Points-to Analysis for Java Using Annotated Inclusion Constraints

Atanas Rountev Ana Milanova Barbara G. Ryder
Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
{rountev,milanova,ryder}@cs.rutgers.edu

1 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 object field. In this paper we define and evaluate a points-to analysis for Java which extends Andersen's points-to analysis for C [4].

Andersen's analysis for C can be implemented efficiently by using systems of set-inclusion constraints and by employing several techniques for constraint representation and resolution. We extend these techniques to efficiently represent and solve systems of *annotated* inclusion constraints. The annotations play two roles in our analysis. *Method annotations* are used to model precisely and efficiently the semantics of virtual calls. *Field annotations* allow us to distinguish the flow of values through different fields of an object. In addition, our analysis keeps track of all methods reachable from the entry point of the program, and avoids analyzing dead library code.

We evaluate the performance of the analysis on a large set of realistic Java programs. Our results show that the analysis is practical and therefore will be useful as a relatively precise general-purpose points-to analysis for Java. The experiments also show that the points-to solution has significant impact on call graph construction, virtual call resolution, elimination of unnecessary synchronization, and stack-based object allocation.

2 Introduction

The goal of points-to analysis is to determine all memory locations whose address may be stored in a given variable p. This information is used to estimate the set of locations indirectly read or written through p. Without points-to information, many subsequent analyses and optimizations would be impossible.

There are many points-to analyses for C with various degrees of cost and precision [25, 24, 18, 4, 31, 39, 33, 32, 19, 26, 34, 21, 30, 14, 20, 10]. There is growing interest in points-to analysis for object-oriented languages; for example, for Java the goal of points-to analysis is to determine the set of objects pointed to by a reference variable or a reference object field. This information can be used (i) to construct the program call graph (needed by various other analyses), (ii) to resolve virtual calls (i.e., to treat them as direct calls if there is only one target method) and enable method inlining, and (iii) to determine whether all references to an object are localized in a specific portion of the program (e.g., a method or a thread) to allow stack-based object allocation and synchronization removal. (These and other applications of points-to analysis for Java are discussed in Section 4).

Object references in Java serve many of the same roles as general-purpose pointers in C; therefore, analysis techniques developed for C can be adapted for Java. The goal of this paper is to define and evaluate a points-to analysis for Java which extends Andersen's points-to analysis for C [4]. Andersen's analysis is a relatively precise flow- and context-insensitive analysis[1]. Even though it has cubic worst-case complexity, recent work has shown that the analysis can be implemented very efficiently using inclusion constraints. As a result, implementations of Andersen's analysis are capable of analyzing hundreds of thousands lines of C code in a few minutes [34, 30]. The goal of our work is to achieve similar performance for Java.

Some of the fastest implementations of Andersen's analysis for C define and solve systems of inclusion constraints of the form $L \subseteq R$, where L and R are points-to sets. Their performance is based on several techniques for efficient constraint representation and constraint resolution [19, 34]. We extend these techniques to efficiently represent and solve systems of *annotated* inclusion constraints of the form $L \subseteq_a R$, where a is an annotation. The annotations play two roles in the analysis. First, *method annotations* are used to model precisely and efficiently the semantics of virtual calls. Second, *field annotations* allow us to distinguish the flow of values through different fields of an object, analogous to tracing flow through fields of a C structure. The existing constraint-based implementations of Andersen's analysis for C do not make this distinction. However, our experiments show that if object fields are not treated as separate entities, the performance of the analysis deteriorates.

One disadvantage of Andersen's analysis is the implicit assumption that all code in the program is executable. Java programs contain large portions of unused library code; including such dead code can have negative effects on the analysis cost and precision. In our points-to analysis, we keep track of all methods reachable from the entry point of the

[1] A flow-insensitive analysis ignores the flow of control between program points. A context-insensitive analysis does not distinguish between different invocation sites of a procedure.

```
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);
  }
```

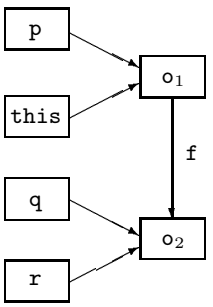

Figure 1: Sample points-to graph.

program, and only analyze reachable methods. An additional advantage of this approach is that it produces the reachability information as part of an integrated analysis.

We have implemented our analysis and evaluated its performance on a large set of realistic Java programs. On 15 out of the 23 data programs, the analysis runs in less than 100 seconds using less than 40Mb of memory. To the best of our knowledge, these are the first published empirical results showing that a relatively precise points-to analysis can run in practical time and space on realistic Java programs. This practicality shows that the analysis will be useful as a relatively precise general-purpose points-to analysis for Java.

Our experiments show that the computed points-to information can significantly improve the precision of the program call graph. Furthermore, using the points-to solution, we have been able to determine that in our multi-threaded data programs about 50% of the object allocation sites correspond to objects for which synchronization is unnecessary and can be safely removed. The points-to information also allowed us to show that for about 45% of all allocation sites, the objects can be stack-allocated instead of heap-allocated.

The contributions of our work are the following:

- We define a points-to analysis for Java that extends Andersen's points-to analysis for C [4].
- We show how to implement the analysis using annotated inclusion constraints. The implementation efficiently and precisely models virtual calls and flow of values through object fields, and only analyzes reachable methods.
- We evaluate the cost and precision of our analysis on a large set of realistic Java programs. We show that the analysis is practical both in time and memory usage. We also show that the points-to solution has significant impact on call graph construction, virtual call resolution, synchronization removal, and stack-based object allocation.

The rest of the paper is organized as follows. Section 3 defines the semantics of our points-to analysis. Section 4 discusses several applications of points-to analysis for Java. Section 5 describes the general structure of our annotated inclusion constraints, and Section 6 shows the specific kinds on constraints and annotations. Section 7 describes the analysis implementation. The experimental results are presented in Section 8. Section 9 discusses related work, and Section 10 presents conclusions and future work.

3 Points-to Analysis for Java

In this section we define the semantics of our points-to analysis for Java by extending the semantics of Andersen's points-to analysis for C. The implementation of the analysis using annotated inclusion constraints is described in Section 6.

$$\langle s_i\colon\ l = new\ C, G\rangle \Rightarrow G \cup \{(l, o_i)\}$$

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

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

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

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

$$resolve(G, m, o_i, r_1, \ldots, r_n, l) =$$
$$\texttt{let}\ \ m_j(p_0, p_1, \ldots, p_n, ret_j) = dispatch(o_i, m)\ \ \texttt{in}$$
$$\{(p_0, o_i)\} \cup \langle p_1 = r_1, G\rangle \cup \ldots \cup \langle l = ret_j, G\rangle$$

Figure 2: Points-to effects of program statements.

The analysis is defined in terms of three sets. Set R contains all reference variables in the analyzed program. Set O contains names for all objects created at object allocation sites. For each allocation site s_i, we use a separate object name $o_i \in O$. Set F contains all instance fields in program classes.

