Practical Virtual Method Call Resolution for Java

Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa,
Raja Vallée-Rai, Patrick Lam, Etienne Gagnon and Charles Godin*
Sable Research Group (www.sable.mcgill.ca)
School of Computer Science
McGill University
Montreal, Quebec, CANADA H3A 2A7
[vijay,hendren,razafima,rvalleerai,plam,gagnon,cgodin]@sable.mcgill.ca

ABSTRACT

This paper addresses the problem of resolving virtual method and interface calls in Java bytecode. The main focus is on a new practical technique that can be used to analyze large applications. Our fundamental design goal was to develop a technique that can be solved with only one iteration, and thus scales linearly with the size of the program, while at the same time providing more accurate results than two popular existing linear techniques, *class hierarchy analysis* and *rapid type analysis*.

We present two variations of our new technique, *variable-type analysis* and a coarser-grain version called *declared-type analysis*. Both of these analyses are inexpensive, easy to implement, and our experimental results show that they scale linearly in the size of the program.

We have implemented our new analyses using the Soot framework, and we report on empirical results for seven benchmarks. We have used our techniques to build accurate call graphs for complete applications (including libraries) and we show that compared to a conservative call graph built using class hierarchy analysis, our new variable-type analysis can remove a significant number of nodes (methods) and call edges. Further, our results show that we can improve upon the compression obtained using rapid type analysis.

We also provide dynamic measurements of monomorphic call sites, focusing on the benchmark code excluding libraries. We demonstrate that when considering only the benchmark code, both rapid type analysis and our new declared-type analysis do not add much precision over class hierarchy analysis. However, our finer-grained variable-type analysis does resolve significantly more call sites, particularly for programs with more complex uses of objects.

*All work reported in this paper was done while all authors were at McGill University. Currently Vijay Sundaresan is at IBM Toronto, Chrislain Razafimahefa is at the University of Geneva, and Patrick Lam is at MIT.

1. INTRODUCTION

As the Java(tm) programming language and Java bytecode 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. A more accurate call graph can be used to: (1) compact applications by removing methods that are never called, and (2) improve the efficiency and accuracy of subsequent interprocedural analyses.

Of course, virtual method resolution is not a new problem. It has been widely studied for a variety of object-oriented languages[7, 8, 9, 10, 12, 14, 18, 16, 21, 24, 28, 29, 30]. The focus of this paper is the development and evaluation of a new simple and inexpensive technique for resolving virtual method calls in Java. A main design objective was to develop a technique that would produce a solution in one iteration and thus scales linearly in the size of the program. Further, we wanted a technique which was simple, easy to implement, could be applied to large Java applications, but yet could also yield more precision than two efficient existing techniques. In particular, we wanted to measure static and dynamic improvements over *class hierarchy analysis*[8, 14, 21] and *rapid type analysis*[8].

Our technique is based on an analysis that builds a *type propagation graph* where nodes represent variables and edges represents flow of types due to assignments, including the implicit assignments due to method invocation and method returns. The first variation is called *declared-type analysis*, where the nodes represent the declared type of variables. For this analysis the type propagation graph contains at most one node for each class in the application. The second variation is called *variable-type analysis* and it is finer-grained and more accurate, although still efficient. In this variation the type propagation graph contains at most one node for each variable with an object (reference) type. Both of these analyses can be thought of as more refined versions of rapid type analysis. Whereas rapid type analysis simply collects the types of all objects allocated, and uses this to prune the call graph, declared-type and variable-type analysis find which types of objects reach each variable (i.e. which allocated objects might be assigned to this variable).

In keeping with our desire for a simple and efficient analysis, the analyses were carefully designed so that one iteration over the type propagation graph results in a safe solution. Further, our algorithm is simple to implement and could easily be added to compilers that already have class hierarchy analysis and/or rapid type analysis.

All of the analyses were implemented using the Soot framework that provides Jimple, a typed three-address code representation of Java bytecode [1]. Since our framework operates on bytecode, our analysis is not restricted to Java, but can be used for bytecode produced from a wide variety of languages. The benchmarks used in our evaluation are meant to be representative of real applications and they include four SPECjvm benchmarks plus three other large, object-oriented benchmarks. We use these benchmarks to show both static and dynamic results that illustrate the accuracy of our analyses. When considering the whole applications, including library code, we found that the existing analyses did perform quite well, but our variable-type gave additional improvements. When considering only the benchmark code, factoring out the library code, we found that the existing analyses performed poorly, but our variable-type analysis performed significantly better.

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 class hierarchy analysis and rapid type analysis as implemented in our system. In Section 3 we outline variable-type analysis and the coarser-grain version, declared-type analysis. We present our experimental framework and empirical measurements in Section 4. Finally, in Section 5 we discuss related work, concentrating mostly on other efficient techniques, and in Section 6 we give our conclusions and future work.

2. FOUNDATIONS

2.1 The Soot 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 Application Programming Interfaces (APIs) for manipulating Java code in various forms [1]. We analyze complete applications, so our implementation works by first reading all class files that are required by an application, 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 such as 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 [22]. 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.

After analysis and transformation we convert the Jimple representation back to Java bytecode, making our entire system a bytecode to bytecode optimizer[37].

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. Another important point is that Jimple splits variables according to D/U and U/D webs, so that two unrelated variables of the same name will not be confused in our analyses.

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 Class 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 *class hierarchy analysis*.

2.2.1 Class Hierarchy Analysis

Class hierarchy analysis is a standard method for conservatively estimating the run-time types of receivers [8, 14, 21]. Given a receiver o of with a declared type d, *hierarchy_types(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(C)*, includes C plus all subclasses of C.
- If receiver o has a declared interface type I, the possible run-time types of o, *hierarchy_types(I)*, includes: (1) the set of all classes that implement I or implement a subinterface of I, which we call *implements(I)*, plus (2) all subclasses of *implements(I)*.

To implement this analysis, we simply build an internal representation of 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. For a single-threaded application, the call graph must include one node for each method that can be reached by a computation starting from the `main` method. If the program

```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

is an applet or has threads, then the call graph must also include all methods that can be reached starting at any entry point. 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 $C.M[1]$ to $C.M[n]$. In our example, the call graph node for method `C.main` contains two call sites, $C.main[1]$ which is `a.m()`, and $C.main[2]$ 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 $C.M[i]$ calls method $C'.M'$, then there must be an edge between $C.M[i]$ and $C'.M'$ in the call graph. In the example call graph there are three edges from the call site `a.m()` corresponding to 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_types(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 for 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 run-time 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. Note that entire subpieces of the call graph could become spurious if the subgraph becomes disconnected from the roots of the graph. In the example, if the edge from `a.m()` to `A.m` was spurious, then both the nodes for `A.m` and `Object.toString` 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 (for example, `main`, `start`, `run`). As each node (method) is added to the call graph, edges from

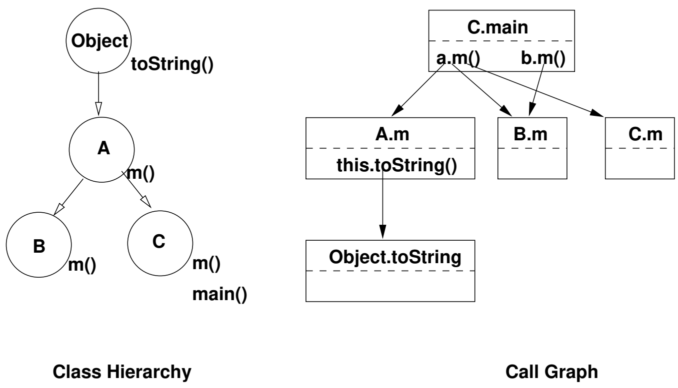
```
class A extends Object {
  String m() {
    return(this.toString());
  }
}

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 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`. For the call site `b.m()`, the declared type of `b` is `B`, *hierarchy_types(B)*=$\{B\}$, and so there is only one 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.

