PRACTICAL TECHNIQUES FOR VIRTUAL CALL RESOLUTION IN JAVA

by
Vijay Sundaresan

School of Computer Science
McGill University, Montreal

June 1999

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Virtual method calls are a fundamental feature offered by Java, an object-oriented programming language. However, they are also a source of degradation of performance at run time and imprecision in interprocedural analyses. There are several well known, inexpensive analyses that have been developed for virtual call resolution. However, they have been observed to be effective in resolving method calls in library code, while not being very effective in the benchmark code excluding libraries.

We present a new flow insensitive and context insensitive analysis called *reaching type analysis* in this thesis. We present the analysis rules for two variations of this analysis, *variable type analysis* and a coarser grained version *declared type analysis*. Reaching type analysis is based on an analysis that builds a *type propagation graph* where nodes represent variables and edges represent the flow of types due to assignments.

We have implemented variable type analysis and declared type analysis, and two existing analyses, *class hierarchy analysis* and *rapid type analysis*, in the Soot framework and compared their relative success at building accurate call graphs for complete applications. We present static empirical results for call graph improvement for the whole application as well as for the benchmark code alone. We have also made dynamic measurements focusing on the benchmark code excluding libraries.

Method inlining is a compiler optimization in which the method call is replaced by the body of the method being called. Method inlining is very effective in improving performance of benchmarks that have many small methods and in which a large proportion of instructions executed are virtual calls. We have implemented method inlining (automatic and profile guided) at the Java bytecode level using the Soot framework. We demonstrate the effectiveness of our analyses and method inlining on a set of 15 benchmarks whose bytecodes were generated from Java, ML, Ada, Eiffel and Pizza compilers.

Résumé

Les appels de methodes virtuelles sont une des caractéristiques très utile et appréciée offerte par le langage de programmation Java. Cependant, ils sont non seulement la source d'une dégradation des performances lors de l'exécution mais aussi une source d'imprécision lors d'analyses inter-procédurales. Il existe un certain nombre d'analyses peu coûteuses développées pour la résolution d'appels de methodes virtuelles. Cependant, on observe qu'elles sont efficaces pour résoudre les appels qui se trouvent dans les librairies mais qu'elles le sont beaucoup moins pour ceux situés hors des librairies.

Dans cette thèse, on présente une nouvelle analyse indépendante du flux et du contexte appelée *reaching-type analysis*. Nous décrivons les règles d'analyse pour deux variantes de l'analyse, *variable type analysis* et *declared type analysis*, cette dernière étant une version dont la précision est moindre. Reaching-type analysis est basé sur une analyse qui construit un *graphe de propagation de types* dont les noeuds représentent des variables et les arcs des flux de type dûs aux assignations.

Dans le cadre du projet Soot, nous avons developpé variable type analysis et declared type analysis ainsi que deux autres analyses déjà présentes dans la littérature, *class hierarchy analysis* et *rapid type analysis* et nous avons comparé leur succès relatif à la construction du graphe d'appel pour des applications complètes, c'est-à-dire des applications analysées en tenant compte du code des librairies. Nous présentons des résultats expérimentaux sur l'amélioration de la précision du graphe d'appel d'une part pour des applications complètes et d'autre part pour ces mêmes applications sans tenir compte des librairies.

L'*inclusion de methode* est une optimisation de compilateur qui consite à substituer un appel de méthode par la méthode appelée. L'inclusion de methode est très efficace pour améliorer les performances de programmes composés de méthodes de petite taille et dans lesquels une grande partie des instructions exécutées sont des appels

virtuels. Nous avons réalisé l'inclusion de methode (automatique ou guidée par collecte d'information) au niveau du bytecode Java dans le cadre du projet Soot. Nous montrons son efficacité sur un ensemble de 15 programmes test dont les bytecodes ont été générés depuis des compilateurs Java, ML, Ada, Eiffel et Pizza.

Acknowledgments

Sincere thanks to:

My advisor Laurie Hendren, for her encouragement and support during my studies at McGill. She has been instrumental in sparking my interest in compiler research and I shall remain grateful to her for providing the much needed focus to my career. Her cheerful nature and enthusiasm have always rubbed off on the group, and it has been a joy interacting with her. Laurie's genuine concern and kind help in easing my financial situation have enabled me to concentrate my mind solely on research. I look up to her as my mentor, and will always have the highest amount of respect for her.

My colleagues at the Sable Research Group interacting with whom I have learnt so much about Java and compilers. Etienne Gagnon, whose views I have tried to seek out during the course of my thesis. His in depth knowledge of the Java Virtual Machine specifications, and his advice on other career related matters have been invaluable. Raja Vallee-Rai, whose dedication and attention to detail ensured that the Soot framework was both easy to use and understand. I have learnt about the importance of object oriented software design by observing his programming style. Chrislain Razafimahefa, for being a good friend, and helping me along during my earlier days at McGill when I was still learning Java. The many discussions that I had with him regarding our analyses and optimizations, our constant interaction during the coding process, and the soccer and dinner sessions are all memories I will cherish. Laleh Tajrobehkar for providing many a light moment, and Patrick Lam for helping with problems cheerfully on several occasions.

The friends with whom I had many good times during my stay in Montreal, and who helped me adjust to life here. Prasad Kakulavarapu for being a trusted and ever helpful friend. The long and heated discussions on all manner of topics were always enjoyable. Tallman Nkgau, Ian Garton, Mike Soss and Charles Abety for all the get togethers and friendly banter that were a source of relaxation. Rakesh Ghiya for introducing me to Laurie and for the encouragement when I was still finding my feet at McGill.

The staff in the administrative office: Vicki Kierl, Franca Cianci and Lise Minogue for being very helpful throughout my studies at McGill.

My mother who is the dearest person in my life for her constant affection and support at all times. My father who has been my guiding spirit; I miss him deeply and wish he was alive to see me now as I finish studies and start working. My brothers Subash and Prakash, for helping me in more ways than I can ever hope to repay. My uncle Giri, who has always been a pillar of support.

Dedicated to my late father Mahalingam Sundaresan

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Java is a general-purpose concurrent class-based object-oriented programming language, that allows application developers to write a program once and then be able to run it everywhere on the Internet. The extensive functionality offered by the Java library API, the platform independence of Java bytecode, and the applicability of objected oriented software design in large projects have all contributed to the growth of Java.

However the features that have contributed to the growth of Java come with a certain performance penalty that makes Java suffer in comparison to other popular languages like C/C++. Platform independence is achieved through an interpreter that interprets the Java bytecodes before executing them. But interpreting the bytecodes at execution time is much slower than executing native code that has been compiled using a traditional compiler, as the overhead of the interpreter's execution must be paid at run time. Just In Time (JIT) compilers are becoming increasingly popular for this reason though they are not yet available for all of the common platforms in use. The highly object oriented features that make software maintenance and debugging easier for Java applications also mean that the run time penalty must be paid in the form of virtual method calls and type inclusion checks that are quite expensive as compared to other instructions. Thus it is clear that although the design of the source language itself and the bytecode are quite clean, there is a significant amount of work that needs to be done before Java can exhibit the same run time performance as its competing languages.

The problem of improving performance can be solved in two ways. It is possible to perform static analyses on Java bytecode and apply traditional program transformations like method inlining, common subexpression elimination, and loop invariant removal. This approach requires no interaction with the Java Virtual Machine that is being used to execute the bytecodes. Another possibility is to use the results from static analyses to annotate the class file that is being executed. This requires a Java Virtual Machine that would be able to understand the annotations that are part of the class file, and perform run time optimizations like register allocation, and eliminating array bounds checks as it is executing the bytecodes. We have adopted the first approach in our attempt to improve performance, and we present the analyses, optimizations and the benefits that we observed. The focus of this thesis is on improving performance of Java bytecode by trying to reduce the overhead associated with virtual method calls. The rest of this chapter is organized as follows. Section 1.1 introduces the problem we are addressing. Section 1.2 describes the framework that we have used to perform our analyses/optimization and Section 1.3 discusses the related work in this area. In Section 1.4, we briefly describe the contributions of this thesis in addressing this problem, and Section 1.5 outlines the organization of this thesis.

1.1 Virtual Method Calls in Java

In this section we discuss some of the issues pertaining to virtual method calls in Java and how they impact performance. Java is an object oriented language and applications written in Java typically contain many classes, methods, and fields. Every class in Java (except the cosmic superclass `java.lang.Object`) must extend some unique superclass. Subclasses inherit all the features of parent classes and might also contain additional methods/fields that are used to perform specific functions of the subclass. A form of multiple inheritance is achieved through the use of interfaces that classes are allowed to implement. Thus every class in Java is part of a inheritance hierarchy with `java.lang.Object` at the root of the hierarchy. This follows the standard object oriented design model in which classes that are more general are near the top of the hierarchy and subclasses have more specialized functionality. Subclasses are allowed to declare methods whose implementation overrides that of a method in a superclass. The method invoked at run time depends on the actual class of the receiver object and not on the Java declared type of the variable referring to the object. Call sites for which the compiler producing bytecode from the source language cannot fix the target of the call statically are the source of *virtual* method calls.

```
class A {
 public static void main ( String[] args ) {
   for ( int i=0;i<2;i++ )
   {
     A a = null;
     if ( i == 0 )
       a = new A();
     else
       a = new B();
     a.m(); // CS  ( polymorphic call site )
   }
 }

 public void m() { System.out.println ( ``In A'' ); }

}

class B extends A {
 public void m() { System.out.println ( ``In B'' ); }
}
```

Figure 1.1: An example of a polymorphic call site

Consider the call site `CS` shown in Figure 1.1. The class `A` is extended by class `B` and both classes have different implementations of the method `m()`. It is possible that a variable declared to be of type `A` can point to an object the run time type of which is either `A` or `B`. Thus the variable `a` shown in the example refers to an object of class `A` in the first iteration while it refers to an object of class `B` in subsequent iterations. Since the method invoked depends only on the actual type of the object referred to by the variable, the method `m()` declared in class `A` is invoked in the first iteration, and the method `m()` declared in class `B` is invoked in successive iterations.

Call sites that can invoke more than one target method at run time depending on the class of the receiver are termed *polymorphic* call sites. Call sites that can only invoke a single target method at run time are termed *monomorphic* call sites. If the target method invoked from a particular call site can be fixed statically, then the call site is said to have been *resolved*.

The method to be invoked at run time is determined by the Java Virtual Machine by examining the virtual method tables of the class of the receiver object. There are entries in the virtual method table of a class for each non-private method that might be invoked on an object belonging to that class. There are two possible bytecodes that might be generated for virtual calls. The first is the `invokevirtual` bytecode instruction which is generated when the declared type of the receiver in the source code was a class. If the declared type of the receiver was an interface then the bytecode that would be generated is the `invokeinterface` bytecode instruction. Both kinds of virtual calls are expensive but the calls made using the `interfaceinvoke` bytecode are more so because of reasons that we explain now.

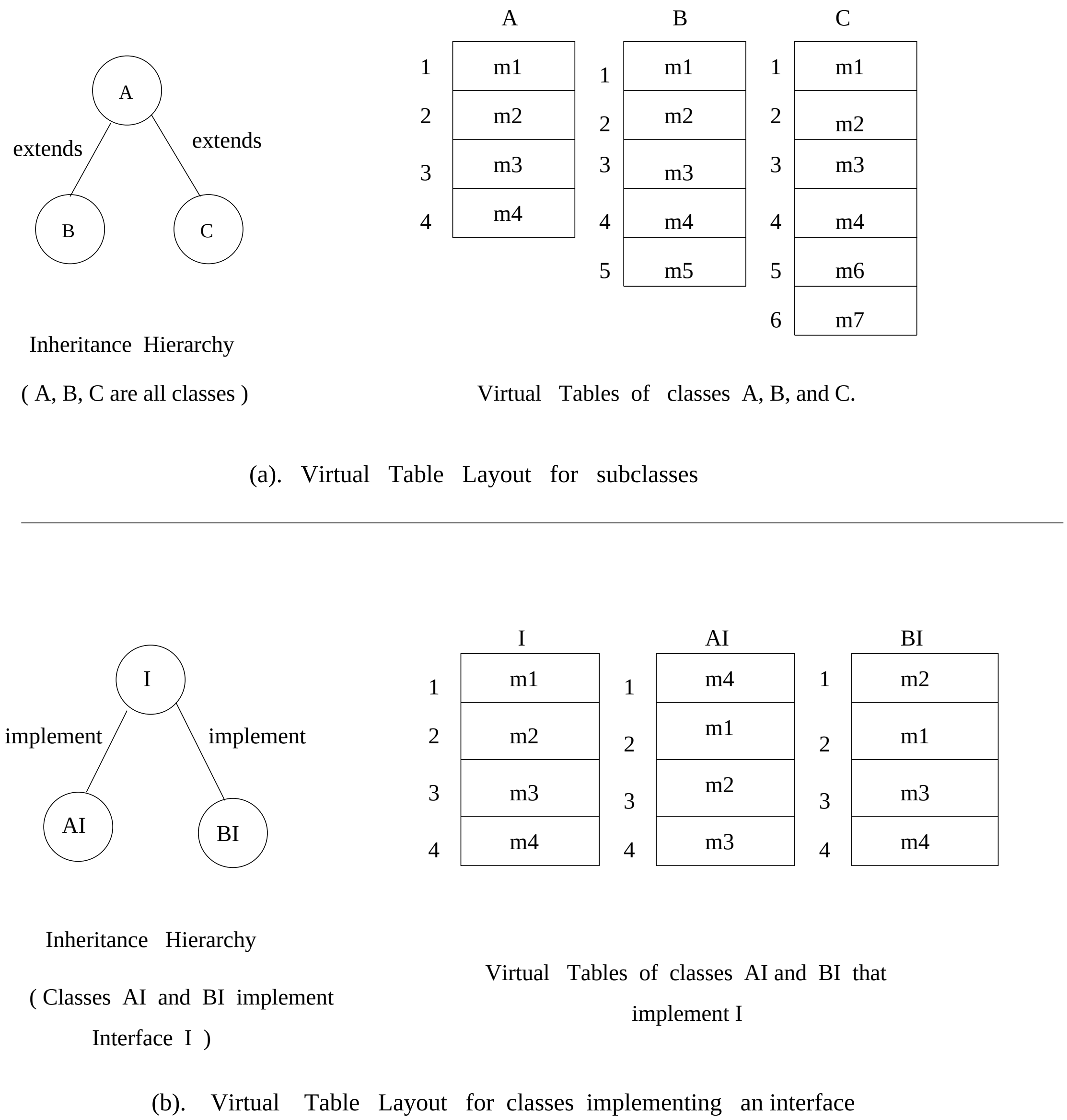


Figure 1.2: Virtual Table layout for subclasses and interfaces

```
class A {
 public static void main ( String[] args ) {
   for ( int i=0;i<2;i++ )
   {
     A a = null;
     if ( i == 0 )
       a = new B();
     else
       a = new C();
     a.m1(); // Index into Virtual Tables can be reused after first execution
   }
 }

 public void m1() { System.out.println ( ‘‘In A’’ ); }
}

class B extends A {
 public void m1() { System.out.println ( ‘‘In B’’ ); }
}

class C extends A {
 public void m1() { System.out.println ( ‘‘In C’’ ); }
}
```

Figure 1.3: Example of JVM only needing to compute the index once for invokevirtuals

In order to understand the reason for the virtual calls being expensive, it is necessary to understand the actions taken by the Java Virtual Machine when it executes a virtual method call. We first explain the actions to be performed for executing the `invokevirtual` bytecode instruction. In this case, the virtual method table of the class referred to in the `invokevirtual` instruction is examined and the index of the method matching the method signature is obtained. For subclasses it is guaranteed by the Java Virtual Machine Specification that the index of a particular method in the virtual method table is the same in the subclass as it was in the parent class. In Figure1.2(a), we observe that classes `A`, `B` and `C` have different number of methods, but the methods that they have in common (`m1, m2, m3, m4`) have the same offsets in the virtual tables of all three classes. Thus once the index of the method in the class referred to in the signature has been found, the same index can be used to access the entry in the virtual method table of the class of the receiver object. The entry in the virtual method table would tell the Java Virtual Machine the exact method that is to be invoked. Note that the index needs to be computed only the first time

```
class AI implements I {
 public static void main ( String[] args ) {
   for ( int i=0;i<2;i++ )
   {
     I a = null;
     if ( i == 0 )
       a = new AI();
     else
       a = new BI();
     a.m1(); // Index into Virtual Tables cannot be reused
   }
 }

 public void m1() { System.out.println ( ‘‘In A’’ ); }
}

class BI implements I {
 public void m1() { System.out.println ( ‘‘In B’’ ); }
}

interface I {
 public void m1();
}
```

Figure 1.4: Example of JVM needing to compute the index every time for invokeinterfaces

the `invokevirtual` instruction is executed, and on all subsequent executions of the instruction the index computed the first time can be used, as it guaranteed to be the same no matter what the class the receiver object (it must be the class referred to in the `invokevirtual` instruction or a subclass). Refer to the example in Figure 1.3 in which a call site is shown for which the index into the virtual tables needs to be calculated only the first time it is executed.

In the case of an `invokeinterface` bytecode instruction, the actions to be taken are slightly different. There is no relationship between the index corresponding to a particular method entry in the virtual method table of an interface, and the index of the same method entry in the virtual method table of any class that implements that interface. Thus, in Figure 1.2(b), we observe that even though classes `AI` and `BI` both implement the interface `I` and have four methods each (names identical for methods in both classes), there is no correlation between the offsets into the virtual tables for the methods in the two classes. Note the difference in this case as compared to the case shown in Figure 1.2(a). Thus the virtual method table of the class of the receiver object must be searched *each* time the `invokeinterface` instruction is executed. The index computed the previous time it was executed may not be the same as the index of the method in the virtual method table of the receiver object this time as the class of the receiver might be different on different executions of the `invokeinterface` instruction. The class of the receiver on different executions must of course be some class that implements the interface but the index of the method entries in the virtual method table of the class need not be the same. So the `invokeinterface` bytecode instruction is expensive as compared to `invokevirtual` bytecode instruction, and even more so than a static method call that requires no method lookup. An example is shown in Figure1.4 where the index into the virtual method table must be recalculated each time the call site is executed.

Hence, it should be clear that virtual method calls are expensive at run time and replacing them wherever possible them might improve performance. Possible ways of avoiding the overhead associated with virtual method calls are either eliminating them completely or by replacing them by less expensive instructions. Eliminating a method call completely is possible if the target of the method call is known statically and we are allowed to inline the code of the called method into the caller and remove the call instruction. There are other bytecode instructions (namely `invokestatic` and `invokespecial`) for method invocations at call sites where the target method is known statically. These are less expensive to execute and can be used instead of the virtual call instructions if the virtual call has been resolved. Note that if the conversion is to the `invokestatic` bytecode, then a new static method might have to be created and added to the class. This static method would be similar in functionality to the method being invoked by the virtual call instruction, but would differ from it in that it would have one extra explicit parameter corresponding to the implicit `this` parameter.

1.2 The Soot Framework

We have used the *Soot* framework [1] to perform our analyses and optimizations. Soot is a framework for analyzing, optimizing and annotating Java bytecode. More concretely Soot offers three alternative representations for Java bytecode that are designed to be easier to work with as compared to using the bytecode directly.

The Baf representation: The first representation is *Baf* which is a compact and simpler bytecode representation that is useful when it is necessary to deal with bytecode as stack code. The Baf intermediate representation hides some of the encoding issues in bytecode, such as the constant pool and multiple variants of virtually the same instruction. Baf is currently in use for performing peephole transformations and for a final re-ordering phase. Refer to Figure 1.5(b) for a simple example of Baf code.

The Jimple representation: The second representation is *Jimple* which is a compact three-address code representation of Java bytecode that is unstructured. It is much simpler to develop analyses and optimizations on the Jimple representation than on Java bytecode for the following reasons :

- Resembles simple Java: instructions are in three-address code form.
- Typed: Like Java, Jimple's local variables are typed (the types are inferred from the bytecode).

In Figure 1.5(c) we show an example of Jimple code that would be produced for the Java code shown in Figure 1.5(a).

The Grimp representation: The third representation is *Grimp* (aggregated Jimple) which is similar to Jimple except that it represents statements as trees. This is extremely useful where Jimple's fractured nature is inappropriate. Grimp is used in the framework for decompilation and generation of bytecode. Figure 1.5(d) shows an example of Grimp code. Note that the Grimp code is very similar to the original Java code.

The fact that all three intermediate representations in Soot are constructed directly from the Java bytecode in the class files, and not from the high level Java programs allows us to analyze Java bytecode that has been produced by any compiler, optimizer, or other tool. There are front ends available for languages such as Ada, ML, Eiffel and Pizza that produce Java bytecode.

We analyze complete applications, so our implementation starts by reading all the classes required by a particular application, starting with the main class and recursively loading all the classes used in each new class. As each class is read it is converted into the Jimple intermediate representation. After this conversion each class is stored in an instance of a `SootClass` which contains information like its

```
if ( x + y != z )
return;
else
System.out.println ( ‘‘foo’’ );
```

(a). Original Java code

```
t = x + y;
if t == z goto label0;
return;

label0 :
ref = System.out;
ref.println ( ‘‘foo’’ );
```

(c). Jimple code

```
iload x
iload y
iadd
iload z
icmpge label0
return

label0:
getfield System.out
push foo
invokevirtual println
```

(b). Baf code

```
if ( x + y == z ) goto label0;
return;

label0 :
System.out.println ( ‘‘foo’’ );
```

(d). Grimp code

Figure 1.5: Different representations offered by Soot for Java code

name, superclass, list of interfaces that it implements, and a collection of `SootFields` and `SootMethods`. Each `SootMethod` in turn contains information such as its name, modifiers, locals, parameters, and a list of Jimple three-address code instructions. At the beginning of the Jimple instructions list for each method there are special *identity statements* that provide explicit assignments from the parameters (including the implicit `this` parameter) to locals within the `SootMethod`.

The Soot framework includes some basic intraprocedural optimizations like *copy propagation*, *constant propagation*, and *dead code elimination* that are very useful in cleaning up the Jimple code produced from a naive bytecode to Jimple translation. Since the operand stack that is present at the stack code (bytecode) level is completely eliminated in the Jimple representation, the stack locations must be represented in Jimple as local variables. Types for locals are inferred [26] using the explicit references to types present in method and field signatures, and instantiations. To avoid having too many locals in the Jimple representation, a local packing pass is made over the code that tries to pack as many locals as possible into one local.