The semantics of the analysis is expressed as manipulations of *points-to graphs* with 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 simple program and its points-to graph are shown in Figure 1.

We assume that the program is represented by statements of the following form:

- Direct assignment: `l = r`
- Instance field write: `l.f = r`
- Instance field read: `l = r.f`
- Object creation: `l = new C`
- Virtual invocation: `l = r0.m(r1,...,rk)`

At a virtual call (i.e., method invocation site), 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 $\texttt{r}_0$ [22, Section 15.11.3]. At runtime, 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` [22, Section 15.11.4].

The analysis semantics is defined in terms of rules for adding new edges to points-to graphs. Each rule represents the semantics of a program statement. The analysis solution is the closure of the empty graph under the edge-addition rules. The rules for different program statements are shown in Figure 2, in the format $\langle s, G\rangle \Rightarrow G'$. Here s is a statement, G is the 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 effects on the points-to graph are straightforward, except for virtual call sites where resolution is performed for every possible receiver object. Function *dispatch* uses the class of the 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 value of m_j.

4 Applications of Points-to Analysis for Java

In this section we briefly discuss several applications of points-to analysis for Java. In our experiments, we have measured the precision of the analysis solution with respect to some of these applications.

Call Graph Construction and Virtual Call Resolution Points-to information can be used to determine the target methods of a virtual call by examining the classes of all possible receiver objects. The information about target methods is needed to construct the program call graph, which is a prerequisite for all interprocedural analyses and optimizations. If the call has only one target method, it can be resolved by replacing the virtual call with a direct call; furthermore, the target method can be inlined. In Section 8, we empirically evaluate the impact of our points-to solution on call graph construction and virtual call resolution.

Read-Write Information Points-to analysis can be used to determine what objects are read or written by every program statement. This information is needed to perform side-effect analysis and def-use analysis, which in turn are needed for various optimizations (e.g., partial redundancy elimination). For example, better points-to analysis could improve the def-use information used for removal of array bounds checks [7], the side-effect information needed to implement lazy exceptions [12], and the elimination of redundant cast checks and tests through `instanceof`.

Synchronization Removal Synchronization constructs in Java allow multiple threads to access shared objects safely. Synchronization operations can have considerable overhead; however, the synchronization can be removed for objects accessed by only one thread. Several kinds of escape analysis [11, 6, 8, 38] have been used to identify thread-local objects.

Points-to analysis can be used as an alternative to escape analysis in detecting thread-local objects. Consider an object o_i such that in the points-to graph computed by the analysis, o_i is not reachable from (i) static (i.e., global) reference variables, or (ii) objects of classes that implement interface `java.lang.Runnable`[2]. It is easy to see that such o_i does not escape the thread that created it. Thus, given the output of the points-to analysis, we can identify thread-local objects by traversing the points-to graph. In Section 8 we show empirical results from our experiments with this approach.

Stack Allocation If the lifetime of an object does not exceed the lifetime of the method that created it, the object can be allocated on the method's stack frame. This transformation reduces garbage collection overhead and opens up opportunities for additional optimizations in the method. Some escape analyses [11, 6, 38] have been used to identify objects that do not escape the methods that created them.

Similarly to thread-local objects, points-to analysis allows easy identification of method-local objects. Consider a thread-local object o_i such that in the points-to graph computed by the analysis, o_i is not reachable from the formal parameters or the return variable of the method that created o_i. In this case, o_i does not escape its creating method and can be allocated on the stack. Section 8 shows the empirical results from our experiments with this application.

[2]The `run` methods of such objects are the starting points of new threads.

5 Annotated Inclusion Constraints

This section describes the general structure of the annotated inclusion constraints used in our points-to analysis; the details about the specific kinds of constraints and annotations are discussed in Section 6.

Previous work on Andersen's analysis for C [19, 34] is based on non-annotated inclusion constraints and uses several techniques for efficient constraint representation and resolution. We extend this work by introducing annotations that allow us to model object fields and virtual calls in Java. By adapting the existing techniques for efficient constraint-based points-to analysis for C, we have been able to perform precise and practical points-to analysis for Java.

5.1 Constraint Language

We consider annotated set-inclusion constraints of the form $L \subseteq_a R$, where a is chosen from a given set of annotations. We assume that one element of this set is defined as the "empty" annotation, and will use $L \subseteq R$ to denote constraints labeled with the empty annotation. In our analysis, the annotations are used to model the flow of values through fields of objects, as well as the flow of values between a virtual call and the run-time target methods of the call.

L and R are expressions representing sets, defined by the following grammar:

$$L, R \rightarrow v \mid c(v_1, \ldots, v_n) \mid proj(c, i, v) \mid 0 \mid 1$$

Here v and v_i are set variables, $c(\ldots)$ are constructed terms and $proj(\ldots)$ are projection terms. Each *constructed term* is built from an n-ary constructor c. A constructor is either *covariant* or *contravariant* in each of its arguments; the role of this variance in constraint resolution will be explained shortly. Constructed terms may appear on both sides of inclusion relations. 0 and 1 represent the empty set and the universal set; they are treated as nullary constructors. *Projections* of the form $proj(c, i, v)$ are terms used to select the i-th argument of any constructed term $c(.., v_i, ..)$. Projection terms may appear only on the right-hand side of an inclusion.

5.2 Annotated Constraint Graphs

Systems of constraints from the above language can be represented as directed (multi)graphs. This representation is a natural extension of the graph representation for non-annotated constraints used to implement Andersen's analysis for C [19]. Constraint $L \subseteq_a R$ is represented by an edge from the node for L to the node for R; the edge is labeled with the annotation a. There could be multiple edges between the same pair of nodes, each with a different annotation.

The nodes in the graph can be classified as variables, sources, and sinks. *Sources* are constructed terms that occur on the left-hand side of inclusions. *Sinks* are constructed terms or projections that occur on the right-hand side of inclusions. The graph only contains edges that represent *atomic constraints* of the following forms: $Source \subseteq_a Var$, $Var \subseteq_a Var$, or $Var \subseteq_a Sink$. If the constraint system contains a *structural* (non-atomic) constraint, the resolution rules from Figure 3 are used to generate new atomic constraints.

$$c(v_1,...,v_n) \subseteq_a c(v'_1,...,v'_n) \Rightarrow$$

$$\begin{cases} v_i \subseteq_a v'_i & \text{if } c \text{ is covariant in } i \\ v'_i \subseteq_a v_i & \text{if } c \text{ is contravariant in } i \end{cases}$$

$$c(v_1,...,v_n) \subseteq_a proj(c,i,v) \Rightarrow$$

$$\begin{cases} v_i \subseteq_a v & \text{if } c \text{ is covariant in } i \\ v \subseteq_a v_i & \text{if } c \text{ is contravariant in } i \end{cases}$$

Figure 3: Resolution rules for structural constraints.

