McGill University
School of Computer Science
Sable Research Group

Practical Virtual Method Call Resolution for Java

Sable Technical Report No. 1998-7

Vijay Sundaresan, Chrislain Razafimahefa, Raja Vallée-Rai and Laurie Hendren

November 6, 1998

Contents

List of Figures

List of Tables

Abstract

This paper addresses the problem of resolving virtual method and interface calls in Java. The main focus is on practical, flow-insensitive techniques that can be used to analyze large applications.

We present a new flow-insensitive analysis called *reaching-type analysis*, which is used to estimate the set of types that reach the receiver of virtual method/interface calls. We present two variations of this analysis, *variable-type analysis* and a coarser-grain version called *declared-type analysis*. We also demonstrate how a points-to style analysis, called *refers-to analysis*, can be used to resolve the types of receivers.

We have implemented our techniques using the Soot framework, and we report on empirical results for 9 Java benchmarks, including the 7 benchmarks from SPECjvm98. We have measured the success of the various analyses at building accurate call graphs, and we conclude that reaching-type analysis leads to call graphs with 17% to 44% fewer edges and 14% to 48% fewer nodes than the corresponding call graph built using a standard class hierarchy analysis.

1 Introduction

As the Java(tm) programming language becomes more popular, it is becoming important to provide optimizing compilers and more efficient runtime systems. One important optimization problem for Java, as for other object-oriented languages, is that of statically determining what methods can be invoked by virtual method calls. The results of such an analysis can be used to reduce the cost of virtual method calls, to detect potential sites for method inlining, and to provide an accurate call graph that can be used in subsequent analyses.

Of course, virtual method resolution is not a new problem, it has been widely studied for a variety of object-oriented languages, and related pointer analyses have been studied for C. The focus of this paper is a study of the effectiveness of five context-insensitive, flow-insensitive algorithms for resolving virtual method calls in Java. We have chosen to concentrate on relatively cheap analyses since we wish to apply the analyses to reasonably sized Java benchmarks.

Our analyses fits into three groups. The first group can be considered to be the *baseline analyses*. These two analyses, *hierarchy analysis*[7, 10, 9] and *rapid type analysis*[9] are existing techniques that have been previously used as inexpensive ways of getting conservative estimates, and form a baseline for the comparison of our other methods.

We propose a second group of analyses, called *reaching-type analyses*, which are based on an analysis that builds a *type propagation graph* where nodes represent variables and edges represents flow of types due to assignment. The first variation is called *declared-type analysis*, where the nodes represent the declared type of variable, and the second variation is called *variable-type analysis* where the nodes represent variable names. Both of these analyses can be thought of as more refined versions of rapid type analysis.

The third group of analyses, called *refers-to analysis*, are based on simplified *points-to* algorithms originally developed for C programs[23]. We use the term refers-to rather than points-to, because the pointer relationships in Java are all of the form of variables referring to instances of objects. It should be noted that refers-to analysis is really designed for determining side-effects, and not necessarily for resolving virtual function calls. However, as we show later, the refers-to solution can be used for this purpose, and we were interested in seeing how its performance compared with the reaching-type strategies that were designed specifically for resolving virtual function calls.

All of the analyses were implemented using the Soot framework that provides Jimple, a typed three-address code representation of Java bytecode. We ran all analyses on a set of 9 Java applications, including 7 SPECjvm98 benchmarks, and 2 other benchmarks[1]. The benchmarks are meant to be representative of real applications, they include some applications with threads, and they range in size from about 14,000 Jimple statements to about 45,000 Jimple statements.

[1]We have several other large benchmarks that we can include in the final paper.

Our experimental results confirm that class hierarchy analysis does do a reasonable job for building an initial conservative call graph for our benchmarks. Rapid type analysis has been shown to be quite effective for C++ benchmarks [9], and our results confirm that rapid type analysis also gives a significant improvement for Java programs, removing from 7% to 30% of the call edges from the conservative call graph. Further, our new reaching-type analyses give even better results, removing from 17% to 44% of the call edges from the conservative call graph.

The remainder of this paper is structured as follows. In Section 2 we give an overview of Soot and Jimple, and we give a very brief summary of hierarchy analysis and rapid type analysis as implemented in our system. In Section 3 we outline the two variations of reaching-type analysis, and in Section 4 we outline the two variations for refers-to analysis. We present our experimental framework and empirical measurements in Section 5. Finally, in Section 6 we discuss related work, concentrating mostly on similar empirical studies, and in Section 7 we give our conclusions and future work.

2 Foundations

2.1 The Soot(Jimple) Framework

Our analyses are built on top of the Jimple intermediate representation, which is part of the Soot framework. The Soot framework is a set of Java APIs for manipulating Java code in various forms.[2] We analyze complete applications, so our implementation works by first reading all class files that are required by an application, by starting with the main root class and recursively loading all classes used in each newly loaded class. As each class is read, it is converted into the Jimple intermediate representation. After conversion, each class is stored in an instance of a `SootClass`, which in turn contains information like its name, its superclass, a list of interfaces that it implements, and a collection of `SootFields` and `SootMethods`. Each `SootMethod` contains information including its name, modifier, parameters, locals, return type and a list of Jimple three-address code instructions. All parameters and locals have declared types. Figure 1(a) shows a Java method, and Figure 1(b) shows a textual representation of the Jimple representation. It is important to note that we produce the Jimple intermediate representation directly from the Java bytecode in class files, and not from the high-level Java programs. This means that we can analyze Java bytecode that has been produced by any compiler, optimizer, or other tool.

In terms of our analysis, there are several important points to note. Firstly, there are relatively few kinds of Jimple statements, and each statement has a simple format. Thus, our analyses can be specified by giving the rules for each kind of Jimple statement. Further, all operands in Jimple are either variable references or constants. Since we have a declared type for each variable, and each constant has a type, our analyses can use this type information in a straightforward manner. Figure 1(b) shows examples of assignment statements, conditional statements, method calls, and return statements. Also note that at the beginning of each method there are special *identity statements* that provide explicit assignments from parameters (including the implicit "this" parameter), and locals.

2.2 Hierarchy Analysis and the Conservative Call Graph

The objective of all of our analyses is to determine, at compile-time, a call graph with as few nodes and edges as possible. All of our analyses start with a *conservative call graph* that is built using *hierarchy analysis*.