In terms of our analyses, the fact that in the Jimple representation, each statement has a relatively simple format means that the rules are not as complex as they would have been if the statements could contain large expressions. Also there is a fixed set of different kinds of statements in Jimple, and the set of analysis rules can be assumed to be complete once rules have been specified for each statement. Further all the operands in Jimple are either variable references or constants. Since there is a declared type (that is inferred) for each local and constant, our analyses can use this information in a straightforward manner.

Figure 1.6 provides an overview of all the components of Soot and some of the applications that Soot is being used for.

1.3 Related Work

The problem of virtual call resolution has been studied before for other languages. In this section we discuss some of the work we found that was relevant to our research. We first discuss work related to call graph improvement and receiver class prediction, and later in this section we discuss the research into method inlining.

The study of analyses to improve the call graph and elimination of virtual calls is discussed in some detail in the work by Grove et al [15]. 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 which is of particular interest to us. The results that they obtained give an indication about the time and space 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. They also formally introduce some interesting properties of a *call graph lattice domain*, and discuss the conditions when a call graph is sound. The ideas set forth in this discussion were used by us in formulating our analyses as call graph improvement techniques. This ensures that our call graph is always in some part of the call graph lattice where it is guaranteed to be sound (conservatively correct). Alternative approaches are possible in which the call graph might be unsound (incorrect) at certain points during the analyses, but we have not chosen those techniques in our implementation.

We have focused on the study of fast analyses for call graph improvement in our

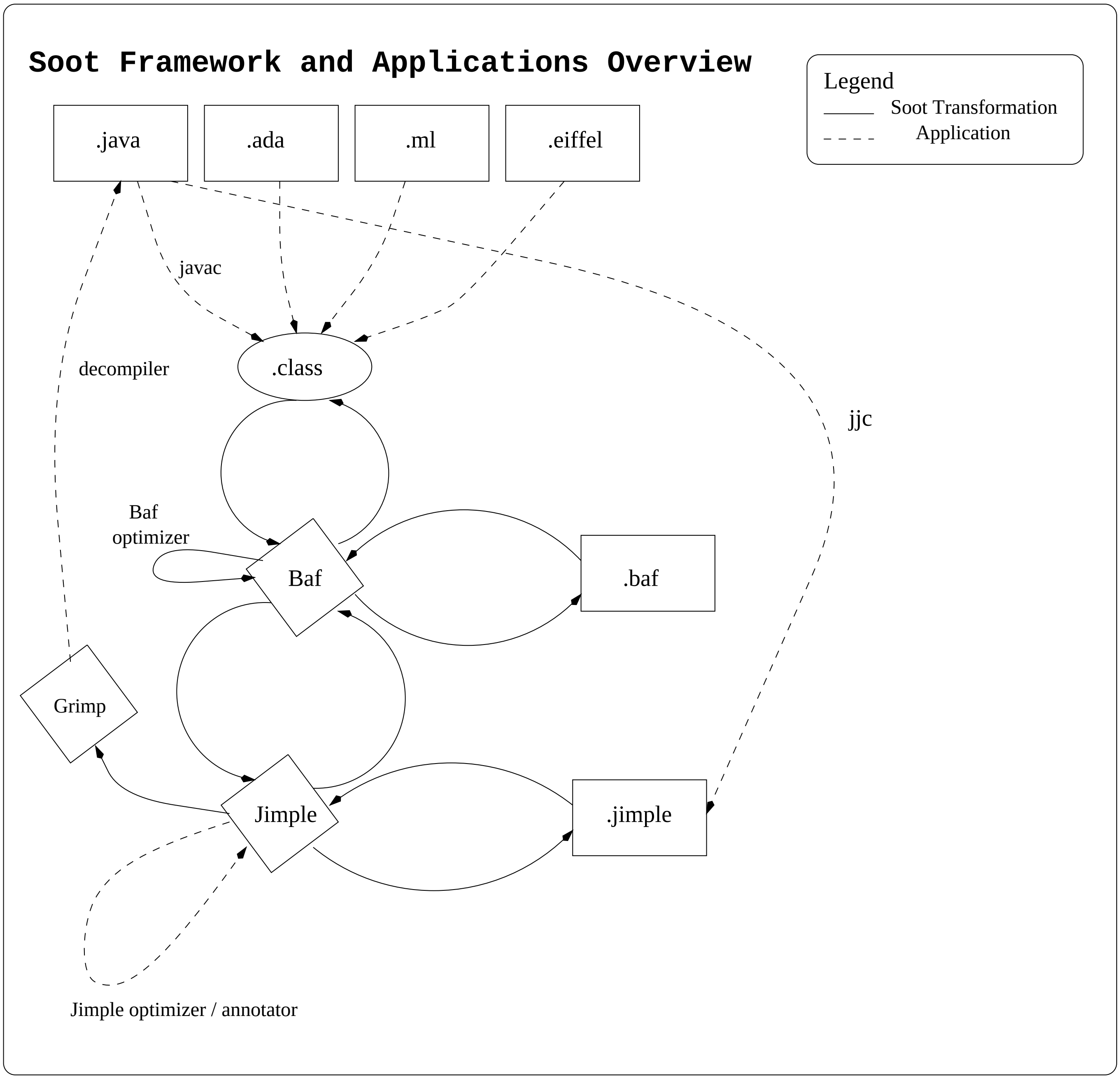


Figure 1.6: The Soot Framework

work while some of their analyses for call graph construction are context sensitive or flow sensitive and are consequently more expensive. Their experiments demonstrated that scalability becomes a crucial issue when context sensitive algorithms for call graph improvement are applied to medium to large sized programs. Grove et al. conclude that a scalable and effective call graph construction algorithm for programs that make extensive use of polymorphism and dynamic dispatching is still an open problem. Our work focuses on scalable call graph construction algorithms 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 discovered that for both the languages they considered, the additional precision of the call graphs constructed by the interprocedurally flow-sensitive algorithms had a significant impact on the effectiveness of the client interprocedural analyses and resulting optimizations. Their speedups are about 5 to 10 percent on average for Java benchmarks (achieved through a combination of aggressive intraprocedural and limited interprocedural optimizations). They also managed to reduce the size of their executables for Java benchmarks by about 10 to 20 percent performing dead member elimination.

Diwan [24] 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. Their *intraprocedural type propagation* improves the results obtained by type (class) hierarchy analysis by using data flow analysis to propagate types from type events to method invocations within a procedure. Type events create or change type information (e.g. `new()` statements, or assignment statements). Since the analysis is flow sensitive, type information is merged at control flow points. Type propagation only propagates types to scalars, and assumes the conservative worst case (the declared type) for the allocated types of record and object fields, and array and pointer references. They also present an interprocedural version of the type propagation analysis which uses a conservative call graph built by their less precise analysis. The algorithm operates by maintaining a work list of procedures that need to be analyzed. A procedure needs to be reanalyzed if new information becomes available about its parameters or about the return value of one of its callees. Some call graph edges may be removed if the analysis refines the type of a method receiver. The interprocedural strategy used by them is context insensitive.

Their other analysis called *aggregate analysis* is aimed towards finding monomorphic use of a general data structure. It circumvents the difficulty of analyzing records and heap allocated objects by merging all instances of an object or record type. As an example, the statements

```
v: T;
v.f := <rhs>
```

propagate the types of `<rhs>` to the field `f` of all possible types of `v`. The possible types of `v` can be determined by another analysis (such as type propagation) or may be conservatively approximated as `T` and its subtypes.

These analyses developed by them for Modula III bear the closest resemblance to our reaching type analysis. However, there are significant differences between their approach and our reaching-type analyses. First, we analyze Java bytecode, and experimented with a wide variety of benchmarks, including some very object oriented ones. 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.

Their results showed that they could resolve almost 92% of the method invocation sites on average, and in some cases improve the performance of these programs by up to 19%. Their cause analysis approach aimed at discovering the reasons for imprecision of their analyses. It found that polymorphism, insufficiently powerful aggregate analysis, and context insensitivity were the main factors that prevented them from resolving all the call sites at compile time.

Also closely related to our work is the work by Bacon and Sweeney [12] on fast static analysis of C++ virtual function calls. Their study considers three relatively simple analysis techniques *unique name*, *class hierarchy analysis* (CHA), and *rapid type analysis* (RTA).

They observed that in some cases there is only one implementation of a particular virtual function anywhere in the program, and this can be detected by comparing the mangled names of C++ functions in the object files. When a function has a unique name, the virtual call is replaced by a direct call. This is called unique name analysis and has the advantage that it does not require access to source code and can optimize virtual calls in library code; however working with object code means that it is not possible to implement optimizations like inlining.

Class hierarchy analysis (CHA) uses the statically declared type of an object with the class hierarchy of the program to determine the set of possible targets of virtual calls. We have implemented class hierarchy analysis (CHA) in our framework as the simplest analysis for building the call graph.

The third analysis studied in the paper is rapid type analysis (RTA). Rapid type analysis starts with a call graph generated by performing class hierarchy analysis, and uses information about instantiated types to further reduce the set of executable virtual functions. The analyses that we have considered are slightly more expensive and complex, but we have also implemented RTA and studied the results for Java applications. Their results for RTA show that it is actually most effective when analyzing library code as it is more likely that there would be classes that are not instantiated in the library. They also measure the further potential for improvement by considering the dynamic information for calls in a program trace.

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 overall 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.

Calder and Grunwald [13] examined characteristics of C++ programs and observed that at a given call site the class of the receiver tends to belong to a set containing a small number of classes. Thus they concluded that profile-guided receiver class prediction would be beneficial though they did not have an implementation in a compiler to prove this hypothesis. Holzle and Ungar [29] describe transformations to convert method invocations to direct calls (profile-guided) for Self programs.

Aigner and Holzle [9] in their work in evaluating techniques for resolving method invocations in C++ found that type feedback and type hierarchy analysis are both effective at resolving method invocations in C++.

Plevyak and Chien's iterative algorithm [33] 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 III, and Cecil. Some of the algorithms that are context insensitive are 0-CFA[36, 37], Palsberg and Schwartzbach's algorithm [32], Hall and Kennedy's call graph construction algorithm for Fortran [27], and Lakhotia's algorithm [30] for building a call graph in languages with higher order functions. Other related work includes Shiver's k-CFA family of algorithms [36, 37] for selecting the target contour based on k-enclosing calling contours at each call site, Agesen's Cartesian Product Algorithm (CPA) [8], and Ryder's [35] call graph construction algorithm for Fortran 77. Agesen[7] describes constraint-graph-based instantiations of k-CFA, and Plevyak's algorithm.

We have not been able to find any published work done on method inlining in Java, which takes as input classfiles and produces optimized classfiles. There has been some research into the benefits of method inlining for other programming languages; in general inlining is traditionally considered to be a source code level optimization.

Carini [14] suggested some useful heuristics to perform automatic inlining in Fortran and their work shows that they could almost attain the performance levels of profile guided inlining in most cases. Their improvements in performance are about 2% on average and about 6% in the best case. Their heuristics are based on two cost functions. The first cost function attempts to accurately estimate the cost of inlining a certain function. Their function heuristics take into account the size of a function, and the the number of loops, number of call sites and the number of I/O calls within the function. While these are the parameters in calculating the cost associated with inlining a certain method, there are also several (user specified) constants in their formula that allow them to tune the automatic inlining strategy. Their second cost function tries to estimate the benefits of inlining at a particular call site, based on the level of nesting inside loops, and the size of the control flow block that the call site resides in. They then traverse the call graph in a bottom-up manner selecting call sites to be inlined based on the cost functions for the call sites and the methods they call. This is very similar to our strategy for automatic inlining, which is also based on selecting potentially important call sites, and inlining at only these call sites provided the method being inlined is not very large.

Our work on inlining differs from theirs in two important ways. There are many complex inlining safety issues involved when inlining is performed at the Java bytecode level. Access restrictions have to be kept in mind, and there are also many Java Virtual Machine specification dependent issues. The fact that we only inline call sites within the benchmark and do not alter the Java class libraries in any manner also impedes our efforts to inline at every possible call site. Inlining library methods might not be allowed if the method accesses some library field/method that is inaccessible from the benchmark class (note that we are not allowed to change access restrictions of class members in the library). Also all calls are statically resolved in Fortran, while in Java it might be that many of the important call sites cannot be statically resolved because of polymorphic virtual calls.

The other important difference is that we try to inline methods at the Java bytecode level (which is almost equivalent to inlining on an executable), whereas they inline at the source code level. Consequently, they are concerned about the size of their source files as compilation time can increase significantly with increase in the size of their source files. In our case larger classfiles mean an increase in class loading time but we have in fact observed that this is not a significant factor in the total execution time of a benchmark (when the class files are available locally).

The work by Ayers [11] on aggressive inlining strategies at the intermediate representation level showed that they could get significant speedup (in some cases a factor of 2) for some well known SPEC benchmarks. Their inliner performs possibly multiple passes over the code to inline at the important call sites and also uses profile feedback. It should be noted that their speedup also included the gains as a result of many other global optimizations that became more effective as a result of inlining. Their inliner identifies the important call sites and inlines at these call sites greedily until it exceeds a precomputed budget. The budget is an estimate of how much compilation time would increase because of inlining (taking into account the fact that several optimization phases have non linear complexity). They try to limit compile-time increases to 100% over no inlining. Since they are compiling down to native code, they also measure the I-cache and D-cache miss rates, and register pressure. One of their conclusions is that profile feedback while inlining is crucial in achieving good performance. They also attack the widely held notion that inlining is only effective if the methods that get inlined are small in size.

Their work is similar to ours in that they perform inlining at the intermediate representation level, and their strategy for inlining involves changing the scopes for program entities wherever required (similar to changing access modifiers of class members in our case). There are also some differences compared to our work; their front end can build the intermediate representation for Fortran, C, and C++ programs, whereas we try to optimize Java bytecode. Also their inlining strategy is dictated to a large extent, by architecture related issues (caches, register allocation) which might not be applicable for Java in the presence of interpreters. They are able to include the cumulative effects of inlining itself and all the other compiler optimizations that become more effective in the presence of inlining in their performance results. We only report the raw improvement that method inlining has on performance as the rest of our framework is not developed enough to perform some of the global optimizations that they perform.

The work by Hwu et al.[31] focuses on the effectiveness of inlining in reducing the number of dynamic function calls for C programs on their IMPACT system. In related work on the same system Chang et al.[16] report a mean speedup of 11% with a maximum speedup of 46%. Their benchmarks are largely under 5000 lines of C code, whereas we try to analyze larger sized benchmarks. They use profiling information to assign weights to different call sites in their call graph, and use these weights while making inlining decisions. The other factor that they consider while deciding whether to inline at a call site is their cost function value at that call site. The cost function tries to estimate the effect on code size and cache performance if inlining is allowed at a particular call site. The cost function for methods are updated after each inline expansion.

Their experiments show that a large percentage of function calls/returns (about 59%) can be eliminated with a modest code expansion cost (17%). They suggest that the reduction in function call frequency would result in larger basic blocks that could be exploited effectively by instruction scheduling. The main source of the function calls remaining after inline expansion are system calls to the operating system and they express the need for further research in that area.

Davidson and Holler [20] described INLINER, an implementation of a C source to source automatic inliner and achieve a mean speedup of 12% with a maximum speedup of about 35%. Their main observation was that the increased register pressure as a result of inlining can have detrimental effects on performance. They do not compare their results with a profile feedback, whereas we provide experimental data for profile guided inlining.

Dean and Chambers [21] describe a novel approach that essentially involves training the compiler to make good inlining decisions. The compiler uses a database to record the results of inlining experiments conducted in the past. The potential benefit of an inline is estimated by consulting the database.

More research into inlining and related issues can be found in the work by Cooper et al. [18, 19, 17], Richardson et al. [34], Holler [28], and Allen et al. [10].

1.4 Thesis Contributions

We have focused on reducing the overhead associated with virtual method calls in Java bytecode in this thesis. We have adopted two distinct approaches to address this problem. First, we have tried to improve the precision of existing and future interprocedural analyses in our compiler framework by developing and implementing analyses that build a reasonably accurate call graph for the program. Second, we have implemented a compiler optimization that eliminates virtual calls and improve the performance of programs compiled to Java bytecode. We now briefly describe our work in both these approaches.

Simple techniques for call graph construction in the presence of virtual calls can be inaccurate especially in the case of highly object oriented programs, as they would have to assume that a particular call site might invoke many different methods at run time. Since the call graph is the basis of all interprocedural analyses, an inaccurate call graph can severely limit the effectiveness of interprocedural analyses. We implemented two well known and relatively inexpensive techniques for building call graphs, *class hierarchy analysis*[23, 25, 12] and *rapid type analysis*[12]. These two analyses serve as a baseline for comparison with the new techniques we propose.

Our original contribution is a new group of analyses, called *reaching-type analyses*, which are based on a *type propagation graph* where nodes represent variables and edges represent the flow of types due to assignments. The variations of reaching-type analyses that we have implemented are both *flow-insensitive* and *context-insensitive* and are consequently not expected to be very expensive if implemented in an actual compiler.

The first variation is called *declared type analysis*, where nodes represent the declared type of a variable. This is designed to be a coarse grained analysis that tries to limit the number of nodes in the constraint graph, so that the cost of solving the constraints is not high.

The second variation is more fine grained and is called *variable type analysis*. In variable type analysis, nodes represent variable names. It is more expensive to solve the constraints in this case as there are more nodes and edges in the constraint graph, but our empirical results show that this analysis does build a call graph that is significantly more accurate than any of the other analyses.

We have implemented declared type analysis and variable type analysis using the Soot framework[1], that provides several intermediate representations and APIs for analysis and transformation of Java bytecode.

We statically measured the precision of the call graphs built by each of the four analyses we implemented in our experiments using a set of 13 benchmarks generated from Java, Ada, ML, Eiffel and Pizza. These benchmarks are meant to be representative of real applications and vary in size from 1,000 to 42,000 Jimple statements without library code. As our analysis requires the whole application (including Java class libraries), the size of the program under analysis is actually about 70,000 Jimple statements for our largest benchmark.

In addition to the static analyses, we also produced a dynamic profile that is used to determine which methods are actually called at run time, and to obtain the frequency of execution of each call site. We then used these results to give us a bound on what can be achieved statically, and to compare the dynamic results of the baseline analyses with our reaching-type analyses.

We have implemented an optimization called *method inlining* aimed at improving performance of bytecode. Method inlining involves replacing a method invocation instruction by the code of the method that it invokes (if it can be determined at compile time). We provide a detailed and clear specification of the safety issues that are specific to performing method inlining at the Java bytecode level. We also discuss some important inlining criteria and our own static inlining strategy. We have added an option in our inliner that would allow for profile guided inlining. We have measured the run time improvement in performance for the set of benchmarks we mentioned earlier, and also compared our inlining strategies.

In summary, the main contributions of this thesis are :

- Design of reaching-type analyses used to estimate the set of run time types for the receiver of virtual method calls. Development of a coarse grained variation called declared type analysis, and a more accurate variation called variable type analysis, both of which are flow insensitive and context insensitive.

- Implementation of these analyses along with two well known analyses using Soot, which is a Java bytecode analysis and transformation framework. Study of the effectiveness of these analyses on a set of real, large sized benchmarks. Comparison with profiling results to estimate the bound for the best that can be achieved by any analysis.

- Implementation of method inlining with automatic and profile guided options and comparison of the two strategies. Measurement of the run time performance impact of this optimization on our set of benchmarks.

1.5 Thesis Organization

The rest of this thesis is organized as follows. In chapter 2 we introduce the well known analyses, class hierarchy analysis and rapid type analysis that we have implemented, and we provide detailed rules for our new analyses. In chapter 3 we discuss issues related to the optimization we have implemented. We present empirical data demonstrating the effectiveness of our analyses and optimization on real benchmarks derived from different languages in chapter 4. Finally, we state our conclusions and discuss the scope for future work in chapter 5.

Chapter 2

Analyses

In this chapter we introduce the analyses we have implemented that provide more precise information at virtual call sites. We also present the rules associated with each analysis and use examples to illustrate the differences between the analyses.

Our study is directed toward relatively cheap analysis techniques as we want to apply these techniques to large programs. Thus, it is essential that the techniques scale well with program size. Also the fact that we are doing whole application analysis means that we have to analyze classes that belong to the Java class library, and this can have a significant effect on the memory/time requirements of the analysis.

The analyses we have studied can be grouped into two categories. The first category consists of baseline analyses, which are known techniques that are among the cheapest for the problem of virtual method call resolution. The two analyses in this category are *class hierarchy analysis* (CHA)[23, 25, 12] and *rapid type analysis* (RTA)[12]. These techniques have been studied for other object oriented languages like Cecil[15] , Modula III[24], and C++[12] and hence form a baseline for comparison with our other analyses. The second group of analyses called *reaching-type analyses* is proposed by us and is based on an analysis that builds a type propagation graph where nodes represent program variables and edges represent the flow of types as a result of assignments. There are two analyses that fall in this category, *declared type analysis* (DTA) and *variable type analysis* (VTA). In declared type analysis, nodes represent declared types of program variables, whereas in variable type analysis the nodes represent program variables.

Our aim in performing each of the analyses is to determine the methods that can be invoked at virtual method call sites. The results of such analyses have many uses. We have used the results to perform a compiler optimization known as method inlining that aims to improve program performance. We discuss method inlining in greater detail in Chapter 3. Another consequence of improving the call graph of the program at virtual call sites is that it improves the precision of subsequent analyses like side effect analyses.

We now present details of our implementation of the analyses. In section 2.1 we explain how we build the class hierarchy, followed by a description of the structure of the call graph built using class hierarchy analysis. We then explain our implementations of rapid type analysis, reaching-type analysis and discuss some limitations in sections 2.2, 2.3 and 2.4 respectively. We discuss the comparison of our results with dynamic results in section 2.5.

2.1 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.

2.1.1 Class Hierarchy Analysis

Class hierarchy analysis is a standard method for conservatively estimating the run-time types of receivers. 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 inheritance hierarchy, and then we use this hierarchy to compute the appropriate *hierarchy_types* sets.

We now present details specific to our implementation of the inheritance hierarchy using the Soot framework.

Step 1 : Include all the classes that might be accessed starting from the main class. Note that in this step we are actually including all the classes in the set that is the *transitive closure* of all the classes that can be accessed. This is achieved through the use of a `SootClassManager` that automatically builds `SootClass` representations for all classes that belong to the transitive closure. We will refer to this set as `SootClassList`.