We use annotated constraint graphs based on the *inductive form* representation [3]. Inductive form is an efficient sparse representation that does not explicitly represent the transitive closure of the constraint graph. The graphs are represented with adjacency lists $pred(n)$ and $succ(n)$ stored at each node n. Edge (n_1, n_2, a), where a is an annotation, is represented either as a predecessor edge by having $\langle n_1, a\rangle \in pred(n_2)$, or as a successor edge by having $\langle n_2, a\rangle \in succ(n_1)$, but not both. $Source \subseteq_a Var$ is always a predecessor edge and $Var \subseteq_a Sink$ is always a successor edge. $Var \subseteq_a Var$ is either a predecessor or a successor edge, based on a fixed total order $\tau : Vars \to \mathcal{N}$. Edge (v_1, v_2, a) is a predecessor edge if and only if $\tau(v_1) < \tau(v_2)$. The order function is typically based on the order in which variables are created as part of building the constraint system [34].

5.3 Solving Systems of Annotated Constraints

Every system of annotated inclusion constraints can be represented by an annotated constraint graph in inductive form. The system is solved by computing the closure of the graph under the following transitive closure rule:

$$\left.\begin{array}{l} \langle L, a\rangle \in pred(v) \\ \langle R, b\rangle \in succ(v) \\ Match(a, b) \end{array}\right\} \Rightarrow L \subseteq_c R \qquad \text{(Trans)}$$

The closure rule can be applied locally, by examining $pred(v)$ and $succ(v)$. The new transitive constraint is created *only if* the annotations of the two existing constraints "match"—that is, only if $Match(a, b)$ holds, where $Match$ is a binary predicate on the set of annotations. Intuitively, the Trans rule uses the annotations to filter out some flow of values in the constraint system. The $Match$ predicate is defined as follows:

$$Match(a,b) = \begin{cases} \text{true} & \text{if } a \text{ or } b \text{ is empty} \\ \text{true} & \text{if } a = b \\ \text{false} & \text{otherwise} \end{cases}$$

The annotation c of the new constraint is

$$c = \begin{cases} a & \text{if } b \text{ is empty} \\ b & \text{if } a \text{ is empty} \\ \text{empty} & \text{otherwise} \end{cases}$$

Intuitively, an annotation is propagated until it is matched with another instance of itself, after which the two instances cancel out.

If the new constraint generated by the Trans rule is atomic, a new edge is added to the graph. Otherwise, the resolution rules from Figure 3 are used to transform the constraint into several atomic constraints and their corresponding edges are added to the graph.

The closure of a constraint graph under the Trans rule is the *solved inductive form* of the corresponding constraint system. The least solution of the system is not explicit in the solved inductive form [3], but is easy to compute by examining all predecessors of each variable. For constraint graphs without annotations, the least solution $LS(v)$ for a variable v is

$$LS(v) = \{c(\ldots) \mid c(\ldots) \in pred(v)\} \cup \bigcup_{u \in pred(v)} LS(u)$$

In this case, $LS(v)$ can be computed by transitive acyclic traversal of all predecessor edges [19]. For an annotated constraint graph, the traversal is done similarly, but the annotations are used as in rule Trans:

$$\begin{aligned} LS(v) = {} & \{\langle c(\ldots), a\rangle \mid \langle c(\ldots), a\rangle \in pred(v)\} \cup \\ & \{\langle c(\ldots), z\rangle \mid \langle u, x\rangle \in pred(v) \wedge \langle c(\ldots), y\rangle \in LS(u) \\ & \qquad \wedge Match(x, y)\} \end{aligned}$$

Here annotation z is computed from annotations x and y as in the Trans rule.

6 Constraint-based Points-to Analysis for Java

In this section we show how to implement the points-to analysis from Section 3 using annotated inclusion constraints. Recall that the analysis is defined in terms of the set R of all reference variables and the set O of names for all objects created at object allocation sites. Every element of $(R \cup O)$ is essentially an abstract memory location representing a set of run-time memory locations.

To model the analysis with annotated inclusion constraints, we extend a technique developed for Andersen's analysis for C [19, 34]. For each abstract location x, a set variable v_x represents the set of abstract locations pointed-to by x. The representation of each location is through a trinary constructor *ref* which is used to build constructed terms of the form $ref(x, v_x, \overline{v_x})$. The last two arguments are the same variable, but with different variance—the overline notation is used to denote a contravariant argument. Intuitively, the second argument is used to read the values of locations pointed-to by x, while the last argument is used to update the values of locations pointed-to by x. Given a reference variable $r \in R$ and an object variable $o \in O$, constraint

$$ref(o, v_o, \overline{v_o}) \subseteq v_r$$

shows that r points to o.

We use *field annotations* to model the flow of values through fields of objects. Field annotations are unique identifiers for all instance fields defined in program classes. For any two object variables o_1 and o_2, constraint

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_1}$$

shows that field f in object o_1 points to object o_2.

6.1 Constraints for Assignment Statements

For every program statement, our analysis generates annotated inclusion constraints representing the semantics of the statement. Figure 4 shows the constraints generated for assignment statements. The first two generation rules are

$$\langle l = new\ o_i \rangle \Rightarrow \{ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq v_l\}$$

$$\langle l = r \rangle \Rightarrow \{v_r \subseteq v_l\}$$

$$\langle l.f = r \rangle \Rightarrow \{v_l \subseteq proj(ref, 3, u),\ v_r \subseteq_f u\},\ u \text{ fresh}$$

$$\langle l = r.f \rangle \Rightarrow \{v_r \subseteq proj(ref, 2, u),\ u \subseteq_f v_l\},\ u \text{ fresh}$$

Figure 4: Constraints for assignment statements.

straightforward. The rule for "$l.f = r$" uses the first constraint to access the points-to set of l, and the second constraint to update the values of field f in all objects pointed-to by l. Similarly, the last rule uses two constraints to read the values of field f in all objects pointed-to by r.

Example Consider the statements in Figure 5 and their corresponding points-to graph. After processing the statements, our analysis creates the following constraints:

$$\begin{array}{ll} ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_p & ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_q \\ v_p \subseteq proj(ref, 3, u) & v_q \subseteq_f u \\ v_p \subseteq proj(ref, 2, w) & w \subseteq_f v_r \end{array}$$

where u and w are fresh variables. For the purpose of this example we assume that the variable order τ (defined in Section 5.2) is $\tau(v_p) < \tau(v_q) < \tau(v_r) < \tau(v_{o_1}) < \tau(v_{o_2}) < \tau(u) < \tau(w)$. Consider the indirect write in `p.f = q`. Since we have

$$ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_p \subseteq proj(ref, 3, u)$$

we can use the TRANS rule and the resolution rules from Figure 3 to generate a new constraint $u \subseteq v_{o_1}$. Thus,

$$v_q \subseteq_f u \subseteq v_{o_1}$$

and using rule TRANS we generate $v_q \subseteq_f v_{o_1}$. Intuitively, this new constraint shows that some of the values of field f in object o_1 come from variable q. Now we have

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_q \subseteq_f v_{o_1}$$

Since both constraint edges are predecessor edges, we cannot apply rule TRANS. Still, in the least solution of the constraint system (as defined in Section 5.3), we have the constraint $ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_1}$, which shows that field f of o_1 points to o_2.