In our work we use the number of call edges from a call site (and **not** the number of run-time types of the receiver) to determine if the call site is monomorphic or polymorphic. If there is only one edge from a call site, we categorize the call site as *monomorphic*, whereas if there are two or more edges we categorize the call site as *polymorphic*. In the call graph in Figure 2, the call `a.m()` is polymorphic, whereas the calls `b.m()` and `this.toString()` are monomorphic.

2.3 Rapid Type Analysis

Rapid type analysis [8] 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 an 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 can use *rapid_types(C,P)* = *hierarchy_types(C)* $\cap$ *instantiated_types(P)* as a better estimate of the runtime types for o.

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 to `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`.

This particular version of rapid type analysis should be called *pessimistic rapid type analysis* since it starts with the complete conservative call graph built by CHA and looks for all instantiations in method in that call graph. This may, therefore, find an instantiation which is in a method that should really be removed from the call graph. The original approach suggested by Bacon and Sweeney [8] is *optimistic rapid type analysis*. In the optimistic approach the call graph is iteratively created, and only instantiations in methods already in the call graph are considered as possible set for computing *instantiated_types(P)*. We have implemented both variations and give experimental results comparing them in Section 4.

3. VARIABLE-TYPE AND DECLARED-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 class hierarchy analysis. In this section we propose two analyses that result in 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.[1] For a type A to *reach* a receiver o there must be some execution path through the program which starts with a call of a constructor of the form $v = newA()$ followed by some chain of assignments of the form $x_1 = v, x_2 = x_1, \ldots, x_n = x_{n-1}, o = x_n$. 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* where nodes represent variables, and each edge `a` $\rightarrow$ `b` represents an assignment of the form `b = a`, (2) initializing reaching type information generated by assignments of the form `b = new A()` (i.e. the node associated with `b` is initialized with the type `A`) and, (3) propagating type information along directed edges corresponding to chains of assignments.

For a program P, each variable a with an object (reference) type is associated with some node in the type propagation graph, called *representative(a)*. After propagating the types, each node n in the type propagation graph is associated with a set of types, called *reaching_types(n)*. Thus, after propagating types we can find out the set of types reaching any variable. For building call graphs we are particularly interested in types reaching variables used as receivers. Given a receiver o, with declared type C, we approximate the runtime types of o using *reaching_types(representative(o))* $\cap$ *hierarchy_types(C)*. Note that we filter out impossible reaching types by intersecting with possible types as indicated by hierarchy types.

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 variable a is the name of a, and then explain a coarser-grain variant called *declared-type* analysis where the representative for a is the declared type of a.

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 (more complex references are simplified into a combination of these simple ones), 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 `C.f` where `C` is the name of the class defining field `f`. Note that we ignore `a`, so this means that we are approximating all instances of objects with this field 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.

[1] We discuss why we do not have to consider aliasing in Section 3.1.2.

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 (reference) type
    create a node labeled with $C.f$
- 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 can 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. This does not affect the accuracy of the result, it just leaves out nodes that will have no edges to them.

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 explicit assignments via assignment statements, and implicit assignments due to method invocation and returns. Edges are added as follows:

Assignment Statements: are all in the form *lhs = rhs;* or *lhs = (C) 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`. Each argument 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 rep. of the matching parameter of $C'.m'$.

Note that we handle native methods by summarizing their effect on our analyses. None of the benchmarks for which we present results have any native methods; but there are some native methods in the Java library that are called by these benchmarks. We have examined the code for these Java library native methods in the open source Kaffe OpenVM [2] in order to find the appropriate summary.

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`.[2] 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. In fact, this is one of the key properties that makes our analysis simple. This property 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 `C.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`. For example, consider the following small example: `A[] a = new A[10]; Object o1 = a; Object o2 = o1; A[] b = (A[]) o2;` .
In this case `a`, `o`, `o1`, `o2` and `b` are all referring to the same array. So, an assignment to `a[i]` is also assigning to `b[i]`.

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. We did not find this situation occurring very frequently in our benchmarks.

[2] In the actual analysis the names are qualified by their surrounding class name and method name, we use the unqualified variable name to keep our example simple.

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

3.1.3 Size of the propagation graph

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 with an object type, L is the total number locals with an object type, and F is the number of fields with an object type. Thus, 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 behavior, and in fact the graphs are quite sparse (see Table 4 in Section 4).

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 the representative node for *lhs*. Figure 3(c) shows the type initialization for the example program.

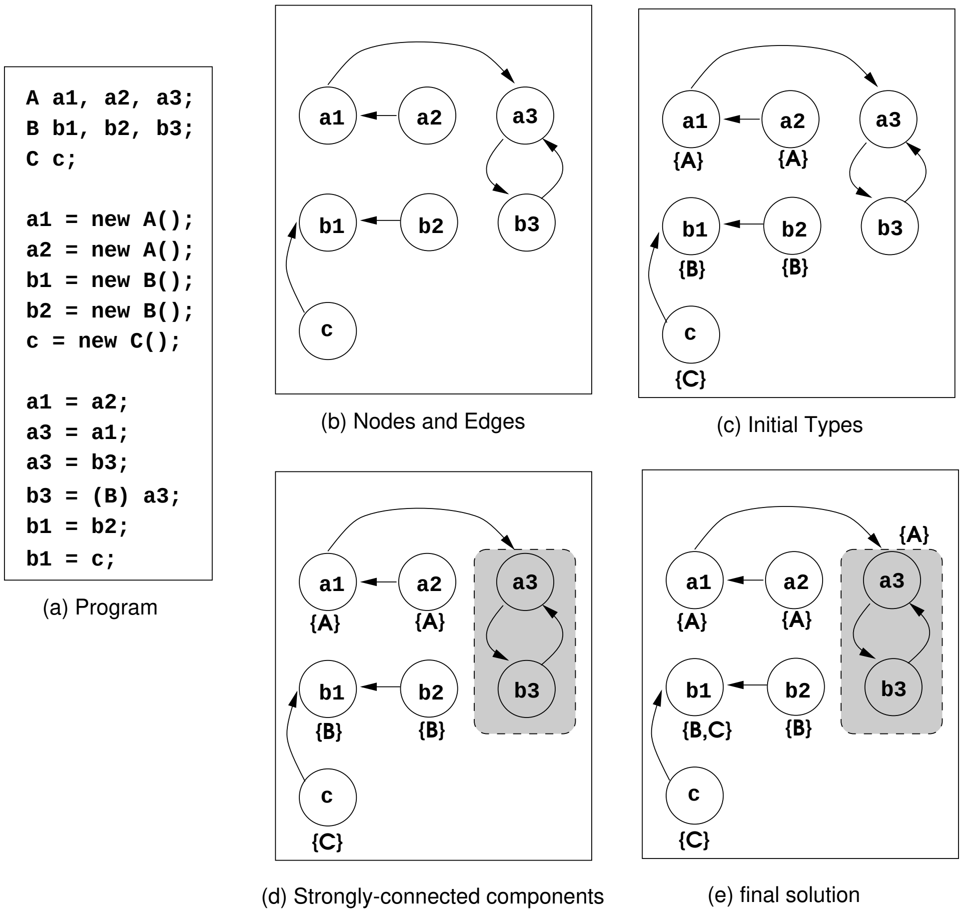
After initialization, we propagate types. This is accomplished in two phases. The first phase finds strongly-connected components in the type propagation graph. Each strongly-connected component is then collapsed into one supernode, with *ReachingTypes* of this collapsed node initialized to the union 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 topological manner, where a node is processed only after all of its predecessors have already been processed. Note that both the strongly-connected component detection and propagation on the DAG has complexity of $O(max(N, E))$ 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 `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



(a) Program (b) Nodes and Edges (c) Initial Types (d) Strongly-connected components (e) final solution

Figure 3: Example of a variable-type analysis

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 4 we present empirical results to evaluate these two analyses with respect to accuracy and the size of the graph problem to be solved.