Step 2 : Establish all the direct superclass-subclass relationships between the classes included in Step 1. Each class maintains a list of its immediate subclasses in `SubClassList` at this stage. Note that the superclass of a particular class may be obtained directly from its `SootClass` representation. See method `EstablishDirectRelations` in Figure 2.1

Step 3 : Each class maintains a list of all its subclasses (all the classes in that subgraph of the inheritance graph of which the class is the root). Note that the subclasses are included in a transitive manner. See method `includeAllSubClasses` in Figure 2.2.

Step 4 : Each interface maintains a list of the `SootClasses` that implement the interface. See method `setImplementors` in Figure 2.3.

```
void EstablishDirectRelations() {

  while ( SootClassList.hasNext() )
  {
    nextclass = next( SootClassList );
    superclass = SuperClass(nextclass);
    add the SootClass corresponding to nextclass to superclass.SubClassList;
  }
}
```

Figure 2.1: Establishing direct parent child relationships

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.

```
void includeAllSubClasses() {

  while ( SootClassList.hasNext() )
  {
    nextclass = next( SootClassList );

    subclassQ = nextclass.SubClassList;

    while ( ! subclassQ.isEmpty() )
    {
      nextsubclass = head( subclassQ );

      if ( ! nextclass.SubClassList.contains(nextsubclass) )
      add nextsubclass to nextclass.SubClassList;

      subsubclasslist = nextsubclass.SubClassList;
      while ( subsubclasslist.hasNext() )
      {
        nextsubsubclass =  next( subsubclasslist );
        if ( nextsubsubclass does not belong to nextclass.SubClassList )
        add nextsubsubclass to subclassQ;
      }
    }
  }
}
```

Figure 2.2: Including all the subclasses transitively into the SubClassList of a parent

2.1.2 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 2.4(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 $B.main[2]$

```
void setImplementors() {

  while ( SootClassList.hasNext() )
  {
    nextclass = next( SootClassList );
    interfacelist = nextclass.getInterfaces();
    /* returns the list of SootClasses corresponding
       to the interfaces implemented by nextclass    */

    while ( interfacelist.hasNext() )
    {
      implementedinterface = interfacelist.next();
      add nextclass to implementedinterface.ImplementorList;
    }
  }
}
```

Figure 2.3: Establishing (interface) directly implemented-by relationships

which is `b.m()`.

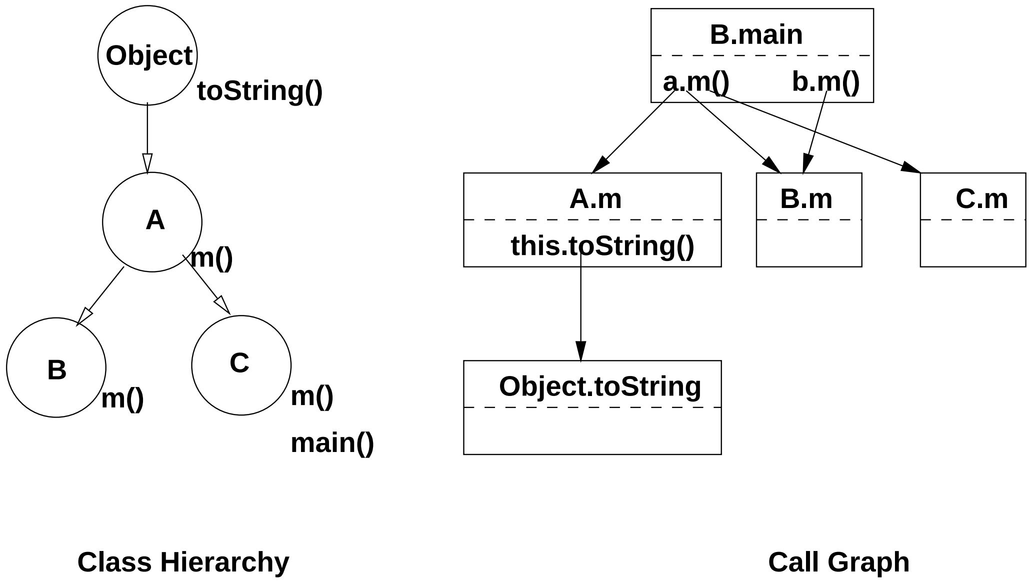


Figure 2.4: An example of the Call Graph built for the program.

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 $C.M.c[i]$ and $C'.M'$ in 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. An 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 2.4. 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.

2.1.3 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 2.4. 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.

We now explain our implementation of the call graph by performing Class Hierarchy Analysis using the Soot framework.

In Java bytecode, there are four different kinds of invoke expressions, and we explain the algorithm to build the call graph for each kind of invoke expression :

- *invokestatic:* This invoke expression is generated when the method call in the source language from which the bytecode was produced was to a method declared to be static. These method calls are already resolved statically and the the target method of the call is known before run time. The method in the invoke expression is the target of the call.

- *invokespecial:* This invoke expression is generated for calls to constructors, private methods, or methods in some parent class of the class in which the invoke expression occurs. The first two cases are simple as the method invoked at run time is the method referred to in the invoke expression. However there is some lookup performed in the case of calls to methods in a parent class of the current class. See Figure 2.6 for the algorithm.

- *invokevirtual:* This invoke expression is generated when the method call cannot be resolved at compile time and the target of the call is dependent on the run time type of the object that is the receiver of the call. In this case, we obtain the Jimple type of the local corresponding to the receiver of the method call, and use this type to build the call graph.
- *invokeinterface:* This invoke instruction is generated when the receiver of the call is of interface type in the Java source code. In this case too, we use the type of the local corresponding to the receiver in Jimple. See Figures 2.8 and 2.9 for the algorithm.

```
public SootMethod performMethodLookup ( searchingclass, invoke expression )

  searching = true;

  while ( searching )
  {
    if ( searchingclass declares a method m which has the same name,
         parameter types, and return type as the method in the invoke expression )
    {
      searching = false;
      declaredmethod = searchingclass.getMethod( m );
      return declaredmethod;
    }
    else
    searchingclass = SuperClass( searchingclass );
  }
}
```

Figure 2.5: Method to perform method lookup

2.2 Rapid Type Analysis (RTA)

Rapid type analysis [12] 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 with declared type

```
public void buildCallGraphForInvokeSpecials ( invoke expression ) {

  if ( ( the name of the method in the invoke expression is <init> )
        || ( method in invoke expression is private ) )
  add an edge between the callsite and the MethodNode corresponding
  to the method in the invoke expression;
  else
  {

    if ( the declaring class of method in invoke expression is
         a superclass of the currentmethod )
    {
      /* Method lookup is performed as it is a call to superclass's method */
      targetmethod = performMethodLookup ( currentclass, invoke expression );
      add an edge between the callsite and the MethodNode corresponding
      to targetmethod;
    }
    else add an edge between the callsite and the MethodNode corresponding
    to the method in the invoke expression;
  }
}
```

Figure 2.6: Building the call graph for invokespecial

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 2.4(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(A,P)* = $\{A,B\}$ 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. Note that our implementation of rapid type analysis is based on a *pessimistic* approach, as it starts with a call graph that is correct (built using class hierarchy analysis) and does not alter it in any manner during the analysis (detection of instantiations). After the analysis is complete, the call graph is pruned in one pass over the original call graph.

The alternate approach to performing rapid type analysis is termed the *optimistic* approach. In this approach it is initially assumed that no methods except `main` are called and no objects are instantiated, and therefore no virtual call sites call any target

```
public void BuildCallGraphForInvokeVirtuals ( invoke expression ) {

  declaredclass = type of the Jimple local corresponding to the receiver of the
                  method call;

  /* if base of the call is of ArrayType, call reaches method in java.lang.Object */
  if ( declaredclass is an ARRAY type )
  add an edge between the callsite and the MethodNode corresponding to the
  method in the invoke expression;
  else
  {
    targetmethod = performMethodLookup ( declaredclass, invoke expression );
    add an edge between the callsite and the MethodNode corresponding to
    targetmethod;

    subclasslist = declaredclass.SubClassList;
    while ( subclasslist.hasNext() )
    {
      subclass = subclasslist.next();
      if ( subclass declares a method m which has the same name, parameter types,
           and return type as the method in the invoke expression )
      {
        submethod = subclass.getMethod( m );
        add an edge between the callsite and the MethodNode
        corresponding to submethod;
      }
    }
  }
}
```

Figure 2.7: Building the call graph for invokevirtual

methods. The call graph created by class hierarchy analysis is traversed starting at `main`. Virtual call sites are initially ignored. When an object is created, any of the virtual methods of the corresponding class that were left out are then traversed as well. The live portion of the call graph and the set of instantiated classes grow iteratively in an interdependent manner as the algorithm proceeds.

The pessimistic approach has the advantage that the call graph is always correct during the analysis, and hence the analysis can be terminated at any point, and the call graph can be safely used for performing subsequent analyses or optimizations. Also this approach is relatively efficient as compared to the optimistic approach as the algorithm is not iterative and statements are examined only once, and the complexity of pruning the call graph is linear in the number of edges in the call graph. The

```
public void BuildCallGraphForInvokeInterfaces ( invoke expression ) {

  /* if base of the call is of ArrayType, call reaches method in java.lang.Object */
  if ( declared type of the receiver is an ARRAY type )
  add an edge between the callsite and the MethodNode corresponding to the
  method in the invoke expression;
  else
  {
      declaringinterface = ( INTERFACE ) declared type of the receiver;
      implementorlist = declaringinterface.ImplementorList;

      while ( implementorlist.hasNext() )
      {
        implementorclass = implementorlist.next();

        targetmethod = performMethodLookup ( implementorclass, invoke
        expression );
        add an edge between the callsite and MethodNode corresponding
        to targetmethod;

        implementorsubclasslist = implementorclass.SubClassList;
        while ( implementorsubclasslist.hasNext() )
        {

          implementorsubclass = implementorsubclasslist.next();

          if ( implementorsubclass declares a method m which has the same name,
          parameter types, and return type as the method in the invoke expression )
          {
            implementorsubmethod = implementorsubclass.getMethod( m );
            add an edge between the callsite and MethodNode corresponding
            to implementorsubmethod;
          }
        }
      }
    }
```

Figure 2.8: Building the call graph for invokeinterface

optimistic approach arrives at an answer that corresponds to the least fixed point in the call graph lattice domain whereas the pessimistic approach would terminate at the greatest fixed point. In other words, the optimistic approach is guaranteed to produce a call graph at least as precise as the one produced by the pessimistic approach. However, the algorithm must terminate to ensure that the resultant call graph is correct, as the call graph might be incomplete at intermediate steps. Also as

```
    interfacesubclasslist = declaringinterface.SubClassList;
    while ( interfacesubclasslist.hasNext() )
    {
      subinterface = interfacesubclasslist.next();

      implementorlist = subinterface.ImplementorList;
      while ( implementorlist.hasNext() )
      {
        subintimplementorclass = implementorlist.next();

        implementedmethod = performMethodLookup ( subintimplementorclass,
        invoke expression );
        add an edge between the callsite and the MethodNode corresponding
        to implementedmethod;

        subintimplementorsubclasslist = subintimplementorclass.SubClassList;

        while ( subintimplementorsubclasslist.hasNext() )
        {
          subintimplementorsubclass = subintimplementorsubclasslist.next();

          if ( subintimplementorsubclass declares a method m which has the same name,
          parameter types, and return type as the method in the invoke expression )
          {
            subintimplementorsubmethod = subintimplementorsubclass.getMethod( m );
            add an edge between the callsite and the MethodNode corresponding to
            subintimplementorsubmethod;
          }
        }
      }
    }
  }
```

Figure 2.9: Building the call graph for invokeinterface (continued)

the algorithm is iterative it might need to examine a particular call site several times.

Typically in benchmark code, methods are rarely created without being called; hence we expect both approaches of rapid type analysis to produce similar results. In library code, the optimistic approach might perform better as there are often methods created for use by developers (that are not actually called within the library itself). We have chosen to implement the pessimistic approach as we are using rapid type analysis as a baseline for comparison purposes only, and as we are interested more in analyzing and optimizing the benchmark alone and not the libraries.

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. RTA is expected to perform well on applications that contain many abstract classes (which are never instantiated in Java). Abstract classes are expected to be present in library code that is created to be used by other applications.

A detail about RTA that needs to be explained is what happens when an array is instantiated : `A[] a = new A[10];` In this case RTA would consider it simply as if an instance of the class `A` had been created. A method call with an array element as the receiver e.g. `a[i].m()` would be recognized as possibly reaching `A.m()` by RTA. Method calls with `a` as the receiver (e.g. `a.toString()`) actually reach `java.lang.Object`'s methods at runtime. If the base of the method call is is an array, we recognize the fact that the call graph edge to the method in class `java.lang.Object` must be retained. Thus call sites with the array variable `a` or an array element `a[i]` as the receiver are both handled correctly.

2.3 Reaching Type Analysis

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 `new()` 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.

2.3.1 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. `C.f`) where `C` is the name of the class in which `f` is declared. This means that we are approximating all instances of objects with field `C.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.

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 $C.f$
- 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 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 2.10(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 2.10(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`.

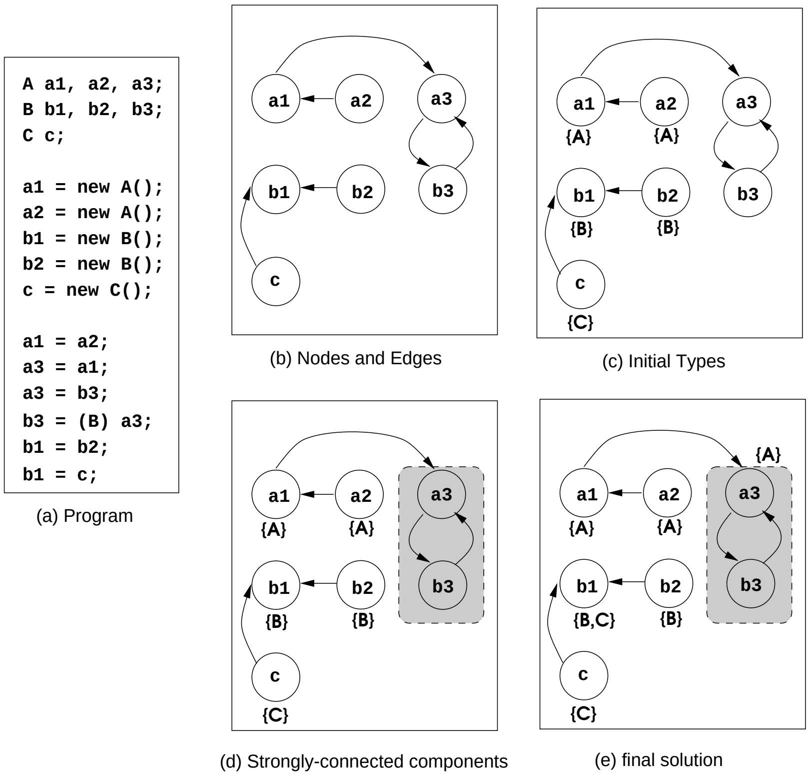


Figure 2.10: An example of the type propagation graph for Variable Type Analysis.

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.[1] It is also true for field references because we represent all instances of objects

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

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`.[2] 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.

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 behavior, and in fact the graphs are quite sparse.

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 2.10(c) shows the type initialization for the example program.

[2]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.

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 2.10(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 has complexity $O(max(N, E))$ operations, where the most expensive operation is a union of two ReachingType sets.

Figure 2.10(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.

We have present the rules showing the effect of each Jimple statement on the constraint graph constructed by variable type analysis in Appendix A.

2.3.2 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 2.11 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 Chapter 4 we present empirical results to evaluate these two analyses with respect to accuracy and the size of the graph problem to be solved.

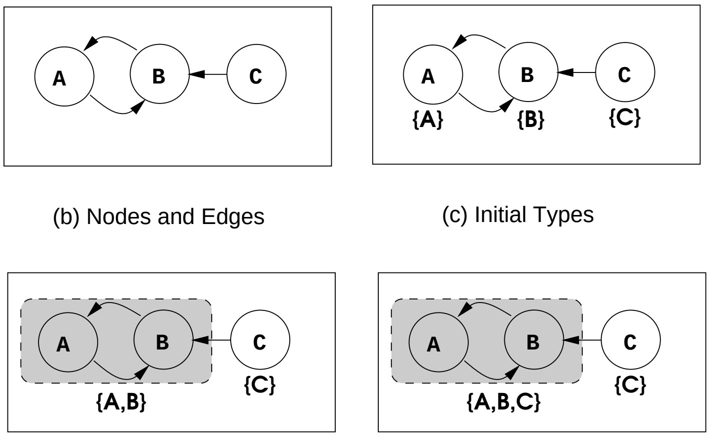
We present the rules showing the effect of each Jimple statement on the constraint graph constructed by declared type analysis in Appendix B.

```
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 2.11: An example of the type propagation graph in Declared Type Analysis.

2.4 Assumptions and Limitations

In this section we discuss some issues that limit the precision of our analyses and how we have attempted to reduce their impact.

Native Methods

It is not possible to jimplify native methods and so our analyses are unable to analyze the statements in these methods. We have therefore summarized the effect of native methods by manually going over the source code for native methods (available with the open source JVM *Kaffe* [2]), and checking for class instantiations and writes to fields. Since native methods are typically encountered in library code only and not in benchmark applications, we can use the results that we obtained for library native methods repeatedly when we are analyzing different benchmarks. In the presence of native methods in the benchmark application itself, the results of the analysis might be incorrect if the native methods have significant side effects.

To give a better feel of the summarizing that we have done for native methods we present the following example. An example of a native method that is commonly invoked is the `clone()` method in `java.lang.Object`. This method has no impact on rapid type analysis, as objects are only being duplicated and the set of instantiated classes does not change as a result of cloning. For the reaching type analyses, we need to recognize the fact that the object that is the receiver of the call is being duplicated and returned by the method. Thus all the runtime types that were associated with the receiver must also be associated with the object returned by the method and there is no need to be any more conservative.

The task of summarizing native methods is simplified to an extent by the observation that native methods rarely write to fields. In fact there are not very many library classes that declare fields of reference type (most of the fields in libraries are of primitive types). Thus we found that in practice, apart from detecting instantiated objects that are returned by native methods, the summarizing was not a very involved process.

The Closed Application Assumption

The basis for all the analyses that we have implemented apart from CHA is that classes can be instantiated only by `new` statements. However this is not strictly true in Java because of the native library method `java.lang.Class.newInstance()`. It is possible to use this method to instantiate any class as shown in Figure 2.12

```
class ArbitraryInstantiation {
  public static void main ( String[] args ) throws
  java.lang.InstantiationException,java.lang.IllegalAccessException,
  java.lang.ClassNotFoundException
  {
    String arbitraryclass = args[0];
    java.lang.Class c = java.lang.Class.forName ( arbitraryclass );
    Object o = c.newInstance();
    System.out.println ( o.toString() );
  }
}

class ArbitraryClass {
  public String toString() { return "Arbitrary"; }
}
```

Figure 2.12: Example of class instantiation without a call to a constructor

When the `ArbitraryInstantiation` class is run with the argument `args[0]` equal to `ArbitraryClass`, the output is the string ``Arbitrary'' produced by the `toString()` method of `ArbitraryClass`. In general depending on the argument, any class that is accessible to the Java Virtual Machine might be instantiated in the above manner.

It is impossible to detect class instantiations that occur in this manner through static analyses, and we make the assumption that applications we are analyzing must be closed, i.e. all the classes that are accessed from the class must be referred to in the code explicitly. We compensate for calls to `java.lang.Class.newInstance()` in Java library code by manually checking the methods in which there are calls to this method, and observing the actual class that is being instantiated by looking at the source code. This task was also not very hard in practice, as at all places where there are calls to this method, it is followed in the source code by explicit casts that allow us to detect the actual class of the object. If the benchmark itself loads some classes dynamically, then a possible solution might be to specify the names of these classes in a file. At the time of building the inheritance hierarchy this file could be examined and intermediate representations for the classes referred to in the file could be built to be used by subsequent analyses and optimizations.

2.5 Comparison with Dynamic Results

The focus of the analyses presented in this chapter is in obtaining a precise set of methods called from each method call site. One possible measure that could be used to evaluate different analyses being compared is the number of edges that get removed from the call graph in each case. This comparison is based entirely on the static results that are obtained by measuring the effect of the analyses, and is enough to establish the extent of improvement that is to be expected by increasing the complexity of analysis. However it does not offer any insight about the extent of improvement that is still possible after performing the analysis. A measure of the gap between the "best possible answer" and the answer obtained by performing an analysis is very important for a variety of reasons.

If it can be shown that the scope for improvement of the analysis results is very little, that information might save the extra time and effort that might otherwise have been invested in implementing increasingly complex and expensive analyses that would not lead to any significant gain in precision. Another application of knowing the "best possible answer" for each call site is that we can fix the exact cause of imprecision in the results obtained through the analysis. This can offer significant indications about the features that a more complex analysis should have in order to be more effective. For example if it is observed that the imprecision in the analysis results is a result of context insensitivity of the analysis then subsequent analysis can be designed with the knowledge that including information about calling context would be beneficial. This is similar to the cause analysis used by Diwan et al. [24].

We now briefly explain the approach we have taken to arrive at the estimate for the "best possible answer" of interest in our case. We are interested in fixing the methods that can be called from a particular call site in our analysis and we insert profiling code into the bytecode that keeps track of the methods actually invoked during some sample execution of the program, and generate a trace. The trace of execution produced contains data grouped into pairs, a callsite (identified by a unique ID) and the method that was called from the callsite each time it was executed. We generate the traces by injecting Jimple code into the benchmark classes being profiled, and using classes produced by Soot (with profiling code in them) to run the benchmark. We show a small class that has the profiling code inserted in it in Figure 2.13

We have inserted a call to a static method `printMethodCalled()` (in a class `Profiler` that we have defined) preceding each virtual call site. The parameters to the static method are the ID of the call site (parameters 1 and 2), the signature of the method being invoked (parameters 3 and 4), and the object that is the receiver of the call. Note that the call site ID consists of the caller method signature and a number i representing the fact that it is the ith call site. The parameter types of the method being invoked are passed as a string to the static method. The static method we have defined uses the library classes `java.lang.Class`, and `java.lang.reflect.Method` to obtain the actual class of the receiver object during execution and then perform method lookup using the method signature to arrive at the actual method being invoked. In order to reduce the size of the traces we generate in this way, we only print out a method reached from a particular call site the first time it is invoked from that call site. This involves keeping track of the methods that were invoked from the call site on previous executions, and comparing the method invoked currently to the methods that have been invoked previously. This makes the trace generation process slower but it has the significant benefit of generating much smaller traces. Smaller traces mean that the comparison with results from different analyses takes much lesser time.

We use the profiled results for each call site to examine the need for better analyses, as well as to perform cause analysis on our own analyses to examine their

```
class toy extends java.lang.Object
{

    public static void main(java.lang.String[])
    {
        java.lang.Object r0;
        toy r1;

        r0 := @parameter0;
        r1 = new toy;
        specialinvoke r1.[toy.<init>():void]();

        // STATIC CALL INSERTED TO Profiler.printMethodCalled

        staticinvoke [Profiler.printMethodCalled
        (java.lang.String,long,java.lang.String,java.lang.String,java.lang.Object):void]
        ("toy.main(java.lang.String[]):void", 2L, "m", "/int/toy/", r1 );

        virtualinvoke r1.[toy.m(int,toy):void](3, r1);
        return;
    }

    void m(int, toy)
    {
        int i0;
        toy r1, r0;

        r1 := @this;
        i0 := @parameter0;
        r0 := @parameter1;
        return;
    }
}
```

Figure 2.13: Example of class with profiling code inserted

shortcomings. We present these dynamic results in Chapter 4.

Chapter 3

Method Inlining

In this chapter we introduce the optimization known as *method inlining* and discuss the issues involved in our implementation. This optimization is based on the analyses that we discussed in Chapter 2. As has been mentioned before, the invokevirtual and invokeinterface bytecode instructions are expected to be expensive at run time. There are several optimizations that can be implemented to reduce this overhead once it is definitely known that the call site can only call a particular method. For each call site, our static analyses determine the set of methods that are potential run time targets with varying degrees of accuracy. The opportunity for our optimization arise only at those call sites for which the set of target methods is a singleton. There are some possibilities for other optimizations when this set is not a singleton, and we will briefly discuss them at the end of this chapter.

In Java, method inlining is a complex optimization that is only safe to apply if certain safety criteria are satisfied. Also the precise algorithm used to perform method inlining is critical in achieving performance improvement. We explain these issues involved in performing method inlining in detail in this chapter and also present empirical results for the actual run-time improvement in performance as a result of performing our optimization on a set of benchmarks in Chapter 4. This optimization has been implemented on the Jimple intermediate representation and the Soot framework is used to produce optimized classfiles.

3.1 Method Inlining

Method inlining [14, 11, 31, 16, 20, 21, 18, 19, 17, 34, 28, 10] is an optimization technique that has been used by optimizing compilers traditionally for both procedural and object oriented languages. The basic idea in method inlining is to statically replace a method invocation instruction by the code representing the body of the method that is the target of the call. By performing this transformation, the overhead associated with executing the method invocation instruction can be avoided. We illustrate this optimization by a simple example in Figure 3.1.