To model indirect reads, we use the second argument of the *ref* constructor. For example, for the constraints above we have

$$ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_p \subseteq proj(ref, 2, w)$$

and therefore $v_{o_1} \subseteq w \subseteq_f v_r$, which through TRANS generates $v_{o_1} \subseteq_f v_r$. This new constraint shows that the value of r comes from field f of object o_1. Now we have

$$v_q \subseteq_f v_{o_1} \subseteq_f v_r$$

Since the annotations of the two constraints match—that is, they represent accesses to the same field—we generate $v_q \subseteq v_r$ to represent the flow of values from q to r. Thus, in the least solution of the system we have

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_r$$

which shows that reference variable r points to o_2. This example illustrates how field annotations allow us to model flow of values through object fields.

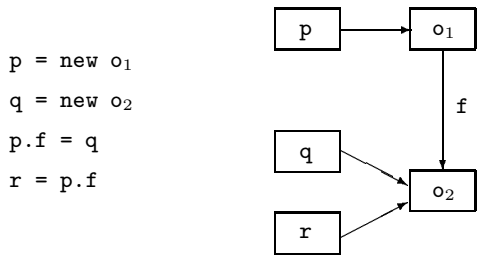

Figure 5: Accessing fields of objects.

6.2 Handling of Virtual Calls

For every virtual call in the program, our analysis generates a constraint according to the following rule:

$$\langle l = r_0.m(r_1, \ldots, r_k) \rangle \Rightarrow \{v_{r_0} \subseteq_m lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)\}$$

The rule is based on a *lam* (lambda) constructor. The constructor is used to build a term that encapsulates the actual arguments and the left-hand side variable of the call. The annotation on the constraint is a unique identifier of the *compile-time* target method of the call. This annotation is used during the analysis to find all appropriate *run-time* target methods.

To model the semantics of virtual calls as defined in Section 3, we separately perform virtual dispatch for every possible receiver object. In order to do this efficiently, we use a precomputed *lookup table*. For a given receiver object at a virtual call site, the lookup table is used to determine the corresponding run-time target method, based on the class of the receiver object.[3] Such a table is straightforward to precompute by analyzing the class hierarchy; the table is essentially a representation of the *dispatch* function from Section 3.

Given the class of the receiver object and the unique identifier for the compile-time target of the virtual call, the lookup table returns a lambda term of the form

$$lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret})$$

Here p_i are the formal parameters of the run-time target method; p_0 corresponds to the implicit parameter `this`. We assume that each method has a unique variable *ret* that is assigned the value returned by the method. At the beginning of the analysis, lambda terms of this form are created for all non-abstract methods in the program and are stored in the lookup table.

To model the effects of virtual calls, we define an additional closure rule VIRTUAL. This rule encodes the semantics of virtual calls described in Section 3 and is used together with the TRANS rule to obtain the solved form of the constraint system. VIRTUAL is applied whenever we have two constraints of the form

$$ref(o, v_o, \overline{v_o}) \subseteq v \qquad v \subseteq_m lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

As described in Section 5.2, the edge from the *ref* term is a predecessor edge, and the edge to the *lam* term is a successor edge. Thus, the VIRTUAL closure rule can be applied locally, by examining sets $pred(v)$ and $succ(v)$. Whenever two such

[3] Every object is tagged with its class; this tag is used when performing lookups.

```
class A { X n() { ... return rA; } }
class B extends A
        { X n() { ... return rB; } }
     A a = new A(); // object o1
     A b = new B(); // object o2
     B c = b;
s1:  X x = b.n();
s2:  X y = c.n();
     if (...) a = b;
s3:  X z = a.n();
```

(1) $ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_b \subseteq_{A::n} lam(\overline{0}, v_x) \Rightarrow$
$\{ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_{B::n::this},\ v_{rB} \subseteq v_x\}$

(2) $ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_c \subseteq_{B::n} lam(\overline{0}, v_y) \Rightarrow$
$\{ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_{B::n::this},\ v_{rB} \subseteq v_y\}$

(3) $ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_a \subseteq_{A::n} lam(\overline{0}, v_z) \Rightarrow$
$\{ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_{A::n::this},\ v_{rA} \subseteq v_z\}$

(4) $ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_a \subseteq_{A::n} lam(\overline{0}, v_z) \Rightarrow$
$\{ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_{B::n::this},\ v_{rB} \subseteq v_z\}$

Figure 6: Example of virtual call resolution.

constraints are detected, the lookup table is used to find the lambda term for the run-time method corresponding to object o and compile-time target method m. The result of applying VIRTUAL are two new constraints:

$$\begin{aligned} ref(o, v_o, \overline{v_o}) &\subseteq v_{p_0} \\ lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret}) &\subseteq lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l) \end{aligned}$$

The first constraint creates the association between parameter `this` of the invoked method and the receiver object. The second constraint immediately resolves to $v_{r_i} \subseteq v_{p_i}$ (for $i \geq 1$) and $v_{ret} \subseteq v_l$, plus the trivial constraint $0 \subseteq v_{p_0}$. These new atomic constraints model the flow of values from actuals to formals, as well as the flow of return values to the left-hand side variable l used at the call site.

Example Consider the set of statements in Figure 6. For the purpose of this example, assume that $\tau(v_a) < \tau(v_c) < \tau(v_b)$. Since the declared type of `b` is `A`, at call site $s1$ the compile-time target method is `A::n`[4]; thus, we have

$$v_b \subseteq_{A::n} lam(\overline{0}, v_x)$$

When rule VIRTUAL is applied as shown in (1), the lookup for receiver object o_2 and compile-time target `A::n` produces run-time target `B::n`. The resolution with the *lam* term for `B::n` creates the two new constraints shown in (1).

The declared type of `c` is `B`, and for call site $s2$ we have

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_c \qquad v_c \subseteq_{B::n} lam(\overline{0}, v_y)$$

where the first constraint is obtained through the TRANS rule.[5] By applying VIRTUAL, we create the constraints shown in (2).

[4]We use `X::z` to denote method `z` defined in class `X`.

[5]Note that if $\tau(v_b) < \tau(v_c)$, instead of propagating the *ref* term to v_c we would propagate the *lam* term to v_b.

For call site $s3$, the receiver object can be either o_1 or o_2. As shown in (3) and (4), separate lookup and resolution is performed for each receiver.

6.3 Correctness

For every program statement, our analysis generates constraints representing the semantics of the statement. This initial constraint system is solved by closing the corresponding constraint graph under closure rules TRANS and VIRTUAL. Let A^* be the solved inductive form of the constraint system. Recall that the least solution of the system is not explicit in A^* and can be obtained through additional traversal of predecessor edges, as described in Section 5.3.