3.3 Tradeoffs

We have designed our approach to work well with Java, particularly for large, object-oriented benchmarks. In order to keep our algorithm simple and efficient, yet effective, we have made several design decisions:

Avoiding solving the aliasing problem: We avoid having to solve the general aliasing problem by representing all instances of field f of class C as one variable name (as described in section 3.1.2). This keeps the analysis simple. Arrays do introduce one restricted sort of aliasing, and we handle this by introducing bidirectional edges for assignments that may involve arrays.

No killing based on casts or declared type: For each assignment statement $lhs = rhs$ or $lhs = (C)rhs$, we always propagate **all** types from the node for rhs to the node for lhs. One could imagine an algorithm that removed impossible types based on the declared type of rhs or the type given in the cast expression (C). However, this would lead to information being killed along some edges, and it would require either an iterative worklist solver or a more complex constraint solver (i.e. it would no longer be possible to collapse strongly connected components and solve simply in one pass over the graph).

It should be noted that we do filter out impossible types **after** we have the final solution. That is, for each variable we use the declared type of the variable and class hierarchy analysis to eliminate any reaching types that are not possible.

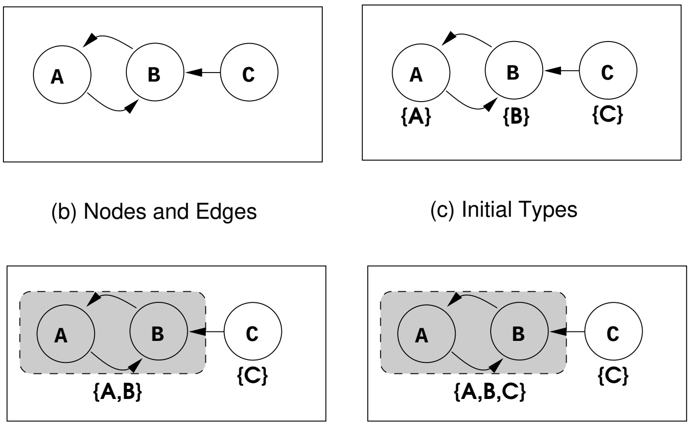
A pessimistic algorithm: Our algorithm is pessimistic in the sense that it adds edges for all method calls that are indicated by the conservative call graph. This means that we may include spurious edges, and types may propagate along those edges. The opposite approach would be to optimistically assume that method call of the form $o.m()$ could only call those methods

```
A a1, a2, a3;
B b1, b2, b3;
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

that correspond to the types currently reaching o at each step of the analysis. This set would increase as the analysis proceeds, and once again iteration would be required.

We can improve our pessimistic algorithm by giving it a better conservative call graph to start with. We experimented with two variations: (1) using the call graph generated using optimistic rapid type analysis as input to variable-type analysis; and (2) using the output of variable-type analysis as the input for a second iteration of variable-type analysis.

An interprocedural, whole program, approach: Another alternative to our approach is to propagate reaching type information intraprocedurally, and perform conservative approximations for the effect of method calls. By studying the more object-oriented benchmarks, we found that many of the reaching types were really being propagated interprocedurally, and so we designed our analysis to work on a type propagation graph that encodes the whole program, with all edges for method calls and returns.

Based on a typed 3-address representation: Our approach was implemented using Jimple, an intermediate representation that provides explicit names and types for all local variables. This allows our analysis to be very simple. Since all assignments are between explicit, named locations, we can represent the type propagation graph in an obvious fashion. The fact that local variables have a type is also useful for two reasons [22]. First, it makes the definition of declared-type analysis trivial. Second, it helps to improve the precision of the conservative call graph, since the declared type of a receiver may be tighter than the type in the signature encoded in the corresponding invokevirtual or invokeinterface bytecode instruction.

4. EXPERIMENTAL RESULTS

We have experimented with seven benchmarks, as outlined in Table 1. The four SPECjvm benchmarks include `raytrace` which is a graphics raytracer, `compress` which is a compression program based on a modified Lempel-Ziv method, `jack` which is a Java parser generator based on the Purdue Compiler Construction Tool Set (PCCTS), and `javac` which is the Java compiler from Sun's JDK 1.0.2. The other three benchmarks include `sablecc` which is a compiler front end generator written in Java[3], `soot` is an earlier version of our compiler framework[1], and `pizza` is the Pizza compiler [4]. For all benchmarks, the Java library used was from the Blackdown linux port, JDK1.1.7.

The statistics in Table 1 provide an insight into the nature of the benchmarks for which we have conducted experiments. In the column labeled **# Stmts**, we show the number of Jimple statements in the whole application (benchmark plus Java libraries accessed by the application), and the number of Jimple statements in only the benchmark (without libraries). In the column labeled **Hierarchy** we give the average and maximum depth of the inheritance hierarchy for the whole application and benchmark only. These numbers not only measure the extent of object orientedness of the whole application, but are also useful in discovering whether it is the benchmark itself that has been written in an object oriented manner, or if the Java libraries are the source of object orientedness. For example, we can see that `raytrace` and `compress` are not very object-oriented. The column labeled **Classes and Interfaces** gives the number of classes and interfaces that come from the library, the benchmark code only, and the overall total.

Table 2 gives a summary of the conservative call graph built for each benchmark using Class Hierarchy Analysis (CHA). We have measured the conservative call graph characteristics for the whole application (including the library) as well as the portions of the call graph related to the benchmark alone. Accordingly, Table 2 is divided into 2 distinct parts.

Benchmark		# Stmts		Hierarchy				Classes and Interfaces				
				avg. depth		max. depth		library		bench. only		whole app.
		whole	bench.	whole	bench.	whole	bench.					
lang.	name	app.	only	app.	only	app.	only	class	int.	class	int.	(total)
java	_205_raytrace	49239	5347	3.0	1.3	6	3	274	41	34	1	350
java	_201_compress	46619	2727	3.0	1.1	6	2	274	41	21	1	337
java	_228_jack	55107	11215	3.0	1.6	6	3	274	41	62	5	382
java	_213_javac	69585	25304	3.5	3.2	8	7	277	41	177	5	500
java	sablecc-w	68575	24621	3.2	2.3	6	5	276	41	298	13	628
java	soot-c	63506	33396	3.3	2.1	6	4	185	11	497	34	727
pizza	pizza compiler	73130	42805	3.0	1.7	6	5	187	11	207	11	416

Table 1: Benchmark Characteristics

Name	Whole Application							Benchmark Only						
		Call Sites			Edges				Call Sites			Edges		
	$\|N\|$	mono.	pot. poly.	total	mono.	pot. poly.	total	$\|N\|$	mono.	pot. poly.	total	mono.	pot. poly.	total
raytrace	1729	6582	377	6959	6576	2591	9167	207	2037	12	2049	2037	46	2083
compress	1583	5450	369	5819	5444	2556	8000	76	927	6	933	927	30	957
jack	1857	7191	779	7970	7185	3619	10804	337	2672	396	3068	2672	992	3664
javac	2821	10570	1276	11846	10564	13707	24271	1188	5933	848	6781	5933	10306	16239
sablecc	3737	11151	1332	12483	11140	24553	35693	1955	5920	889	6809	5920	20736	26656
soot	2828	11653	1738	13391	11653	25331	36984	2001	9070	1545	10615	9070	22620	31690
pizza	2660	13729	799	14528	13729	6024	19753	1756	11115	577	11692	11115	4069	15184

Table 2: Conservative Call Graph Characteristics

First consider the characteristics of the whole application, including libraries. Column 1 shows the number of methods that are in the call graph. Note that this number measures the number of methods that might be called starting at all possible entry points, based on CHA, and does not include methods that cannot be reached from a root in the conservative call graph. Column 2 shows the number of *monomorphic* call sites in methods in the call graph. The monomorphic sites include call sites for `invokestatic` and `invokespecial` instructions as well as call sites for `invokevirtual` and `invokeinterface` instructions that have been resolved to exactly one method by CHA. Column 3 shows the number of *potentially-polymorphic* sites i.e. `invokevirtual` and `invokeinterface` instructions that have more than 1 target after performing CHA. Column 4 shows the total number of call sites in the whole application. Column 5 shows the number of monomorphic edges (edges from monomorphic call sites), while column 6 shows the number of potentially-polymorphic edges (edges from potentially-polymorphic call sites). Column 7 shows the total number of edges in the whole application.

