameterization framework that allows precision improvement through object sensitivity without incurring the cost of non-discriminatory replication of all variables. The analysis is parameterized by the set of variables for which the analysis designer wants to maintain multiple points-to sets. This *targeted replication* allows analysis designers to tune directly the cost of the analysis. The framework space ranges from context-insensitive analysis to precise object-sensitive analysis for which every local variable is replicated for every possible receiver object of its enclosing method.

In this paper we discuss parameterized object-sensitive points-to analysis that is based on an Andersen-style points-to analysis for Java. Andersen's analysis for C [4] is a well-known flow- and context-insensitive points-to analysis. Recent work [25, 15, 19] shows how to extend this analysis for Java. Although we demonstrate our technique on Andersen's analysis, parameterized object sensitivity can be trivially applied to enhance the precision of other flow- and context-insensitive analyses for Java (e.g., analyses that are based on flow- and context-insensitive points-to analyses for C [24, 23, 8]).

Modification side-effect analysis (MOD) determines, for each statement, the set of objects that may be modified by that statement. Similarly, USE analysis computes the set of objects that may be read by a statement. This information plays an important role in optimizing compilers and software productivity tools. Side-effect analysis requires the output of a points-to analysis. We define and evaluate a new object-sensitive MOD analysis that is based on the parameterized object-sensitive points-to analysis. Although we omit the discussion, our approach also applies to the corresponding USE analysis.

We have implemented one instantiation of our parameterized object-sensitive analysis. We compare this instantiation with an Andersen-style flow- and context-insensitive points-to analysis. For a set of 23 realistic Java programs, our experiments show that the cost of the two analyses is comparable. In some cases the object-sensitive analysis is actually faster than the context-insensitive analysis. We also evaluate the precision of the two analyses with respect to several client applications. MOD analysis based on object-sensitive points-to analysis is significantly more precise than the corresponding MOD analysis based on context-

```
class Y {...}
class X {
   Y f;
   void set(Y r)
       { this.f = r; }
   static void main() {
     s1: X p = new X();
     s2: Y q = new Y();
         p.set(q);
   }
}
```

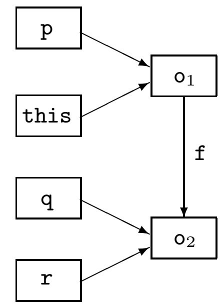

Figure 1: Sample program and its points-to graph.

insensitive points-to analysis. In addition, object sensitivity improves the precision of call graph construction and virtual call resolution. Our experimental results show that object-sensitive analyses are capable of achieving significantly better precision than context-insensitive ones, while at the same time remaining efficient and practical.

Contributions. The contributions of our work are the following:

- We propose object sensitivity as a new approach for context-sensitive, flow-insensitive points-to analysis for Java. We also define a parameterization framework that allows analysis designers to control the degree of object sensitivity and the cost/precision tradeoffs of the analysis.
- We define a new object-sensitive side-effect analysis for Java which is based on our parameterized object-sensitive points-to analysis.
- We compare one instantiation of our parameterized object-sensitive analysis with an Andersen-style flow- and context-insensitive analysis. Our experiments on a large set of programs show that the object-sensitive analysis is practical and significantly improves the precision of MOD analysis, call graph construction, and virtual call resolution.

Outline. The rest of the paper is organized as follows. Section 2 describes Andersen's analysis for Java and discusses some sources of imprecision due to context insensitivity. Section 3 defines our object-sensitive analysis. Section 4 discusses parameterized object sensitivity and Section 5 describes techniques for its efficient implementation. The new MOD analysis is defined in Section 6. The experimental results are presented in Section 7. Section 8 discusses related work and Section 9 presents conclusions and future work.

2. FLOW- AND CONTEXT-INSENSITIVE POINTS-TO ANALYSIS FOR JAVA

Previous work proposes various flow- and context-insensitive analyses for Java [18, 25, 15, 19]. These analyses are typically derived from similar analyses for C. This section discusses a flow- and context-insensitive points-to analysis for Java which is derived from Andersen's points-to analysis for C [4]. It also illustrates how context insensitivity compromises analysis precision.

$$f(G, s_i\colon l = \mathit{new}\ C) = G \cup \{(l, o_i)\}$$

$$f(G, l = r) = G \cup \{(l, o_i) \mid o_i \in Pt(G, r)\}$$

$$f(G, l.f = r) = \\ G \cup \{(\langle o_i, f\rangle, o_j) \mid o_i \in Pt(G, l)\ \wedge\ o_j \in Pt(G, r)\}$$

$$f(G, l = r.f) = \\ G \cup \{(l, o_i) \mid o_j \in Pt(G, r)\ \wedge\ o_i \in Pt(G, \langle o_j, f\rangle)\}$$

$$f(G, l = r_0.m(r_1, \ldots, r_n)) = \\ G \cup \{\mathit{resolve}(G, m, o_i, r_1, \ldots, r_n, l) \mid o_i \in Pt(G, r_0)\}$$

$$\mathit{resolve}(G, m, o_i, r_1, \ldots, r_n, l) = \\ \texttt{let}\ \ m_j(p_0, p_1, \ldots, p_n, \mathit{ret}_j) = \mathit{dispatch}(o_i, m)\ \ \texttt{in} \\ \{(p_0, o_i)\} \cup f(G, p_1 = r_1) \cup \ldots \cup f(G, l = \mathit{ret}_j)$$

Figure 2: Points-to effects of program statements for Andersen's analysis.

2.1 Analysis Semantics

Andersen's analysis for Java is defined in terms of three sets. Set R contains all reference variables in the analyzed program (including static variables). Set O contains names for all objects created at object allocation sites; for each allocation site s_i, there is a separate object name $o_i \in O$. Set F contains all instance fields in program classes. The analysis constructs *points-to graphs* containing two kinds of edges. Edge $(r, o_i) \in R \times O$ shows that reference variable r points to object o_i. Edge $(\langle o_i, f\rangle, o_j) \in (O \times F) \times O$ shows that field f of object o_i points to object o_j. A sample program and its points-to graph are shown in Figure 1.

For brevity, we only discuss the kinds of statements listed below. Other kinds of statements (e.g. calls to constructors and static methods) are handled in a similar fashion.