```
class Example {
 public static void main ( String[] args ) {
  A a = new A();
  a.m(5);
 }
}
```

(a). Example method before inlining

```
class A {
 int f;
  public m ( int i ) {
  this.f = i;
 }
}
```

(b). Callee method

```
class Example {
 public static void main ( String[] args ) {
  A a = new A();
  a.f = 5; // Method inlining done here
 }
}
```

(c). Example class after method inlining

Figure 3.1: An example of method inlining in Java code

In this simple program, the method call `a.m(5);` in the original program was statically determined to be invoking the method `A.m()`. In this case it is possible for inlining to be done and as is shown in Figure 3.1(c), the call to the method has been replaced by the actual code from the method `m`.

3.1.1 Applications of Method Inlining

Method inlining is expected to lead to greater improvement in performance for object oriented languages like Java/C++ as compared to procedural languages like C. In programs developed in an object oriented manner, the frequency of invocation instructions is expected to be considerably greater. Further, in object oriented languages the overhead of method lookup associated with these instructions make them expensive at run time.

Another factor that makes method inlining a useful optimization is that it eliminates the control flow edges because of the invocation instruction from the Control Flow Graph (CFG). Frequent branches in the code mean that some of the techniques used for optimization at the architecture level, like pipelining, cannot be performed optimally. Instruction scheduling and pipelining are techniques that are very effective if the program has a relatively simple flow of control. In the presence of complex control flow, the CPU will be idle during some cycles even if pipelining is being done, thus degrading performance. Architecture issues are not as important in the case of interpreters as they are if a Just-In-Time (JIT) compiler is used (which produces native machine code). This effect on performance is even more significant in object oriented languages because typically programs have methods with small bodies. These small methods are expected to contain mostly instructions to manipulate fields within the declaring class of the method, and not very many method invocation instructions. In such cases inlining calls to these small methods could be beneficial in increasing the size of basic blocks (or extended basic blocks) and make the instruction pipeline proceed without stalling to account for branches in control flow. Thus replacing the invocation instruction by the body of the method is expected to lead to better overall performance.

Another possible area of application for method inlining is in its interaction with other static analyses. Interprocedural static analysis is more complex and more expensive than intraprocedural analysis. Intraprocedural analysis has better scalability but is more imprecise as compared to interprocedural analysis as it only considers one method body at a time for its analysis. Method inlining can be used to inline code from other methods into the method being analyzed. If method inlining is performed on a method before performing intraprocedural analyses, this would improve the precision of the results obtained while at the same time, avoid having to do interprocedural analysis.

An area unrelated to compiler optimizations in which method inlining could be of use is in model checking (verifying the correctness of a program). It is more suitable for model checkers if they are provided with a program with a few large methods (with inlining done in them to the maximum extent possible) to analyze, rather than one in which there are many small methods.

3.1.2 Disadvantages of Method Inlining

There are also some possible disadvantages of doing method inlining that need to be mentioned.

The size of the program can increase substantially if method inlining is performed aggressively. This is an undesirable feature in general, and particularly so in the case of Java. Larger classfiles mean that the time required to fetch them from a remote host would be greater than the case when no inlining is done. This may be unacceptable in terms of performance as the time to download applets over the network before executing them is a serious concern in applications such as web browsers. Larger classes mean an increase in the class loading time, as well as an increase in the amount of memory utilized when the class is being used. In extreme cases, the size of a method with inlining performed in it might exceed the maximum allowed size (65535 locals and 65536 bytes in size) of a method in Java bytecode as specified by the Java Virtual Machine Specification.

Method inlining introduces binary compatibility issues. Once inlining has been performed, the method that was inlined can no longer be changed, as otherwise, the behavior of the program would be different as compared to what was intended (as the original code would have been inlined at several call sites). Thus, in case the method has to be changed, then the whole program has to be reanalyzed and the optimization must be performed taking the changes into account. Clearly this could be quite expensive, and therefore, undesirable.

Another disadvantage is that inlining aggressively could lead to a program that is extremely difficult to understand. In the case of Java, decompilers attempt to reconstruct the original Java source files from the classfiles, and this is often used to understand the behavior of the program in the absence of the original source files. If the classfile used has one large method with very few method calls as a result of inlining, then the Java source file produced by the inliner would be bereft of some programmer friendly features like method calls to library methods (for I/O, Math, and String operations etc.) or to other user defined methods. Also the source file produced by the decompiler would bear little resemblance to the original source file and is of limited use in program understanding.

There are some methods in classes (e.g. `java.lang.SecurityManager`) belonging to the Java library that return values dependent on the contents of the execution stack. Since the execution stack is dependent on the actual call chain executed at run time, altering the call chain (by inlining, we get rid of caller/callee relationship between methods) might result in a change in the run time behaviour of the program. This is clearly a drawback and should be avoided. We have adopted a conservative strategy that is based on the fact that calls to these library methods typically occur infrequently. We have observed that for our set of benchmarks, these methods never get called. We have chosen to disable method inlining in case calls to these methods are detected in the call graph, in order to ensure the behaviour of the program is unchanged. An alternate strategy would have been to allow inlining in any case, but not make any guarantees regarding the behaviour of the program after inlining.

Therefore it should be clear that the criteria for inlining a method need to be chosen with some care. We have studied the impact of some inlining strategies on the size and performance of the resulting classfiles, and we discuss them in Chapter 4.

We now discuss, in detail, some of the structural issues that are of interest in our implementation of method inlining, as well as our approach to detecting which invocation instructions are safe to inline.

3.1.3 Structural issues in method inlining

We discuss some of the issues in the actual inlining of code from the method that is being invoked into the caller method. These are the issues involved once the application has been analyzed and the method invocation instruction is found to satisfy all the inlining safety rules/criteria. We shall discuss them with reference to the Jimple representation of the example program we have shown in Figure 3.2 and the Jimple representation of the same program with inlining performed as shown in Figure 3.3.

The following are the steps involved in the inlining process :

1. Duplicate and Add Locals : Create a new local in the caller method for each of the locals declared in the callee method that is being inlined. This essentially involves cloning each local in the callee method, adding the cloned local (which has the same type as the local in the callee method) to the method body of the caller method, and storing the mapping from the locals in the callee method to the corresponding locals that have been created in the caller method. This mapping for locals is implemented using a hash table (*LocalHashMap*) which takes a local in the callee method as the key, and returns the new local created for inlining purposes in the caller method as the value. The mapping is used when statements are to be

```
class Example extends java.lang.Object
{
    public java.lang.String s;

    public static void main(java.lang.String[])
    {
        Example r2, r1;
        int i0, i1;
        java.lang.Object r0;

        r0 := @parameter0;
        r1 = new Example;
        specialinvoke r1.[Example.<init>():void]();
        i1 = virtualinvoke r1.[Example.m1():void]();
        i0 = 0;
        goto label1;

        label0:
        r2 = new Example;
        specialinvoke r2.[Example.<init>():void]();
        i1 = virtualinvoke r2.[Example.m1():void](); // CALL THAT IS BEING INLINED
        i0 = i0 + 1;

     label1:
        if i0 < 2 goto label0;

        return;
    }

    public int m1()
    {
        Example r0;
        r0 := @this;
        r0.[Example.s:java.lang.String] = "EXAMPLE";
        goto label0;
     label0:
        return 1;
    }

    void <init>()
    {
        Example r0;
        r0 := @this;
        specialinvoke r0.[java.lang.Object.<init>():void]();
        return;
    }
}
```

Figure 3.2: Jimple representation of the class in which inlining is being performed (before inlining)

```
class Example extends java.lang.Object
{
    public java.lang.String s;
    public static void main(java.lang.String[])
    {
        Example r1, r2;
        Example inline$0;
        int i0, i1, inlinereturn$0;
        java.lang.Object r0;
        java.lang.NullPointerException inlinenull$0;

        r0 := @parameter0;
        r1 = new Example;
        specialinvoke r1.[Example.<init>():void]();
        virtualinvoke r1.[Example.m1():void]();
        i0 = 0;
        goto label4;

     label0:
        r2 = new Example;
        specialinvoke r2.[Example.<init>():void]();
        if r2 != null goto label1;
        inlinenull$0 = new java.lang.NullPointerException;
        specialinvoke inlinenull$0.[java.lang.NullPointerException.<init>():void]();
        throw inlinenull$0;

     label1:
        inline$0 = r2;
        inline$0.[Example.s:java.lang.String] = "EXAMPLE";
        goto label2;
     label2:
        inlinereturn$0 = 1;
        goto label3;
     label3:
        i1 = inlinereturn$0;
        i0 = i0 + 1;
     label4:
        if i0 < 2 goto label0;

        return;
    }

    public int m1()
    {
        Example r0;
        r0 := @this;
        r0.[Example.s:java.lang.String] = "EXAMPLE";
        goto label0;
     label0:
        return 1;
    }

    void <init>()
    {
        Example r0;
        r0 := @this;
        specialinvoke r0.[java.lang.Object.<init>():void]();
        return;
    }
}
```

Figure 3.3: Jimple representation of the class in which inlining is being performed (after inlining)

duplicated for inlining. Name conflicts must be avoided between locals present in the method originally and the cloned locals that are added to the method. Note that the local names of cloned locals in the example program do not conflict with any of the original locals.

2. New local for method return : For callee methods that return a value (return type is not `void`), we create a new local called `inlinereturn$N` in the caller method before we inline the method invocation instruction. The type of this new local is the return type (obtained from the signature) of the callee method. This variable is used to temporarily hold the value returned by the callee method (it is possible that there is more than one return statement in the method) along different control flow paths. We will soon discuss the exact manner in which the inlining process gets simplified because of this new local. In Java bytecode locals cannot be of type `boolean, byte, char, or short`. All variables declared to be of any of these four types in the source code are actually of type `int` in bytecode. However the return type of a method could be one of these four types, and it is the actual return type that is present in the signature of the method. Thus we need to be careful while creating the new local corresponding to the return value, so that the type of the new local is `int` in case the return type of the method that is being inlined is any of the four types we have mentioned. Note that we have introduced the new local `inlinereturn$0` (of type `int` at line 8 which is the return type of method `m1` that is being inlined) in our example program.

3. Explicit Null Check : Inlining a method call involves replacing the invocation instruction in the method body of the caller method. A method invocation of a non-static method can result in a `java.lang.NullPointerException` object being thrown if the receiver of the method call is null. Since this check is implicitly performed by the Java Virtual Machine at runtime when it is executing the method invocation instruction, simply replacing the invocation instruction by the method body of the callee would not capture the effect of the instruction accurately. This inaccuracy might be significant as the `java.lang.NullPointerException` might be caught somewhere in the benchmark code and a series of instructions might be executed in the exception handler code. Hence in order to ensure that all the `java.lang.NullPointerException` objects that were being thrown in the original program are also thrown in the transformed program, we introduce code before the beginning of the inlined code that performs the null checks explicitly. In the example program the local `inlinenull$0` (of type `java.lang.NullPointerException`) has been introduced and is being used to perform the null check (lines 22 to 25 in the example program in Figure 3.3). Note that it should be possible to reduce the number of explicit null checks to be inserted based on the results of a static analysis that determines nullness.

At this stage all the new locals that are needed to start inlining have been introduced and the null pointer check for the receiver has also been added. We can begin duplicating the statements from the callee method to the caller method.

4. Duplicate and Add statements : Assignment statements are inserted to copy each parameter. In the Jimple representation this involves creating a new `AssignStmt` in the caller method corresponding to each `IdentityStmt` involving parameters in the callee method. The copying of the implicit parameter `this` is slightly more complex. The type inference algorithm in the Java Virtual Machine infers a type for the receiver of the method call based on an intraprocedural dataflow analysis. The type inferred by the Java Virtual Machine might be a superclass of the declaring class of the method that is the run time target of the call. In the presence of interfaces, the Java Virtual Machine might infer the type of the receiver to be an interface, whereas the run time target of the call would be some method declared in a class implementing the interface. If we naively inline the target method at a call site where the inferred type of the receiver is higher in the inheritance hierarchy than the declaring class of the method, the Java Virtual Machine would raise a verification error. This is because the local representing `this` in the method being inlined would be inferred to be of the same type as the receiver of the call if a simple assignment statement is used to copy the `this` parameter.

We can inline at this call site without violating verification constraints by introducing an explicit cast while copying the `this` parameter (casting the receiver to the declaring class of the method being inlined). The explicit cast would enable the Java Virtual Machine to infer that the type of the local in the inlined code representing `this` (in the inlined method) is the declaring class of the inlined method, thus avoiding the verification error. We do not introduce the casts at each call site where inlining is performed; we introduce the casts only at those call sites where the Jimple type of the receiver (inferred by the type inference algorithm in Jimple) is neither the declaring class of the method being inlined, nor a subclass of the declaring class of the inlined method.

Each statement in the method body of the callee method is cloned and the locals accessed in the statement are adjusted using *LocalHashMap* that contains the mapping between locals in the callee method and the new locals created for inliningin the caller method. Refer to the example where the cloned local `inline$0` in method `main()` corresponds to the local `r0` in method `m()`. Also observe how the cloned statements introduced into the method `main()` as a result of inlining (for statements that use `r0` in `m()`) use the cloned local `inline$0`. While cloning each statement, the mapping between each statement in the callee method (that is being inlined) and the corresponding cloned statement introduced into the caller method is stored in a hashtable *StmtHashMap* for use in the following steps. Note that in the example class in Figure 3.3, we have replaced the statement `return 1;` in the callee method by two statements `inlinereturn$0 = 1; goto label3;` in the caller method (lines 32 and 33). We employ this technique in order to mimic the actual flow of control in the callee method. The value being returned is assigned to the special local `inlinereturn$0` and control passes to the last statement in the inlined code (`i1 = inlinereturn$0;` at line 35 in Figure 3.3) that mimics the return from the callee method to the caller method.

Although statements are cloned and added to the method body of the caller method, certain relationships between statements in the callee method like flow of control, and exception ranges have not yet been captured in the case of the cloned statements in the caller. The flow of control is adjusted using *StmtHashMap* to establish the targets of branch statements correctly in the inlined code. In the example in Figure 3.3, the target of the cloned `GotoStmt goto label2;` at line 30 is determined to be the statement `inlinereturn$0 = 1;` at line 32 at this stage. Note that *StmtHashMap* is used to get the cloned statement corresponding to the original target in the callee method. Another pass over the inlined code adjusts the trap table of the caller method such that exceptions in the callee method are included in the caller method and the range of statements in which exceptions might be raised is also entered in the trap table (after referring to *StmtHashMap*).

With this fixup step, the inlining process is complete for the method invocation instruction under consideration. These are the exact steps followed for inlining the callee methods at each call site that is determined to satisfy the inlining criteria/rules. Once all the inlining in a particular method has been completed, the method can be used to produce correct bytecode that has the same semantic behaviour as the original method in the original class. However the code that we have produced after inlining suffers as a result of some clear inefficiencies. Therefore we choose to make two more passes over the method (containing inlined code) at this stage to clean up the code created as a result of inlining. Note that these passes can be omitted and the program would still behave in exactly the same manner as the original uninlined program (except that execution might take longer).

Cleanup Passes

Our procedure for inlining a method call, though reasonably simple, is not as efficient as possible for the following reasons. In the inlining process, new locals are created in large numbers, and the method that is the target of inlining could have the number of its locals increased by several factors. In fact the more the number of call sites at which inlining is done, the more the number of locals in the method in the caller method. So while the benefits from inlining increase with more call sites being inlined, the overhead associated with the resultant increase in locals (increase in the size of the class, as well as the performance penalty explained in Section 3.1.4) must also be paid. Another example of inefficiencies introduced as a result of our inlining procedure are the explicit assignments in the inlined code that are used to mimic the implicit assignments as a result of parameter passing. The use of the buffer local `inlinereturn$N` (the assignment statements involving this local) to hold the return value is also inefficient as compared to return statements that were present in the original program. In addition, the replacement of the `return` statements by `goto` statements might be redundant in some cases.

We now explain the 2 cleanup passes and the effect they have on the inlined code in detail.

Local Packing : The first cleanup pass performs local packing which is one of the standard transformations present in the Soot framework. This pass has the effect of packing sets of two or more locals that are found to satisfy certain properties into one local. The locals that are packed into one local should satisfy the property that they should be in disjoint def/use chains and they should have compatible types. This ensures that they can be packed into the same local without any conflicts. One of the big advantages of doing local packing at this stage is that the many locals of type `java.lang.NullPointerException` that we introduced for the null checks while inlining, now get packed into considerably fewer locals. This pass also results in the reduction of extra locals introduced as a result of cloning the locals in the callee method that was inlined. A conceptually simple case in which benefits of local packing might be significant is the packing of locals of type `int`, or `java.lang.String` in inlined code obtained as a result of inlining at more than one call site. Most methods declare locals of these basic types, and it is possible to pack many such locals introduced into the caller method because of inlining at different call sites in the caller method (if they satisfy the packing criteria mentioned earlier).

Redundant Statements Cleanup : The second cleanup pass is used to remove redundant statements that have been introduced into the code as a result of inlining. Some standard optimizations available in the Soot framework are copy propagation, constant propagation and dead code elimination. Note that these optimizations are performed in an intraprocedural manner. These transformations are extremely useful in removing the copy statements (for parameters) at the beginning of the inlined code.

Redundant Goto Elimination : Another optimization that reduces the number of statements in the method is redundant goto elimination. This is a peephole optimization that is useful in the case of inlined code in particular. Our procedure for inlining mimics control flow in the case of `return` statements by replacing them by `goto` statements. In practice, it is quite common for methods to have just one `return` statement as the last statement in the code. In such cases our naive inlining procedure would insert a `goto` statement whose target is the very next statement in control flow. This optimization is aimed at detecting and eliminating such redundant `goto`'s.

It needs to be stressed that these transformations are crucial to obtaining maximal benefits from inlining, and omitting them might lead to insignificant speedup of the program (or even a slight slowdown in some cases). Their effectiveness is heightened when the methods that have been inlined are relatively small in size. The reason for this is that in the case of small methods, the extra statements introduced while inlining are almost as many as the number of statements in the method itself. In such a situation the inlining penalty is significant in the absence of redundant statements cleanup.

The final stage in the inlining process is the generation of optimized classes with method inlining performed in them. We have inlined method calls in the Jimple representation and used the Soot API for SootClass to produce classes containing the inlined method calls.

Additional Structural issues for inlining synchronized methods

Invoking a synchronized method makes the current thread acquire a monitor on the receiver of the method call. The monitor is released by the current thread when the method has finished executing or if an exception (that is not caught anywhere in the body of the synchronized method) is raised. Note that these actions are performed internally by the Java Virtual Machine at the time of invoking a synchronized method. While inlining synchronized methods, we mimic these implicit actions performed by the Java Virtual Machine by explicitly introducing code that has the same effect. A method that is declared to be synchronized has exactly the same functionality as a method (that is not declared synchronized) having its entire body enclosed within a synchronized block. We use this fact while generating the inlined Jimple code when we inline synchronized methods.

3.1.4 Safety Criteria for Method Inlining

We now introduce the criteria that we check for before we decide to inline the target method of the invocation instruction. Note that in order to be able to inline, **all** of the criteria we specify must be satisfied. The method in which the invocation instruction occurs is referred to as the current method in this discussion, and the method that is the target of the invocation instruction is referred to as the target method.

We now explain our reasons for choosing each of these criteria one by one.

Rule 1 : *There must be exactly one target method for the invocation instruction in the call graph.*

This test can be made using a call graph built using any of the analyses in Chapter 2 since they all result in a call graph that is conservative and correct.

Rule 2 : *The target method must not be the same as the current method.*

This is to avoid inlining recursive calls to the current method as there would be little performance benefit in doing so. Note that inlining might occur at other call sites in the recursive method.

Rule 3 : *The target method must not be a native method.*

The code for native methods is not available in the form of Java bytecode, therefore it is not possible to inline calls to these native methods.

Rule 4 : *The invocation instruction must not result in an illegal access error in the original program.*

We do not inline invocation instructions that might result in an illegal access error in the original program, as inlining may result in the error no longer being thrown. This is because the access modifiers are checked by the run time system when the invocation instruction is executed, and by replacing the invocation instruction we are eliminating those checks. To preserve the same semantic behaviour as the original program, we need to detect all the invocation instructions in the original program that might result in an illegal access error being thrown, so that we can avoid inlining at these call sites. We show an example of a program where inlining is not allowed at a particular call site for this reason in Figure 3.4. Note that method `m()` was being accessed illegally in the original program, but after inlining the access to `m()` has been eliminated and replaced by an access to the field `f` which is `public`.