Now consider the second part of Table 2, which shows the characteristics of the benchmark only, not including any library methods. This part of the table includes all methods from the call graph that do not belong to the Java library, call sites inside these methods, and the edges attached to these call sites. These figures give a clear idea about the performance of CHA on the benchmark classes. For example, it is clear that there is hardly any scope for improvement of the benchmark portion of the call graph in benchmarks like `raytrace` or `compress`, whereas in benchmarks like `javac`, `soot`, or `sablecc` there are many unresolved call sites.

Table 3 summarizes the effect of applying a variety of techniques on the conservative call graph. In this table pRTA is pessimistic rapid type analysis, oRTA is optimistic rapid type analysis, DTA is declared-type analysis, and VTA is variable-type analysis. We also gave two combinations: oRTA+VTA is the combination of first using oRTA to build a pruned call graph, and then applying VTA; and VTA+VTA is the result of first using one application of VTA to get a pruned call graph, and then applying VTA on that pruned graph.

4.1 Reducing the size of the Conservative Call Graph

One use of our analyses is to reduce the size of the call graph. Eliminating methods from the call graph means that these methods do not need to be included in the application. This leads to smaller, compacted class files for applications, or smaller executables for compilers that translate class files for complete applications to native code. Further, reducing methods and call edges results in smaller call graphs which can make subsequent interprocedural analyses more efficient and more accurate. In Table 3, the columns labeled **Nodes Removed** and **Edges Removed** summarizes the number and percentage of nodes/edges removed for each analysis.

Rapid type analysis has been shown to be quite effective for C++ benchmarks [8], particularly for removing unused methods and call edges from the call graph for complete applications (including libraries). In this case the library code often contains many methods that are never called by a particular application. Our results confirm that rapid type analysis also does give a significant improvement for our Java

		Whole Application							Benchmark Only						
		Nodes Removed		Edges Removed		Callsites Resolved			Nodes Removed		Edges Removed		Callsites Resolved		
			(%tot.)		*(%tot.)*		*(%poly)*	*(%tot.)*		*(%tot.)*		*(%tot.)*		*(%poly)*	*(%tot.)*
`raytrace`	pRTA	808	(46%)	3585	(39%)	292	(77%)	(4.2%)	15	(7%)	46	(2%)	5	(41%)	(0.2%)
	oRTA	884	(51%)	4128	(45%)	300	(79%)	(4.3%)	15	(7%)	46	(2%)	5	(41%)	(0.2%)
	DTA	925	(53%)	4375	(47%)	304	(80%)	(4.4%)	18	(8%)	55	(2%)	5	(41%)	(0.2%)
	VTA	1026	(59%)	5200	(56%)	342	(90%)	(4.9%)	18	(8%)	68	(3%)	5	(41%)	(0.2%)
	oRTA+VTA	1031	(59%)	5242	(57%)	342	(90%)	(4.9%)	18	(8%)	68	(3%)	5	(41%)	(0.2%)
	VTA+VTA	1026	(59%)	5200	(56%)	342	(90%)	(4.9%)	18	(8%)	68	(3%)	5	(41%)	(0.2%)
`compress`	pRTA	814	(51%)	3664	(45%)	293	(79%)	(5.0%)	11	(14%)	40	(4%)	3	(50%)	(0.3%)
	oRTA	890	(56%)	4207	(52%)	301	(81%)	(5.2%)	11	(14%)	40	(4%)	3	(50%)	(0.3%)
	DTA	926	(58%)	4418	(55%)	303	(82%)	(5.2%)	16	(21%)	62	(6%)	4	(66%)	(0.4%)
	VTA	1033	(65%)	5214	(65%)	344	(93%)	(5.9%)	16	(21%)	70	(7%)	4	(66%)	(0.4%)
	oRTA+VTA	1039	(65%)	5256	(65%)	346	(93%)	(5.9%)	16	(21%)	70	(7%)	4	(66%)	(0.4%)
	VTA+VTA	1033	(65%)	5214	(65%)	344	(93%)	(5.9%)	16	(21%)	70	(7%)	4	(66%)	(0.4%)
`jack`	pRTA	820	(44%)	3763	(34%)	313	(40%)	(3.9%)	17	(5%)	121	(3%)	21	(5%)	(0.7%)
	oRTA	896	(48%)	4306	(39%)	321	(41%)	(4.0%)	17	(5%)	121	(3%)	21	(5%)	(0.7%)
	DTA	924	(50%)	4475	(41%)	323	(41%)	(4.1%)	20	(5%)	184	(5%)	21	(5%)	(0.7%)
	VTA	1027	(55%)	5719	(52%)	734	(94%)	(9.2%)	21	(6%)	565	(15%)	382	(96%)	(12.5%)
	oRTA+VTA	1033	(55%)	5769	(53%)	735	(94%)	(9.2%)	21	(6%)	565	(15%)	382	(96%)	(12.5%)
	VTA+VTA	1027	(55%)	5719	(52%)	734	(94%)	(9.2%)	21	(6%)	565	(15%)	382	(96%)	(12.5%)
`javac`	pRTA	823	(29%)	4516	(18%)	319	(25%)	(2.7%)	30	(2%)	713	(4%)	30	(3%)	(0.4%)
	oRTA	886	(31%)	5056	(20%)	327	(25%)	(2.8%)	30	(2%)	738	(4%)	30	(3%)	(0.4%)
	DTA	931	(33%)	5460	(22%)	337	(26%)	(2.8%)	33	(2%)	855	(5%)	30	(3%)	(0.4%)
	VTA	1001	(35%)	6639	(27%)	489	(38%)	(4.1%)	35	(2%)	1136	(6%)	135	(15%)	(2.0%)
	oRTA+VTA	1005	(35%)	6682	(27%)	489	(38%)	(4.1%)	35	(2%)	1144	(7%)	135	(15%)	(2.0%)
	VTA+VTA	1001	(35%)	6639	(27%)	489	(38%)	(4.1%)	35	(2%)	1136	(6%)	135	(15%)	(2.0%)
`sablecc`	pRTA	657	(17%)	4145	(11%)	407	(30%)	(3.3%)	42	(2%)	1077	(4%)	164	(18%)	(2.4%)
	oRTA	708	(18%)	4720	(13%)	421	(31%)	(3.4%)	49	(2%)	1220	(4%)	168	(18%)	(2.5%)
	DTA	773	(20%)	5670	(15%)	456	(34%)	(3.7%)	75	(3%)	1854	(6%)	192	(21%)	(2.8%)
	VTA	867	(23%)	10723	(30%)	635	(47%)	(5.1%)	91	(4%)	5943	(22%)	311	(34%)	(4.6%)
	oRTA+VTA	918	(24%)	11092	(31%)	663	(49%)	(5.3%)	91	(4%)	5951	(22%)	311	(34%)	(4.6%)
	VTA+VTA	1016	(27%)	11141	(31%)	680	(51%)	(5.4%)	92	(4%)	6005	(22%)	317	(35%)	(4.7%)
`soot`	pRTA	212	(7%)	2635	(7%)	137	(7%)	(1.0%)	60	(2%)	1362	(4%)	38	(2%)	(0.4%)
	oRTA	224	(7%)	2814	(7%)	143	(8%)	(1.1%)	60	(2%)	1362	(4%)	38	(2%)	(0.4%)
	DTA	282	(9%)	4061	(10%)	172	(9%)	(1.3%)	68	(3%)	2168	(6%)	60	(3%)	(0.6%)
	VTA	328	(11%)	7447	(20%)	657	(37%)	(4.9%)	89	(4%)	5027	(15%)	510	(33%)	(4.8%)
	oRTA+VTA	335	(11%)	7669	(20%)	662	(38%)	(4.9%)	90	(4%)	5076	(16%)	510	(33%)	(4.8%)
	VTA+VTA	348	(12%)	8380	(22%)	829	(47%)	(6.2%)	109	(5%)	5960	(18%)	682	(44%)	(6.4%)
`pizza`	pRTA	213	(8%)	2097	(10%)	123	(15%)	(0.8%)	17	(1%)	643	(4%)	3	(0.3%)	(0.0%)
	oRTA	213	(8%)	2097	(10%)	123	(15%)	(0.8%)	17	(1%)	643	(4%)	3	(0.3%)	(0.0%)
	DTA	233	(9%)	2566	(12%)	155	(19%)	(1.1%)	20	(1%)	830	(5%)	23	(3%)	(0.2%)
	VTA	270	(10%)	3462	(17%)	270	(32%)	(1.9%)	32	(1%)	1418	(9%)	109	(17%)	(0.9%)
	oRTA+VTA	270	(10%)	3462	(17%)	270	(32%)	(1.9%)	32	(1%)	1418	(9%)	109	(17%)	(0.9%)
	VTA+VTA	270	(10%)	3462	(17%)	270	(32%)	(1.9%)	32	(1%)	1418	(9%)	109	(17%)	(0.9%)

Table 3: Improvement of Call Graph over Conservative Call Graph

bytecode benchmarks.

When considering the whole application, the number of dead method nodes removed by pRTA varies between 7% of the total number of methods in the conservative call graph (`soot`) to about 51% (`compress`), and the number of edges removed by pRTA varies from 7% (`soot-c`) to 45% (`compress`). The optimistic version, oRTA, does perform better than pRTA on several benchmarks, giving a high of 56% nodes and 52% edges reduced (`compress`). However, when you consider the benchmark code only, we see that there is much less scope for improvement, and we see very little difference between oRTA and pRTA.

Both of our new analyses show additional benefit over oRTA, with VTA performing the best. When considering the whole application, VTA removes 10% (`pizza`) to 65% (`compress`) of the methods and 17% (`pizza`) to 65% (`compress`) of the edges. The most notable improvements due to VTA are for the large object-oriented benchmarks. For example, for `sablecc` oRTA removed 13% of the edges, whereas VTA removed 30%, and for `soot` oRTA removed 7% whereas VTA removed 20%.

Our combined analyses, oRTA+VTA and VTA+VTA, show small improvements over VTA, with the largest impact for the very object-oriented benchmarks, `sablecc` and `soot`.

These results show that VTA is quite useful for further reducing the size of the call graph, and in getting more compaction by removing additional methods. Note that for large benchmarks, where a greater proportion of the code is from the benchmark itself and not from the library, a much smaller percentage of methods can be removed by all analyses, although VTA does perform slightly better.