- Direct assignment: `l = r`
- Instance field write: `l.f = r`
- Instance field read: `l = r.f`
- Object creation: `l = new C`
- Virtual invocation: $\texttt{l = r}_0\texttt{.m(r}_1\texttt{,...,r}_k\texttt{)}$

At a virtual call, name `m` uniquely identifies a method in the program. This method is the *compile-time* target of the call, and is determined based on the declared type of $\texttt{r}_0$ [12, Section 15.11.3]. At run-time, the invoked method is determined by examining the class of the receiver object and all of its superclasses, and finding the first method that matches the signature and the return type of `m` [12, Section 15.11.4].

Analysis semantics is defined in terms of *transfer functions* that add new edges to points-to graphs. Each transfer function represents the semantics of a program statement. The functions for different statements are shown in Figure 2 in the format $f(G, s) = G'$, where s is a statement, G is an input points-to graph, and G' is the resulting points-to graph. $Pt(G, x)$ denotes the points-to set (i.e., the set of all successors) of x in graph G. The solution computed by the analysis is a points-to graph that is the closure of the empty graph under the application of all transfer functions for program statements.

```
  class X {...}
  class Y {
      X f;
1     void set(X x) { this.f = x; } }

2 s1: X x1 = new X();
3 s2: X x2 = new X();
4 s3: Y y1 = new Y();
5 s4: Y y2 = new Y();
6     y1.set(x1);
7     y2.set(x2);
```

Figure 3: Imprecision due to field encapsulation.

For most statements, the effects on the points-to graph are straightforward; for example, statement $l = r$ creates new points-to edges from l to all objects pointed to by r. For virtual call sites, resolution is performed for every receiver object pointed to by r_0. Function *dispatch* uses the class of the receiver object and the compile-time target of the call to determine the actual method m_j invoked at run-time. Variables $p_0, \ldots, p_n$ are the formal parameters of the method; variable p_0 corresponds to the implicit parameter `this`. Variable ret_j contains the return values of m_j (we assume that each method has a unique variable that is assigned all values returned by the method; this can be achieved by inserting auxiliary assignments).

2.2 The Imprecision of Context-Insensitive Analysis

This section presents several examples of basic object-oriented features and programming idioms for which context-insensitive analysis produces imprecise results.

2.2.1 Encapsulation

Figure 3 illustrates the typical situation when an encapsulated field is written through a modifier method. At the call site at line 6, $\texttt{y}_1$ points to o_3 and $\texttt{x}_1$ points to o_1. After the analysis applies the transfer function for the virtual call (as shown in Figure 2), the implicit parameter `this` of method `set` points to o_3 and formal parameter `x` points to o_1. After the analysis processes the call at line 7, `this` points to o_4 and `x` points to o_2. Thus, at statement `this.f=x` at line 1, the analysis erroneously infers points-to edges $(\langle o_3, f\rangle, o_2)$ and $(\langle o_4, f\rangle, o_1)$.

The imprecision can be avoided if the analysis distinguishes invocations of `set` on o_3 from invocations of `set` on o_4. This could be achieved if the analysis were able to associate *multiple* points-to sets with `this` and with `x`, one for each of the objects on which `set` is invoked. This would allow statement `this.f=x` to be analyzed separately for each of the receiver objects, and would avoid creating spurious points-to edges. In Section 3 we show how object-sensitive analysis achieves this goal.

During context-insensitive analysis, there is a single copy of every method for all possible invocations. Therefore, field `f` of *each* receiver object will point to *all* objects passed as arguments to the method which sets the value of `f`. In object-oriented languages, encapsulation and information hiding are strongly supported, and fields are almost always accessed

```
   class X { void n() {...} }
   class Y extends X { void n() {...} }
   class Z extends X { void n() {...} }

   class A {
       X f;
1      A(X xa) { this.f = xa; } }

   class B extends A {
2      B(X xb) { super(xb); ...  }
       void m() {
3          X xb = this.f;
4          xb.n(); } }

   class C extends A {
5      C(X xc) { super(xc); ...  }
       void m() {
6          X xc = this.f;
7          xc.n(); } }

8  s1: Y y = new Y();
9  s2: Z z = new Z();
10 s3: B b = new B(y);
11 s4: C c = new C(z);
12     b.m();
13     c.m();
```

Figure 4: Field assignment through a superclass.

indirectly through method invocations. As a result, context-insensitive analysis can incur significant imprecision.

2.2.2 Inheritance

Consider the example in Figure 4. At line 2, `B.this` points to o_3 and `B.xb` points to o_1. After the analysis processes the call to the superclass constructor, `A.this` and `A.xa` point to o_3 and o_1, respectively. Because of the call at line 5, `A.this` will point to o_4 and `A.xa` will point to o_2. Thus, at statement `this.f=xa` at line 1, spurious edges $(\langle o_3, f\rangle, o_2)$ and $(\langle o_4, f\rangle, o_1)$ are added to the graph. The imprecision propagates further, as the analysis infers that `xb` at line 3 points to both o_1 (of class `Y`) and o_2 (of class `Z`). Therefore, it appears that the possible targets of the virtual call at line 4 are `Y.n` and `Z.n` (the same problem also occurs at line 7). As a result, the calls at lines 4 and 7 cannot be devirtualized[1] using the solution computed by the context-insensitive analysis. The imprecision is due to statement `this.f=xa` in the constructor of superclass `A`, which merges the information for all possible receiver objects.

In the presence of inheritance, instance fields are often located in superclasses and are written through invocations of superclass constructors or methods. During context-insensitive analysis, fields of subclass instances are perceived to point to objects intended for instances of other subclasses. In the presence of wide and deep inheritance hierarchies, context insensitivity can lead to substantial imprecision.

[1]If a virtual call has only one target, it can be replaced with a direct call. This optimization eliminates the run-time overhead of virtual dispatch and enables method inlining.

```
   class Container {
       Object[] data;
       Container(int size) {
1       s1: Object[] data_tmp = new Object[size];
2          this.data = data_tmp; }
       void put(Object e,int at) {
           Object[] data_tmp = this.data;
3          data_tmp[at] = e; }
       Object get(int at) {
4          Object[] data_tmp = this.data;
5          return data_tmp[at]; } }

6  s2: Container c1 = new Container(100);
7  s3: Container c2 = new Container(200);
8  s4: X x = new X();
9      c1.put(x,0);
10 s5: X y = new Y();
11     c2.put(y,1);
```

Figure 5: Simplified container class.