[2]Refer to `www.sable.mcgill.ca` for more information on the APIs.

```java
public int stepPoly(int x)
{ if(x < 0)
   { System.out.println("error");
     return -1;
   }
  else if(x <= 5)
    return x * x;
  else
    return x * 5 + 16;
}
```

(a) java source

```
public int stepPoly(int)
{ java.io.PrintStream r1;
  Example r0;
  int i0, i1, i2, i3;

  r0 := @this;
  i0 := @parameter0;
  if i0 >= 0 goto label0;

  r1 = java.lang.System.out;
  r1.println("error");
  return -1;

label0:
  if i0 > 5 goto label1;

  i1 = i0 * i0;
  return i1;

label1:
  i3 = i0 * 5;
  i2 = i3 + 16;
  return i2;
}
```

(b) Jimple representation

Figure 1: Example of Jimple

2.2.1 Hierarchy Analysis

Hierarchy analysis is a standard method for conservatively estimating the run-time types of receivers [7, 10, 9]. Given a receiver o of with a declared type d, *hierarchy_types(o,d)* for Java is defined as follows:

- If receiver o has a declared class type C, the possible run-time types of o, *hierarchy_types(o,C)*, includes C plus all subclasses of C.
- If receiver o with a declared interface type I, the possible run-time types of o, *hierarchy_types(o,I)*, includes: (1) the set of all classes that implement I or implement a subinterface of I, call this set *implements(I)*, plus (2) all subclasses of *implements(I)*.

To implement this analysis, we simply build an internal representation of the the inheritance hierarchy, and then we use this hierarchy to compute the appropriate *hierarchy_types* sets.

2.2.2 Call Graphs

For our purposes a *call graph* consists of nodes and directed edges. The call graph must include one node for each method that can be reached by a computation starting from the `main` method (or if the program has threads, then the call graph must also include all methods that can be reached starting at any `start` or `run` method in a class that implements `java.lang.Runnable`). An example call graph is given in Figure 2(b).

Each node in the call graph contains a collection of call sites. Consider a method M from class C with n method calls in its body. Method M is represented in the call graph by a node labeled $C.M$, and this node will contain entries for each call site, which we denote $C.M[c_1]$ to $C.M[c_n]$. In our example, the call graph node for method `B.main` contains two call sites, $B.main[1]$ which is `a.m()`, and $B.main[2]$ which is `b.m()`.

Edges in the call graph go from call sites within a call graph node, to call graph nodes. The call graph must contain an edge for each possible calling relationship between call sites and nodes. If it is possible that call site $C.M.c[i]$ calls method $C'.M'$, then there must be an edge between $C.M.c[i]1$ and $C'.M'$ in

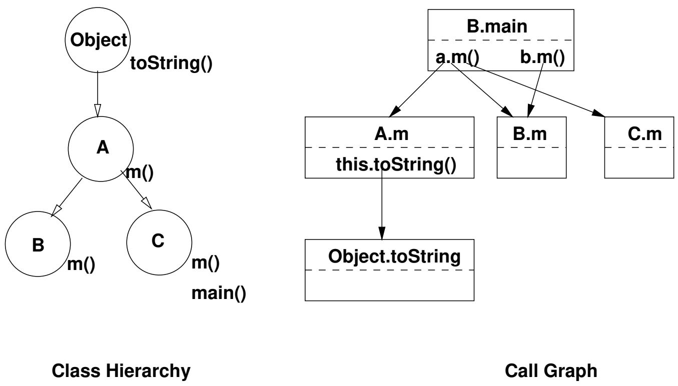
```
class A extends Object {
  String m() {
    return(this.toString());
  }
}

class B extends A {
  String m() { ... }
}

class C extends A {
  String m() { ... }
  public static void main(...) {
   A a = new A();
   B b = new B();
   String s;

   ...
   s = a.m();
   s = b.m();
  }
}
```

(a) Example Program



(b) Class Hierarchy and Call Graph

Figure 2: Example of a conservative call graph

the call graph. In the example call graph there are three edges from the call site `a.m()` corresponding the fact that the virtual call `a.m()` might resolve to calls to `A.m`, `B.m` or `C.m`.

Special attention is required when adding calling edges from a virtual method or interface call and this is done using an approximation of the run-time types of the receiver. Given a virtual call site $C.M[i]$ of the form $o.m(a_1, \ldots, a_n)$, and a set of possible runtime types for receiver o, call this *runtime_types(o)*, we find all possible targets of the call as follows. For each type C_i in *runtime(o)*, look up the class hierarchy starting at C_i until a class C_{target} is found that includes a method $C_{target}.m$ that matches the signature of m. The edge from $C.M[i]$ to $C_{target}.m$ is added to the call graph.

Consider the the call `a.m()` in the example in Figure 2. If the possible runtime types for receiver `a` includes $\{A, B, C\}$, then in each case a matching method `m` is found in the class itself (without looking further up the hierarchy), and thus the call edges to `A.m`, `B.m`, and `C.m` are added. However, sometimes the target method is found further up the hierarchy. Consider the call `this.toString()`. If the possible runtime types the receiver `this` are $\{A, B, C\}$, then looking up the hierarchy in each case will result in the target `Object.toString()`.

Note that a call graph may contain spurious nodes and edges. Spurious edges may be included for virtual method calls. When adding call edges from a virtual method call site $C.M[i]$ of the form $o.m(a_1, \ldots, a_n)$, an edge must be placed between this call site and every method $C'.m$ corresponding to the possible run-time types of the receiver o. If we use a conservative approximation of the run-time types for o, then we may include spurious types in our approximation, and this may lead to spurious edges. In our example, if the type of the receiver `a` in the call `a.m()` can only have a runtime type of `A`, then the edges to `B.m` and `C.m` are spurious.

Spurious nodes are included when all incoming edges to the node are spurious. In the example, if the edge from `a.m()` to `C.m` is spurious, then the node `C.m` would also become spurious.

The analyses presented in this paper are designed to reduce the number of spurious edges and nodes by providing better approximations of the runtime types of receivers.

2.2.3 Building the Conservative Call Graph

In our implementation, call graphs are built iteratively using a worklist strategy. The worklist starts with nodes for all possible entry points (i.e. `main`, `start`, `run`). As each node (method) is added to the call graph, edges from the call sites in the node are also added. If the target of an edge is not already in the call graph, then it is added to the call graph and to the worklist. Conservative call graphs are built using *hierarchy_types* as the estimate for *runtime_types* for determining the edges from virtual method call sites.

Consider the example in Figure 2. The conservative call graph starts with the entry method `C.main` which includes two call sites `a.m()` and `b.m()`. Next, edges are added from `a.m()`. The type of receiver `a` is estimated using hierarchy analysis on the declared type of `a`, *Hierarchy_types(A)*=$\{A,B,C\}$. For each element of this set, the appropriate method `m` is located, leading to three call edges to `A.m`, `B.m` and `C.m`. The edges from call site `b.m()` are added similarly, leading to one edge to `B.m`. There is one remaining call site, `this.toString()` which is inside method `A.m`. The declared type of `this` is `A`, and *hierarchy_types(A)*= $\{A,B,C\}$. However, in this case all three types lead to the same call edge to the method `Object.toString()`. This illustrates the point that a tighter estimate of run-time types may not necessarily lead to fewer edges. Thus, our experimental measurements concentrate on measuring the number of call edges, and not the accuracy of the type resolution.