```
class A {
 public static void main ( String[] args ) {
  try {
       B b = new B();
       b.m(); // Call site raises IllegalAccessError
  } catch ( java.lang.IllegalAccessError e ) {
       System.out.println (``ILLEGAL ACCESS'');
  }
 }
}
```

(a). Caller method before inlining

```
class B {
 public int f = 5;
 private void m() {
  System.out.print(f);
 }
}
```

(b). Callee method (private)

```
class A {
 public static void main ( String[] args ) {
  try {
         B b = new B();
         System.out.print ( b.f );
  } catch ( java.lang.IllegalAccessError e ) {
          System.out.println (``ILLEGAL ACCESS'');
  }
 }
}
```

(c). Caller method after inlining

Figure 3.4: An example of a call site violating Rule 4

In this example we can see that a `java.lang.IllegalAccesserror` would be raised when `B.m()` is invoked from the call site `b.m();` in method `main` in class `A` (as the method `m()` declared in class `B` is `private`). The method `m()` in class `B` accesses the `public` field `f`. After inlining is performed in method `main` as shown in Figure 3.4(c) the call to method `m()` (which was the source of the `java.lang.IllegalAccessError` in the original program) has been eliminated. Furthermore the access to the field `f` has been shifted to class `A`, but as the field `f` is `public`, the access to field `f` is not illegal. Thus the `java.lang.IllegalAccessError` is no longer raised in the inlined program whereas it was being raised in the original program, and this alters the semantic behaviour of the program as the error was being caught and there is a call to `System.out.println()` that is in the error handler code.

Most of the implementations of the Java Virtual Machine (e.g. Sun Microsystem's implementation) follow a "lazy" linking model. The Java Virtual Machine can check for errors (illegal access errors or others) using one of two possible schemes. In the first scheme, the JVM would check all the accesses in the method that is being linked before successfully linking the method. If there were any errors detected by the Java Virtual Machine at the linking stage, then an error would be raised, and linking would not succeed. This is the "early" linking model. The other linking scheme that could be used by the JVM is to perform the checks for errors as a result of an instruction only at the time of executing it. Thus a method that contains an instruction that might result in an error, would be successfully linked in. However before the instruction that results in the error is executed, the check would be performed and an error would be raised. It must be understood that the instruction that results in an error might never get executed if it is not along a control flow path that gets executed. If it is not executed, then the error would not be raised and therein lies the difference between the two schemes. The checks for errors are done at a later stage and only when required (at the time of execution) in the second scheme, and hence the name lazy model. The lazy scheme imposes less stringent restrictions as some programs that might raise an error using the early scheme might not do so if the lazy scheme is used. We will try to detect the illegal access errors that might be raised in the program statically; thus we would detect those illegal access errors that would be raised if the Java Virtual Machine followed an early linking model (which being more strict than the lazy model, means that we would detect all possible errors that might be detected by a Java Virtual Machine that follows either linking model).

Checks for Illegal Access Errors for method accesses: We referred to the Java Virtual Machine Specifications in order to check the exact conditions in which an illegal access error might be raised. It can be determined statically whether a particular method invocation instruction could result in an illegal access error. The checks to detect an illegal access are performed on the method whose signature appears in the invocation instruction. Note that the method referred to in the method signature may not be the method that is actually invoked at run time, but still it is the method whose access flags are checked.

Given an invocation instruction of the form *invokeinstruction(* `m` *)* occurring in a class C_{CALLER}, an illegal access error can be raised in each of the following 3 cases :

1. If the method `m` is declared to be `private`, and if it is invoked from any class other than the one in which the method `m` is declared.

2. If the method `m` has `default` access (not `private`, `protected`, or `public`), and if it is accessed from any package other than the package containing the declaring class.

3. If the method `m` is `protected`, an illegal access error would be raised if either or both of the following conditions is violated :

(a). The method `m` must be either a member of C_{CALLER} or a member of a superclass of C_{CALLER}.

(b). The class of the receiver object must be C_{CALLER} or a subclass of C_{CALLER}.

The checks specified in condition 3(b) are slightly different from those in the other other conditions because they involve the class of the receiver object. The class of the receiver object is inferred by the Java Virtual Machine by performing a simple dataflow analysis. According to the Java Virtual Machine specification it obtains the class for an object at control flow merge points by using the class hierarchy to obtain the least common superclass of the classes associated with the object along different control flow paths. We perform the checks in condition 3(b) using the type for the receiver object that is inferred by the type inference algorithm in Jimple. The type inference algorithm in Jimple assigns types to each local in the Jimple representation of a method. We can use the type assigned to a local as a good estimate of the class of the object (as would be inferred by the Java Virtual Machine).

The type inference algorithm in Jimple does not necessarily produce the best solution for the class of the object represented by a local, but instead attempts to assign a type that satisfies all the constraints imposed by the statements using the local. Note that the type assigned to the local by the type inference algorithm in Jimple might therefore be higher up in the class hierarchy than the class of the object (represented by the local) inferred by the Java Virtual Machine. This is because the inference algorithm used by the Java Virtual Machine is guaranteed to produce the most precise estimate for the class of the object (it must use the least superclass at control flow points where it has to merge classes).

Since the typing algorithm in Jimple always infers a type that is either a superclass of, or the same as the class obtained by the inference algorithm in the Java Virtual Machine, any illegal error arising because of condition 3(b) that would be detected by the Java Virtual Machine would also be detected by us if we use the Jimple type of the receiver. This is because if the class of the receiver object inferred by the Java Virtual Machine was neither the current class, nor any subclass of the current class, then it must be either a strict superclass of the current class or be completely unrelated in the class hierarchy to the current class. If it is completely unrelated to the current class in the class hierarchy then the type obtained through the typing algorithm in Jimple would also be completely unrelated to the current class as the class hierarchy used by both inference algorithms is identical. If the class inferred by the Java Virtual Machine is a strict superclass of the current class, then the type inferred by the Jimple type inference algorithm would also be a strict superclass of the current class, as the type inferred by the Jimple type inference algorithm must be at the same level or higher up in the class hierarchy as the class obtained by the inference algorithm in the Java Virtual Machine.

Thus, we have shown that if an illegal access error would be detected by the Java Virtual Machine, then we would also detect such an error using our scheme. Note that our scheme might regard a legal access as an illegal access; in such cases we are being conservative but correct. We have observed that such cases occur extremely rarely in practice in the benchmarks that we have tested.

Rule 5 : *If the target method contains any accesses to classes, methods, or fields that result in illegal access errors, then the errors must still be raised as a result of inlining the target method into the current method.*

We refer to the Java Virtual Machine specifications to detect if a class, method or field access from the target method is illegal. The checks for determining whether a field or class access is illegal are nearly identical to the checks we have explained for method access in the discussion of the previous rule. If we find an illegal access from the callee method $m_{ToBeInlined}$, we check if that access would result in an illegal access error if it was made from the caller method $m_{InlinedInto}$ (into which inlining is being done). If the access would no longer be illegal after inlining then the method is not inlined. We show a simple example of a method which cannot be inlined because it violates Rule 5 in Figure 3.5. The access to method `m2()` declared in class `B` is illegal when made from class `C` (which is in a different package). But after inlining is performed, the access to method `m2()` is moved to class `A`, and the access is no longer illegal as classes `A` and `B` belong to the same package.

Rule 6 : *It must be legal to access all the classes or class members (fields and methods) that are accessed from the method that is the target of the invocation instruction, from the current method. Modifiers*

```
package Package1;

 public class A {
  public static void main ( String[] args ) {
   C c = new C();
   c.m1();
  }
 }
}
```

(a). Class A before inlining

```
package Package1;

 public class B {
  public int f = 5;
  void m2() { }
 }
```

(b). Class B

```
package Package2;

 public class C {
  public void m1() {
   try {
        B b = new B();
        b.m2(); // Raises Illegal Access
       }
   catch ( java.lang.IllegalAccessError e ) {
        System.out.print(``ILLEGAL ACCESS'');
       }
  }
 }
```

(c). Callee method m1() in class C

```
package Package1;

 public class A {
  public static void main ( String[] args ) {
   C c = new C();
   try {
        B b = new B():
        b.m2();  // No Illegal Access
       }
   catch ( java.lang.IllegalAccessError e ) {
        System.out.print(``ILLEGAL ACCESS'');
       }
  }
 }
```

(d). Class A after inlining

Figure 3.5: An example of a method violating Rule 5

of class members can be changed in order to ensure that no illegal access errors occur, but this is subject to the restriction that any illegal access errors that were being thrown in the original program must still be thrown.

Just as it is required to preserve accesses that resulted in illegal access errors in the original program it is also necessary to ensure that accesses that were legal in the original program do not become illegal as a result of the code duplication/motion that occurs during inlining. This is a real concern while inlining because there are many accessor (`get()`) and mutator (`put()`) methods in object oriented programs that access private fields. Inlining these small methods would not be allowed if the class into which inlining is being done is not the same as the class in which the private fields are declared. We adopt an aggressive inlining strategy that includes changing the modifiers of classes, methods, and fields in order to allow us to perform inlining if possible. One of the options that might seem feasible is to change all the modifiers to public so that we are free to inline wherever we can. But changing modifiers in this way, though simple, might result in some illegal access errors (that were being raised in the original program) no longer being raised. Moreover it is preferable to optimize the original program without changing it any more than is really required. We now explain how we achieve these goals while inlining.

In order to ensure that all the illegal access errors in the original program are still preserved in the optimized program after we have changed modifiers and performed inlining, we make one pass over each method in the call graph even before we begin examining methods to perform inlining. The purpose of this pass is to check each access in each method and fix the maximum extent to which we are allowed to change modifiers of each class, method, and field subject to the constraint that all illegal access errors in the original program are still raised. For example, if a `private` field was being accessed from some class other than its declaring class, then we would note that we should not change the modifier of the field to anything less restrictive. In the absence of any constraints (arising out of illegal accesses) attached with a particular class, method, or field we note that we can change the modifier to `public` if required. In this manner this pass fixes the extent of freedom we are allowed while changing modifiers.

When we are considering a method for inlining we check each access in the target method, and find out the extent to which we need to change modifiers in order to inline without introducing any new illegal accesses. Thus for each class, method and field accessed from the target method, we fix the modifier required in order to successfully inline the method. Note that this required modifier is fixed keeping in mind that we should only modify the original program to the extent that is necessary in order to inline. Thus a field that is `private`, would only have its modifiers changed to default if an access to it was being inlined into a class in the same package (other than the declaring class of the field). There is no need to make the field `public` in this case and we do not attempt to do so. If however we are not allowed to change the modifier to the extent that is required to inline, then we cannot inline the method at this call site. We have fixed the maximum extent to which we can change modifiers in the pass we explained before.

Before changing the modifier of a method `m()` in class `C`, it is necessary to ensure that any method that overrides the method in subclasses of class `C` can also have its modifiers changed to the same extent (or less restrictive) as method `m()`. The Java Virtual Machine imposes the restriction that methods cannot be overridden to be more private in subclasses than they are in the superclass. This means that before inlining a callee method into the caller method, if any method `m()`'s modifiers need to be changed, then the extent of freedom that we are allowed in changing the modifiers of the method `m()` and *all* overriding methods must be considered (instead of just method `m()`). If even one overriding method of `m()` cannot have its modifiers changed to the extent required, then the modifiers of method `m()` would not be changed. If the modifiers of method `m()` and all its overriding methods can be changed to the required extent, then they are changed simultaneously to enable inlining to occur.

Note that changing modifiers within a class can result in problems if the program used serialization (the calculated SUID could change). This could be handled by adding an explicit SUID field into the classfile. In our implementation, we do not change the modifiers of classes (or their class members) that can be serialized (implement `java.io.Serializable` or `java.io.Externalizable`).

Rule 7 : *It must be safe to move all of the invokespecial instructions in the target method to the callee method without changing the behaviour of the program.*

We also need to check if the invokespecial bytecode instructions in the target method can be safely inlined into the current method without changing the semantic behaviour of the program. In order to understand why we need to be careful about duplicating/moving invokespecial instructions, we refer to the following portion of the Java Virtual Machine specification for the invokespecial bytecode :

After resolving the invokespecial instruction, the Java Virtual Machine determines if all the following conditions are true for the method m() whose signature appears in the instruction :

- *The name of the method m() is not <init>, an instance initialization method.*
- *The method m() is not a private method.*
- *The class of the method m() is a superclass of the class containing the method in which the invokespecial instruction is present.*
- *The ACC_SUPER flag is set for the class containing the method in which the invokespecial instruction is present.*

If all four conditions are true, the Java Virtual Machine selects the method (to invoke) with the identical descriptor in the closest superclass, possibly selecting the method just resolved. If even one of these four conditions is not true, then the Java Virtual Machine selects the method that it has just resolved as the one that is to be invoked.

Thus the invokespecial instruction might result in the Java Virtual Machine performing a method lookup in case all the four conditions mentioned above are satisfied. Note that this method lookup is entirely dependent on the class where the invokespecial instruction occurs, rather than the class of the receiver (as in the case of virtual method calls). This makes the actual location of the invokespecial instruction in the code critical in how it behaves at run time. This also implies that we have to be careful while moving invokespecial instructions while inlining, to avoid altering the behavior of the program.

We adopt a simple strategy that is based on the assumption that the Java Virtual Machine rarely needs to perform method lookup while executing invokespecial instructions. The method lookup is only required when a method is invoked using a statement like `super.m();` in the source code. In practice, programmers seldom invoke methods using the `super` keyword, and so most invokespecials are usually for constructors or `private` methods. We have chosen not to inline methods that contain an invokespecial instruction that might require method lookup either originally or after inlining. We have made this choice because we feel that it does not impact the number of methods inlined significantly, but simplifies the checks required to be done. We consider a particular invokespecial instruction capable of requiring method lookup at run time if it satisfies all four of the conditions that we have mentioned earlier.

We then perform two checks for each invokespecial instruction :

(a). If the invokespecial instruction might require method lookup in the original program.

(b). If the invokespecial instruction might require method lookup in the program once the instruction is (possibly) duplicated and moved as part of inlining.

If either or both of the above two conditions is true then we decide not to inline the method containing the invokespecial instruction. If exactly one of the two conditions is true, then the invokespecial instruction required method lookup originally/after inlining, but not after inlining/originally. If both the above conditions are satisfied, then method lookup was required both originally and after inlining but since the location of the invokespecial instruction has changed, the result of the method lookup might be different in the two cases. Note that it is possible to check if the result of the method lookups are the same, but we have chosen to avoid this overhead. We only inline methods in which none of the invokespecial instructions require method lookup both before and after inlining.

Note that it is illegal (according to the Java Virtual Machine specifications) for an invokespecial instruction to be used to invoke a method that is not `private`, or a constructor, or a method declared in a superclass of the class where the invokespecial bytecode occurs. Thus an invokespecial instruction that was originally invoking a `private` method `m()` might result in a verification error after inlining, if the modifier of method `m()` was changed during inlining. We do not attempt to change the modifiers of a `private` method if we detect an invokespecial instruction invoking it.

3.1.5 Inlining Criteria

Once it has been determined that a method invocation instruction can be inlined based on the inlining safety rules, the next step is to check if it would be useful to inline it. If the invocation instruction is not an important factor in the overall execution time of the application, then the potential benefit of inlining it might not be worth the cost of actually inlining it. Therefore we have attempted to come up with some heuristics relying on static, compile-time characteristics of the application that would help us in deciding whether we should inline a particular method call or not. Our sole aim while making this decision is to improve performance to the maximum extent possible. It should be obvious that the criteria we mention here may not be the same if the aims of performing inlining are different (see section 3.1.1 for different uses of inlining). The complexity of inlining a large method is not a factor as our implementation can successfully inline methods that are relatively large.

We now specify the characteristics of the application that our heuristics are based on :

1. Number of statements in the callee method.
2. Number of statements in the caller method.
3. Number of locals in the callee method.
4. Number of locals in the caller method.
5. Number of invocation instructions in the callee method.
6. Number of loops in the caller method.
7. Whether the caller method is recursive or not.

We now explain the impact of each of these factors.

1. Number of statements in the callee method : If the callee method is very small then it is expected to be beneficial to inline calls to it. This is because the time to execute the method invocation instruction would be a significant overhead in the overall time required to execute the method call. Thus eliminating the invocation instruction is likely to lead to significant benefits if the method call was executed frequently. Conversely, if the callee method had a large number of statements, then it is expected to be relatively complex and the invocation instruction itself is unlikely to be the main overhead in the method call.

2. Number of statements in the caller method : Our intent in choosing this program characteristic is to arrive at some measure of the complexity of the caller method and how this complexity is being affected by inlining. There are some overheads associated with increasing the size of a method beyond certain limits, like the cost of the extra locals (soon to be explained). If the method was a simple one to start with, then we need to be careful that inlining is not responsible for making it very complex. To this end we might decide to stop inlining in a particular caller method if the number of Jimple statements has crossed a certain threshold number. Another possibility is to stop inlining when the size of the caller method after inlining has crossed a certain multiple of the size of the original uninlined method, even if there are further inlining candidates.

3. Number of locals in the callee method : If the number of locals in the callee method is more than a certain threshold number, then we should not inline the method. It is expected that a method having many locals in the Jimple representation is reasonably complex since redundant locals are removed by the standard analyses in Jimple. Apart from the fact that eliminating the method invocation instruction might not lead to that much improvement if it is complex method, there is also another interesting issue specific to Java bytecode that needs to be kept in mind.

In Java bytecode, there are different instructions to load/store objects and integers to and from the execution stack. Some of these instructions `aload_0`, `aload_1`, `aload_2`, `aload_3`, `iload_0`, `iload_1`, `iload_2`, `iload_3`, `astore_0`, `astore_1`, `astore_2`, `astore_3`, `istore_0`, `istore_1`, `istore_2`, `istore_3` are 1 byte in length. There are also other kinds of the load/store instructions that take 2 or more bytes (more than 2 bytes if used in conjunction with the `wide` bytecode). The rationale behind having the 1 byte load/store instructions is to provide fast access to the locals occupying the first few local slots of the method. It is beneficial to put the most accessed locals in the first few local slots so that the JVM can load/store these important locals with minimal overhead. Fast access to these locals might be achieved in different ways depending on whether the JVM is an interpreter, or a Just In Time (JIT) compiler. An interpreter might have precomputed indices for the important locals into the data structure in which the locals are stored. This might enable it to access these locals faster than the other locals for which it may have to do some computation to arrive at the index. Also in the case of the 1 byte loads/stores there are fewer bytecodes to interpret than the longer instructions. A JIT might be storing these important locals in machine registers to enable faster access over the other locals that might be in memory. In that case the difference in access times between the "fast" locals and the other locals is likely to be considerable.

Inlining a method introduces more locals into the caller method; the locals in the callee method get cloned and invariably occupy local slots in the caller method that have to be accessed by the 2 byte load/store instructions. This is true whether the callee method is a relatively simple method (few locals) or a complex one (many locals). However the reason why we do not want to inline large methods is the following. In the case of inlining small methods the overhead incurred by the cloned locals getting assigned to the "slow" local slots in the caller method (whereas they were in the "fast" local slots in the callee method) is offset by the benefit of eliminating the method invocation instruction. In the execution cost of the method call the loads/stores are an insignificant proportion compared to the invocation instruction. In the case of large callee methods, this might not be the case. It is very likely that the large method does some computation intensive task and the execution time of the eliminated invocation instruction is not a significant factor in the execution time of the method call. Moreover in a complex method, the time spent in accessing locals might be significant, and the overhead of having some cloned locals in "slow" local slots as a result of inlining might be enough to offset the relatively little gain of eliminating the invocation instruction.

In summary we are making the following points. Method inlining has a harmful side effect in that locals that were in "fast" local slots get assigned to "slow" local slots when they are cloned and added to the caller method. However this harmful effect is expected to be insignificant compared to the benefit of eliminating the invocation instruction for small methods, whereas it might be enough to offset the benefits of inlining in the case of larger methods.

4. Number of locals in the caller method : We might decide to stop inlining into a particular method if the number of locals becomes greater than a certain threshold value. One possibility for such a threshold value might be 256, as some locals would have to be accessed using the `wide` bytecode instruction as the number of locals becomes greater than what can be captured in 1 byte. It needs to be mentioned that the number of locals in the method in the Jimple representation while we are inlining is not the same as the number of local slots that would be there in the bytecode produced by Jimple for the inlined method. This is because before we emit bytecodes for the inlined method we perform passes over the Jimple code that do local packing and code cleanup, which might reduce the number of locals considerably.

5. Number of invocation instructions in the callee method : This is a characteristic that offers clues about the nature of the callee method. If the number of invocation instructions is large, then it is unlikely that inlining the method would be beneficial because in all likelihood, the method is quite complex, and performs significant computation.

6. Number of loops in the caller method : The number of loops in a methods is in general a very good indication of where the program might be spending most of the time in computations. By identifying the methods that have the most number of loops we might be able to perform optimizations at the places in the program that really matter. This might be especially useful for an optimization like inlining which is effective only when it is performed selectively. Thus we might choose to inline only at call sites within loops, and in order to try to maximize the gains, we can relax the other restrictions on callee method size etc. so that maximal inlining can be done at the important sites. We use a simple algorithm for detecting loops in the Jimple code which is based on detecting `IfStmt`'s that branch backwards. We present results for the case when inlining is done only in caller methods that have at least one loop.

7. Whether the caller method is recursive or not : This is also a characteristic that can be used to determine if a particular method is a "hot spot" in the program, and holds the key to better performance. We might choose to relax some of the other inlining restrictions in order to inline maximally in a recursive method. Note that we can only determine if a method is possibly recursive at compile-time as we use the conservative call graph, which is not precise at some virtual method call sites.

3.1.6 Inlining Orders

We have discussed the criteria that can be used to make decisions on whether or not to inline a particular method call. However in order for inlining to be effective we need to fix some criteria that would determine the order in which methods should be considered for inlining. Inlining method calls within certain methods first, could be extremely crucial in improving performance. In this section we discuss some of the inlining orders we have experimented with. Each order is described in terms of assigning inlining priorities to methods. The higher the priority of a method, the earlier it should be inlined.

1. Bottom Up Order : In this order methods that are at the bottom of the call chain (leaves in the call graph) are assigned maximum priority, and the priorities decrease as we move up the call chain. This order is expected to be extremely effective in applications that have many small methods or in applications with relatively short call chains. This order is not likely to be useful in applications with many large methods or long call chains. Performing inlining in bottom up order without any restrictions on the size of the callee/caller methods gave us valuable insight into its limitations/effectiveness. In applications with many small methods, inlining maximally could be quite beneficial, as the invocation instructions eliminated were the main overhead in the original program. The harmful effects of introducing more locals as a result of inlining are not as pronounced because the methods being inlined are small in size. The effects are also not very harmful in applications with short call chains as the increase in locals in caller methods is not expected to be very high because inlining is done at fewer levels in the call chain. The overhead due to the cloned locals is high in the case of applications with predominantly large methods, and this may result in degraded performance. The bottom up order is most useful when used in conjunction with the restrictions on callee/caller method size and the locals limit. In such cases, inlining would be performed extensively in methods at the end of the call chain. Methods near the top of the call chain would have little or no inlining done in them as most of the callee methods would have grown large enough (as a result of inlining earlier) that they no longer satisfy the restrictions on size and locals limit that would permit inlining. Still since every control flow path must terminate in a method at the end of some call chain, bottom up order is a good scheme.

2. Top Down Order : This order of inlining is the exact opposite of bottom up order. Methods near the top of the call chain are assigned maximum inlining priority, and the priorities decrease as we go down the call chain. This order is likely to be quite beneficial even in the complete absence of any restrictions on the size of the callee methods being inlined. It must be noted that in this scheme, a particular callee method's code never gets inlined into any method other than the caller method.

Methods near the top of the call chain never get code from methods near the bottom of the call chain because inlining is done first for methods at the higher levels in the call graph. Clearly this order leads to lesser code explosion than bottom up order, and is therefore quite effective when inlining is done maximally even in applications that have many large methods or long call chains. It may not be as effective as bottom up order for applications with many small methods or short call chains.

3.1.7 Our Static Inlining Strategy

We have attempted to come up with one static inlining strategy that we apply on every input application. The main requirement of this strategy is that it should be adaptive enough to be equally effective for completely different classes of applications. The strategy we present combines many of the ideas we have discussed earlier. The most important point in our strategy is that we only inline at the call sites that are determined to be important. We chose not to inline in bottom up order because of the high rate of increase of the number of locals in this case. As a result of most locals being in slow local slots in the class file, all the benefits from inlining were lost and there was a significant slowdown (almost 40 percent) in some cases. Top down order was felt to be too restrictive in that we really wanted a strategy that would be aggressive at important call sites and not try to increase the code size too much on account of inlining at unimportant call sites. It is imaginable though, that one might still want to inline following these orders if the aim of performing method inlining was not performance improvement. We have presented our algorithm to detect important call sites in Figure 3.6 and our inlining strategy in algorithmic form in Figure 3.7.

Our inlining strategy starts by identifying call sites in recursive methods and call sites that are inside loops. These call sites would clearly be executed frequently but it also needs to be realized that all the methods that could be called from these call sites might also be executed many times. Thus we recursively include all the methods attached to the important call sites, and add all the call sites within these methods into our list of important call sites. While adding an important call site and sites along some call chain beginning at the call site, it is worth pointing out that the call sites deeper down in the call chain are added first to the list of important call sites and the call sites higher up in the chain are added afterwards. Thus call sites lower down in the call chain would be considered for inlining first. This is equivalent to inlining in bottom up order but only at the important call sites.