2.2.3 Collections and Maps

Consider the example in Figure 5. There is a single object name o_1 which represents the `data` arrays of both instances of `Container`. Therefore, objects stored in individual containers appear to be shared between the two containers. In order to avoid this imprecision, the `data` array of every instance of `Container` should be represented by a distinct object name. In addition, the analysis should be able to assign distinct points-to sets to `put.this` and `put.e` for every possible receiver object of `put`.

Context insensitivity causes data that is stored in one instance of a collection or a map to be retrieved from every other instance of the same class, and very likely from all instances of its subclasses. Since collections (e.g., `Vector`) and maps (e.g., `Hashtable`) are commonly used in Java, context insensitivity can seriously compromise analysis precision.

3. OBJECT-SENSITIVE ANALYSIS

In context-sensitive analysis, a method is analyzed separately for different calling contexts. We define a new form of context-sensitive points-to analysis for Java which we refer to as *object-sensitive analysis*. With object sensitivity, each instance method (i.e., non-static method) and each constructor is analyzed separately for each object on which this method/constructor may be invoked. More precisely, the analysis uses a set of *object names* to represent objects allocated at run time. If a method/constructor may be invoked on run-time objects represented by object name o, the object-sensitive analysis maintains a separate contextual version of that method/constructor that corresponds to invocation context o.

Our object-sensitive analysis is based on Andersen's analysis for Java from Section 2.1. However, the same approach can be trivially applied to other flow- and context-insensitive analyses for Java (e.g., analyses derived from flow- and context-insensitive points-to analyses for C [24, 23, 8]). Section 3.1 defines the semantics of the object-sensitive analysis.

Section 3.2 discusses why object sensitivity is appropriate for flow-insensitive analysis of object-oriented programs, and compares this approach with other context-sensitive analyses.

3.1 Analysis Semantics

Our object-sensitive analysis is defined in terms of five sets. Recall from Section 2.1 that set R contains all reference variables in the analyzed program (including static variables), and set F contains all instance fields in program classes. Set S contains all object allocation sites in the program. We also use a set of object names O' and a set of replicated variables R'; both sets will be discussed shortly.

To simplify the presentation, we define a relation α which shows that a method or a constructor m may be invoked on instances of a given class C. Suppose that m is defined in some class D. Relation $\alpha(C, m)$ holds if and only if C and D are the same class or C is a subclass of D. Note that $\alpha(C, m)$ should hold even if m is overridden somewhere on the inheritance chain between D and C, because m could still be invoked on instances of C through `super`. We extend the notation to object names: for any $o \in O'$ which represents instances of class C, $\alpha(o, m)$ if and only if $\alpha(C, m)$.

The analysis uses a set of object names $O' \subseteq S \times (S \cup \{\epsilon\})$. Consider an allocation site $s_i \in S$ in method m. If m is a static method, the objects allocated by s_i are represented by a single object name $o_{i\epsilon}$. If m is an instance method or a constructor, the objects allocated by s_i are represented by a set of object names $o_{ij} \in O'$. There is a separate name o_{ij} for each allocation site $s_j : l = new\ C$ for which $\alpha(C, m)$ holds. Name o_{ij} represents all run-time objects that were created at s_i when m was invoked on an object created at s_j. For example, allocation site s_1 in Figure 5 appears in constructor `Container`. Sites s_2 and s_3 create instances of `Container`; thus, there are two object names o_{12} and o_{13} which correspond to s_1.

Set $\mathcal{C} = O' \cup \{\epsilon\}$ represents the space of all possible contexts for our object-sensitive analysis. A static method is always analyzed under the empty context ϵ. Any instance method or constructor m is separately analyzed for each context $o \in O'$ for which $\alpha(o, m)$ holds. This separation is achieved by maintaining *multiple replicas* of reference variables for each possible context. The set of replicated reference variables R' is defined by a function $map : R \times \mathcal{C} \to R'$. If $r \in R$ is a static variable or a local variable in a static method, r is mapped to itself. If r is a local variable in an instance method or a constructor m, r is mapped to a "fresh" variable r^o for every context $o \in O'$ for which $\alpha(o, m)$ holds. For example, in Figure 4 we have $\alpha(o_{3\epsilon}, \texttt{A.A})$ and $\alpha(o_{4\epsilon}, \texttt{A.A})$, and there are two copies of `A.this` and `A.xa` corresponding to contexts $o_{3\epsilon}$ and $o_{4\epsilon}$. For the rest of the paper we will refer to the elements of R' as *context copies*.

The object-sensitive analysis constructs points-to graphs in which the nodes are elements of R' and O'. Analysis semantics can be defined by transfer functions that add new edges to these points-to graphs. For statements that are located inside static methods, the transfer functions are identical to those in Figure 2. For statements located in instance methods and constructors, the transfer functions are presented in Figure 6. The effects of $F(G, s)$ are equivalent to applying the corresponding $f(G, s)$ from Figure 2 for each context from the set $\mathcal{C}_m = \{o \in \mathcal{C} \mid \alpha(o, m)\}$, where m is the method in which s is located.[2]

$$F(G, s_i\!: l = new\ C) = G \cup \bigcup_{o_{jk} \in \mathcal{C}_m} \{(l^{o_{jk}}, o_{ij})\}$$

$$F(G, l = r) = G \cup \bigcup_{c \in \mathcal{C}_m} f(G, l^c = r^c)$$

$$F(G, l.f = r) = G \cup \bigcup_{c \in \mathcal{C}_m} f(G, l^c.f = r^c)$$

$$F(G, l = r.f) = G \cup \bigcup_{c \in \mathcal{C}_m} f(G, l^c = r^c.f)$$

$$F(G, l = r_0.m(r_1, \ldots, r_n)) = G \cup \bigcup_{c \in \mathcal{C}_m} \{resolve(G, m, o_{ij}, r_1^c, \ldots, r_n^c, l^c) \mid o_{ij} \in Pt(G, r_0^c)\}$$

$$\begin{aligned} &resolve(G, m, o_{ij}, r_1^c, \ldots, r_n^c, l^c) = \\ &\quad \texttt{let}\ c' = o_{ij} \\ &\qquad m_j(p_0, p_1, \ldots, p_n, ret_j) = dispatch(o_{ij}, m)\ \texttt{in} \\ &\quad \{(p_0^{c'}, o_{ij})\} \cup f(G, p_1^{c'} = r_1^c) \cup \ldots \cup f(G, l^c = ret_j^{c'}) \end{aligned}$$

Figure 6: Points-to effects of statements in instance methods and constructors for object-sensitive analysis. $\mathcal{C}_m$ is the set of possible contexts for the enclosing method m. r^c denotes $map(r, c)$.