2.3 Rapid Type Analysis

Rapid type analysis [9] is a very simple way of improving the estimate of the types of receivers. The observation is that a receiver can only have a type of a object that has been instantiated via a `new`. Thus, one can collect the set of object types instantiated in the program P, call this *instantiated_types(P)*. Given a receiver o with declared type C with respect to program P, we define *rapid_types(C,P)* = *hierarchy_types(C)* $\cap$ *instantiated_types(P)*.

As an example, consider the program P given in Figure 2(a), and assume that the program contains instantiations of objects of type `A` and `B`. Now consider the call site `a.m()`, where `a` has declared type `A`. In this case we would use *rapid_types(P,A)* = $\{A,B\}$ to find the runtime types for receiver `a`. This leads to only two call edges, to `A.m` and `B.m`. So, using rapid type analysis the call graph would not include the call edge to `C.m`, nor would it include the node for `C.m`.

We have implemented rapid type analysis in our framework in order to give us a baseline for comparison with our other methods.

3 Reaching-type Analyses

Rapid type analysis can be considered to be a very coarse-grain mechanism for approximating which types reach a receiver of a method invocation. In effect, rapid type analysis says that a type A reaches a receiver o if there is an instantiation of an object of type A (i.e. an expression `new A()` anywhere in the program, and A is a plausible type for o using hierarchy analysis. In this section we propose two analyses that results finer-grain approximations by taking into consideration chains of assignments between instantiations of A and the receiver o.

Assuming an intermediate form like Jimple, where all computations are broken down into simple assignments, and assuming no aliasing between variables, we can state the following property. For a type A to *reach* a receiver o there must be some execution path through the program which starts with a a call of a constructor of the form $v = newA()$ followed by some chain of assignments of the form $x_1 = v, x_2 = x_1, \ldots x_{n-1} = x_n, o = x_n$. The individual assignments may be regular assignment statements, or the implicit assignments performed at method invocations and method returns.

We propose two flow-insensitive approximations of this reaching-types property. Both analyses proceed by: (1) building a *type propagation graph*, (2) initializing the graph with type information generated by $\widehat{n}$ew() statements, and, (3) propagating type information along directed edges.

For a program P, each receiver o is associated with some node in the type propagation graph, called *representative(o)*. Further, after propagating the types, each node n in the type propagation graph is associated with a set of types, called *reaching_types(n)*. Given a receiver o, the types reaching o is the set *reaching_types(representative(o))*.

In the following subsections we describe the analysis in more detail. We first present the more accurate analysis, called *variable-type* analysis, where the representative for a receiver o is the name of o, and then explain a coarser-grain variant called *declared-type* analysis where the representative for o is the declared type of o.

3.1 Variable-type analysis

Variable type analysis uses the "name" of a variable as its representative. In Jimple we can have three kinds of variable references, and we assign representative names as follows:

Ordinary references: are of the form `a`, and refer to locals or parameters. The name `C.m.a` is used as our representative, where `C` is the enclosing class and `m` is the enclosing method.

Field references: are of the form `a.f` where a could be a local, a parameter, or the special identifier `this`. We use as the representative the name of the field only (i.e. `f`). This means that we are approximating all instances of objects with field `f` by one representative node in the type propagation graph.

Array references: are of the form `a[x]`, where a is a local or parameter, and x is a local, parameter, or constant. We treat arrays as one large aggregate, so the name `C.m.a` is used, similar to the ordinary reference case.

3.1.1 Constructing the type propagation graph

Given a program P, where P consists of all classes that are referred to in the conservative call graph, nodes are created as follows:

- for every class C that is included in P
  - ⊙ for every field f in C, where f has an object type
    create a node labeled with f

- for every method $C.m$ that is included in the conservative call graph of P
  - ⊙ for every formal parameter p_i of $C.m$, where p_i has an object type
    create a node labeled $C.m.p_i$
  - ⊙ for every local variable l_i of $C.m$, where l_i has an object type
    create a node labeled $C.m.l_i$
  - ⊙ create a node labeled $C.m.this$ to represent the implicit first parameter
  - ⊙ create a node labeled $C.m.return$ to represent the return value $C.m$

  Note that the last two rules could be optimized to add the $C.m.this$ node only when the method refers to `this`, and to add $C.m.return$ only when the method returns an object type. Our current implementation does not perform this optimization.

Once all of the nodes have been created, we add edges for all assignments that involve assigning to a variable with an object type. These may be either direct assignments via assignment statements, and indirect assignments via method invocation and returns. Edges are added as follows:

Assignment Statements: are all in the form *lhs = rhs;*, where the *lhs* and *rhs* must be an ordinary, field or array reference. For each statement of this form, we add a directed edge from the representative node for *rhs* to the representative node of *lhs*.

Method Calls: are in the form of $lhs = o.m(a_1, a_2, \ldots, a_n)$; or $o.m(a_1, a_2, \ldots, a_n)$;. The receiver o must be a local, a parameter, or the special identifier `this`. The arguments must be a constant, a local, or parameter name.

The method call corresponds to some call site, call it *C.m[i]*, in the conservative call graph. Assignment edges are added as follows:

for each $C'.m'$ that is the target of $C.m[i]$ in the conservative call graph
- ⊙ add an edge from the representative of o to $C'.m'.this$
- ⊙ if the return type is not void
  add an edge from $C'.m'.return$ to the representative for lhs
- ⊙ for each argument a_i that has object type
  add an edge from the representative of a_i to the representative of
  the matching parameter of $C'.m'$.

In Figure 3(a) we give the important parts of an example program. Note that since our analysis is flow-insensitive, the order of assignments is not important, nor is control flow. Thus, this list of assignments represents a program that contains those assignments. This program has only ordinary variables of the form `a1`, `a2`, `a3`, `b1`, `b2`, `b3`, `c`. Figure 3(b) shows the initial graph. There is one node per variable, and one edge per assignment. For example, the assignment `a3 = b3;` corresponds to the edge from `b3` to `a3`.

3.1.2 Aliases

All of the assignment rules assume that a variable reference, and all of its aliases, are represented by exactly one node in the type propagation graph. That is, if `a` and `b` are aliases, then they should correspond to the same node in the graph. This is true for ordinary references because locals and parameters cannot be aliased in Java.[3] It is also true for field references because we represent all instances of objects with that field as one node in the graph. So, if two field references `a.f` and `b.f` are aliased (`a` and `b` refer to the same object) it is fine because we are representing them both with a field called `f`. However, it is not true for array references because several different variable names may refer to the same array. Further, references to arrays can be stored in variables with type `java.lang.Object`.[4] Thus, when adding edges for assignments of the form *lhs* = *rhs*, where both sides are of type `java.lang.Object`, or when at least one side has an array type, edges are added in **both** directions between the representatives of *rhs* and *lhs*. This encodes the aliasing relationship, and both nodes are guaranteed to be assigned the same solution.