```
void getImportantCallSitesAccessedFrom ( method m ) {

  List impSitesAccessedFromMethod = new ArrayList();

  List impSitesAccessedDirectlyFromMethod = new ArrayList();

  if ( m is a recursive method )
  add all the call sites in m to the list impSitesAccessedDirectlyFromMethod;
  else if ( m contains loops )
  add the call sites inside loops to the list impSitesAccessedDirectlyFromMethod;

  List impSitesAccessedIndirectlyFromMethod = new ArrayList();

  for ( each call site cs in impSitesAccessedDirectlyFromMethod )
  {
    List reachablemethods = getReachableMethodsInReverseTopologicalOrder(cs);

    /* getReachableMethodsInReverseTopologicalOrder() returns the list of
       methods that could be called along all call chains starting at cs.
       Note that it returns the methods that are lower down in the call
       chain at the head of the list and methods are higher up in the
       call chain at the end of the list. */

    for ( each method meth in reachablemethods )
    add all the call sites in meth to the list impSitesAccessedIndirectlyFromMethod;
   }

   add all the call sites in the list impSitesAccessedIndirectlyFromMethod to the
   list impSitesAccessedFromMethod;
   add all the call sites in the list impSitesAccessedDirectlyFromMethod to the
   list impSitesAccessedFromMethod;

   return impSitesAccessedFromMethod;
}
```

Figure 3.6: Locating important call sites to attempt inlining

There need to be limitations though, on the amount of code explosion allowed, even at the important call sites. Thus we prevent any further inlining in a particular method if the number of statements in the method has increased by a certain factor as compared to the original untransformed method. The code increase factor can be specified by the user in our implementation. The default value for the code increase factor is eight (fixed after experiments), if the user does not specify a particular value. There is also a strict bound on the allowable size of a method after inlining (10000

```
void performInlining ( List methods ) {

  Set importantCallSites = new HashSet();

  for ( each method m in methods )
  {
    List importantSitesAccessedFromMethod = getImportantCallSitesAccessedFrom(m);
    add all the call sites in importantSitesAccessedFromMethod
    to the set importantCallSites;

    /* Note that if a call site is determined to be important because
       of accesses to it from more than one method, then it is not
       considered twice as importantCallSites is a set.  */

  }

  for ( each call site cs in importantCallSites )
  {
    if ( cs has exactly 1 callee method )
    {
      Method declaringmethod = cs.getDeclaringMethod();
      Method calleemethod = cs.getCalleeMethod();

      if ( satisfiesSafetyCriteria(cs) )
      {

        if ((declaringmethod.size() < (declaringmethod.originalsize())*EXPANSION_FACTOR)
          && (declaringmethod.size() < MAX_ALLOWED_SIZE ))
        {

          if (calleemethod.size() < AVG_MTHD_SIZE)
          {

           InlineMethod(cs, calleemethod);