Example. Consider the set of statements in Figure 4. Since we have $\alpha(\texttt{B}, \texttt{B.B})$ and $\alpha(\texttt{B}, \texttt{A.A})$, set R' includes

$$\{\texttt{B.this}^{o_{3\epsilon}},\ \texttt{B.xb}^{o_{3\epsilon}},\ \texttt{A.this}^{o_{3\epsilon}},\ \texttt{A.xa}^{o_{3\epsilon}}\}$$

Similarly, R' also includes

$$\{\texttt{C.this}^{o_{4\epsilon}},\ \texttt{C.xc}^{o_{4\epsilon}},\ \texttt{A.this}^{o_{4\epsilon}},\ \texttt{A.xa}^{o_{4\epsilon}}\}$$

At line 2, $\texttt{B.this}^{o_{3\epsilon}}$ points to $o_{3\epsilon}$ and $\texttt{B.xb}^{o_{3\epsilon}}$ points to $o_{1\epsilon}$. When the analysis processes the call to `A.A` at line 2, `A.this` and `A.xa` are mapped to the context copies corresponding to $o_{3\epsilon}$, and points-to edges ($\texttt{A.this}^{o_{3\epsilon}}$, $o_{3\epsilon}$) and ($\texttt{A.xa}^{o_{3\epsilon}}$, $o_{1\epsilon}$) are added to the graph. Similarly, because of line 5, $\texttt{A.this}^{o_{4\epsilon}}$ points to $o_{4\epsilon}$ and $\texttt{A.xa}^{o_{4\epsilon}}$ points to $o_{2\epsilon}$. Statement `this.f=xa` at line 1 occurs in the context of $o_{3\epsilon}$ and $o_{4\epsilon}$. Thus, we have

$$\texttt{A.this}^{o_{3\epsilon}}\texttt{=A.xa}^{o_{3\epsilon}} \qquad \texttt{A.this}^{o_{4\epsilon}}\texttt{=A.xa}^{o_{4\epsilon}}$$

which produces edges $(\langle o_{3\epsilon}, f\rangle, o_{1\epsilon})$ and $(\langle o_{4\epsilon}, f\rangle, o_{2\epsilon})$.

3.2 Advantages of Object Sensitivity

In object-oriented languages such as Java, one of the primary roles of instance methods is to access or modify the state of the objects on which they are invoked. Instance methods typically work on encapsulated data, using implicit parameter `this` to modify or retrieve data from the object structure rooted at the receiver object. If points-to analysis does *not* distinguish the different receiver objects of instance methods, the states of these objects are essentially merged and any access/modification of the state of one object is

[2]For simplicity, we present the semantics as if *all* elements of $\mathcal{C}_m$ are possible contexts. As discussed in Section 5, analysis implementations only need to consider contexts that actually occur at calls to m.

propagated to all other objects. Therefore, it is crucial to distinguish the different objects pointed to by `this` and to analyze instance methods separately for different receiver objects. Similarly, the role of a constructor is to create the initial state of an object. To avoid merging the initial states of all objects pointed to by `this`, points-to analysis should distinguish the different objects on which a constructor is invoked.

Context sensitivity mechanisms of finer granularity than a receiver object may create redundant contextual versions. For example, one of the most popular mechanisms for context sensitivity is the *call string* approach, which represents invocation context using a string of k enclosing call sites. For $k = 1$, a method is analyzed separately for each call site that invokes that method. For many statements, it is redundant to distinguish between *distinct* call sites that have the *same* receiver object. For example, if statement `this.f=formal` were analyzed separately for distinct call sites that have the same receiver object, the effect would be the same as if it were analyzed once for that object: field f of the receiver would point to all objects in the points-to sets of the corresponding actual parameters at all call sites. Clearly, because of the flow insensitivity of the analysis, the effects of the distinct per-call-site versions of the statement are merged. The same kind of redundancy also occurs for statements that read the value of any field of the receiver object (e.g., `l=this.f`), as well as for certain method invocations on the receiver (e.g., `l=this.m()`). Therefore, such redundancies cause the call string approach to incur increased analysis cost without any precision gain. On the other hand, object-sensitive analysis performs exactly the necessary amount of work for such statements.

In certain cases, distinguishing calling context by a chain of enclosing call sites can be less precise than distinguishing context per receiver object. To illustrate such a case, recall the set of statements from Figure 4. Suppose that the following new statement is added at line 14:

14 s_5 : `C c`$_2$ `= new C(y);`

If calling context is distinguished per call site ($k = 1$), the effects of constructor `A.A` invoked at line 5 are merged for receivers o_4 and o_5. Thus, there are redundant points-to edges $(\langle o_4, f\rangle, o_1)$ and $(\langle o_5, f\rangle, o_2)$. The imprecision propagates and affects both the points-to analysis and its clients: the virtual call at line 10 cannot be resolved.

4. PARAMETERIZED OBJECT SENSITIVITY

In this section we define a parameterized framework for object-sensitive analysis. The framework encompasses a family of analyses which range from the least precise and least costly context-insensitive Andersen's analysis to the most precise and costly object-sensitive analysis described in Section 3.

The framework is parameterized in two dimensions. First, the analysis designer can select the set of object allocation sites for which a more precise naming scheme should be used. The analysis uses multiple object names for the selected sites and single object names for all other sites. Second, the analysis designer can specify the set of reference variables for which multiple points-to sets should be maintained. The analysis replicates only these selected variables.