Let G^* be the points-to graph computed by the algorithm in Section 3. Consider a reference variable r and an object variable o such that $(r, o) \in G^*$. We can show that the least solution constructed from A^* contains

$$ref(o, v_o, \overline{v_o}) \subseteq v_r$$

Similarly, consider two object variables o_i and o_j such that $(\langle o_i, f\rangle, o_j) \in G^*$. Again, we can show that the least solution contains

$$ref(o_j, v_{o_j}, \overline{v_{o_j}}) \subseteq_f v_{o_i}$$

The above relationships hold only if we impose some restrictions on the order of variables in the constraint system. As described in Section 5.2, the graph representation of the system is based on a total order $\tau : Vars \rightarrow \mathcal{N}$ on the set of variables. This order determines whether a variable-variable edge is a predecessor edge or a successor edge. Suppose $r \in R$ is an arbitrary reference variable. For every object variable $o_i \in O$, we should have $\tau(v_r) < \tau(v_{o_i})$. Similarly, for every "fresh" variable u created due to field access (see Figure 4), we should have $\tau(v_r) < \tau(u)$. We enforce these restrictions as part of building the constraint system.

Given these restrictions, it can be proven that the least solution of the constraint system represents all points-to pairs from G^*. The proof of this claim is outlined in Appendix A.

6.4 Cycle Elimination

Cycle elimination [19] is an approach for reducing the cost of constraint resolution. It is an important technique for achieving good performance for Andersen's analysis for C. In this section we show that a specific form of cycle elimination can be used during our analysis for Java.

The idea behind cycle elimination is to detect a set of variables that form a cycle in the constraint graph:

$$v_1 \subseteq v_2 \subseteq \ldots \subseteq v_k \subseteq v_1$$

Clearly, all such variables have equal solutions and can be replaced with a single variable. Whenever a cycle is detected during the resolution process, one variable from the cycle is chosen as a witness variable, and the rest of the variables are redirected to the witness. This transformation has no effect on the computed solution, but can significantly reduce the cost of the analysis.

Cycle detection is performed every time a new edge is added between two variables v_i and v_j. The detection algorithm essentially performs depth-first traversal of the constraint graph and tries to determine whether v_i is reachable from v_j. Cycle detection is partial and does not detect all cycles. Nevertheless, for Andersen's analysis for C this

technique has significant impact on the running time of the analysis (we refer the reader to [19] for more details).

In our analysis for Java, we use a restricted form of cycle elimination. The cycle detection algorithm is the same as in [19], but is invoked only when a new edge is added between two reference variables—that is, when the new edge is (v_{r_i}, v_{r_j}), where $r_i, r_j \in R$. It can be shown that in this case the detected cycles contain only reference variables, and all edges in the cycles have empty annotations. If we performed cycle detection for all variable-variable edges, we would discover cycles in which some edges have field annotations. However, the variables in such cycles do not have the same solution, and cannot be replaced by a single witness variable.

6.5 Tracking Reachable Methods

Andersen's analysis implicitly assumes that all code in the program is executable. Since Java programs heavily use libraries that contain many unused methods, we have augmented our analysis to keep track of all reachable methods, in order to avoid analyzing dead code. Thus, we take into account the effects of a method only if the method has been shown to be reachable from one of the entry methods of the program. The set of entry methods M_0 contains the `main` method of the starting class and the class initialization methods for all classes.[6]

We can augment the semantic rules from Section 3 to include this reachability computation. The original rules are of the form $\langle s, G\rangle \Rightarrow G'$, and define the effects of statement s on points-to graph G. We augment these rules to include sets of reachable methods. The new rules have the form $\langle s, G, M\rangle \Rightarrow (G', M')$, where M and M' are sets of reachable methods. If the method to which s belongs is not in M, we have $G' = G$ and $M' = M$. Otherwise, G' is defined as in Figure 2, and M' contains all elements of M plus all methods that become reachable when s is a call statement.[7]

It is straightforward to implement this augmented semantics in our constraint-based analysis. We maintain a list of reachable methods, initialized with M_0. Whenever a method becomes reachable, all statements in its body are processed and the appropriate constraints are introduced in the constraint system. It is easy to show that this approach produces a solution that is safe with respect to the augmented semantics.

7 Analysis Implementation

For our experiments we used an implementation of the analysis described in Section 6. The implementation uses the Soot framework[8] to process Java bytecode and to build a typed intermediate representation [37]. The constraint-based analysis uses BANE (Berkeley ANalysis Engine) [2]. BANE is a toolkit for constructing constraint-based program analyses. The public distribution of BANE[9] contains an efficient constraint-solving engine that employs inductive form [3] and cycle elimination [19]. We modified the constraint engine to attach annotations to the constraints, to implement the appropriate resolution and closure rules, and to perform cycle elimination as described in Section 6.4. The analysis works on top of the constraint engine, by processing every newly discovered reachable method and generating the appropriate constraints.

The reflection mechanism in Java presents problems for all compile-time analyses, including ours. At run-time, reflection may introduce new classes into the class hierarchy. This mechanism also allows indirect access to objects, methods and fields. One way to handle this problem is to supply the analysis with a list of classes that could be accessed through reflection (some experience with this approach is described in [35]). Our current implementation does not handle reflection, but we are in the process of implementing the approach described above, and we plan to have complete handling of reflection for our future work.

Since the bytecode does not contain bodies for native methods, their effects on the points-to solution cannot be taken into account. In our experience, native methods only occur in the standard libraries. We are currently working on generating summaries that encode the points-to effects of native library methods. Such summaries are commonly used in points-to analysis for C to model the effects of library calls. We are using the same approach for Java, and the summaries will be incorporated in our future work.

Program	User Class	Size (Kb)	Whole-program Class	Method	Stmt
`proxy`	18	56.6	59	831	14968
`compress`	22	76.7	206	740	15070
`db`	14	70.7	200	780	15907
`mtrt`	35	115.9	221	892	17660
`raytrace`	35	115.9	221	892	17993
`jlex`	25	95.1	159	752	18849
`echo`	17	66.7	234	1068	19070
`javacup`	33	127.3	163	936	19971
`rabbit`	24	88.4	196	1217	21962
`jess`	117	319.0	312	1231	23757
`jack`	67	191.5	251	1023	24528
`mpegaudio`	62	176.8	246	958	26729
`volano`	95	249.6	335	1956	32448
`jjtree`	72	272.0	200	1455	32943
`jflex`	54	198.2	276	1668	33230
`jtar`	64	185.2	306	1645	34236
`javac`	182	614.7	370	1997	38977
`mindterm`	69	279.9	312	2070	39725
`muffin`	158	391.5	394	2485	43012
`creature`	65	259.7	292	1725	43115
`sablecc`	312	532.4	541	3162	45387
`soot`	677	1070.4	802	3067	46385
`javacc`	63	502.6	193	1582	56465

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.

8 Empirical Results

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 (excluding library classes). We used 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 popu-

[6] Class initialization methods contain the initializers for static fields [27, Section 3.9].

[7] For multi-threaded programs, a call to `Thread::start` also makes the corresponding `run` method reachable.