3.1.3 Size of the propagation graph

Note that the type propagation graph includes at most $2M + P + L + F$ nodes, where M is the number of methods, P is the total number of parameters, L is the total number of locals, and F is the number of fields in the program under analysis. Thus, it seems reasonable to conclude that the number of nodes grows linearly with the size of the program.

The number of edges is slightly more difficult to estimate. There is at most one edge for each assignment statement in the program. However, the number of edges due to method calls depends on the number of targets for call sites. In the worst case a method call may have C targets, where C is the number of classes in the program under analysis. Thus, each method call could result in $C \times (2 + num\_params)$ edges being added to the type propagation graph. So, it is possible to have $O(C \times M_c)$ edges, where C is the number of classes and M_c is the number of method calls in the program under analysis. In practice we do not find this behaviour, and in fact the graphs are quite sparse (see Section 5).

[3] That is, two locals `a` and `b` must represent different locations, and there is no mechanism for getting a pointer to those locations.

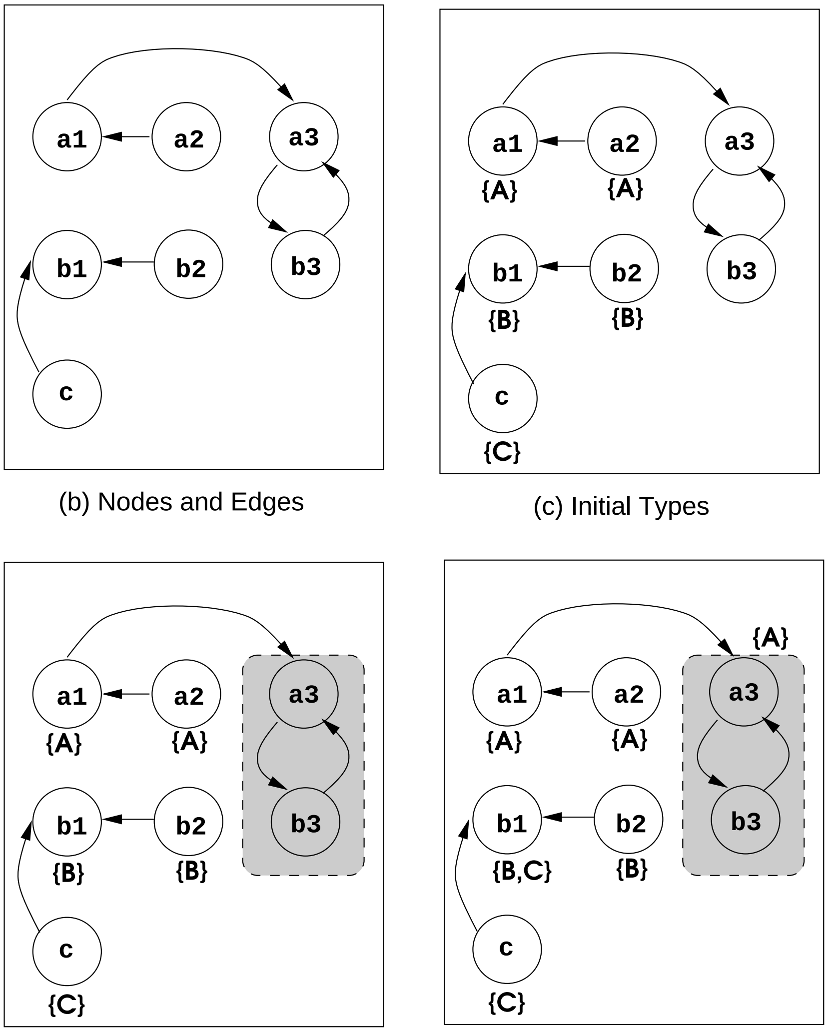
[4] For example, consider `A[] a = new A[10]; Object o1 = a; Object o2 = o1; A[] b = o2;` In this case `a`, `o`, `o1`, `o2` and `b` are all referring to the same array.

```
A a1, a2, a3;
B b1, b2;
C c;

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

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;
```

(a) Program



(b) Nodes and Edges

(c) Initial Types

(d) Strongly-connected components

(e) final solution

Figure 3: Example of a variable-type analysis

3.1.4 Initializing and propagating types

In the initialization phase, we visit each statement of the form *lhs = new A();* or *lhs = new A[n];*. For each such statement we add the type A to the *ReachingTypes* set of representative node for *lhs*. Figure 3(c) shows the type initialization for the example program.

After initialization, we propagate types. This is accomplished in two phases. The first phase finds strongly-connected components in the type propagation graph. Each strongly-connected component is then collapsed into one supernode, with *ReachingTypes* of this collapsed node initialized to the union of all *ReachingTypes* of its constituent nodes. Figure 3(d) shows two nodes collapsed. In this case neither node had an initial type assignment, so the collapsed node has no type assignment either.

After collapsing the strongly-connected components, the remaining graph is a DAG, and types are propagated in a single pass starting from the roots in a breadth-first manner. Note that both the strongly-connected component detection and propagation on the DAG is has complexity $O(max(N, E))$ operations, where the most expensive operation is a union of two ReachingType sets.

Figure 3(e) shows the final solution for our small example. From this solution we can infer that variables `a1`, `a2`, `a3` and `b3` have a reaching type `A`(i.e. they can only refer to objects of type `A`). Variable `b2` has a reaching type type `B`, `c` has a reaching type of `C`, and `b3` has a reaching type of A,B.

3.2 Declared-Type Analysis

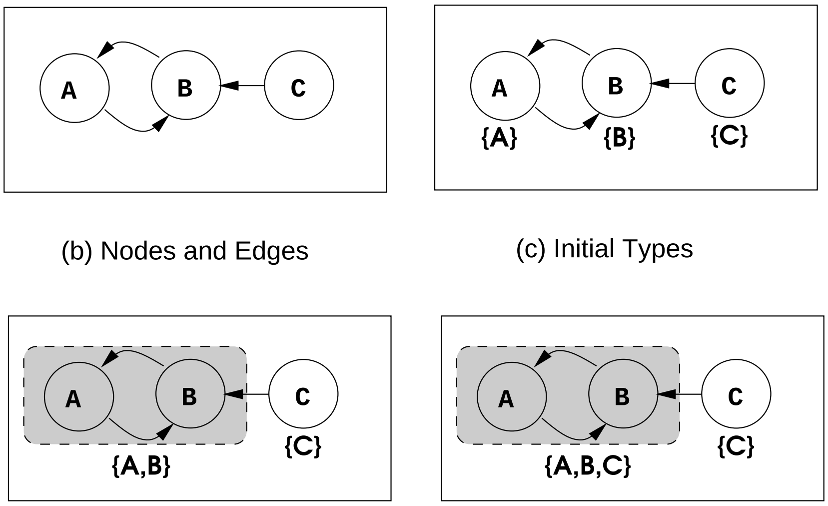
Declared-type analysis proceeds exactly as variable-type analysis, except for the way in which we allocate representative nodes for variables. In declared-type analysis we use the declared type of the variable as the representative, instead of the variable name. Basically, this is just putting all variables with the same declared type into the same equivalence class. Figure 4 shows the declared-type analysis for same program for which we previously computed the variable-type analysis. Note that the size of the graph is considerably smaller, but also the final answer is not as precise. The declared-type analysis concluded that all variables with declared type of `C` must point to `C` objects. However, it conservatively concludes that variables with a declared type of `A` or `B` might point to `A`, `B` or `C` objects. In Section 5 we present empirical results to evaluate these two analyses with respect to accuracy and the size of the graph problem to be solved.

```
A a1, a2, a3;
B b1, b2;
C c;

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

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;
```