The goal of the parameterization is to enhance the flexibility of the object-sensitive analysis. By varying the number of selected allocation sites and variables, the analysis designer can control directly the size of the points-to graph and the cost of the analysis. The parameterization also allows *targeted replication* rather than global non-discriminatory replication. The analysis designer can choose objects and variables for which keeping more precise information is likely to improve the points-to solution (e.g., implicit parameters `this`, formal parameters, return variables, etc.).

The parameterization is based on two sets S^* and R^*. Set $S^* \subseteq S$ contains the object allocation sites for which the analysis designer wants to use the more precise naming scheme from Section 3.1. Set $R^* \subseteq R$ contains the set of reference variables that should be replicated during the analysis.

The set of object names $O' \subseteq S \times (S \cup \{\epsilon\})$ is defined as follows. If $s_i \in S$ is located in an instance method or a constructor m and $s_i \in S^*$, there is an object name $o_{ij} \in O'$ for each allocation site $s_j : l = new\ C$ for which $\alpha(C, m)$. For any other s_i, there is a single object name $o_{i\epsilon} \in O'$. Function $map : R \times \mathcal{C} \to R'$ constructs R' based on parameter set $R^* \subseteq R$. If $r \in R^*$ is a local variable in an instance method or a constructor m, r is mapped to a "fresh" variable r^o for every context $o \in O'$ such that $\alpha(o, m)$. Any other variable is mapped to itself. Thus, *map* replicates variables in R^* for all applicable contexts, and preserves variables not in R^* (i.e., $map(r, c) = r$ for any $r \notin R^*$).

The transfer functions from Figure 6 can be modified in a straightforward fashion for the parameterized analysis. For example, the transfer function for object creation becomes

$$F(G, s_i\colon l = new\ C) =$$
$$\begin{cases} G \cup \bigcup_{o_{jk} \in \mathcal{C}_m} \{(map(l, o_{jk}), o_{ij})\} & \text{if } s_i \in S^* \\ G \cup \bigcup_{o_{jk} \in \mathcal{C}_m} \{(map(l, o_{jk}), o_{i\epsilon})\} & \text{otherwise} \end{cases}$$

The transfer functions for other program statements are identical to the ones from Figure 6, except for the use of the modified function *map* based on parameter set R^*.

5. IMPLEMENTATION TECHNIQUES

A typical implementation of Andersen's flow- and context-insensitive analysis for Java uses a *statement processing routine* which processes different kinds of program statements, and a *virtual dispatch routine* which models the semantics of virtual calls. The parameterized object-sensitive analysis can be build on top of any such existing implementation I of Andersen's analysis for Java. This can be achieved by (i) implementing function $map(v, c)$, (ii) augmenting the statement processing routine in I to process each statement once for every possible context in accordance with the rules from Figure 6, and (iii) augmenting the virtual dispatch routine in I to map the formal parameters and return variable of the invoked method to the corresponding invocation context.

Let I' be an implementation of the parameterized analysis which augments I with function *map* and alters the statement processing routine and the virtual dispatch routine.

Any such I' can be optimized in several ways.

First, the semantics in Figure 6 implicitly assumes that all possible contexts of a method m are actually used at calls to that method—that is, m is invoked with every context o for which $\alpha(o, m)$ holds. Clearly, I' can keep track of which contexts actually occur at calls to m. Thus, I' would take into account the effects of a statement in m for context o if and only if m has been invoked with receiver object o.

Second, whenever the points-to set of a replica $\texttt{this}^o$ is needed, the analysis can return the singleton set $\{o\}$. Thus, I' can avoid storing replicas $\texttt{this}^o$ and redundant points-to edges as well as retrieving the points-to set of $\texttt{this}^o$.

Third, whenever I' processes a statement s which contains only non-replicated variables, there is no need to analyze s multiple times for different contexts. In other words, transfer function $F(G, s)$ from Figure 6 can be replaced with $f(G, s)$ from Figure 2. For the rest of this paper we refer to such statements as *context-independent*, while statements that need to be analyzed multiple times for different contexts are referred to as *context-dependent*.

Fourth, some further simplifications of transfer functions can be carried out. Let l be a replicated local variable. The simplification can be performed as follows: (i) create a new non-replicated variable l', (ii) create a new (context-dependent) statement `l'=l`, and (iii) replace l with l' in all assignment statements of the form `l.f=p`, `p=l` and `p.f=l` for which $p \notin R^*$. As a result, all such statements become context-independent and therefore are inexpensive to process. Similarly, replacement of l with l' can also be performed for virtual call $\texttt{l.m(p}_1\texttt{,..,p}_n\texttt{)}$ if $p_i \notin R^*$ for every i, as well as for $\texttt{r.m(p}_1\texttt{,..,l,..,p}_n\texttt{)}$ if $r \notin R^*$. It is straightforward to show that this optimization does not affect analysis correctness or precision.

We conclude this section with a brief note on the relative complexity of object-sensitive analysis. In general, for every statement s which involves access through `this`, the work done by a context-insensitive analysis is proportional to the size of the points-to set of `this` (i.e., the number of receivers of the method enclosing s). Clearly, the work done by an object-sensitive analysis for s is also proportional to the number of receivers of the enclosing method. Therefore, assuming equivalent object naming schemes, the two analyses perform comparable amount of work on program statements involving `this`. Because field access and method invocation through `this` are prevalent in object-oriented programs, one might expect that the cost of object-sensitive analysis may be comparable to the cost of context-insensitive analysis.

6. SIDE-EFFECT ANALYSIS

In this section we present a MOD analysis based on object-sensitive points-to analysis. Our MOD algorithm computes a set of modified objects $Mod(s, c) \subseteq O'$ for each statement s and for each context c of the method containing s. The algorithm is shown in Figure 7. $Pt(x)$ denotes the set of objects pointed to by context copy x. We say that statement s *appears* in context c if α holds between c and the enclosing method of s. $MMod(m, c)$ stores the sets of objects modified by each contextual version of a method (i.e., objects that are modified when m is invoked with context c). For virtual calls (lines 6-10) the target methods are determined for each receiver object o_{ij} in context c, based on the class