We also studied how many methods and edges could be removed when considering only the benchmark code and factoring out the library code. For methods, oRTA eliminates 1% to 14% and VTA eliminates 1% to 21%. For edges, oRTA eliminates 2% to 4% and VTA eliminates 3% to 22%. VTA works particularly well for `jack`, `sablecc`, `soot` and `pizza`. Overall, when we consider only the benchmark code, a smaller percentage of methods and edges can be eliminated, but the gap between RTA and VTA can be more significant, and the gap between pRTA and oRTA is less significant.

4.2 Resolving Virtual Calls

The second major measurement is how many potentially polymorphic call sites can be resolved to exactly one method. Below we present both static and dynamic results.

4.2.1 Static Results

Given the conservative call graph built by CHA, we have measured how many of the **remaining** potentially polymorphic sites can be resolved or eliminated by RTA, DTA and VTA. These results are found in Table 3, in the columns labelled **Callsites Resolved**. We say that a call site is resolved if it was potentially polymorphic after CHA analysis, but resolves to exactly one method after RTA/DTA/VTA.[4] We have presented the number of call sites resolved, as well as two percentages. The column labelled *%poly* gives the percentage with respect to the number of potentially polymorphic call sites in the conservative call graph, whereas the column labelled *%tot.* gives the percentage with respect to **all** call sites. A call site is eliminated if the method containing the call site is eliminated due to RTA/DTA/VTA.

First consider results of the whole benchmark. VTA performs significantly better than pRTA and oRTA, in some cases resolving more than twice as many call sites (i.e. `jack`, `soot` and `pizza`). Next, consider the behavior of methods that are part of the benchmark only (i.e. not part of the Java library). Here we see that the benchmarks `raytrace` and `compress` do not have any interesting behavior. Even though the analyses resolves a high percentage of the potentially polymorphic call sites (high *%poly*), these call sites were not very important in the overall picture (low *%tot.*). For the remaining five benchmarks we note that pRTA, oRTA and DTA do not perform very well, giving less than 5% (*%poly*) on four of the benchmarks. However, VTA can resolve a significant number of call sites with a high of 96%(*%poly*) or 12%(*%tot.*) for `jack`. Also, note that the gap between RTA and VTA is quite large on all five benchmarks. This seems to indicate that RTA and DTA are not good at resolving call sites in the benchmark part of the code, whereas VTA can resolve a significant number.

4.2.2 Dynamic Results

We have used profiling to estimate the possible run time-impact of the analyses. We instrumented the bytecode produced by our compiler to produce a summary of which methods were actually called at each invokevirtual and invokeinterface call, and to collect the execution frequency for each call site. We have concentrated on the run time behavior of call sites in the benchmark classes (excluding the Java libraries).[5] Figure 5 summarizes the percentage of dynamic calls that correspond to invokevirtual/invokeinterface call sites that can be resolved to one method (monomorphic call sites). For each benchmark, the first four bars correspond to call sites that could be resolved using CHA, RTA, DTA and VTA. The rightmost bar for each benchmark shows the result of our dynamic profile (i.e. how many call sites only resolved to one method during an actual execution). For example, in `jack`, almost 100% of all invokevirtual/invokeinterface calls are monomorphic at runtime, whereas in `javac` only about 90% are monomorphic at runtime. In general, we can see some interesting trends. First, for benchmarks that are not very object-oriented, like `raytrace` and `compress`, a simple method like CHA finds all monomorphic call sites. Second, it appears that RTA and DTA give very little or no improvement on all benchmarks, confirming our static measurements. However, our VTA analysis does give some improvement, with significant improvement on several of them. In some cases (`jack` and `pizza`), we observe that the number of call sites resolved by VTA is almost the same as the number of monomorphic calls obtained with the profile, and in these cases there is no need for any more sophisticated analyses.

For two benchmarks, `soot` and `javac`, we observe that while VTA did resolve substantially more call sites than any of the other analyses, it is not able to perform well enough to approach the results obtained in the profile. We studied the reasons for this gap on `soot` as the difference is greater for this benchmark, and as it is an analysis framework developed by us, we had the source code with which we were familiar. We illustrate the reason for VTA's inability to find all monomorphic calls with a typical example. The `soot` framework has an abstract class `AbstractValueBox` that is a container class that declares a field holding an object of class `Value`. `Value` is also an abstract class that is overridden by specific classes like `Local`, `InstanceField`, `InvokeExpr`. `AbstractValueBox` is extended by specific container classes like `LocalBox`, `InstanceFieldBox` and `InvokeExprBox`. These specific container classes do not declare any fields and the values that are held in these boxes are stored in the `Value` field of `AbstractValueBox`. Thus objects belonging to many classes that override `Value` reach the `Value` field declared in `AbstractValueBox`. The accessor method to get the `Value` stored in a box is defined only in `AbstractValueBox` and it returns the `Value` field. Thus whenever a specific kind of `Value` object is put into a box and retrieved, all the classes that reached the `Value` field are in the set of possible types (computed by VTA) for the object retrieved. We believe that this would be a problem for even more sophisticated analyses because the statements that put values in the boxes are often very far from statements taking the values out, and

[4] We have shown the number of potentially polymorphic call sites left by CHA analysis in columns 3 and 10 of Table 2.

[5] One common scenario is that one would want to perform compiler optimizations on the benchmark code alone, and leave the Java library classes unchanged (for example, when performing class file to class file optimization on user code). This was the main reasoning behind our decision to profile the benchmark classes only, as this would give us a good indication of the possible performance impact of optimizing the benchmark. Also we felt that it would be interesting to measure the difference in performance of the analyses on the benchmark classes dynamically, given that the static results indicate that our VTA analysis does substantially better than CHA and RTA in the benchmark code.

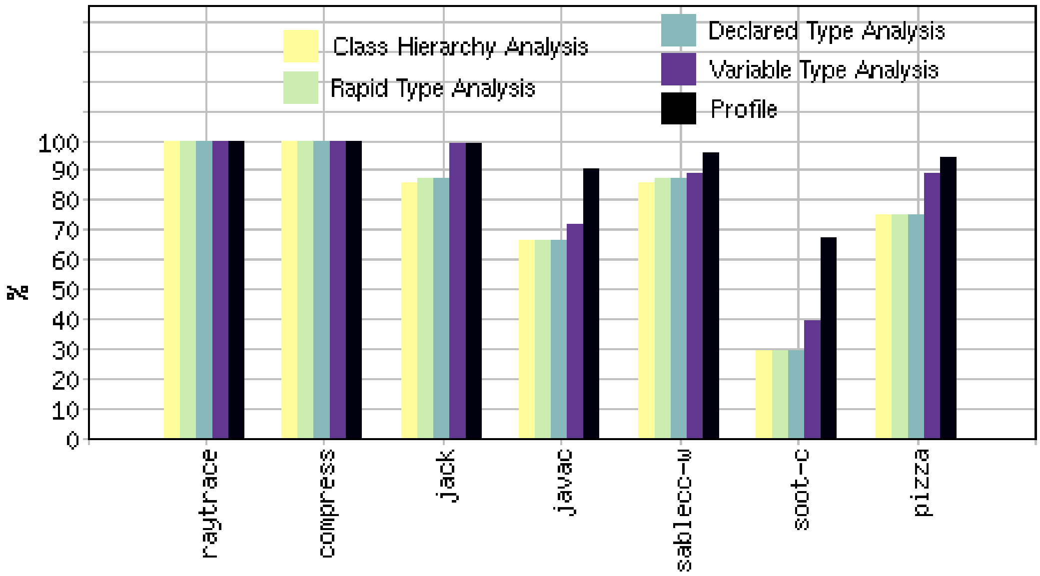


Figure 5: % Dynamic Monomorphic Calls (Benchmark Only)

it would be difficult to pair the definitions and uses up correctly.

Another explanation for the gap is the presence of several run time flags in this benchmark. For a particular option, there is usually an abstract class performing the basic functionality associated with the option, and it is extended by different classes that perform a specific function. Depending on the particular choice for the runtime flag one of the possible classes is instantiated. Thus, this is an example where the call site is monomorphic for a particular run of the program, but polymorphic over many different runs. This sort of monomorphism cannot be determined by a static analysis, but would be a good candidate for runtime optimizations.