(a) Program



(b) Nodes and Edges

(c) Initial Types

(d) Strongly-connected components

(e) final solution

Figure 4: Example of a declared-type analysis

4 Refers-to Analyses

Reaching-type analysis was specially designed to estimate the types of receivers, and it does not give any useful answers for aliasing. In fact, it starts with the assumption that all objects referencing the same field name are aliased.

On the other hand, there are analyses that are specifically designed to capture aliasing relationships, like the various flow-insensitive points-to analyses previously designed for C/C++ [23, 4, 20, 24]. The results of such an analysis can also be used to estimate the types of receivers. In a points-to analysis, each allocation (or each invocation of a constructor in Java) is associated with a *allocation site*, which can be considered to be a unique label. In Java, unlike in C, each allocation site allocates an object of a specific type (i.e. the constructor `new A()` is guaranteed to produce an object of type `A`. Thus, given a points-to set for a receiver o (i.e. *points_to(o)* is the set of all variables and allocation sites pointed-to by o), we can deduce the possible runtime types of o by merging the types of all the allocation sites in *points_to(o)*.

When designing our optimizer for Jimple, our original intent was to use reaching-type analysis to prune the conservative call graph, and then, based on an improved call graph, use a variation of points-to analysis to compute alias and side-effects. However, an interesting question is whether or not the points-to analysis itself can effectively prune the call graph. Can we get precise enough estimates of the runtime types of receivers using a flow-insensitive points-to analysis?

In order to examine this question we implemented a a simple version of Steensgaard's points-to algorithm[23], modified to work with our Jimple representation of Java. We call this variation refers-to analysis because it does not need to deal with general pointer relationships.

The general idea of the algorithm is as follows. There are two kinds of nodes, *reference nodes* and *abstract location nodes*. Reference nodes represent Java variable references (locals, parameters, instance fields), whereas abstract locations represent heap locations. Reference nodes are said to refer-to abstract locations. At object instantiation sites we create a special kind of reference node, called a *new_reference node*, which also contains the type of the object instantiated.

Initially each reference node refers-to a unique abstract location. The analysis proceeds by examining all assignments (where assignments include explicit assignments and assignments due to method calls, just as in the reaching-types analysis). Each assignment of the form $lhs = rhs$ causes the abstract locations for lhs to be merged with the abstract locations for rhs. As in Steensgaard's approach we perform these merges using the fast set-union find algorithm, and the notion of an Equivalence Class Representative (ECR) for each set of abstract locations. After performing the analysis each reference will refer to an ECR. If a receiver o refers-to the same ECR as a new_reference node with type `A`, then we can conclude that o might have a run-time type of A.

In the analysis that we used for our experiments, arrays and objects as a single aggregate (we did not distinguish between different fields of an object), thus we call it the *aggregate refers-to analysis*. We have also implemented another variation that distinguishes between fields, call *withfield refers-to analysis*.

For the same example program previously presented in Figures 3 and 4, the equivalence classes computed by this algorithm are `newA.1`, `newA.2`, `a1`, `a2`, `a3`, `b3` and `newB.1`, `newB.2`, `newC.1` `b1`, `b2`, `c`, where `newA.1` represents the first allocation site with type `A`. This is more precise than declared-type analysis, because it successfully finds that all of the `a` variables and `b3` can only refer-to objects of type `A`. However, it is less precise than variable-type analysis because it conservatively grouped `c` with objects of type `B`.

5 Experimental Results

5.1 Benchmarks

We have experimented with 9 benchmarks, as outlined in Table I. The first two benchmarks are from the McGill benchmark set, `puzzle` is an image processing application that takes four files describing puzzle pieces and attempts to fit them together, `jimple` is an earlier version of our compiler software. The last seven benchmarks are from the SPECjvm benchmark suite. The program `mtrt` is a raytracer, `jess` is an expert shell system based on NASA's CLIPS expert system, `compress` is a compression program based on a modified Lempel-Ziv method, `db` performs multiple database functions on a memory-resident database, `mpegaudio` decompresses audio files, `jack` is a java parser generator based on the Purdue Compiler Construction Tool Set (PCCTS), and `javac` is the Java compiler from the JDK 1.0.2.

The statistics in Table I provide an insight into the nature (the extent of object orientedness of) the benchmarks for which we have conducted experiments. We have shown the number of Jimple statements in each benchmark in column 1, while columns 2 and 3 show the average and maximum depths respectively in the class hierarchy. We have broken down the total number of classes in the benchmark into different categories, column 4 shows the number of library classes in each benchmark while column 5 shows the number of classes that were in the actual application. Columns 6 and 7 show the number of classes and interfaces in the benchmark. We have also shown the total number of methods in the benchmarks and computed the number that are abstract and native (methods that are not abstract or native are concrete methods).

Name	**Stmt**	**Hierarchy**		**Classes and Interfaces**					**Methods**			
		avg. depth	max depth	lib.	app.	class	int.	total	concrete	abstract	native	total
puzzle	14679	3.2	6	186	3	179	10	189	1808	129	173	2110
jimple	42539	3.3	6	194	503	667	30	697	4108	261	190	4559
_227_mtrt	27094	3.0	6	320	35	313	42	355	3298	402	184	3884
_202_jess	32916	2.9	6	325	116	396	45	441	3690	419	184	4293
_201_compress	24181	3.0	6	320	22	300	42	342	3168	399	184	3751
_209_db	24948	3.0	6	322	14	294	42	336	3197	401	184	3782
_222_mpegaudio	36182	3.0	6	320	62	332	50	382	3404	441	184	4029
_228_jack	30769	3.0	6	320	67	341	46	387	3436	412	184	4032
_219_javac	44646	3.5	8	323	182	459	46	505	4331	410	193	4934

Table I: Benchmark Characteristics

Table II gives a summary of the conservative call graph built for each benchmark using Class Hierarchy Analysis (CHA).