input *Stmt*: set of statements map: $R \times \mathcal{C} \rightarrow R'$
Methods: set of methods Pt: $R' \rightarrow \mathcal{P}(O')$
output Mod: $Stmt \times \mathcal{C} \rightarrow \mathcal{P}(O')$
declare $MMod$: $Methods \times \mathcal{C} \rightarrow \mathcal{P}(O')$
[1] **foreach** indirect write $s: p.f = q \in Stmt$ **do**
[2] **foreach** context c in which s appears **do**
[3] $Mod(s, c) := \{o_{ij} \mid o_{ij} \in Pt(map(p, c))\}$
[4] add $Mod(s, c)$ to $MMod(EnclMethod(s), c)$
[5] **while** changes occur in Mod or $MMod$ **do**
[6] **foreach** virtual call $s: l = r.m(...) \in Stmt$ **do**
[7] **foreach** context c in which s appears **do**
[8] **foreach** object $o_{ij} \in Pt(map(r, c))$ **do**
[9] $Mod(s, c) := Mod(s, c) \cup$ $\{o_{kl} \mid o_{kl} \in MMod(target(o_{ij}, m), o_{ij})\}$
[10] add $Mod(s, c)$ to $MMod(EnclMethod(s), c)$
[11] **foreach** static call $s: l = C.m(...) \in Stmt$ **do**
[12] **foreach** context c in which s appears **do**
[13] $Mod(s, c) := Mod(s, c) \cup MMod(m, \epsilon)$
[14] add $Mod(s, c)$ to $MMod(EnclMethod(s), c)$

Figure 7: Object-sensitive MOD analysis. $\mathcal{P}(X)$ denotes the power set of X.

of o_{ij} and the compile-time target m. In addition, object o_{ij} determines which set of modified objects associated with the target method will be added to the *Mod* set at line 9. For static calls (lines 11-14) we use ϵ to denote the special empty context in which the statements in those methods appear.

Example. Consider the example in Figure 4. MOD analysis based on context-insensitive points-to analysis erroneously determines that the *Mod* sets for statements 1, 2, and 5 are $\{o_{3\epsilon}, o_{4\epsilon}\}$. Consider a MOD analysis based on the precise object-sensitive points-to analysis from Section 3. The statement at line 1 appears in two contexts: $o_{3\epsilon}$ and $o_{4\epsilon}$. Therefore, $MMod(A.A, o_{3\epsilon})$ is $\{o_{3\epsilon}\}$ and $MMod(A.A, o_{4\epsilon})$ is $\{o_{4\epsilon}\}$. The receiver for the call statement at line 2 is $o_{3\epsilon}$; therefore the MOD analysis infers that $Mod(2, o_{3\epsilon})$ is $\{o_{3\epsilon}\}$. Similarly $Mod(5, o_{4\epsilon})$ is $\{o_{4\epsilon}\}$.

7. EMPIRICAL RESULTS

We chose to implement one particular instantiation of the parameterized object-sensitive points-to analysis. In this instantiation we replicate implicit parameters `this`, formal parameters, and return variables of instance methods and constructors (i.e., S^* is empty and R^* contains `this`, formals, and return variables of non-static methods). Given that instance methods and constructors in Java are usually short, keeping precise information for these variables has the potential to improve considerably the points-to solution without significant increase in analysis cost. This instantiation, which we denote by *ObjSens*, was compared with Andersen's context-insensitive points-to analysis for Java (denoted by *And*).

The object-sensitive analysis is built on top of an existing constraint-based implementation of Andersen's analysis [19], using the optimization techniques described in Section 5. We use the Soot framework (`www.sable.mcgill.ca`) to process Java bytecode and to build a typed intermediate representation [28]. The points-to analysis implementation

Program	User Class	Size (Kb)	Whole-program Class	Method	Stmt
proxy	18	56.6	565	3283	58837
compress	22	76.7	568	3316	60010
db	14	70.7	565	3339	60747
jb-6.1	21	55.6	574	3393	60898
echo	17	66.7	577	3544	62646
raytrace	35	115.9	582	3451	62755
mtrt	35	115.9	582	3451	62760
jtar-1.21	64	185.2	618	3583	65112
jlex-1.2.5	25	95.1	578	3381	65437
javacup-0.10	33	127.3	581	3564	66463
rabbit-2	52	157.4	615	3770	68277
jack	67	191.5	613	3573	69249
jflex-1.2.2	54	198.2	608	3692	71198
jess	160	454.2	715	3973	71207
mpegaudio	62	176.8	608	3531	71712
jjtree-1.0	72	272.0	620	4078	79587
sablecc-2.9	312	532.4	864	5151	82418
javac	182	614.7	730	4470	82947
creature	65	259.7	626	3881	83454
mindterm1.1.5	120	461.1	686	4420	90451
soot-1.beta.4	677	1070.4	1214	5669	92521
muffin-0.9.2	245	655.2	824	5253	94030
javacc-1.0	63	502.6	615	4198	102986

Table 1: Characteristics of the data programs. First two columns show the number and bytecode size of user classes. Last three columns include library classes.

Program	*And* Time [sec]	*And* Memory [Mb]	*ObjSens* Time [sec]	*ObjSens* Memory [Mb]
proxy	11.9	40	8.1	38
compress	22.8	46	23.5	46
db	23.4	47	24.0	46
jb	9.0	43	10.7	41
echo	44.2	60	47.2	59
raytrace	26.1	50	24.7	51
mtrt	27.0	50	25.1	51
jtar	45.0	58	44.5	56
jlex	13.1	44	17.5	46
javacup	29.6	56	34.0	55
rabbit	29.9	53	28.6	52
jack	85.5	62	88.6	62
jflex	40.2	68	39.5	70
jess	48.8	67	54.1	67
mpegaudio	32.0	53	29.7	52
jjtree	23.7	53	24.4	52
sablecc	136.6	112	73.1	94
javac	973.4	122	956.9	122
creature	176.1	90	126.3	87
mindterm	82.3	91	93.0	88
soot	146.1	130	171.8	131
muffin	236.3	144	214.0	133
javacc	165.2	110	169.5	112

Table 2: Running time and memory usage of the analyses.

is based on the BANE toolkit (`bane.cs.berkeley.edu`) for constraint-based program analysis [3].

All experiments were performed on a 360MHz Sun Ultra-60 machine with 512Mb physical memory. The reported times are the median values out of three runs. We used 23 publicly available data programs, ranging in size from 56Kb to about 1Mb of bytecode. The same set of programs was used in our previous work on Andersen's analysis [19]. The set includes programs from the SPEC JVM98 suite, other benchmarks used in previous work on analysis for Java, as well as programs from an Internet archive (`www.jars.com`) of popular publicly available Java applications.