The benchmarks `javac` and `soot` are examples where there exists a substantial number of polymorphic calls, even after an analysis like VTA has been used to devirtualize as many calls as possible. The gap between the result of VTA and the profile corresponds to calls that are monomorphic at runtime, but were not determined to be monomorphic by the static analysis. From Figure 5 we can see that for `javac` this gap is about 18% (90-72), and for `soot` this gap is about 28% (67-39). One can try to close this gap by applying more expensive static analyses, or one can use dynamic call optimization techniques, like branch target prediction [11], or inline caching [17]. For example, a hardware-based branch target buffer (BTB), like the 512-entry BTB of a Pentium III, reduces the overhead of calls that seldomly switch targets by storing the last target of every call site. The prediction hit rate of a large BTB is equal to the number of times that a target at an executed call site does not change, and therefore it gives an upper bound to the dynamic frequency of monomorphic calls. In Driesen and Hölzle's study on the direct cost of virtual function calls in C++ [19], a BTB predicts 75% of the calls on a suite of C++ programs. When all member functions are declared virtual (as in Java), the prediction rate climbs to 90%, which is similar to the Java profiles obtained in the current study. A static analysis technique like VTA can be used to remove all provably monomorphic call sites, after which a BTB optimizes dynamically monomorphic calls. A BTB can handle provably monomorphic calls, but since it is a limited resource close to the processor core, it is likely to remain limited in capacity, and therefore monomorphic call site removal by static analysis can increase the program workload that a particular processor can handle efficiently.

Figure 5 also demonstrates how many truly polymorphic calls exist in the benchmark, these correspond to 100 minus the height of the profile bar. For `javac` there are about 10% (100-90), and for `soot` there are about 33% (100-67). Dynamic techniques can also optimize truly polymorphic calls by using more sophisticated branch target prediction, that exploits correlations between the current call site and a previously executed call site, both polymorphic. For example, cascaded two-level prediction has been shown to reduce the number of unpredictable polymorphic calls from 25% (for a 256-entry BTB) to 6% (3-stages of 512-entry two-level), thereby optimizing 75% of the remaining truly polymorphic calls [20].

4.2.3 Performance Improvements

One might wonder if the increased precision of VTA is useful in further optimizations. Certainly reducing the size and complexity of the call graph will improve subsequent interprocedural analyses and helps to compact applications, but is it also useful for performance improvement of the benchmark? We don't expect it to make a large difference on any one optimization, but we do expect it to give small improvements on different optimizations. Currently we have implemented method inlining, where we use our framework to read class files, inline methods based on the call graph produced by CHA or the call graph after pruning using VTA, and then generate the inlined class files [34]. We executed the original, and inlined benchmark class files using the Blackdown linux JDK1.2, with the JIT turned on. Two of the benchmarks show better performance when the inlining is based on the call graph using VTA rather than CHA. For `soot` we observe 1% speedup when inlining is done using the CHA call graph but 3% speedup when the VTA call graph is used. For `javac` we see no improvement for inlining when based on the CHA call graph, but 2% speedup when using the VTA call graph. This leads us to believe that some of the extra call sites found by VTA could be important ones for inlining. We hope to see other benefits as more optimizations are added to our framework.

4.3 Measuring the Analysis

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, plus some execution numbers for our untuned implementation. In Table 4, we show our measurements.[6] Note that for DTA and VTA, the time required to obtain the solution is 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 DTA and VTA. In comparing DTA and VTA, we observe that VTA has about 4 times the number of nodes, and about 8 times the number of edges as in DTA. This gives a good indication about the relative costs of these 2 analyses. The last column of Table 4 gives the time, in seconds, for solving the constraint graph. The interesting point is not so much the absolute time [7], but the fact that the analysis scales well, and behaves linearly in practice. This also shows that VTA is indeed a constant factor (around 10) more expensive than DTA, and so the increased precision of VTA over DTA does come at a price.

5. RELATED WORK

There has been considerable 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. One of the classic algorithms is 0-CFA which has $O(n^3)$ complexity. Other context-insensitive approaches include Palsberg and Schwartzbach's algorithm [28], Hall and Kennedy's call graph construction algorithm for Fortran [23], and Lakhotia's algorithm [26] for building a call graph in languages with higher order functions. Other related work includes Shiver's k-CFA family of algorithms [32, 33] for selecting the target contour based on k enclosing calling contours at each call site, Agesen's Cartesian Product Algorithm [6], and Ryder's [31] call graph construction algorithm for Fortran 77. Plevyak and Chien's iterative algorithm [30] tries to improve a safe call graph to begin with and tries to refine it to the desired extent by creating new contours. Chatterjee et. al. give a method for finding relevant contexts for a subset of C++/Java [12]. Agesen [5] describes constraint-graph-based instantiations of k-CFA, and Plevyak's algorithm.

Our work has focused on a technique that can find a solution that does not require any iteration and scales linearly in the size of the program. Thus, previous work that focuses on the effectiveness of inexpensive analyses is more directly related to this paper. In this field, the goal is to find simple, inexpensive, yet effective analyses. The results of Dean et. al. [14] suggest that class hierarchy analysis is a good technique for resolving many method invocations for the Cecil language. Fernandez [21] implemented virtual call elimination and used an idea that is essentially Class Hierarchy Analysis (CHA). Aigner and Hölzle [7] find that type feedback and class hierarchy analysis are both effective at resolving method invocations in C++. Our work confirms that CHA does work well for Java bytecode, and we use CHA to get our original conservative call graph. However, our VTA methods can substantially improve the conservative call graph, removing 10% to 63% of the nodes, and 17% to 64% of the edges.

Bacon and Sweeney's work on fast static analysis of C++ virtual function calls [8] considers three relatively simple analysis techniques called: Unique Name, Class Hierarchy Analysis, and Rapid Type Analysis (RTA). They have dynamically measured the results for resolution of user virtual calls, and have given an estimate for the number of dead call sites. They concluded that rapid type analysis is extremely effective in resolving function calls, reducing code size, and is fast. Our results seem to confirm that rapid type analysis does also work well with Java when complete applications including library code are analyzed. However, we show that rapid type analysis does not perform well when considering the benchmark code only. Further, our results indicate that variable-type analysis gives better results for both cases, the complete application, and the benchmark only.