Column 1 shows N, the number of methods in the call graph. We categorise the call sites based on the kind of invoke expression at the call site. In Java there are 4 possible kinds of invoke expressions, invokestatic, invokespecial, invokevirtual, and invokeinterface and we have shown the number of call sites in each category. Furthermore since invokevirtual and invokeinterface callsites may have edges to more than one callee method in the conservative call graph, we also categorise call sites in these 2 categories based on the number of callee methods in the call graph. In the last 2 columns we show the number of monomorphic edges and the number of polymorphic edges in the call graph.

Polymorphic edges originate from potentially polymorphic callsites. A callsite is potentially polymorphic if there is more than 1 callee method attached to that callsite in the conservative call graph. Call sites that are not polymorphic are termed monomorphic and edges originating from these call sites are called monomorphic edges.

It must be noted that a substantial number of invokevirtual and invokeinterface callsites have edges to only 1 callee method in the CHA call graph that we start with indicating that these callsites could possibly be optimized even without any further analysis. It needs to be emphasized that the call graph that we obtain from CHA is already reasonably precise, and the edges that are present in this call graph are required to ensure correctness, and cannot be removed unless a more complex analysis is used.

Name	$\|N\|$	**Call Sites**											**Edges**		
		static	special	virtual				interface				total	mono.	poly.	total
				1	2	>2	total	1	2	>2	total		(#)	(#)	(#)
puzzle	682	298	641	1489	26	109	1629	0	1	9	11	2579	2428	1178	3606
jimple	2662	1533	2629	6217	51	554	6837	17	163	693	1079	12078	10396	16909	27305
mtrt	1473	582	1795	3486	55	172	3720	91	26	40	171	6268	5954	2021	7975
jess	1797	852	2345	4188	66	187	4457	112	26	45	198	7852	7497	2865	10362
compress	1327	543	1638	2548	55	166	2776	91	26	40	171	5128	4820	1992	6812
db	1359	554	1705	2710	55	169	2943	91	26	60	191	5393	5060	2144	7204
mpegaudio	1570	580	1850	2961	65	191	3224	106	26	40	186	5840	5497	2432	7929
jack	1586	669	2388	3258	377	183	3930	107	26	90	252	7239	6422	2994	9416
javac	2406	750	2511	5639	219	764	6732	92	26	102	244	10237	8992	12576	21568

Table II: Conservative Call Graph Characteristics

5.2 Improvements over the Conservative Call Graph

We have obtained the results shown in Table III from our analysis.

The number of dead method nodes removed by RTA varies between 9 percent of the total number of methods in the conservative call graph (for jimple) to about 35 percent (for javac). The analyses that we have considered are all expected to perform better at removing dead methods if there are more library classes in the application. This is because CHA builds the call graph based on the class hierarchy. If a certain library class O and its subclasses all implemented method *m()* and if all these classes are part of the class hierarchy for the application, CHA would add edges from *o.m()*(o is of declared type O) to each of the *m()*'s in the class O and its subclasses. It is extremely likely that the application would only instantiate a few of the suclasses of O (that it requires) and so most of the edges that are present in the CHA call graph are not needed in actual fact. This scenario is more likely to occur for library classes in our opinion and so we believe that the more complex analyses are expected to do better when there are many callsites to library methods.

This expected behaviour is observed in practice as there are a greater proportion of methods removed in applications like javac that involve many library classes as compared to an application like jimple. Refers-To analysis performs only slightly better than RTA in removing dead methods while Declared Type Analysis shows an increase of 15 to 20 percent in the number of dead methods removed. Variable Type Analysis shows a similar improvement over Declared Type Analysis.

The result for the number of dead edges removed by each of the competing analyses follows a pattern similar to what was observed for dead methods removal. Variable Type Analysis and to a lesser degree, Declared Type Analysis show a clear improvement over RTA in terms of edges removed. We feel that the reason for Refers-To analysis not performing well is that in a majority of the references in the benchmarks that we analysed were classified in one single ECR resulting in a considerable loss in precision. This behaviour suggests that Refers-To analysis may not be the ideal approach to solve the problem of call graph improvement.

We have also presented results for the number of call sites reduced to 0 (implying that the method invocation at that call site would never actually be executed), and the number of call sites reduced to 1 callee method as a result of edge removal (these are candidates to be considered for optimizations like method inlining).

Variable Type Analysis is clearly observed to be the most effective analysis of the ones we have studied for improving the precision of the call graph. In most cases about 30 percent of the edges in the original call graph are removed by Variable Type Analysis. Declared Type Analysis is observed to perform better than Rapid Type Analysis but is clearly not as effective as Variable Type Analysis. Refers-To analysis is observed to be relatively ineffective in producing substantial improvement over RTA.

The results of the SPEC VM benchmarks are very similar to one another because these benchmarks consist of many library classes and only a few classes from the specific application. Since the results for the libraries are likely to be similar irrespective of the calling application the overall results seem to quite similar for these benchmarks.

Our implementation is not yet tuned for speed, so in order to give an estimate of the time required for each analysis, we gathered information about the size of the data structures built for each algorithm. In Table IV, we show our measurements. It needs to be noted that for Declared Type Analysis and Variable Type Analysis, the time required to obtain the solution is directly proportional to the number of edges in the constraint graph after the graph has been transformed such that each strongly connected component in the original constraint graph is replaced by special SCC nodes. The number of edges in the constraint graph is observed to grow linearly with the size of the application for both Declared Type Analysis and Variable Type Analysis. For Refers-To analysis we can observe that the number of ECR unions grows linearly with the number of variables in the application. The time required to obtain a solution from Refers-To analysis is proportional to the number of merges. We can clearly see that all the three relatively