          }
        }
      }
    }
  }
}
```

Figure 3.7: Our static inlining algorithm

Jimple statements) so that the method does not grow beyond the limit imposed by the Java Virtual Machine specification. Thus, inlining into a particular method is stopped if it has grown to a size greater than the limit imposed by the code increase factor or 10000 Jimple statements.

Since we are preventing any further inlining after the code size has reached a certain limit, it is important that the increase in code is a result of inlining as many call sites as possible. For example, if one of the call sites was a call to a large method then the code explosion in the caller as a result of inlining at that one call site might have increased the size of the caller past the threshold. This would result in none of the remaining call sites being inlined; this might be significant if there are many call sites that call small methods, as many such calls could have been eliminated if we had not inlined at the call site that resulted in code explosion. Thus we inline only at those call sites where the callee method is sufficiently small in size; the size we choose as a threshold is the measured average value for the number of Jimple statements in methods in the application being optimized.

3.1.8 Profile Guided Inlining

We have also implemented an option to enable profile guided inlining in our implementation. Profile guided inlining only considers the call sites that were executed frequently in a profile run for inlining. The input to the profile guided inliner is a file which contains the unique IDs for call sites that were executed during the profile run, along with the frequency of execution of each call site. The inliner considers the call sites for inlining in decreasing order of frequency of execution, thus inlining at the most important call sites first. By only inlining at the call sites that are known to have a performance impact, we reduce the amount of code explosion that typically occurs when inlining is performed based on static criteria. Also most of the program characteristics we mentioned earlier (like number of locals in a method after inlining etc.) are not altered significantly. In some sense, the improvement in performance as a result of inlining based on the profile is the maximum that can be achieved as a result of this optimization. The comparison of the effect of profile guided inlining with inlining based on static criteria indicates the effectiveness of our static criteria in identifying the important call sites to inline.

In Chapter 2, we explained our profiling strategy when the aim was to identify the run time targets of each call site. Our profiling strategy in identifying the call sites that are executed frequently is similar. We insert profiling code into each method at the Jimple level and then use the Soot framework to generate classes with profiling code in them. The code that is inserted before each call site is a call to a static method in a special profiler class created by us. The static method takes as parameters the unique ID of the call site and keeps track of the number of times each call site was executed. This static method also writes the call site ID and the frequency count for each call site periodically into a results file. When the program being profiled has executed for a significant amount of time, we can assume that the frequency counts in the results file mirror the actual behavior of the program (i.e. "hot spots" in the program can now be easily identified).

Chapter 4

Experimental Results

In this chapter we present and discuss the experimental results that we have obtained. Our results can be grouped into two distinct categories :

1. The impact of the static analyses presented in Chapter 2 in improving the precision of the call graph for an application.

2. The effectiveness of method inlining, the optimization we presented in Chapter 3, in improving run time performance.

We have performed our experiments on a set of 15 benchmark programs drawn from five different source languages, namely, Java, ML, Ada, Eiffel and Pizza, all of which have compilers that produce bytecode. The benchmark characteristics of particular interest are shown in Table 4.1.

4.1 Benchmark Characteristics

The 15 benchmarks are grouped based on the source language and brief description of each benchmark's functionality is also given.

4.1.1 Java

There are seven Java benchmarks, two of which are from the Sable benchmark set while the rest are from the SPECjvm benchmark suite. The `sablecc` benchmark is a compiler front end generator written in Java[3], and `soot` is an earlier version of our compiler framework[1]. The five SPECjvm benchmarks include `raytrace` which is a graphics raytracer, `jess` which is an expert shell system based on NASA's CLIPS expert system, `compress` which is a compression program based on a modified Lempel-Ziv method, `db` which is a database application, `mpegaudio` which is an obfuscated commercial application that decompresses audio files conforming to the ISO MPEG Layer-3 audio specification, `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.

4.1.2 Eiffel

The `illness` benchmark, which simulates the spread of disease among a population was compiled with the SmallEiffel compiler[4] and the benchmark comes from the SmallEiffel benchmark programs distributed with the compiler. The SmallEiffel compiler does some whole program analysis and produces relatively optimized code.

4.1.3 Ada

The `rudstone` benchmark is a large Ada floating-point intensive benchmark that was derived from a satellite ground control system, and it was compiled using Appletmagic(tm)[38].

4.1.4 ML

There are three ML benchmarks which come from the Standard ML of New Jersey benchmark set. The benchmark `lexgen` reads a specification of a lexer for SML, and generates the SML code for the lexer, `ray` is a graphics ray tracing program, and `nucleic` solves an anticodon problem. All three benchmarks were compiled using the MLJ compiler[5], which actually performs whole program analysis.

4.1.5 Pizza

The `pizza` benchmark is the Pizza compiler[6] written in the Pizza programming language, compiled with the Pizza compiler.

The statistics in Table 4.1 provide an insight into the nature (the extent of object orientedness 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. 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. Note that the Ada, Eiffel and ML benchmarks all appear to be very non object-oriented since the maximum depth of their hierarchies is 2, and none of them have any interfaces in the benchmark part of the code.

Benchmark		**# Stmts**		**Hierarchy**				**Classes and Interfaces**				
				avg. depth		**max. depth**		**library**		**bench. only**		**whole app.**
lang.	name	whole app.	bench. only	whole app.	bench. only	whole app.	bench. only	class	int.	class	int.	(total)
java	`sablecc`	68575	24621	3.2	2.3	6	5	308	44	299	13	664
java	`soot`	63506	33396	3.3	2.1	6	4	186	11	498	34	729
java	`_205_raytrace`	49239	5347	3.0	1.3	6	3	307	44	35	1	387
java	`_202_jess`	56163	11137	2.8	1.3	6	3	316	44	157	4	521
java	`_201_compress`	46619	2727	3.0	1.1	6	2	307	44	22	1	374
java	`_228_jack`	55107	11215	3.0	1.6	6	3	307	44	63	5	419
java	`_209_db`	49876	3002	3.0	1.0	6	1	309	44	14	1	368
java	`_222_mpegaudio`	56744	10923	3.0	1.4	6	4	307	44	54	9	414
java	`_213_javac`	69585	25304	3.5	3.2	8	7	310	44	178	5	537
eiffel	`illness`	29568	1372	3.2	1.8	6	2	174	10	11	0	195
ml	`nucleic`	33096	4900	3.1	1.6	6	2	174	10	47	0	231
ml	`lexgen`	33397	5201	3.1	1.4	6	2	174	10	67	0	251
ml	`ray`	34186	3721	3.1	1.6	6	2	178	10	83	0	271
ada	`rudstone`	75250	31413	2.9	1.2	7	2	312	44	141	0	497
pizza	`pizza compiler`	73130	42805	3.0	1.7	6	5	188	11	207	11	417

Table 4.1: Benchmark Characteristics

4.2 Conservative Call Graph Characteristics

Table 4.2 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 4.2 is divided into two distinct parts. We present conservative call graph and call graph improvement statistics for all our benchmarks in this section.

Name	Whole Application							Benchmark Only						
	$\|N\|$	Call Sites			Edges			$\|N\|$	Call Sites			Edges		
		mono.	pot. poly.	total	mono.	pot. poly.	total		mono.	pot. poly.	total	mono.	pot. poly.	total
`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
`raytrace`	1729	6582	377	6956	6576	2591	9167	207	2037	12	2049	2037	46	2083
`jess`	2230	8871	467	9338	8865	3804	12669	627	4209	89	4298	4209	994	5203
`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
`db`	1615	5688	393	6081	5682	2713	8395	80	1090	26	1116	1090	110	1200
`mpegaudio`	1828	6127	404	6531	6121	3072	9193	311	1602	38	1640	1602	179	1781
`javac`	2821	10570	1276	11846	10564	13707	24271	1188	5933	848	6781	5933	10306	16239
`illness`	746	2494	164	2658	2494	1318	3812	56	144	1	145	144	9	153
`nucleic`	800	3500	172	3672	3500	1353	4853	103	1149	6	1155	1149	32	1181
`lexgen`	916	3633	200	3833	3633	1438	5071	196	1250	22	1272	1250	67	1317
`ray`	973	3203	195	3398	3203	1505	4708	206	713	19	732	713	94	807
`rudstone`	1707	6014	358	6372	6004	2311	8315	207	1637	1	1638	1633	7	1640
`pizza`	2660	13729	799	14528	13729	6024	19753	1756	11115	577	11692	11115	4069	15184

Table 4.2: Conservative Call Graph Characteristics

4.2.1 Conservative Call Graph for Whole Application

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 can not be called. 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.

4.2.2 Conservative Call Graph for Benchmark Only

Now consider the second part of Table 4.2, 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 `rudstone`, `illness`, and `raytrace`, whereas in benchmarks like `javac`, `soot`, or `pizza` there are many unresolved (potentially polymorphic) call sites.

4.3 Improvements over the Conservative Call Graph

We have obtained the results shown in Table 4.3 from our analyses. For each benchmark we show the result of applying rapid type analysis (RTA), declared-type analysis (DTA), and two variations of variable-type analysis VTA(1) and VTA(2). VTA(1) is the analysis as presented in Chapter 2. VTA(2) uses the result of VTA(1) to prune the conservative call graph, and then uses this pruned call graph to run the VTA algorithm again. One could imagine repeating this process until no further improvement is gained, but in practice we have found that 2 iterations works quite well.

4.3.1 Call Graph Improvement for Whole Application

Rapid type analysis (RTA) has previously been shown to give good results on complete C++ applications [12], and our study shows that this is the case for many of our Java benchmarks as well. The number of dead method nodes (whole application) removed by RTA varies between 7% of the total number of methods in the conservative call graph (for `soot`, `pizza`) to about 51% (for `compress`). We expect most of the improvement to come from removing dead methods in library classes. 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 benchmark application would only instantiate a few of the subclasses of O and so most of the edges that are present in the CHA call graph are not actually needed. This is exactly the sort of information that RTA can find. This expected behavior is observed in practice as there are a greater proportion of methods removed in applications like `raytrace` and `compress` that involve many library classes as compared to the number of classes in the benchmark itself.

The next important question is to see if our new analyses, declared-type analysis (DTA) and variable-type analysis (VTA) can do even better than RTA for complete applications. The number of methods removed by DTA varied between 9% (`soot`) and 58% (`compress`), whereas the number of edges removed varied between 10% (`compress`) and 54% (`soot`). Thus DTA does shows some improvement over RTA both in terms of nodes removed (at best 7%) and edges removed (at best 9%). VTA shows a clearer improvement over RTA. The number of nodes removed by VTA(2) varied between 10% and 65% of the total number of nodes in the conservative graph. The number of edges removed as compared to RTA is greater on average by about 15% (best case improvement 19%).

We have also shown the number of call sites resolved by each analysis, and VTA does considerably better than any of the other analyses in this respect. Before discussing the improvement, we would like to clarify that the percentages given (alongside the raw data for the number of resolved call sites) are calculated on the number of potentially-polymorphic call sites (as shown in Table 4.2) that are left unresolved by CHA. As an example, for the benchmark `jack`, RTA resolved 308 of the 774 call sites (39%) that are unresolved by CHA, whereas VTA(2) resolved 730 call sites (94%). Note that the number of call sites resolved by each analysis includes call sites that are determined to have no targets (in other words, they cannot be executed as they occur in an unreachable method), as well as call sites that have precisely one target. The improvement in resolved call sites in most benchmarks is at least 15%, and in some cases (`jack`, `soot`) it is higher than 40%. DTA shows some improvement over RTA in some cases, but performs almost as badly as RTA for some of the more object oriented benchmarks (`javac`, `jack`, `soot`, `pizza`). Thus we observe that for highly object oriented applications, VTA is more effective than the other analyses.

4.3.2 Call Graph Improvement for Benchmark Only

We now discuss the results for methods/edges removed, and call sites resolved only within the benchmark methods (excluding the Java libraries). Compared to complete applications, fewer benchmark methods are removed, and this can be explained by the fact that there are fewer redundant methods in general in benchmark classes, as compared to a library (where there might be many methods that have been created for use by users of the library). The differences between the analyses are more clearly observable in the numbers for the benchmark only. It can be observed that for the benchmark only, RTA has almost the same effectiveness as CHA in most of the benchmarks that are not very object oriented. On the more object oriented benchmarks, RTA does better than CHA on the benchmark code but there is still substantial scope for improvement. DTA has roughly the same effect as RTA on the benchmark code, but VTA performs considerably better than any of the other analyses. The improvement over RTA in terms of methods removed is about 4% on average, and 10% in the best case. VTA removes about 6% more edges than RTA on average, and 18% more edges in the best case (`soot`). The differences in call sites resolved is more marked and RTA does not perform as well on the highly object oriented benchmarks as it does on the non object oriented benchmarks. VTA resolves a substantial number of call sites in the highly object oriented benchmarks (as high as 96% of virtual call sites for `jack` and 44% for `soot`). In some of the non object oriented benchmarks like `raytrace`, `rudstone`, and `compress` there are not very many virtual call sites in the benchmark that can be resolved, and so none of the analyses do particularly well on them. Note that the number of call sites resolved includes call sites with one target and call sites with no targets.

4.4 Comparison with 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). One common scenario is that one would want to perform compiler optimizations on the benchmark code alone, and leave the Java library classes unchanged. This was the main reasoning behind our decision to profile the benchmark classes only, as this

		Whole Application			Benchmark Only		
		Nodes Removed	Edges Removed	Callsites Resolved	Nodes Removed	Edges Removed	Callsites Resolved
sablecc	rapid-type	657 (17%)	4145 (11%)	407 (30%)	42 (2%)	1077 (4%)	164 (18%)
	declared-type	773 (20%)	5670 (15%)	456 (34%)	75 (3%)	1854 (6%)	192 (21%)
	variable-type(1)	867 (23%)	10723 (30%)	635 (47%)	91 (4%)	5943 (22%)	311 (34%)
	variable-type(2)	1016 (27%)	11141 (31%)	680 (51%)	92 (4%)	6005 (22%)	317 (35%)
soot	rapid-type	212 (7%)	2635 (7%)	137 (7%)	60 (2%)	1362 (4%)	38 (2%)
	declared-type	282 (9%)	4061 (10%)	172 (9%)	68 (3%)	2168 (6%)	60 (3%)
	variable-type(1)	328 (11%)	7447 (20%)	657 (37%)	89 (4%)	5027 (15%)	510 (33%)
	variable-type(2)	348 (12%)	8380 (22%)	829 (47%)	109 (5%)	5960 (18%)	682 (44%)
raytrace	rapid-type	808 (46%)	3585 (39%)	292 (77%)	15 (7%)	46 (2%)	5 (41%)
	declared-type	925 (53%)	4375 (47%)	304 (80%)	20 (9%)	55 (2%)	5 (41%)
	variable-type(1)	1026 (59%)	5200 (56%)	342 (90%)	18 (8%)	68 (3%)	5 (41%)
	variable-type(2)	1026 (59%)	5200 (56%)	342 (90%)	18 (8%)	68 (3%)	5 (41%)
jess	rapid-type	974 (43%)	4671 (37%)	346 (74%)	145 (23%)	1112 (21%)	49 (55%)
	declared-type	1096 (49%)	5823 (46%)	356 (77%)	157 (25%)	1455 (27%)	50 (56%)
	variable-type(1)	1191 (53%)	6826 (53%)	404 (86%)	162 (25%)	1552 (29%)	54 (60%)
	variable-type(2)	1191 (53%)	6826 (53%)	404 (86%)	162 (25%)	1552 (29%)	54 (60%)
compress	rapid-type	814 (51%)	3664 (45%)	293 (79%)	11 (14%)	40 (4%)	3 (50%)
	declared-type	926 (58%)	4418 (55%)	303 (82%)	16 (21%)	62 (6%)	4 (66%)
	variable-type(1)	1033 (65%)	5214 (65%)	344 (93%)	16 (21%)	70 (7%)	4 (66%)
	variable-type(2)	1033 (65%)	5214 (65%)	344 (93%)	16 (21%)	70 (7%)	4 (66%)
jack	rapid-type	820 (44%)	3763 (34%)	313 (40%)	17 (5%)	121 (3%)	21 (5%)
	declared-type	924 (50%)	4475 (41%)	323 (41%)	20 (5%)	184 (5%)	21 (5%)
	variable-type(1)	1027 (55%)	5719 (52%)	734 (94%)	21 (6%)	565 (15%)	382 (96%)
	variable-type(2)	1027 (55%)	5719 (52%)	734 (94%)	21 (6%)	565 (15%)	382 (96%)
db	rapid-type	812 (50%)	3649 (43%)	291 (74%)	12 (15%)	57 (4%)	3 (11%)
	declared-type	920 (56%)	4426 (52%)	302 (76%)	15 (18%)	84 (7%)	3 (11%)
	variable-type(1)	1002 (62%)	5168 (61%)	360 (91%)	15 (18%)	119 (9%)	23 (88%)
	variable-type(2)	1002 (62%)	5168 (61%)	360 (91%)	15 (18%)	119 (9%)	23 (88%)
mpegaudio	rapid-type	839 (45%)	3908 (42%)	303 (75%)	36 (11%)	89 (4%)	12 (31%)
	declared-type	930 (50%)	4560 (42%)	325 (80%)	46 (14%)	157 (8%)	13 (34%)
	variable-type(1)	1043 (57%)	5549 (60%)	354 (87%)	46 (14%)	180 (10%)	13 (34%)
	variable-type(2)	1043 (57%)	5549 (60%)	354 (87%)	46 (14%)	180 (10%)	13 (34%)
javac	rapid-type	823 (29%)	4516 (18%)	319 (25%)	30 (2%)	713 (4%)	30 (3%)
	declared-type	931 (33%)	5460 (22%)	337 (26%)	33 (2%)	855 (5%)	30 (3%)
	variable-type(1)	1001 (35%)	6639 (27%)	489 (38%)	35 (2%)	1136 (6%)	135 (15%)
	variable-type(2)	1001 (35%)	6639 (27%)	489 (38%)	35 (2%)	1136 (6%)	135 (15%)
illness	rapid-type	189 (25%)	1380 (36%)	104 (63%)	0 (0%)	5 (3%)	0 (0%)
	declared-type	234 (31%)	1743 (45%)	110 (67%)	0 (0%)	5 (3%)	0 (0%)
	variable-type(1)	290 (38%)	2060 (54%)	143 (87%)	5 (8%)	16 (10%)	1 (100%)
	variable-type(2)	290 (38%)	2060 (54%)	143 (87%)	5 (8%)	16 (10%)	1 (100%)
nucleic	rapid-type	194 (24%)	1390 (28%)	107 (62%)	1 (1%)	1 (0.1%)	0 (0%)
	declared-type	244 (30%)	1757 (36%)	113 (65%)	7 (6%)	6 (0.6%)	1 (16%)
	variable-type(1)	300 (37%)	2062 (42%)	150 (87%)	12 (11%)	11 (1%)	5 (83%)
	variable-type(2)	300 (37%)	2062 (42%)	150 (87%)	12 (11%)	11 (1%)	5 (83%)
lexgen	rapid-type	202 (22%)	1421 (28%)	112 (56%)	1 (1%)	1 (0.1%)	0 (0%)
	declared-type	261 (28%)	1804 (35%)	121 (60%)	14 (7%)	13 (1%)	1 (4%)
	variable-type(1)	314 (34%)	2129 (41%)	172 (86%)	18 (9%)	45 (3%)	21 (95%)
	variable-type(2)	314 (34%)	2129 (41%)	172 (86%)	18 (9%)	45 (3%)	21 (95%)
ray	rapid-type	197 (20%)	1415 (30%)	111 (56%)	1 (1%)	1 (0.1%)	0 (0%)
	declared-type	259 (26%)	1817 (38%)	119 (61%)	15 (7%)	16 (1%)	1 (5%)
	variable-type(1)	320 (32%)	2176 (46%)	165 (84%)	25 (12%)	55 (6%)	15 (78%)
	variable-type(2)	320 (32%)	2176 (46%)	165 (84%)	25 (12%)	55 (6%)	15 (78%)
rudstone	rapid-type	830 (48%)	3680 (44%)	293 (81%)	0 (0%)	3 (0.2%)	0 (0%)
	declared-type	936 (54%)	4364 (52%)	302 (84%)	0 (0%)	4 (0.4%)	0 (0%)
	variable-type(1)	1014 (59%)	4903 (58%)	337 (94%)	0 (0%)	9 (0.6%)	1 (100%)
	variable-type(2)	1014 (59%)	4903 (58%)	337 (94%)	0 (0%)	9 (0.6%)	1 (100%)
pizza	rapid-type	213 (8%)	2097 (10%)	123 (15%)	17 (1%)	643 (4%)	3 (0.3%)
	declared-type	233 (9%)	2566 (12%)	155 (19%)	20 (1%)	830 (5%)	23 (3%)
	variable-type(1)	270 (10%)	3431 (17%)	259 (32%)	32 (1%)	1388 (9%)	99 (17%)
	variable-type(2)	270 (10%)	3462 (17%)	270 (33%)	32 (1%)	1418 (9%)	109 (18%)

Table 4.3: Improvement of Call Graph over Conservative Call Graph

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.

We observed that for the non object oriented benchmarks `raytrace`, `compress`, `db`, `mpegaudio`, `illness`, `ray` and `rudstone`, almost 100% of the benchmark call sites are resolved by CHA. Thus, there is no point in considering these any further. The results for the remaining 7 benchmarks are much more interesting, and are presented in Table 4.4. For each benchmark, we provide one row for each of the analyses (CHA,RTA,DTA,VTA(1) and VTA(2)), plus one row for the result obtained by the profile. In each case we give the percentage of calls (dynamic number) in three categories. The first column gives the percentage of calls that were monomorphic. For the analyses rows this means that these calls were determined to be monomorphic by the analysis. For the profile row this means that only one method was resolved for this call site over the entire run. Note that monomorphic calls in the profile may be monomorphic for this particular run, but polymorphic for different input data. Column 2 shows the percentage of calls that have 2 or 3 targets according to the analyses and the profile. This is of interest as it is possible to optimize polymorphic calls that have only a few possible targets by introducing a switch into the code that would call the appropriate method (statically resolved) based on the class of the receiver[22]. Column 3 shows the percentage of calls that are unresolved by the analyses and have more than 3 possible targets, along with the profile number for such polymorphic calls. The final column shows the average number of methods that could be called from each call site according to each analysis, and the same measurement for calls made in the profile.

Table 4.4 shows several interesting trends. First, consider the percentage of monomorphic calls. It appears that RTA gives very little or no improvement on all benchmarks. Thus, as expected, RTA is not effective for benchmark code. Our DTA analysis also does not perform very well on the benchmark code, giving no significant improvement over RTA. However, our VTA analysis does give some improvement on all benchmarks, 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. We also observe that for the ML benchmarks `nucleic` and `lexgen`, RTA and DTA cannot resolve all calls, but

		Benchmark Only			
		Virtual Callsites with 1 target	**Virtual Callsites with 2 or 3 targets**	**Virtual Callsites with > 3 targets**	Average **per Callsite**
`sablecc`	class-hierarchy	86%	5.5%	8.5%	1.69
	rapid-type	86.5%	5%	8.5%	1.60
	declared-type	86.5%	5%	8.5%	1.60
	variable-type(1)	88.7%	5.2%	6.1%	1.42
	variable-type(2)	89.7%	6.2%	4.1%	1.30
	profile	95.5%	0.7%	3.8%	1.13
`soot`	class-hierarchy	29%	16%	55%	11.03
	rapid-type	29%	20.5%	50.5%	9.31
	declared-type	29%	20.5%	50.5%	8.49
	variable-type(1)	39%	14.9%	46.1%	6.45
	variable-type(2)	41%	18.6%	40.4%	6.2
	profile	66.6%	16%	17.4%	1.58
`jack`	class-hierarchy	86.3%	1.1%	12.6%	1.444
	rapid-type	87%	11.6%	1.4%	1.298
	declared-type	87%	11.6%	1.4%	1.158
	variable-type(1)	98.5%	1.5%	0%	1.017
	variable-type(2)	98.5%	1.5%	0%	1.017
	profile	98.5%	1.5%	0%	1.017
`javac`	class-hierarchy	65.9%	7.1%	27%	3.441
	rapid-type	65.9%	7.1%	27%	3.078
	declared-type	65.9%	12.5	21.6%	2.607
	variable-type(1)	72.2%	6.5%	21.3%	2.597
	variable-type(2)	72.2%	6.5%	21.3%	2.597
	profile	90.1%	3.6%	6.3%	1.446
`nucleic`	class-hierarchy	0%	99.1%	0.9%	2.138
	rapid-type	0%	99.1%	0.9%	2.138
	declared-type	0%	99.1%	0.9%	2.138
	variable-type(1)	99.1%	0%	0.9%	1.138
	variable-type(2)	99.1%	0%	0.9%	1.138
	profile	99.1%	0%	0.9%	1.138
`lexgen`	class-hierarchy	81.8%	18.2%	0%	1.182
	rapid-type	81.8%	18.2%	0%	1.182
	declared-type	81.8%	18.2%	0%	1.182
	variable-type(1)	100%	0%	0%	1.000
	variable-type(2)	100%	0%	0%	1.000
	profile	100%	0%	0%	1.000
`pizza`	class-hierarchy	75.5%	9.5%	15%	2.289
	rapid-type	75.5%	20%	4.5%	1.853
	declared-type	75.5%	20%	4.5%	1.798
	variable-type(1)	89%	8.8%	2.2%	1.509
	variable-type(2)	89%	8.8%	2.2%	1.509
	profile	94%	6%	0%	1.074

Table 4.4: Comparison of calls resolved by each analysis with the profile result

VTA resolves almost all the calls. This is because the inheritance hierarchy in the case of these benchmarks consists of an abstract class that contains default definitions for many methods (that raise an exception if called). Many classes directly extend this abstract class but contain redefinitions for only certain methods, and so there are 2 possible targets for a call in many cases. VTA resolves these call sites while the coarser grained analyses do not.

For 2 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 an 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 it would be difficult to pair the definitions and uses up correctly.

Another explanation for the gap is 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 such as specialization.

4.5 Time and Space Complexity of Analyses

Name	Jimple Stmts	Call Graph		Declared Type				Variable Type				Time (seconds)	
				before SCC		after SCC		before SCC		after SCC			
		$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$	DTA	VTA
`sablecc`	68575	3737	35693	7722	8273	6104	3927	25482	75280	20298	43618	13	128
`soot`	63506	2828	36981	6333	6699	5178	3784	24190	68289	19620	43416	15	207
`raytrace`	49239	1729	9167	3540	3139	2989	1931	12496	18125	10700	13329	8	54
`jess`	56163	2230	12669	4320	3943	3634	2453	15563	23695	13232	17059	9	87
`compress`	46619	1583	8000	3235	2832	2741	1745	11010	15734	9471	11461	8	44
`jack`	55107	1857	10801	3828	3474	3284	2274	14293	21361	12320	16131	11	68
`db`	49876	1615	8395	3356	2954	2801	1812	11878	16742	9780	11924	8	46
`mpegaudio`	56744	1828	9193	3696	3371	3115	2182	13416	20200	12101	15665	11	62
`javac`	69585	2821	24271	5872	6061	4741	3374	22220	54930	17019	26417	12	113
`illness`	29568	746	3812	1650	1386	1409	868	5625	8355	4851	6041	4	50
`nucleic`	33096	800	4853	1796	1538	1536	986	7815	12130	6403	8714	4	41
`lexgen`	33397	916	5071	2044	1686	1746	1049	7798	11687	6717	8762	5	41
`ray`	34186	973	4708	2178	1849	1854	1164	7227	10547	6101	7575	5	52
`rudstone`	75250	1707	8315	3609	2758	3122	1690	11956	15764	10440	11504	8	89
`pizza`	73130	2660	19753	7177	7445	6023	3856	28007	50242	17216	23390	11	102

Table 4.5: Size of Data Structures

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 4.5, we show our measurements. 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 3 times the number of nodes, and about 7 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 4.5 gives the time, in seconds, for solving the constraint graph. The interesting point is not so much the absolute time [1], but the fact that the analysis scales well, and behaves linearly in practice.

[1] 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) on a 333Mhz pentium. Thus one can safely assume that a tuned implementation will run faster by a large constant factor.

4.6 Method Inlining Results

We now explain the impact of performing method inlining on the run time performance of our set of benchmarks. We have measured the impact of method inlining on only those benchmarks (out of our set of 15 benchmarks) that execute for long enough for timing measurements to be accurate. We have shown the measurements using the JIT compiler that is part of the linux jdk1.2 release, and also using the linux jdk1.2 interpreter on a 400Mhz pentium.

Before discussing the results, we would like to briefly describe the methodology we followed for obtaining these results. All execution times in the tables are the minimum values obtained over five separate execution runs. We feel that using the minimum value is less likely to include inaccuracies induced by fluctuations in system load due to other processes. We have also observed that the timings are only accurate if measured at high CPU utilization (97% to 99%) when using the Unix *time* command.

We present results in each table for method inlining performed using the call graph obtained through class hierarchy analysis, and using variable type analysis. We have not presented results for method inlining using the call graph built by rapid type analysis or declared type analysis as we have observed that the performance impact in these cases is identical to that observed when inlining is performed after class hierarchy analysis. This is a result to be expected as the call graph characteristics (for benchmark code) built using class hierarchy analysis, rapid type analysis, and declared type analysis are very similar (see Table 4.3).

4.6.1 Automatic Method Inlining

In Table 4.6 we have shown our run time measurements using the linux JIT before and after performing automatic method inlining, while in Table 4.7 we have shown similar results for the interpreter. In columns 1 and 2 we have shown the execution times before and after inlining respectively. In column 3 we have shown the calculated speedup, and in column 4 we have shown the factor by which code size increased as a result of performing our optimization.

We observe an average speedup of 1.05 on our set of benchmarks, with a maximum speedup of 1.21 for the benchmark `compress` using the JIT. Using the interpreter, we observe an average speedup of 1.02 on our set of benchmarks, with a maximum speedup of 1.08 for the benchmark `raytrace`. Method inlining is observed to be a

	Execution Time Before Inlining	JIT results for automatic inlining: Analysis Used For Call Graph	Execution Time After Inlining	Speedup	Code Increase
compress	66.88 s	class-hierarchy	55.14 s	1.21	1.23
		variable-type	55.14 s	1.21	1.23
jess	48.17 s	class-hierarchy	45.55 s	1.05	2.01
		variable-type	45.55 s	1.05	2.01
raytrace	53.97 s	class-hierarchy	48.81 s	1.10	2.22
		variable-type	48.26 s	1.11	2.22
db	131.01 s	class-hierarchy	127.11 s	1.03	1.29
		variable-type	127.11 s	1.03	1.29
mpegaudio	53.95 s	class-hierarchy	50.66 s	1.06	1.34
		variable-type	50.61 s	1.06	1.34
jack	60.91 s	class-hierarchy	61.81 s	0.98	1.87
		variable-type	61.61 s	0.98	1.87
javac	68.40 s	class-hierarchy	68.33 s	1.00	2.14
		variable-type	67.00 s	1.02	2.14
sablecc	38.34 s	class-hierarchy	36.76 s	1.04	1.50
		variable-type	36.70 s	1.04	1.50
soot	126.67 s	class-hierarchy	124.81 s	1.01	2.29
		variable-type	122.01 s	1.03	2.29

Table 4.6: Measurements for automatic inlining using the JIT

	Execution Time Before Inlining	Interpreter results for automatic inlining: Analysis Used For Call Graph	Execution Time After Inlining	Speedup	Code Increase
compress	441 s	class-hierarchy	450 s	0.98	1.23
		variable-type	450 s	0.98	1.23
jess	109 s	class-hierarchy	106 s	1.03	2.01
		variable-type	106 s	1.03	2.01
raytrace	125 s	class-hierarchy	54 s	1.08	2.22
		variable-type	54 s	1.08	2.22
db	229 s	class-hierarchy	229 s	1.00	1.29
		variable-type	229 s	1.00	1.29
mpegaudio	374 s	class-hierarchy	390 s	0.96	1.34
		variable-type	375 s	1.00	1.34
jack	144 s	class-hierarchy	144 s	1.00	1.87
		variable-type	144 s	1.00	1.87
javac	135 s	class-hierarchy	135 s	1.00	2.14
		variable-type	135 s	1.00	2.14
sablecc	45 s	class-hierarchy	45 s	1.00	1.50
		variable-type	45 s	1.00	1.50
soot	184 s	class-hierarchy	179 s	1.02	2.29
		class-hierarchy	178 s	1.03	2.29

Table 4.7: Measurements for automatic inlining using the interpreter

more effective optimization on the JIT than on the interpreter and we feel that this is the case for two reasons.

The first reason is that the JIT might be performing some simple analyses and optimizations that become more effective on the inlined bytecode. Since we replace a method call instruction by code from the method, any intraprocedural analyses that the JIT performs would be more precise. Method calls are a serious impediment for performing simple analyses/optimizations as the analyses would have to conservatively assume that any field could be written as a result of the call. Also the greater the size of basic blocks (straight line code), the greater the possibilities for effective instruction scheduling.

The second reason for the difference in speedups between the JIT and the interpreter is the relative cost of different bytecode instructions in the two cases. Simpler bytecode instructions like `aload`, or `astore` can easily be converted into register operations on a RISC architecture by a JIT. More complex bytecode instructions like `invokevirtual`, or `invokeinterface` are compiled into a sequence of machine instructions as they do not have direct counterparts in machine instructions. Thus in the case of a JIT, simpler bytecode instructions are extremely cheap at run time whereas complex bytecode instructions are relatively more expensive. Moreover with effective register allocation resulting in minimal register spills, the cost of the simpler instructions can be further reduced. In the interpreter though, the relative costs of these instructions are not expected to differ as much. The basic cost of interpreting each bytecode must be paid for even the simple bytecode instructions, and this means that the simple instructions are no longer as cheap as in the case of the JIT. In fact the reduction in cost of the simple bytecode instructions in the case of the JIT is significant as observed in the reduction in execution time for programs (by several factors in some cases) as compared to the interpreter. Therefore in the case of a JIT, eliminating expensive bytecode instructions like `invokevirtual` and `invokeinterface` leads to considerable improvement in performance at run time.

We feel it is a combination of both these factors that makes method inlining such an effective optimization when using the JIT. In the benchmark `compress`, that shows the highest speedup in the case of the JIT, almost all the improvement can be attributed to inlining a few calls within tight loops, where the program spends most of its time. As we have shown the speedup obtained did not result in significant increase in code size. This benchmark demonstrates the effectiveness of our static inlining strategy. Since our static inlining strategy focuses on optimizing potential "hot spots" in the program by detecting loops and recursion, inlining was in fact not performed in many non-critical parts of the program limiting code explosion while at the same time succeeding in speeding up the program. In fact there are several other benchmarks (`raytrace`, `mpegaudio`, `sablecc`) where there are significant speedups using the JIT without excessive code explosion. The slight slowdown in the case of the benchmark `jack` could be attributed to factors such as register pressure, which might degrade performance in this benchmark after inlining. Note that there was a slight slowdown for `jack` even when profile guided inlining was performed indicating that it was very sensitive to changes in the machine code being produced by the JIT.

Speedups obtained using the interpreter are a rough measure of the performance impact of just eliminating the method invocation instruction. Unlike the JIT, there are no analyses that might benefit from the effect of having larger basic blocks and reduced number of method calls. The interpreter can only gain as a result of having fewer bytecode instructions to interpret after inlining. Thus the benchmark `raytrace` that has a large number of method calls (mostly to small methods) shows the maximum speedup (1.08) using the interpreter as most of the calls were eliminated.

It is also interesting to observe that `compress` which showed a speedup of 1.21 with the JIT shows no improvement at all with the interpreter. This benchmark was not observed to be very object-oriented (refer to Table 4.1) and had many field accesses and relatively few virtual calls in the code. The lack of any impact in the case of the interpreter confirms that simply eliminating the virtual call instructions alone was not enough to improve performance. We believe that the substantial improvement in performance with the JIT is because of the analyses/optimizations performed by the JIT on the inlined code.

Automatic inlining inevitably leads to some code increase as a static inlining strategy must estimate the calls that might get executed frequently using some static heuristic. In our case, all the call sites that could be reached from call sites within loops or in recursive methods are identified as important. If there are many call sites inside loops in a method `m()` that is earlier in the call chain, then all the call sites in methods in the call chain starting at method `m()` would be identified as being important. In such cases the amount of code increase could be significant. The maximum code increase is in the case of the benchmark `soot` where the code increases by 129%. On average we can see that the amount of code increase with automatic inlining is about 76% over all the benchmarks.

There are also some other important points that we observed while fine tuning our static inlining strategy. While experimenting with different thresholds for allowable method size for the caller method (into which inlining is being done) and the callee method, we discovered that choosing these thresholds properly was crucial. If the thresholds were too low (restricting the number of call sites where inlining occurred), we observed that inlining did not have any impact as most of the calls that could be eliminated were actually still present. If the thresholds were too high (allowing more code growth while inlining more call sites), then we observed significant slowdowns (up to 40% in some cases). These slowdowns were observed with much lesser code growth in the case of the JIT than in the interpreter. The effects of slow local slots (with the interpreter) in the bytecode were only observed in cases with high code explosion. But in the case of the JIT, the slowdowns were observed even when the allowed size of the callee method (for it to be inlined) was twice the average size of methods in the application. This observation shows us the harmful effects of high register pressure and cache misses, that become significant when basic blocks grow in size. It is quite possible that not choosing these thresholds correctly would lead to a reduction in speedup or even a slowdown in most benchmarks. These issues are very significant when developing a static inlining strategy.

We now discuss the impact of performing variable type analysis and using the call graph produced as a result to perform method inlining. As we see in the case of both the interpreter and the JIT, variable type analysis has minimal effect in improving performance in most of the benchmarks. This was a result to be expected as the call graph built using variable type analysis for the benchmark alone was not very different from that built using class hierarchy analysis for many of the benchmarks. We observe an improvement in performance as a result of performing method inlining using variable type analysis for the benchmarks `soot` (2%), and `javac` (2%) with the JIT. From the benchmark characteristics we have presented in Tables 4.1 and 4.2, it can be observed that `soot` and `javac` are the two benchmarks that have a complex inheritance hierarchy and many virtual calls unresolved by class hierarchy analysis. Thus variable type analysis does not directly improve the performance in the case of most of the benchmarks we experimented with, though in the case of more object oriented benchmarks, it does have some effect. We feel that the extra precision of the call graph as a result of performing variable type analysis can only be exploited to a certain degree by method inlining (there are other factors to be considered too while inlining). In general, interprocedural analyses/optimizations would benefit from the extra precision and method inlining is just one of these optimizations.

4.6.2 Profile Guided Method Inlining

		Measurements using the JIT			
		Automatic inlining		Profile guided inlining	
	Execution Time Before Inlining	**Speedup**	**Code Increase**	**Speedup**	**Code Increase**
compress	66.88 s	1.21	1.23	1.21	1.06
jess	48.17 s	1.05	2.01	1.05	1.16
raytrace	53.97 s	1.10	2.22	1.12	1.79
db	131.01 s	1.03	1.29	1.05	1.09
mpegaudio	53.95 s	1.06	1.77	1.07	1.10
jack	60.91 s	0.98	1.87	0.99	1.09
javac	68.40 s	1.00	2.14	1.02	1.42
sablecc	38.34 s	1.04	1.50	1.05	1.17
soot	126.67 s	1.01	2.29	1.03	1.60

Table 4.8: Comparison between automatic inlining and profile guided inlining using the JIT

In Table 4.8 we have shown our run time measurements using the JIT before and after performing profile guided method inlining, and for comparison, we also show the same measurements before and after automatic inlining. In column 1 we have shown the execution time of the benchmark originally, In columns 2 and 3 we have shown the calculated speedup and factor by which code size increased when automatic inlining was performed. In column 4 we have shown the speedup with profile guided inlining, and in column 5 we have shown the factor by which code size increased as a result of performing profile guided inlining.

The results from profile guided inlining show that we have an average speedup of around 1.06 and a maximum speedup of 1.21 for the benchmark `compress` in the case of the JIT. We have not presented the results with the interpreter as the performance improvements were not substantial in that case (for reasons we have discussed in the previous section) and the differences between profile guided inlining and automatic inlining were insignificant.

We notice that the speedups obtained by profile guided inlining are approximately the same as the speedups obtained through automatic inlining for all the benchmarks. The speedups obtained by profile guided inlining differ most from those obtained using automatic inlining for the benchmarks `raytrace`(2%), and `soot`(2%). These results highlight the effectiveness of our static inlining strategy in finding the calls that were important to optimize. In general it is more elegant to optimize the program without profile feedback, as profiling usually requires some user interaction (to collect the profile).

The amount of code increase in the case of profile guided inlining is expected to be minimal as only the calls that got executed frequently in the profile run get optimized. We see this behavior in practice and the maximum code increase is in the case of the benchmark `raytrace` where the code increases by 79%. When automatic inlining is used, the maximum code increase was by 129% for the benchmark `soot`. On average we can see the amount of code increase is greater for automatic inlining by about 25% over all the benchmarks as compared to the average code increase with profile guided inlining.

Chapter 5

Conclusions and Future Work

In this thesis, we have focused on reducing the overhead associated with virtual method calls in Java bytecode. The first main contribution of this thesis was the design and implementation of reaching type analysis. Two variations of reaching type analysis, declared type analysis and variable type analysis, were implemented and studied in detail. The second main contribution of this thesis was the implementation of the compiler optimization, method inlining, and the study of its impact in improving the performance of programs compiled to Java bytecode. We now discuss our conclusions and scope for future work in both these areas of study.

5.1 Analyses for virtual call resolution

We have developed and implemented a new flow-insensitive analysis, called reaching-type analysis that can be used to estimate the possible types of receivers in virtual and interface method calls in Java. Reaching-type analysis is based on a type propagation graph where nodes represent variables and edges represent the flow of types due to assignments. Two variations of the analysis were presented, variable type analysis that uses the name of the receiver as its representative, and declared type analysis that uses the declared type of a receiver as the representative.

We presented the analysis rules with examples for the two variations of reaching-type analysis. We have implemented both variations of reaching-type analysis in the Soot framework, that translates bytecode into typed three-address code. We have also implemented two analyses that are well known as effective and inexpensive techniques for call graph construction, namely class hierarchy analysis and rapid type analysis. We have studied the effectiveness of these four analyses on a set of real world applications compiled to Java bytecode from Java, ML, Ada, Eiffel and Pizza.

All four analyses implemented by us require complete applications; so they require all the bytecodes in a benchmark to be available. Therefore they do not handle applications where classes can be dynamically loaded, but we feel that optimizing complete applications is reasonably important for several classes of applications like editors, compilers, and server side applications.

For each benchmark, class hierarchy analysis was used to build an initial conservative call graph. Measurement of these graphs indicated that though class hierarchy analysis led to a call graph that is reasonably sparse with a majority of call sites resolving to a single method, there was still scope for further improvement. We applied rapid type analysis, variable type analysis and declared type analysis starting from the initial call graph built by class hierarchy analysis, and found that a significant number of nodes and edges could be removed. Variable type analysis gave the best results removing 12% to 35% edges and 10% to 65% nodes from the conservative call graph. Further variable type analysis also resolved 33% to 94% of the potentially polymorphic calls sites (after CHA) to one method (or discovered that they target no method as it is statically known that they are cannot be executed). We concluded that the results achieved by variable type analysis are better than those achieved by rapid type analysis in the benchmark code.

In order to further investigate the effectiveness of these analyses, we studied the dynamic behavior of the benchmark code alone. In this case, rapid type analysis and declared type analysis did not have much effect at all, though variable type analysis did show improvement, in some cases approaching the best possible result. We concluded that the extra granularity of variable type analysis over the other analyses was crucial. In some other cases, variable type analysis performed well as compared to the other analyses but there was still a substantial gap between the dynamic profile and static result of the analysis. We have presented some reasons for this gap, and we do not feel that a simple refinement of our analyses will be able to bridge the remaining gap.

We plan to study the effectiveness of pessimistic call graph construction schemes and compare them to the optimistic call graph construction schemes in depth in the future. Our implementation of rapid type analysis is a pessimistic version of the algorithm where edges are removed from a conservative call graph constructed by class hierarchy analysis. We are currently implementing an optimistic version of rapid type analysis in order to compare it with variable type analysis. Preliminary results show that there is very little effect between the optimistic and pessimistic approaches in terms of call graph improvement in the benchmark code. In the library code, optimistic rapid type analysis seemed to perform better than the pessimistic scheme. We are also planning an implementation of optimistic variable type analysis and we feel that such an analysis would be harder to implement and more expensive in practice as compared to our current implementation of variable type analysis which scales linearly with the size of the program. We would want to study a local type propagation algorithm that would be intraprocedural and would approximate the effect of method calls and use this analysis for virtual call resolution.

We would also be interested in studying the effect of applying variable type analysis to prune the call graph before other interprocedural analyses use it. The increased precision might lead to other optimizations like loop invariant removal and common subexpression elimination being enabled leading to a more tangible impact on performance.

5.2 Method inlining

We implemented method inlining, an optimization aimed at improving performance of bytecode. Method inlining involves replacing a method invocation instruction by the code of the method that it invokes (if it can be determined at compile time). We provided a detailed and clear specification of the safety issues that are specific to performing method inlining at the Java bytecode level. We also discussed some important inlining criteria and our own static inlining strategy. Our automatic inlining strategy is based on detecting potentially important call sites to inline by examining methods for calls within loops and recursive methods. This approach aims to only perform inlining selectively at potentially important call sites, and thus avoids unnecessary code increase. We observed an average speedup of 5% over our set of benchmarks, and a maximum speedup of 21% using the JIT. Using the interpreter we observed a speedup of about 2% on average and about 10% in the best case. We concluded that the JIT performs some analyses/optimizations while executing the bytecodes, and inlining improves their effectiveness. The performance improvement with the interpreter is roughly the effect of simply eliminating the virtual call instruction as the interpreter does not perform any optimizations.

In the future, we are planning to study the effect of other optimizations similar to inlining like conversion of the virtual call to a static call. As the safety restrictions for such an optimization would be fewer, we might be able to optimize more calls without excessive code increase. Another optimization we are interested in is receiver class prediction, where a call is statically determined to potentially invoke a small number of methods at run-time. Run time type inclusion tests and conversion of virtual calls to static calls could be used to optimize these virtual calls.

Appendix A

Analysis rules for VTA