[8] `http://www.sable.mcgill.ca/soot`

[9] `http://bane.cs.berkeley.edu`

Program	Time (sec)	Memory (Mb)	Time-nf (sec)	Memory-nf (Mb)
`proxy`	19.8	13.1	133.4	16.9
`compress`	12.9	11.3	20.5	11.6
`db`	17.2	12.7	41.8	13.0
`mtrt`	28.3	15.8	344.4	22.9
`raytrace`	28.4	15.8	344.5	22.9
`jlex`	28.9	15.0	2687.6	49.6
`echo`	33.0	15.7	336.6	23.0
`javacup`	160.8	28.9	1128.8	44.8
`rabbit`	51.2	22.1	864.9	40.9
`jess`	70.2	25.5	3387.9	72.1
`jack`	73.0	21.2	420.0	33.5
`mpegaudio`	26.1	17.4	3654.5	126.3
`volano`	85.5	32.2	2148.4	85.7
`jjtree`	46.4	22.1	238.5	33.7
`jflex`	273.9	46.6	6201.6	134.6
`jtar`	22.6	20.9	73.8	23.6
`javac`	2332.5	104.0	13151.0	222.1
`mindterm`	95.6	37.7	9029.0	141.2
`muffin`	135.2	43.2	1681.5	93.3
`creature`	252.6	42.5	4929.6	177.4
`sablecc`	1700.5	110.0	20879.5	292.2
`soot`	1622.5	124.5	26639.3	333.7
`javacc`	857.7	84.9	4931.8	139.9

Table 2: Running time and memory usage of the analysis. Last two columns show the cost when object fields are not distinguished.

lar publicly available Java applications. The data programs represent a mix of different kinds of applications. For example, we included several applications that use graphical interfaces. Such programs are examples of an emerging class of Java applications that has been underrepresented in the previous empirical work on analyzing Java programs.

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 class hierarchy analysis (CHA) [15] has been used to filter out unused classes and methods. 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 from Soot's intermediate representation.

Analysis Cost Our first set of experiments measured the cost of the analysis. The results are shown in Table 2. The first two columns show the running time of the analysis and the amount of memory used. For 15 out of the 23 programs, the analysis runs in less than 100 seconds; for all but 4 programs, the running time is less than five minutes and the memory usage is less than 50Mb. To the best of our knowledge, these are the first published empirical results showing that a relatively precise points-to analysis can run in a few minutes on realistic Java programs. Empirical results for more expensive points-to analyses [23, 9] suggest that they do not scale; for example, the 1-1-CFA analysis from [23] runs in about 2000 seconds on `javacup` and runs out of memory on `javac`. The practicality of our analysis makes it a good candidate for precise points-to analysis for Java.

All four outlier programs in our data set are compiler-related: `javac` is a compiler, `soot` is a compiler-like framework, and `sablecc` and `javacc` are compiler generators. All four programs have parsers that process some input language; different kinds of entities in the language are represented by different classes. For example, `soot` parses Java bytecode; various kinds of bytecode instructions are represented by more than 200 subclasses of the abstract class `Instruction`. The bytecode is then transformed into an intermediate representation; again, there is a large number of classes used to represent different kinds of statements and expressions. Because of these wide inheritance hierarchies, there are many variables with large points-to sets. This significantly increases the size of the points-to graph and the cost of the analysis. We observed similar patterns in the other outlier programs.

We draw two conclusions from these experiments. First, for most Java applications our analysis is practical in terms of running time and memory usage, as evidenced on a variety of realistic Java programs. This practicality shows that the analysis will be useful as a relatively precise general-purpose points-to analysis for Java. The second conclusion is that for programs with wide inheritance hierarchies, the analysis becomes costly. We are currently investigating techniques that would allow us to reduce the cost of the analysis for this particular class of Java programs.

The last two columns in Table 2 show the cost of the analysis when object fields are not distinguished—that is, when all field annotations are replaced with empty annotations, and therefore objects are treated as monolithic entities. The resulting imprecision creates significantly larger points-to sets, which causes the increase in analysis cost. These results show that distinguishing between fields of objects is crucial for reducing the time and space cost of the analysis.

Analysis Precision To measure the precision of the analysis with respect to call graph construction and virtual call resolution, we compared the two versions of the analysis (with and without distinguishing object fields) with rapid type analysis (RTA) [5]. RTA is an inexpensive and widely used analysis for call graph construction. It performs a reachability computation on the call graph generated by CHA. By keeping track of the classes that have been instantiated, RTA computes a more precise call graph than CHA.

The first three columns of Table 3 show the reduction in the number of methods in the call graph, compared to the graph computed by CHA. On average, the reachability computation in our points-to analysis identifies 40% of the methods as dead code. This reduction in the number of methods allows subsequent analyses and optimizations to safely ignore large portions of the program.

To determine the improvements in call graph information for virtual calls, we considered calls that could not be resolved to a single target method by CHA. Let V be the set of all CHA-unresolved calls that occur in methods identified by our analysis as reachable. For our data programs, the size of V is between 3% and 34% (13% on average) of all virtual calls in reachable methods. For each call site from V, we computed the difference between the number of target methods according to CHA and the number of target methods according to each of the three analyses. The average differences are shown in the second section of Table 3. On average, our analysis removes more than twice as many targets as RTA. The improved precision can be beneficial for reducing the cost and improving the precision of subsequent interprocedural analyses.

The last section of Table 3 shows the percentage of call sites from V that were resolved to a single target method. Our points-to analysis performs significantly better than

Program	Removed Methods			Removed Targets for Call Site			Resolved Call Sites		
	Points-to	No fields	RTA	Points-to	No fields	RTA	Points-to	No Fields	RTA
`proxy`	37%	35%	31%	7.0	6.1	3.7	70%	55%	0%
`compress`	65%	64%	55%	13.5	12.8	8.6	93%	66%	0%
`db`	57%	56%	48%	8.7	8.2	4.2	79%	68%	0%
`mtrt`	52%	51%	42%	10.3	9.8	5.3	65%	45%	0%
`raytrace`	52%	51%	42%	10.3	9.8	5.3	65%	45%	0%
`jlex`	46%	43%	42%	9.4	8.0	6.2	83%	54%	15%
`echo`	52%	49%	40%	3.4	3.1	2.3	27%	25%	4%
`javacup`	26%	24%	17%	4.1	3.7	2.1	68%	63%	19%
`rabbit`	16%	13%	7%	7.1	4.3	1.8	59%	33%	10%
`jess`	38%	36%	31%	7.4	6.5	4.4	48%	47%	25%
`jack`	43%	41%	37%	2.3	2.2	1.5	95%	91%	72%
`mpegaudio`	51%	46%	40%	10.0	8.9	5.1	50%	35%	0%
`volano`	38%	35%	29%	5.1	3.2	1.7	49%	24%	7%
`jjtree`	24%	23%	22%	6.8	5.9	5.2	45%	34%	18%
`jflex`	37%	31%	27%	6.2	3.7	2.4	49%	35%	6%
`jtar`	76%	75%	18%	5.3	5.2	1.7	32%	32%	3%
`javac`	22%	20%	17%	2.2	1.9	1.1	19%	17%	0%
`mindterm`	20%	18%	14%	2.9	2.3	1.3	26%	20%	6%
`muffin`	39%	35%	30%	3.7	3.3	1.5	72%	68%	4%
`creature`	60%	53%	14%	3.2	2.5	1.2	47%	21%	2%
`sablecc`	24%	22%	15%	7.0	1.7	0.9	24%	14%	1%
`soot`	14%	13%	9%	3.7	2.8	0.5	39%	20%	1%
`javacc`	21%	21%	20%	3.1	3.0	1.9	88%	85%	4%
Average	40%	37%	28%	6.2	5.2	3.0	56%	43%	9%

Table 3: Improvements over CHA-based call graph. (a) Reduction in the number of reachable methods. (b) Average reduction in the number of target methods per virtual call. (c) Number of resolved virtual calls.