Diwan [18] describes results for simple and effective analysis of statically-typed object-oriented languages, and provides experimental results for Modula III programs. Their analysis is similar to ours in the sense that they also propagate types from allocation sites to uses. However, there are significant differences between their approach and our reaching-type analyses. First, we analyze Java bytecode, and so we have tailored our approach to the specifics of Java, including how to properly handle Java arrays. Further, we have experimented with a wide variety of benchmarks, including some large benchmarks that are very object oriented. Second, we believe that our approach is more efficient since we build a complete constraint graph, and solve it once. Their approach requires iterating a flow-sensitive intraprocedural phase since their interprocedural strategy re-analyzes methods when information about parameters or return values change due to the intraprocedural phase. Third, their interprocedural approach uses the declared type of object fields which can introduce imprecision, whereas we use the reaching types for fields.

[6] Note that the number of Jimple statements reported in Table 4 is less than the numbers reported earlier in Table 1 where we summarized the benchmark characteristics. In Table 1 we included all methods in classes that are referred to by the benchmark, whereas in Table 4 we include only those methods that appear in the conservative call graph.

[7] This implementation is built in Java using very high-level data structures based on collections, and it was run using a relatively slow Java interpreter (linux jdk1.1.7) on a 333Mhz pentium. Thus one can safely assume that a tuned implementation will run faster by a large constant factor.

Name	Jimple	Call Graph		Declared Type				Variable Type				Time	
				before SCC		after SCC		before SCC		after SCC		(seconds)	
	Stmts	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$	DTA	VTA
`raytrace`	27570	1729	9167	3540	3139	2989	1931	12496	18125	10700	13329	8	54
`compress`	24833	1583	8000	3235	2832	2741	1745	11010	15734	9471	11461	8	44
`jack`	33186	1857	10804	3828	3474	3284	2274	14293	21361	12320	16131	11	68
`javac`	47172	2821	24271	5872	6061	4741	3374	22220	54930	17019	26417	12	113
`sablecc`	49421	3737	35693	7722	8273	6104	3927	25482	75280	20298	43618	13	128
`soot`	43530	2828	36984	6333	6699	5178	3784	24190	68289	19620	43416	15	207
`pizza`	55468	2660	19753	7177	7445	6023	3856	28007	50242	17216	23390	11	102

Table 4: Size of Data Structures

More recently, DeFouw et. al. have presented a framework for expressing and experimenting with a variety of fast interprocedural class analyses for Cecil and Java[15, 16]. A key part of their framework is the ability to merge nodes in the constraint graph after they have been visited P times. This approach of merging after a threshold allows them to tune the complexity of the algorithm. They present eight instantiations of the framework, three of which are linear or near linear. Our DTA and VTA algorithms are **not** instantiations of their framework because the various tradeoffs we made (see section 3.3) to make our algorithm efficient are not parameters of their approach. First, our algorithm is pessimistic since it uses an initial conservative call graph (or the call graph generated by optimistic RTA) to insert edges into the constraint graph, whereas their schemes are optimistic, inserting edges into the call graph as object instantiations are found. Second, their algorithm merges a node with all of its successor nodes when it is visited P times (the key design point in their framework). In our algorithm we decide which nodes will be merged by computing the strongly connected components, and merging those together. After this compression step, our algorithm will only visit each node once. In their study they analyze a variety of Cecil benchmarks, but only three Java benchmarks. We have demonstrated our analysis on only Java benchmarks, but on a wider variety of those.

Tip and Palsberg have also been working to find scalable analyses that work well with Java [36]. Their motivation is very similar to ours, to find an analysis that makes some tradeoffs, gives better results than RTA, but scales better than traditional 0-CFA analysis. They present a spectrum of constraint-based techniques that focus on making analyses scalable by limiting the number of sets that must be approximated. In 0-CFA one set is associated with each expression, and in RTA one set is associated with the entire program. Their new analyses suggest intermediate points. For example, CTA uses a distinct set for each class, and XTA uses a distinct set for each method and each field. Although their analyses reduce the number of sets approximated, the underlying solver may still require iteration.

The difference between their approach and ours is in the way in which we enforce scalability. For VTA we use one set for each local variable, and one set for each field, thus giving us a finer-grain abstraction. Instead of coarsening the abstraction level, our design goal was to eliminate iteration in the analysis, and we made various tradeoffs to enforce this (as summarized in Section 3.3). For example: our analyses are conservative; we start with either the conservative call graph or the RTA call graph; and we do not kill based on cast information. We also tried a coarser version of VTA called DTA, where we approximated one set for each declared type, but we found that DTA was not nearly as effective as VTA. So, at least in our approach, the granularity of VTA appears to be necessary for good results.

Currently the experimental results of both the Tip/Palsberg approach and our approach both demonstrate that we achieve improvements over RTA. A head-to-head experimental comparison will only be possible when both approaches are implemented in the same framework, with the same assumptions, and run on the same set of benchmarks. It would be very interesting to perform this experiment. This would also allow us to determine if the approaches find the same sources of improvements, or if both techniques combined together gives even better results. If so, it would be possible to run both analyses, and then use the intersection of their results.

Another interesting area of future work is the combination of static and dynamic techniques. Ishizaki et. al. have studied a wide variety of devirtualization techniques for a Java JIT compiler [25]. Their study shows the promise of combining static techniques like type analysis and dynamic techniques.

Our work builds on the Soot framework under development at McGill. Harissa [27], Vortex [13] and JAX [35] are alternative implementation frameworks.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new technique that can be used to estimate the possible types of receivers for virtual method and interface calls in Java. Two variations of the technique were presented, variable-type analysis that uses the name of a receiver as its representative, and declared-type analysis which uses the declared type of a receiver as its representative. These two analyses, plus class 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 four analyses were applied to seven Java bytecode benchmarks.

Our methods work on complete applications, and so they require having all of the bytecode for the benchmark available. Although this is not useful for situations where classes can be dynamically loaded, we feel that compilation and optimization of complete applications is reasonable in many situations. There are certainly many large applications such as compilers, optimizers, editors and server-side applications that can be compiled in this fashion.

For each benchmark, class hierarchy analysis was used to build an initial conservative call graph. Measurements of these graphs confirm what others have noted, namely that class hierarchy analysis leads to a conservative call graph that is fairly sparse, with a majority of call sites resolving to a single method. However, there is scope for further improvement of these conservative graphs.