Table 1 shows some characteristics of the data programs. The first two columns show the number of user (i.e., non-library) classes and their bytecode size. The next three columns show the size of the program, including library classes, after using class hierarchy analysis (CHA) [9] to filter out irrelevant classes and methods.[3] The number of methods is essentially the number of nodes in the call graph computed by CHA. The last column shows the number of statements in Soot's intermediate representation.

7.1 Analysis Cost

The measurements of analysis cost are presented in Table 2. The first two columns show the running time and memory usage of Andersen's analysis.[4] The last two columns show the cost of *ObjSens*. The empirical results demonstrate that the object-sensitive analysis is practical in terms of running time and memory consumption. For the majority of programs it has comparable performance to Andersen's analysis. In certain cases (e.g., `sablecc` and `creature`) the cost of the object-sensitive analysis is significantly lower than the cost of the context-insensitive analysis.

There are two factors that could explain the cost of the object-sensitive analysis. First, the improved precision produces smaller points-to sets, which results in less work and reduced memory consumption for the analysis. In the case when the points-to sets are significantly smaller, *ObjSens* can actually run faster than *And*, as observed for some of our data programs. Second, even if the points-to sets were the same, for many statements *And* and *ObjSens* would perform comparable amount of work. One might expect that because *ObjSens* analyzes context-dependent statements multiple times (once for each context), *ObjSens* would be more expensive. However, for any statement s that accesses the receiver object (e.g., any s containing `this`), there are as many different contextual versions as the number of receivers of the enclosing method. When *And* processes s, it has to consider all of the possible receivers. The amount of work that *And* has to perform for one receiver roughly corresponds to the amount of work that *ObjSens* performs for one contextual version. Therefore, for this statement *And* and *ObjSens* have comparable cost. Given that many statements in instance methods and constructors access the receiver object, one can explain why the two analyses exhibit comparable costs.

[3]CHA is an inexpensive analysis that determines the possible targets of a virtual call by examining the class hierarchy of the program.

[4]The results for Andersen's analysis reported in this paper differ from those reported in [19] because of some minor improvements in our current implementation.

Program	*And*			*ObjSens*		
	1-3	4-9	≥10	1-3	4-9	≥10
proxy	19%	6%	75%	75%	14%	11%
compress	23%	4%	73%	67%	9%	24%
db	20%	4%	76%	48%	25%	27%
jb	15%	5%	80%	67%	20%	13%
echo	25%	6%	69%	63%	11%	26%
raytrace	23%	5%	72%	66%	9%	25%
mtrt	23%	5%	72%	66%	9%	25%
jtar	18%	8%	74%	61%	15%	24%
jlex	17%	4%	79%	56%	34%	10%
javacup	14%	3%	83%	53%	38%	9%
rabbit	18%	5%	77%	47%	13%	40%
jack	17%	3%	80%	53%	8%	39%
jflex	19%	4%	77%	54%	34%	12%
jess	15%	5%	80%	60%	9%	31%
mpegaudio	23%	4%	73%	65%	9%	26%
jjtree	8%	2%	90%	32%	26%	42%
sablecc	20%	3%	77%	52%	15%	33%
javac	14%	4%	82%	37%	5%	58%
creature	18%	3%	79%	54%	13%	33%
mindterm	20%	8%	73%	55%	16%	29%
soot	16%	4%	80%	43%	15%	42%
muffin	16%	4%	80%	45%	7%	48%
javacc	10%	1%	89%	29%	49%	22%
Average	18%	4%	78%	54%	18%	28%

Table 3: Number of modified objects for program statements. Each column shows the percentage of statements whose number of modified objects is in the corresponding range.

Program	(a) Resolved Call Sites	(b) Removed Targets
proxy	12%	3%
compress	19%	13%
db	17%	14%
jb	45%	5%
echo	10%	13%
raytrace	18%	15%
mtrt	18%	15%
jtar	39%	7%
jlex	40%	5%
javacup	26%	5%
rabbit	31%	11%
jack	5%	12%
jflex	23%	3%
jess	17%	14%
mpegaudio	20%	17%
jjtree	48%	6%
sablecc	24%	183%
javac	7%	10%
creature	21%	5%
mindterm	9%	9%
soot	5%	1%
muffin	3%	7%
javacc	15%	4%
Average	21%	16%

Table 4: Improvements over context-insensitive analysis. (a) Increase in the number of resolved call sites. (b) Reduction in the number of target methods.

7.2 Analysis Precision

We evaluated the precision improvements of object-sensitive analysis over context-insensitive analysis with respect to MOD analysis, call graph construction and virtual call resolution.

7.2.1 MOD Analysis

Using the MOD algorithm described in Section 6, we performed measurements for *ObjSens* and *And* in order to estimate the impact of the analyses on MOD analysis. More precise points-to analyses produce a smaller number of modified objects per statement.

We considered all methods that *ObjSens* determined to be potentially executable (i.e., methods that are not dead). For all statements in such methods, we computed (i) *Mod* sets according to the algorithm from Figure 7, and (ii) *Mod* sets using Andersen's analysis and a corresponding context-insensitive version of the algorithm from Figure 7. In order to compare the output of the two analyses, for each statement we merged the *ObjSens*-based *Mod* sets for different contexts to obtain a single *Mod* set. For example, the aggregate *Mod* set for line 1 in Figure 4 is $\{o_{3\epsilon}, o_{4\epsilon}\}$, which is the union of $Mod(1, o_{3\epsilon})$ and $Mod(1, o_{4\epsilon})$.

Table 3 shows the distribution of the number of modified objects for the two analyses. Each column corresponds to a specific range of numbers. For example, the first column corresponds to statements that may modify one, two or three objects, while the last column corresponds to statements that may modify at least 10 objects. Each column shows what percentage of statements (counting only statements that modify at least one object) corresponds to the particular range of numbers of modified objects.