RTA—on average, 56% versus 9% of the virtual call sites are resolved. The increased precision enables improvements in the removal of run-time lookups and in method inlining.

Unlike RTA and similar class analyses, points-to analysis produces information that is useful for applications other than call graph construction. To investigate the impact of our points-to analysis on synchronization removal and stack allocation, we identified all object allocation sites that correspond to thread-local and method-local objects. To do this, we used the approach from Section 4, augmented with detection of single-threaded programs—if no objects implementing `java.lang.Runnable` are allocated in reachable methods, the program is single-threaded and all objects are thread-local. For every object allocation site in a reachable method, we used the two versions of the analysis (with and without object fields) to determine whether the site is thread-local or method-local. The percentage of thread-local and method-local sites is shown in Figure 7.

The analysis identifies 13 programs as single-threaded, and 100% of their sites are thread-local. For the multi-threaded programs, the analysis detects a significant number of allocation sites for thread-local objects—on average, about 50% of all allocation sites. These results show that our analysis can be very effective in detecting and eliminating unnecessary synchronization in Java programs. The analysis also discovers a significant number of sites for method-local objects—on average, about 45% of all sites. This result indicates that there are many opportunities for stack-based object allocation that can be detected using our points-to analysis.

An interesting result from the experiments is that for the purpose of identifying thread-local and method-local objects, there is little precision improvement from distinguishing object fields. On the other hand, distinguishing object fields does have significant impact on the sizes of the points-to sets. We computed the average points-to set sizes for all programs and determined that the less precise version of the analysis computes points-to sets that are 4.4 times larger on average. This leads to the significant difference in the running times shown in Table 2. We expect that the precision difference between the two versions of the analysis will be more significant for other client applications such as def-use analysis and side-effect analysis.

9 Related Work

Points-to analysis for object references in Java is clearly related to pointer analysis for imperative languages such as C. There are various pointer analyses for C with different tradeoffs between cost and precision [25, 24, 18, 4, 31, 39, 33, 32, 19, 26, 34, 21, 30, 14, 20, 10]. The closest related work from this category are the efficient constraint-based implementations of Andersen's analysis from [19] and [34]. This work shows how to reduce the cost of Andersen's analysis by using the inductive form representation together with efficient resolution techniques such as cycle elimination [19] and projection merging [34]. Our work extends this approach by introducing constraint annotations and by modifying the graph representation and the resolution procedure. Field annotations allow us to track object fields separately; this is not possible with the constraints from [19, 34]. Method annotations allow us to model the semantics of virtual calls. In addition, we avoid analyzing dead library code by including a reachability computation as part of an integrated analysis.

Constraint indices and constraint polarities [20] have been used to introduce context-sensitivity in unification-based flow analysis. This work has similar flavor to our use of annotations for tracking flow of values through object fields.

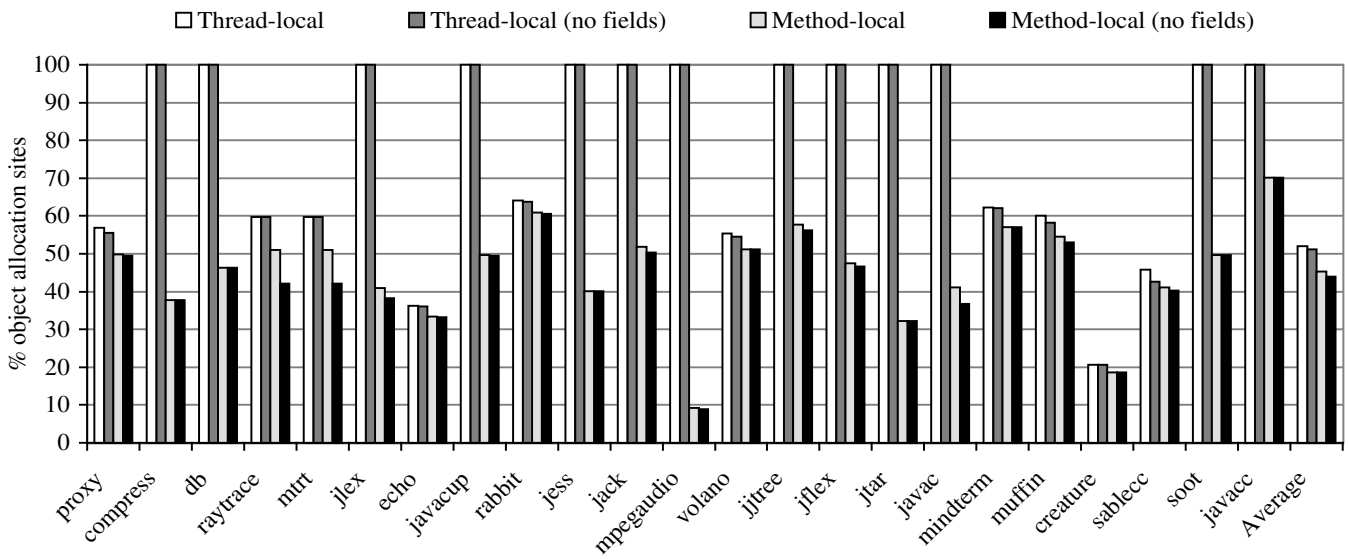

Figure 7: Thread-local and method-local object allocation sites. The average for thread-local sites does not include single-threaded programs.

Conceptually, in both cases the annotations model restricted value flow in constraint systems—either for unification-based constraints in [20], or for inclusion constraints in our analysis.

Previous work on points-to analysis for object-oriented languages includes the flow- and context-sensitive analysis by Chatterjee et al. [9]. This analysis is more precise than ours, but it is unclear whether it will scale for programs as large as the programs from our data set. Points-to analyses with different degrees of precision have been proposed in the context of a framework for call graph construction in object-oriented languages [23]. The closest to our work is the 1-1-CFA algorithm, which incorporates flow- and context- sensitive points-to analysis. The test suite from [23] contains few Java programs, and the analysis times suggest that 1-1-CFA might not be practical. Our empirical results on a large set of realistic Java applications show that our analysis is a practical alternative to the more expensive analyses from [9] and [23]. Plevyak and Chien [29] incorporate points-to analysis in their concrete type inference algorithm; their experiments use small programs written in Concurrent Aggregates. Corbett [13] presents a flow-sensitive points-to analysis for Java in the context of synchronization removal; no implementation is reported.