We applied rapid type analysis, variable-type analysis and declared-type analysis using the initial conservative call graph, and found that a significant number of edges and nodes could be removed. Variable-type analysis gave the best results removing 10% to 65% of the nodes and 17% to 65% of the edges from the conservative call graph. Further, variable-type analysis resolved 32% to 94% of the potentially polymorphic call sites (after CHA) to 1 method. All of these results are better than what was achieved by rapid type analysis. Our declared-type analysis did give some benefit, but not as significant as variable-type analysis.

In order to study the effect of the analyses on the benchmarks, we studied the dynamic behavior of the benchmark code only. In this case we found that neither rapid type analysis nor declared-type analysis had significant impact. However, variable-type analysis did show substantial improvement, in some cases approaching the best possible result. Thus, it seems that the added granularity of variable-type analysis over declared-type analysis is quite important, particularly when optimizing the benchmark code. In other cases variable-type analysis did find significantly more monomorphic call sites, but there was a substantial gap between the static result of the analysis and the dynamic profile. We presented several reasons for this gap, and we do not believe that a simple analysis will be able to close much of the remaining gap.

We observed that the extra call sites resolved by variable-type analysis account for a significant number of calls in the dynamic trace, and we demonstrated that inlining could make use of these extra call sites, giving performance improvement for two benchmarks.

Our techniques were meant to be simple, and we described the various tradeoffs we made to keep the algorithm simple and efficient. We have described our approach in detail, and it should be easy for others to add to their compilers, particularly if they already have CHA and/or RTA analysis. Based on our experimental results, we believe that a good overall strategy would be to use an optimistic RTA-style analysis to get the original pruned call graph. Then, if there are a significant number of polymorphic call sites remaining, our VTA analysis could further prune the graph, giving additional code size reduction, and better virtual call resolution.

We are currently working on tools for tree shaking, pointer analysis, and side-effect analysis based on the call graph produced by variable-type analysis.

7. ACKNOWLEDGEMENTS

This work was supported by NSERC and FCAR. The authors would like to thank Karel Driesen for his input on the connections between static and dynamic techniques. We also wish to express our thanks the OOPSLA reviewers and many others who have given positive suggestions on this and many previous drafts of our paper.

8. REFERENCES

[1] URL: http://www.sable.mcgill.ca/soot/.

[2] URL: http://www.transvirtual.com/kaffe.html.

[3] URL: http://www.sable.mcgill.ca/sablecc/.

[4] URL: http://wwwipd.ira.uka.de/~pizza/.

[5] O. Agesen. Constraint-based type inference and parametric polymorphism. In B. L. Charlier, editor, *SAS'94—Proceedings of the First International Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 78–100. Springer, 28–30 Sep. 1994.

[6] O. Agesen. The Cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In W. G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26, Åarhus, Denmark, 7–11 Aug. 1995. Springer.

[7] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In P. Cointe, editor, *ECOOP'96—Object-Oriented Programming, 10th European Conference*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166, Linz, Austria, 8–12 Jul. 1996. Springer.

[8] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 324–341, New York, Oct. 6–10 1996. ACM Press.

[9] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *21st Symposium on Principles of Programming Languages*, pages 397–408, Jan. 1994.

[10] C. Chambers, D. Grove, G. DeFouw, and J. Dean. Call graph construction in object-oriented languages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, volume 32, 10 of *ACM SIGPLAN Notices*, pages 108–124, New York, Oct. 5–9 1997. ACM Press.

[11] P. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. In *Proceedings of the International Symposium on Computer Architecture*, pages 274–283, June 1997.

[12] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 133–146, New York, NY, USA, 1999. ACM Press.

[13] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. VORTEX: An optimizing compiler for object-oriented languages. In *Proceedings OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31 of *ACM SIGPLAN Notices*, pages 83–100. ACM, Oct. 1996.

[14] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, 7–11 Aug. 1995. Springer.

[15] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. Technical Report TR-97-07-02, University of Washington, Department of Computer Science and Engineering, Jul. 1997.

[16] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, San Diego, California, 19–21 Jan. 1998.

[17] L. P. Deutsch. Efficient implementation of the Smalltalk-80 system. In *Conference record of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, pages 297–302, 1984.

[18] A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 292–305, New York, Oct. 6–10 1996. ACM Press.

[19] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C. In *Proceedings of OOPSLA-1996*, pages 306–323, October 1996.

[20] K. Driesen and U. Hölzle. Multi-stage cascaded prediction. In *EuroPar '99 Conference Proceedings, LNCS 1685*, pages 1312–1321, September 1999.

[21] M. F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 103–115, La Jolla, California, 18–21 Jun. 1995.

[22] E. M. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium 2000*, Lecture Notes in Computer Science, pages 199–219, Santa Barbara, June 2000.

[23] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, Sep. 1992.

[24] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 326–336, New York, NY, USA, Jun. 1994. ACM Press.

[25] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for Java Just-In-Time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-2000)*, 2000.

[26] A. Lakhotia. Constructing call multigraphs using dependence graphs. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284, Charleston, South Carolina, Jan. 10–13, 1993. ACM Press.

[27] G. Muller, B. Moura, F. Bellard, and C. 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.

[28] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 146–161, Nov. 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[29] H. Pande and B. Ryder. Static type determination for C++. In USENIX Association, editor, *Proceedings of the 1994 USENIX C++ Conference: April 11–14, 1994, Cambridge, MA*, pages 85–97, Berkeley, CA, USA, Apr. 1994. USENIX.

[30] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 29(10):324–324, Oct. 1994.

[31] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):216–226, May 1979.

[32] O. Shivers. Control-flow analysis in Scheme. *ACM SIGPLAN Notices*, 23(7):164–174, Jul. 1988. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation.

[33] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, May 1991.

[34] V. Sundaresan. Practical techniques for virtual call resolution in Java. Master's thesis, McGill University, Montreal, Canada, Sep. 1999.

[35] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *Proceedings of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, pages 292–305, Denver, CO), 1999. *SIGPLAN Notices* 34(10).

[36] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-2000)*, 2000.

[37] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In D. A. Watt, editor, *Compiler Construction, 9th International Conference*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, Germany, March 2000. Springer.