Name	Analysis	Nodes Removed	Edges Removed	Virtual Call Edge Reductions			Interface Call Edge Reductions		
				$\to 0$	$\to 1$	$\to 2$	$\to 0$	$\to 1$	$\to 2$
puzzle	rapid-type	173 (25%)	719 (20%)	50	70	6	1	0	9
	aggr. refers-to	188 (27%)	741 (21.5%)	79	28	49	1	0	9
	declared-type	209 (31%)	895 (25%)	60	75	9	1	0	9
	variable-type	249 (37%)	1097 (30%)	215	51	10	2	5	0
jimple	rapid-type	246 (9.2%)	1710 (6.3%)	49	60	52	1	143	5
	aggr. refers-to	258 (9.6%)	1766 (6.5%)	77	48	52	1	143	5
	declared-type	316 (12%)	2430 (8.9%)	93	55	42	1	148	25
	variable-type	365 (14%)	4717 (17%)	158	78	51	3	284	86
mtrt	rapid-type	508 (35%)	1491 (19%)	435	42	50	55	18	0
	aggr. refers-to	517 (35%)	1528 (19.2%)	497	29	50	55	18	0
	declared-type	581 (39%)	1816 (23%)	498	48	43	55	18	13
	variable-type	642 (44%)	2218 (28%)	670	56	48	70	27	0
jess	rapid-type	516 (28.7%)	1654 (16%)	436	42	46	55	18	0
	aggr. refers-to	525 (29.2%)	1698 (16.3%)	486	29	46	55	18	0
	declared-type	595 (33.1%)	2049 (19.8%)	464	48	40	55	18	13
	variable-type	659 (36.6%)	2598 (25%)	688	73	46	73	27	0
compress	rapid-type	509 (38.4%)	1569 (23.0%)	432	42	46	55	18	0
	aggr. refers-to	518 (39.0%)	1606 (23.6%)	479	29	46	55	18	0
	declared-type	577 (43.5%)	1848 (27.1%)	455	48	40	56	18	13
	variable-type	642 (48.4%)	2257 (33.1%)	674	73	46	71	27	0
db	rapid-type	510 (37.5%)	1526 (21.1%)	435	42	15	55	18	0
	aggr. refers-to	518 (38.1%)	1562 (19.6%)	482	29	15	55	18	0
	declared-type	579 (42.6%)	1861 (25.8%)	459	48	40	55	18	33
	variable-type	643 (47.3%)	2329 (32.3%)	668	58	44	73	47	45
mpegaudio	rapid-type	547 (34.8%)	1653 (20.8%)	432	51	31	55	18	0
	aggr. refers-to	556 (35.4%)	1690 (21.3%)	479	38	31	55	18	0
	declared-type	624 (39.7%)	2036 (25.6%)	459	55	56	55	18	13
	variable-type	688 (43.8%)	2514 (31.7%)	668	65	61	76	27	0
jack	rapid-type	518 (33%)	1631 (17%)	433	59	22	55	18	0
	aggr. refers-to	527 (33.2%)	1668 (17.7%)	480	46	22	55	18	0
	declared-type	589 (37%)	1981 (21%)	488	18	14	55	18	61
	variable-type	646 (41%)	2701 (29%)	664	338	36	75	75	0
javac	rapid-type	846 (35%)	6418 (30%)	783	66	34	56	18	5
	aggr. refers-to	852 (35.4%)	7112 (32.9%)	1012	88	104	56	18	5
	declared-type	935 (39%)	7230 (34%)	900	66	46	56	23	0
	variable-type	1008 (41.8%)	9433 (44%)	1187	68	41	79	32	0

Table III: Improvement of Call Graph over Conservative Call Graph

complex analyses that we have studied scale quite well as the size of the application increases. As can be seen from the tables, the constraint graph for Variable Type Analysis has about 3 times the number of nodes, and about 7 times the number of edges that are present in the constraint graph for Declared Type Analysis. This gives a good indication about the relative costs of these 2 analyses.

6 Related Work

Related work to our study of analyses to improve the call graph and elimination of virtual calls is discussed in some detail in the work by Grove [11]. They conduct an empirical study of the effectiveness of many of the commonly known algorithms for call graph construction. The suite of benchmarks that they used for conducting their experiments was composed of medium-sized programs written in Cecil and Java. complexity of some of the well known algorithms for call graph construction. They discuss the different strategies for call graph improvement in a generalized manner, and give the possibilities for the choice of the initial call graph. We have focused on the design and study of fast analyses for call graph improvement while

Name	Jimple Stmts	Call Graph		Declared Type				Variable Type				Aggregated Method	
		$\lvert N\rvert$	$\lvert E\rvert$	before SCC		after SCC		before SCC		after SCC			
				$\lvert N\rvert$	$\lvert E\rvert$	$\lvert N\rvert$	$\lvert E\rvert$	$\lvert N\rvert$	$\lvert E\rvert$	$\lvert N\rvert$	$\lvert E\rvert$	ECRs	merges
puzzle	14679	682	3606	4437	1226	4253	841	13324	8377	12475	6132	7617	11818
jimple	42539	2662	27305	9877	6089	8918	3581	22425	37807	19239	20615	27728	67422
mtrt	27094	1473	7975	8203	2843	7741	1841	24788	17703	23044	13280	14897	24139
jess	32916	1797	10362	9109	3608	8520	2343	28709	23374	26345	16977	18956	31188
compress	24181	1327	6812	7926	2536	7521	1656	23223	15179	21764	11316	13241	21311
db	24948	1359	7204	7982	2594	7574	1707	23584	15919	22056	11876	13669	22163
mpegaudio	36182	1570	7929	8520	2888	8099	1976	25014	17593	23372	13216	17390	26647
jack	30769	1586	9416	8533	3162	8068	2144	26678	20633	24916	16075	16876	28123
javac	44646	2406	21568	10445	5284	9475	3031	29045	37374	25661	24934	25529	69059

Table IV: Size of Data Structures

some of their analyses for call graph construction are context-sensitive or flow-sensitive and are consequently more expensive. Also our work focuses entirely on call graph construction for Java applications, and intends to provide a detailed set of measurements that compare the effectiveness of the relatively cheaper analyses that we have considered. They have also discussed the effectiveness of their techniques with respect to interprocedural optimizations which is what we propose to do in the future. Calder and Grunwald [5], and Holzle and Ungar [14] describe transformations to convert method invocations to direct calls.

Diwan [8] describes results for simple and effective analysis of statically-typed object-oriented languages. Their analysis technique is interesting because they compare their results with an "oracle" that resolves all the possible method invocations, and thus provide a clear bound on the best that can be achieved by using analyses techniques. They also explain their *cause analysis technique* which tries to provide reasons for a particular analysis losing information and accuracy as a result. They conclude that the simpler analyses like type hierarchy analysis are effective in resolving most of the method call sites that can be resolved in Modula-3 programs that they used for experiments. However they differ from us in that they also consider some flow-sensitive analyses in their study while all the analyses in our study are context and flow insensitive. The results of Dean [7] suggest that type hierarchy analysis is a good technique for resolving many method invocations for the Cecil language. Fernandez [10] implemented virtual call elimination and used an idea that is essentially Class Hierarchy Analysis (CHA). Pande [19] discuss a technique that uses a pointer analysis algorithm for static type determination for C++ programs. Our work confirms that CHA does work well in Java, but we also show several linear methods that do substantially better than CHA.

Probably the most closely related work is by Bacon and Sweeney on fast static analysis of C++ virtual function calls[9]. Their study considers three relatively simple analysis techniques Unique Name, Class Hierarchy Analysis, and Rapid Type Analysis. The analyses that we have considered are slightly more expensive and complex, but we have also implemented rapid type analysis in order to show improvement relative to it. Further, we have presented results for Java applications. They also give the analysis time for each of their analysis, that show that the overhead for these analyses is not very significant when compared to the time to compile. They have dynamically measured the results for resolution of user virtual calls, and also try to produce an estimate for the number of dead call sites. They conclude that rapid type analysis is extremely effective in resolving function calls, and reducing code size and it is also proven to be very fast. Our results seem to confirm that rapid type analysis does also work with Java, and we also show that our reaching-type analyses can given even better results. Aigner and Holzle [3] find that type feedback and type hierarchy analysis are both effective at resolving method invocations in C++.