Recent work on escape analysis for Java [11, 8, 38] incorporates several specialized forms of points-to analysis. The scalability of these approaches remains unclear, since the analysis running times are not reported. In contrast, we have a practical general-purpose points-to analysis that can be used by a variety of client applications, including synchronization removal and stack-based object allocation.

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. The traditional client applications of class analysis are call graph construction and virtual call resolution. In general, points-to analysis subsumes class analysis; practical implementations of points-to analysis can be used for a variety of applications, including call graph construction and virtual call resolution. DeFouw et al. [16] present a family of practical interprocedural class analyses. Other work in this area considers more precise and costly analyses with some degree of context- or flow-sensitivity [28, 1, 29, 17, 23], as well as less precise but inexpensive analyses such as RTA [5, 36].

10 Conclusions and Future Work

We have defined a points-to analysis for Java that extends Andersen's points-to analysis for C. The analysis defines and solves systems of annotated set-inclusion constraints. The annotations allow us to model the semantics of virtual calls and the flow of values through object fields. By using several techniques for efficient constraint representation and resolution, we have been able to perform practical and precise points-to analysis on a large set of realistic Java programs. Our experiments show that the analysis performs very well on a wide variety of Java applications. The experiments also show that the points-to solution has significant impact with respect to call graph construction, virtual call resolution, synchronization removal, and stack-based object allocation. These results show that our analysis will be useful as a precise and practical general-purpose points-to analysis for Java.

For programs with wide inheritance hierarchies, the analysis may be relatively costly, with running times in the order of tens of minutes. It is unclear how often such applications occur in practice. Nevertheless, in our future work we plan to investigate techniques for handling such programs. For example, there are various kinds of approximations that could reduce the running time of the analysis without significant loss of precision.

Another direction of future work is to investigate the impact of the analysis solution on def-use analysis and side-effect analysis. For C, these analyses are the traditional clients of points-to information. Currently, there is little work on investigating the uses of these analyses for Java—both in the context of compiler optimizations and in the contexts of software engineering applications such as program slicing and data-flow-based testing.

References

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

[2] 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.

[3] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, June 1993.

[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] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 20–34, 1999.

[7] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *Conference on Programming Language Design and Implementation*, pages 13–26, 2000.

[8] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 35–46, 1999.

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

[10] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[11] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.

[12] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Conference on Programming Language Design and Implementation*, pages 13–26, 2000.

[13] J. Corbett. Constructing compact models of concurrent Java programs. In *International Symposium on Software Testing and Analysis*, pages 1–10, 1998.

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

[15] 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.

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

[17] 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.

[18] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Conference on Programming Language Design and Implementation*, pages 242–257, 1994.

[19] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[20] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

[21] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, 2000.

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

[23] 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.

[24] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Trans. Programming Languages and Systems*, 21(4):848–894, May 1999.

[25] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Conference on Programming Language Design and Implementation*, pages 235–248, 1992.

[26] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 199–215, 1999.

[27] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

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

[29] 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.

[30] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Conference on Programming Language Design and Implementation*, pages 47–56, 2000.

[31] E. Ruf. Context-insensitive alias analysis reconsidered. In *Conference on Programming Language Design and Implementation*, pages 13–22, 1995.

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

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

[34] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Symposium on Principles of Programming Languages*, pages 81–95, 2000.

[35] F. Tip, C. Laffra, P. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305, 1999.

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

[37] R. Vallée-Rai and L. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, McGill University, July 1998.

[38] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.

[39] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

A Proof of Correctness

Let G be a points-to graph as defined in Section 3 and A be an annotated constraint graph in inductive form as described in Section 6. We will define a *representation relation* α between A and the edges in G. If α holds between A and all edges in G, we will write $\alpha(A, G)$.

Consider a reference variable r and an object variable o such that $e = (r, o)$ is an edge in G. We have $\alpha(A, e)$ if and only if A contains a path

$$ref(o, v_o, \overline{v_o}) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \rightarrow v_r$$

such that all v_i correspond to reference variables and all edges are predecessor edges with empty annotations.

Similarly, consider two object variables o_i and o_j such that $e = (\langle o_i, f\rangle, o_j)$ is an edge in G. We have $\alpha(A, e)$ if and only if at least one of the following two conditions is true. The first condition is that A contains a path

$$ref(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \xrightarrow{f} v_{o_i}$$

such that all v_i $(1 \leq i \leq n)$ correspond to reference variables, all edges are predecessor edges, and the only non-empty annotation on the path is f on edge (v_n, v_{o_i}). The second condition is that A contains a path

$$ref(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \xrightarrow{f} u \rightarrow v_{o_i}$$

such that all v_i $(1 \leq i \leq n)$ correspond to reference variables and u is a "fresh" variable created due to field access (see Figure 4). In this path, all edges are predecessor edges, and the only non-empty annotation is f on edge (v_n, u).

Let G^* be the final points-to graph computed by the algorithm in Section 3, and A^* be the solved inductive form of the corresponding annotated constraint system. The least solution of the constraint system is obtained through additional traversal of predecessor edges in A^*, as described in Section 5.3.

Suppose that $\alpha(A^*, G^*)$. It is easy to show that in this case G^* is contained in the set of points-to pairs extracted from the least solution of the constraint system. For example, for every edge $(\langle o_i, f\rangle, o_j) \in G^*$, the least solution contains $ref(o_j, v_{o_j}, \overline{v_{o_j}}) \subseteq_f v_{o_i}$. To prove that $\alpha(A^*, G^*)$, we use the following lemma:

Lemma 1 *Let $\langle s, G\rangle \Rightarrow G'$ as described in Figure 2. If $\alpha(A, G)$, there exists a sequence of applications of closure rules and resolution rules such that A can be transformed into A' for which $\alpha(A', G')$.*

The proof of the lemma requires case-by-case analysis of all statement kinds. For example, consider the assignment `l = r`. In G', there are new edges of the form (l, o_i), where $(r, o_i) \in G$. In A, we have a path from $ref(o_i, v_{o_i}, \overline{v_{o_i}})$ to v_r containing only predecessor edges with empty annotations. Graph A also contains $v_r \subseteq v_l$, which is represented either as a predecessor edge or as a successor edge (depending on the order of v_r and v_l). If $v_r \in pred(v_l)$, then we have the needed path from $ref(o_i, v_{o_i}, \overline{v_{o_i}})$ to v_l. If $v_l \in succ(v_r)$, then a sequence of applications of rule TRANS creates the needed path.

The rest of the statements are handled in a similar manner. For the cases when we have access to object fields, we have to know that all reference variables have order smaller than the rest of the variables (as described in Section 6.3). This restriction ensures that the appropriate kinds of edges (predecessor or successor) are created when transforming A.

Given the above lemma, it is trivial to show that α holds between A^* and G^*.