The measurements in Table 3 show that object sensitivity *significantly improves* analysis precision. For MOD analysis based on *ObjSens*, on average 54% of the statements modify at most three objects. In contrast, for MOD analysis based on *And* this percentage is 18%. It is also significant to note that for *And* nearly 80% of the statements modify at least 10 objects. This indicates substantial imprecision, which can be reduced significantly by using *ObjSens*.

The above empirical results show that object-sensitive analysis is a promising candidate for producing useful side-effect information. Such precise information is important for (i) implementing advanced optimizations in aggressive optimizing compilers, and (ii) improving the precision of software productivity tools, with the corresponding reduction in human time and effort spent on program understanding, reengineering, and testing.

7.2.2 Virtual Call Resolution and Call Graph Construction

One application of points-to analysis is to determine the potential target methods at virtual call sites. This information can be used to construct the program call graph (which is a prerequisite for all interprocedural analyses) and to identify virtual call sites that can be resolved to a single target method. We performed measurements to evaluate the improvement of *ObjSens* over *And* for virtual call resolution and call graph construction. (Andersen's analysis itself already produces precise call graph results [19].)

To determine the improvement in analysis precision, we considered call sites that could not be resolved to a single target method by CHA. Let V be the set of all CHA-unresolved call sites that occur in methods identified by *ObjSens* as executable. We computed the number of sites from V that were resolved to a single target method, according to *And* and according to *ObjSens*. The improvement in the number of resolved call sites for *ObjSens* over *And* is shown in the first column of Table 4. On average, *ObjSens* resolves 21% more sites than *And*. This increased precision allows better removal of redundant run-time virtual dispatch and enables additional method inlining.

We also computed the sum (over all sites in V) of the number of target methods according to *And*, as well as the corresponding sum according to *ObjSens*. The reduction in the total number of target methods (i.e., call edges removed at call sites) is shown in the second column of Table 4. On average, *ObjSens* removes 16% of the target methods determined by *And*. This improved precision is beneficial for reducing the cost and improving the precision of subsequent interprocedural analyses.

8. RELATED WORK

Flow-insensitive context-sensitive alias analysis for Java has been developed by Ruf [20] in the context of a specialized algorithm for synchronization removal. Ruf's analysis uses method summaries to model context sensitivity and, unlike our analysis, requires bottom-up traversal of the call graph (i.e., a called method is analyzed before or together with its callers). Also, our analysis is based on Andersen's analysis, which has cubic time worst case complexity [4]; Ruf's algorithm is based on the almost-linear Steensgaard's points-to analysis for C [24]. Other context-sensitive points-to analyses for Java are presented in [13, 6]. The algorithm in [6] uses method summaries to model context sensitivity, while [13] uses the call string approach. In general, these analyses are more precise and significantly more costly than ours, which is due to their flow sensitivity. Flow-insensitive context-insensitive points-to analyses for Java are described in [18, 25, 15, 19].

Liang et al. [15] propose user-defined models to solve the problem of data sharing between different instances of container classes (recall Figure 5 in Section 2.2). This approach may be more efficient than our proposed approach of using a more precise object naming for container classes. However, the models require human effort to construct and may need reexamination when a modelled class is subclassed.

Class analysis for object-oriented languages computes a set of classes for each program variable; this set approximates the classes of all run-time values for this variable. Typical clients of this information are call graph construction and virtual call resolution. Various practical context-insensitive class analyses are presented in [16, 11, 5, 10, 27, 26]. Different mechanisms for context-sensitivity which typically abstract context by some combination of the parameter class types have been studied in the context of class analysis [1, 17, 2, 13].

Conceptually, our MOD analysis is based on similar MOD analyses for C [22, 14, 21]. Razafimahefa [18] presents algorithms for side-effect analysis for Java which are based on context-insensitive information. The more precise of the algorithms is based on a points-to analysis for Java derived from Steensgaard's analysis for C [24]. These analyses do not take advantage of context-sensitive information and are less precise than our analysis. Clausen [7] investigates side-effect analysis for Java in the context of a Java bytecode optimizer. Similarly to the less precise algorithm developed in [18], Clausen's side-effect analysis does not use points-to information, i.e. a given modification through field f is assumed to write *all* objects whose class contains field f. This results in less precise side-effect information.

9. CONCLUSIONS AND FUTURE WORK

We present a framework for parameterized object-sensitive points-to analysis and a side-effect analysis based on it. The basic idea of our approach is to distinguish among the different receiver objects of a method. We show that object-sensitive analysis is capable of achieving significantly better precision than context-insensitive analysis, while at the same time remaining efficient and practical. Thus, object-sensitive analysis is a better candidate for a relatively precise, practical, general-purpose points-to analysis for Java.

In our future work we plan to investigate other instantiations of our framework, especially instantiations which involve more precise object naming schemes, and to assess their impact on a variety of client analyses. We also plan to develop hybrid analyses which combine object sensitivity with other mechanisms for context sensitivity. Another direction of future work is to investigate techniques for further reduction of the analysis cost (e.g., by analyzing the library code in advance and reusing the results for different client programs). Finally, we would like to investigate applications of points-to and side-effect analyses in the context of software productivity tools (e.g., tools for program slicing or dataflow-based testing). Such tools can play a useful role during the development and maintenance of large production-strength Java systems.

10. REFERENCES

[1] O. Agesen. Constraint-based type inference and parametric polymorphism. In *Static Analysis Symposium*, LNCS 864, pages 78–100, 1994.

[2] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming*, 1995.

[3] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *International Workshop on Types in Compilation*, 1998.

[4] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[5] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on*

Object-Oriented Programming Systems, Languages, and Applications, pages 324–341, 1996.

[6] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[7] L. R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[8] M. Das. Unification-based pointer analysis with directional assignments. In *Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[9] J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[10] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[11] A. Diwan, J. B. Moss, and K. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305, 1996.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[13] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, 1997.

[14] M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[15] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, June 2001.

[16] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, 1991.

[17] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, 1994.

[18] C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, Dec. 1999.

[19] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java based on annotated constraints. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, Oct. 2001.

[20] E. Ruf. Effective synchronization removal for Java. In *Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[21] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, Mar. 2001.

[22] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symposium*, LNCS 1302, pages 16–34, 1997.

[23] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[24] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[25] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an emprirical comparison. Technical report, U. Passau, Sept. 2000.

[26] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280, 2000.

[27] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, 2000.

[28] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, 2000.