Plevyak and Chien's iterative algorithm [15] tries to improve a safe call graph to begin with and tries to refine it to the desired extent by creating new contours.

There has also been work in the area of applying more expensive analyses of varying complexity for call graph construction, especially for languages like C++, Modula-3, and Cecil. Some of the algorithms that are context insensitive are 0-CFA [21, 22], Palsberg and Schwartzbach's algorithm [18], Hall and Kennedy's call graph construction algorithm for Fortran [25], and Lakhotia's algorithm [16] for building a call graph in languages with higher order functions. Other related work includes Shiver's k-CFA family of algorithms [21, 22] for selecting the target contour based on k enclosing calling contours at each call site, Agesen's Cartesian Product Algorithm [2], and Ryder's [12] call graph construction algorithm for Fortran 77. Agesen [1] describes constraint-graph-based instantiations of k-CFA, and Plevyak's algorithm.

Our work is also related to flow-insensitive points-to analysis for C [23, 4, 20]. Our refers-to analysis is a simpler version of the linear points-to algorithm of Steensgaard [23]. There has been considerable work evaluating flow-insensitive and flow-sensitive analyses for C [4, 24, 13, 20]. However, we feel that the tradeoffs in Java will be different, because many pointer relationships in C are simple call-by-reference relationships, while all pointer relationships in Java are always due to references to dynamically-allocated objects.

Our work build on the Soot framework under development at McGill. Alternative implementation frameworks for handling Java code exist, such as Harissa [17], Vortex [6].

7 Conclusions and Future Work

In this paper we have presented a new flow-insensitive analysis called reaching-type analysis that can be used to estimate the possible types of receivers for virtual method/interface calls in Java. Two variations of the analysis were presented, variable-type analysis that uses the name of a receiver as its representative, and declared-type of a receiver as its representative. We also proposed the possibility that a variation of a linear points-to algorithm, called refers-to analysis, could also be used to estimate the types of receivers.

These three analyses, plus hierarchy analysis and rapid type analysis, two previously developed type estimation techniques, were implemented with Soot, an environment that translates Java bytecode to a typed three-address code. All five analyses were applied to 9 Java benchmarks.

Hierarchy analysis was used to build an initial conservative call graph. Measurements of these graphs confirm what others have noted, hierarchy analysis leads to a conservative call graph that is fairly sparse, with a majority of call sites resolving to a single method.

We applied the other analyses based on the initial conservative call graph, and found that a significant number of edges and nodes could be removed, ranging from about 15and nodes. The effectiveness of the variable-type analysis was the best, giving about 10% more reductions that rapid-type analysis, and about 5% more reductions than declared-type analysis.

The results for the refers-to analysis show that it is not a good candidate for virtual method resolution. It performed better than rapid type analysis, but not as well as the reaching-type analyses. This was to be expected, but it was nice to see confirmation of this. Indeed, this means that our original idea of using reaching-type analysis to prune the call graph before computing side-effects using refers-to analysis is correct.

We are currently working on two more experiments. First, we want to collect some dynamic data concerning the actual runtime types of receivers. We have almost completed the part of the Soot framework that translates from Jimple back to bytecode. This translation works for several benchmarks already, but it cannot yet handle all of the library code. Using this translation we intend to instrument the Jimple code to collect the actual set of types for receivers at runtime. This will give us a lower bound on how well the static techniques can do, and will show what gap remains.

Second, we would like to measure the impact of having a better call graph by comparing the results of: (1) refers-to analysis using the conservative call graph, and (2) refers-to analysis using the improved call graph we compute with reaching-variable analysis.

Acknowledgements

This work was supported, in part, by NSERC, FCAR and IBM CAS (Toronto).

References

[1] Ole Agesen. Constraint-based type inference and parametric polymorphism. In *First International Static Analysis Symposium*, September 1994.

[2] Ole Agesen. The cartesian product algorithm : Simple and precise type inference of parametric polymorphism. In *ECOOP '95*, Aarhus, Denmark, August 1995.

[3] Gerald Aigner and Urs Hozle. Eliminating virtual function calls in c++ programs. In *European conference on object-oriented programming*, July 1996.

[4] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, 892*, pages 234–250. Springer-Verlag, 1995. Proceedings from the *7th Workshop on Languages and Compilers for Parallel Computing.*

[5] Brad Calder and Dick Grunwald. Reducing indirect function call overhead in c++ programs. In *Twenty-First Symposium on Principles of Programming Languages*, pages 397 – 408, Portland, Oregon, January 1994.

[6] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. VORTEX: An optimizing compiler for object-oriented languages, Oct. 1996.

[7] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[8] Amer Diwan, J. Eliot B.Moss, and Kathryn S.McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *OOPSLA '96 Conference*, San Jose, CA, October 1996.

[9] David F.Bacon and Peter F.Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96 Conference*, San Jose, CA, 1996.

[10] Mary F.Fernandez. Simple and effective optimization of modula-3 programs. In *Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[11] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object oriented languages. In *OOPSLA '97 Conference*, Atlanta, Georgia, October 1997.

[12] Barbara G.Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, pages 216 – 225, 1979.

[13] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Lecture Notes in Computer Science*. Springer-Verlag, 1998. Proceedings from the *5th International Static Analysis Symposium.*

[14] Urs Holzle and David Ungar. Optimizing dynamically dispatched calls with run-time type feedback. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326 – 336, June 1994.

[15] J.Plevyak and A.Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94*, pages 324 – 340, October 1994.

[16] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Twentieth Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[17] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Berkeley, Jun.16–20 1997. Usenix Association.

[18] Jens Palsberg and Michael I.Schwartzbach. Object oriented type inference. In *OOPSLA '91*, Phoenix, Arizona, October 1991.

[19] Hemant Pande and Barbara G.Ryder. Static type determination and aliasing for c++. Technical report, Rutgers University, July 1995.

[20] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive point-to analysis. In *Conf. Rec. of the 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 1–14, Paris, France, Jan. 1997.

[21] Olin Shivers. Control-flow analysis in scheme. In *ACM SIGPLAN'88 Conference on Programming Language Design and Implementation.*

[22] Olin Shivers. *Control-Flow analysis of Higher-Order Languages.* PhD thesis, May 1991.

[23] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conf. Rec. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 32–41, St. Petersburg, Flor., Jan. 1996.

[24] Philip A. Stocks, Barbara G. Ryder, William A. Landi, and Sean Zhang. Comparing flow and context sensitivity on the modifications-side-effects problem. In *International Symposium on Software Testing and Analysis*, pages 21–31, Mar. 1998.

[25] Mary W.Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, September 1992.