```
( The enclosing method and class in which the statements shown below appear
  are assumed to be m() and C respectively. The variables p, q and r
  represent locals or formal parameters in the method m, the variables
  pa and qa are array variables, f is a reference to an instance field,
  fa is an instance field variable that is of array type, c is a constant
  of a reftype ( e.g. string constants ), i and j are integers.
  Rules involving arrays are shown here for only one dimensional arrays
  but they can be generalized to be applicable for multi dimensional arrays
  in exactly the same manner. )

    Jimple Statement            Effect on Constraint Graph

1.    p = new P;              add the type P to the set InstanceTypes of signature(m)$p;
2.    pa[i] = new P;          add the type P to the set InstanceTypes of signature(m)$pa;

3.    pa = new P[10];         add the type P to the set InstanceTypes of signature(m)$pa;

4.    p.f = new P;            add the type P to the set InstanceTypes of f;

5.    p.fa = new P[10];       add the type P to the set InstanceTypes of fa;

6.    p = q;                  add an edge from rightnode to signature(m)$p;
      p = ( P ) q;            add an edge from signature(m)$p to rightnode if
                              ( ( q or p is of array type ) ||
                              ( p and q are declared to be of type java.lang.Object ) );
                              ( rightnode = C$this if q == this
                                rightnode = signature(m)$q otherwise )

7.    p = pa[i];              add an edge from signature(m)$pa to signature(m)$p;
      p = ( P ) pa[i];        add an edge from signature(m)$p to signature(m)$pa if
                              ( ( pa[i] or p is of array type ) ||
                              ( p and pa[i] are declared to be of type java.lang.Object ) );

8.    pa[i] = p;              add an edge from rightnode to signature(m)$pa;
      pa[i] = ( P ) p;        add an edge from signature(m)$pa to rightnode if
                              ( ( p or pa[i] is of array type ) ||
                              ( p and pa[i] are declared to be of type java.lang.Object ) );
                                ( rightnode = C$this if p == this
                                  rightnode = signature(m)$p otherwise )

9.    pa[i] = qa[j];          add an edge from signature(m)$qa to signature(m)$pa;
      pa[i] = ( P ) qa[j];    add an edge from signature(m)$pa to signature(m)$qa if
                              ( ( qa[j] or pa[i] is of array type ) ||
                              ( pa[i] and qa[j] are declared to be of type java.lang.Object ) );

10.   p = c;                  add the type of the constant c, say C to the
      p = ( P ) c;            set InstanceTypes of signature(m)$p;
```

Figure A.1: Rules for Variable Type Analysis

```
    Jimple Statement                 Effect on Constraint Graph

11.   pa[i] = c;                add the type of the constant c, say C to the
      pa[i] = ( P ) c;          set InstanceTypes of signature(m)$pa;

12.   p.f = c;                  add the type of the constant c, say C to the
      p.f = ( P ) c;            set InstanceTypes of f;

13.   p.f = q;                  add an edge from rightnode to f;
      p.f = ( P ) q;            add an edge from f to rightnode if
                                ( ( q or p.f is of array type ) ||
                                ( p.f and q are declared to be of type java.lang.Object ) );
                                 ( rightnode = C$this if q == this
                                   rightnode = signature(m)$q otherwise )

14.   p = q.f;                  add an edge from f to signature(m)$p;
      p = ( P ) q.f;            add an edge from signature(m)$p to f if
                                ( ( q.f or p is of array type ) ||
                                ( q.f and p are declared to be of type java.lang.Object ) );

15.   pa[i] = p.f;              add an edge from f to signature(m)$pa;
      pa[i] = ( P ) p.f;        add an edge from signature(m)$pa to f if
                                ( ( p.f or pa[i] is of array type ) ||
                                ( p.f and pa[i] are declared to be of type java.lang.Object ) );

16.   p.f = pa[i];              add an edge from signature(m)$pa to f;
      p.f = ( P ) pa[i];        add an edge from f to signature(m)$pa if
                                ( ( p.f or pa[i] is of array type ) ||
                                ( p.f and pa[i] are declared to be of type java.lang.Object ) );

17.   return p;                 add an edge from rightnode to signature(m)$return;
                                add an edge from signature(m)$return to rightnode if
                                ( ( p is of array type ) ||
                                ( p is declared to be of type java.lang.Object );
                                ( rightnode = C$this if p == this
                                  rightnode = signature(m)$p otherwise )

18.   return c;                 add the type of the constant c, say C to the
                                set InstanceTypes of signature(m)$return;


19.   q.method(r);              for ( i = 1; i < N+1; i++ )
                                {
                                 add an edge from rightbasenode to Ci$this;
                                 add an edge from rightparamnode to signature(methodi)$p1;
                                 add an edge from signature(methodi)$p1 to rightparamnode if
                                 ( ( r or p1 is of array type ) ||
                                 ( r and p1 are declared to be of type java.lang.Object ) );
                                }

( rightbasenode = C$this if q == this
              = signature(m)$q otherwise )
( rightparamnode = C$this if r == this
               = signature(m)$r otherwise )

( Ci is the class in which the ith method attached to this callsite ( methodi )
  in the conservative call graph is declared.
  There are N methods in all attached to this callsite in the conservative call graph.
  p1 is the first formal parameter in the definition of methodi )
```

Figure A.2: Rules for Variable Type Analysis (continued)

```
    Jimple Statement                Effect on Constraint Graph

20.  q.method(c);               for ( i = 1; i < N+1; i++ )
                                {
                                  add an edge from rightbasenode to Ci$this;
                                  add the type of c, say C to the set
                                  InstanceTypes of signature(methodi)$p1;
                                }

( rightbasenode = C$this if q == this
                = signature(m)$q otherwise )

( Ci is the class in which the ith method attached to this callsite ( methodi )
  in the conservative call graph is declared.
  There are N methods in all attached to this callsite.
  p1 is the first formal parameter in the definition of methodi )

21.  p = q.method(r);           for ( i = 1; i < N+1; i++ )
                                {
                                  add an edge from rightbasenode to Ci$this;
                                  add an edge from signature(methodi)$return to signature(m)$p;
                                  add an edge from signature(mi)$p to signature(methodi)$return if
                                  ( ( p is of array type ) ||
                                  ( p is declared to be of type java.lang.Object ) );
                                  add an edge from rightparamnode to signature(methodi)$p1;
                                  add an edge from signature(methodi)$p1 to rightparamnode
                                  ( ( r or p1 is of array type ) ||
                                  ( r and p1 are declared to be of type java.lang.Object ) );
                                }

( rightbasenode = C$this if q == this
                 = signature(m)$q otherwise )
( rightparamnode = C$this if r == this
                 = signature(m)$r otherwise )

( Ci is the class in which the ith method attached to this callsite ( methodi )
 in the conservative call graph is declared.
 There are N methods in all attached to this callsite.
 p1 is the first formal parameter in the definition of methodi )

22.  p = q.method(c);           for ( i = 1; i < N+1; i++ )
                                {
                                 add an edge from rightbasenode to Ci$this;
                                 add an edge from signature(methodi)$return to signature(m)$p;
                                 add an edge from signature(mi)$p to signature(methodi)$return if
                                 ( ( p is of array type ) ||
                                 ( p is declared to be of type java.lang.Object ) );
                                  add the type of c, say C to the set InstanceTypes of signature(methodi)$p1;
                                }

( rightbasenode = C$this if q == this
                = signature(m)$q otherwise )

( Ci is the class in which the ith method attached to this callsite ( methodi )
 in the conservative call graph is declared.
 There are N methods in all attached to this callsite.
 p1 is the first formal parameter in the definition of methodi )
```

Figure A.3: Rules for Variable Type Analysis (continued)

Appendix B

Analysis rules for DTA

```
( The enclosing method and class in which the statements shown below appear
  are assumed to be m() and C respectively. The variables p, q and r represent
  locals or formal parameters in the method m, the variables pa and qa are
  array variables, f is a reference to an instance field, fa is an instance field variable
  that is of array type, c is a constant of a reftype ( e.g. string constants )
, i and j are integers. Rules involving arrays are shown here for only one dimensional
  arrays but they can be generalized to be applicable for multi dimensional arrays in exactly the same manner. )


            Jimple Statement                  Effect on Constraint Graph

1.           p = new P;                 add the type P to the set InstanceTypes of
                                        DeclaredType(p);

2.           pa[i] = new P;             add the type P to the set InstanceTypes of
                                        DeclaredType(pa[i]);

3.           pa = new P[10];            add the type P to the set InstanceTypes of
                                        DeclaredType(pa);

4.           p.f = new P;               add the type P to the set InstanceTypes of ;
                                        DeclaredType(p.f);

5.           p.fa = new P[10];          add the type P to the set InstanceTypes of
                                        DeclaredType(p.fa);



6.           p = q;                     add an edge from rightnode to DeclaredType(p);
             p = ( P ) q;               add an edge from DeclaredType(p) to rightnode
                                        if ( q or p is of array type );

                                        ( rightnode = C$this if q == this
                                          rightnode = DeclaredType(q) otherwise )

7.           p = pa[i];                 add an edge from DeclaredType(pa[i]) to DeclaredType(p);
             p = ( P ) pa[i];           add an edge from DeclaredType(p) to DeclaredType(pa[i])
                                        if ( pa[i] or p is of array type );

8.           pa[i] = p;                 add an edge from rightnode to DeclaredType(pa[i]);
             pa[i] = ( P ) p;           add an edge from DeclaredType(pa[i]) to rightnode
                                        if ( p or pa[i] is of array type );

                                        ( rightnode = C$this if p == this
                                          rightnode = DeclaredType(p) otherwise )

9.           pa[i] = qa[j];             add an edge from DeclaredType(qa[j]) to DeclaredType(pa[i]);
             pa[i] = ( P ) qa[j];       add an edge from DeclaredType(pa[i]) to DeclaredType(qa[j]);
                                        if ( qa[j] or pa[i] is of array type );


10.          p = c;                     add the type of the constant c, say C to the
             p = ( P ) c;               set InstanceTypes of DeclaredType(p);
```

Figure B.1: Rules for Declared Type Analysis

Jimple Statement	Effect on Constraint Graph
11. pa[i] = c; pa[i] = (P) c;	add the type of the constant c, say C to the set InstanceTypes of DeclaredType(pa[i]);
12. p.f = c; p.f = (P) c;	add the type of the constant c, say C to the set InstanceTypes of DeclaredType(p.f);
13. p.f = q; p.f = (P) q;	add an edge from rightnode to DeclaredType(p.f); add an edge from DeclaredType(p.f) to rightnode if (q or p.f is of array type); (rightnode = C$this if q == this rightnode = DeclaredType(q) otherwise)
14. p = q.f; p = (P) q.f;	add an edge from DeclaredType(q.f) to DeclaredType(p); add an edge from DeclaredType(p) to DeclaredType(q.f) if (q.f or p is of array type);
15. pa[i] = p.f; pa[i] = (P) p.f;	add an edge from DeclaredType(p.f) to DeclaredType(pa[i]); add an edge from DeclaredType(pa[i]) to DeclaredType(p.f) if (p.f or pa[i] is of array type);
16. p.f = pa[i]; p.f = (P) pa[i];	add an edge from DeclaredType(pa[i]) to DeclaredType(p.f); add an edge from DeclaredType(p.f) to DeclaredType(pa[i]) if (p.f or pa[i] is of array type);
17. return p;	add an edge from rightnode to signature(m)$return; add an edge from signature(m)$return torightnode if ((p is of array type) || (p is declared to be of type java.lang.Object)); (rightnode = C$this if p == this rightnode = DeclaredType(p) otherwise)
18. return c;	add the type of the constant c, say C to the set InstanceTypes of signature(m)$return;

19. q.method(r);	for (i = 1; i < N+1; i++)
	{
	add an edge from rightbasenode to Ci$this;
	add an edge from rightparamnode to DeclaredType(p1);
	add an edge from DeclaredType(p1) to rightparamnode if (r or p1 is of array type);
	}

```
( Ci is the class in which the ith method attached to this callsite ( methodi )
 in the conservative call graph is declared.
 There are N methods in all attached to this callsite.
 p1 is the first formal parameter in the definition of methodi )
 ( rightbasenode = C$this if q == this
   rightbasenode = DeclaredType(q) otherwise )
 ( rightparamnode = C$this if r == this
   rightparamnode = DeclaredType(r) otherwise )
```

Figure B.2: Rules for Declared Type Analysis (continued)

```
20.       q.method(c);                for ( i = 1; i < N+1; i++ )
                                      {
                                       add an edge from rightbasenode to Ci$this;
                                       add the type of c, say C to the set InstanceTypes of
                                       DeclaredType(p1);
                                      }

( rightbasenode = C$this if q == this
  rightbasenode = DeclaredType(q) otherwise )
( Ci is the class in which the ith method attached to this callsite ( methodi )
 in the conservative call graph is declared.
 There are N methods in all attached to this callsite.
 p1 is the first formal parameter in the definition of methodi )

21.      p = q.method(r);             for ( i = 1; i < N+1; i++ )
                                      {
                                       add an edge from rightbasenode to Ci$this;

                                       add an edge from signature(methodi)$return to
                                       DeclaredType(p);
                                       add an edge from DeclaredType(p) to
                                       signature(methodi)$return if
                                       ( ( p is of array type ) || ( p is declared to be of type
                                         java.lang.Object ) );

                                       add an edge from rightparamnode to
                                       DeclaredType(p1);
                                       add an edge from DeclaredType(p1) to
                                       rightparamnode if
                                       ( r or p1 is of array type );
                                      }

( Ci is the class in which the ith method attached to this callsite ( methodi )
 in the conservative call graph is declared.
 There are N methods in all attached to this callsite.
 p1 is the first formal parameter in the definition of methodi )
( rightbasenode = C$this if q == this
  rightbasenode = DeclaredType(q) otherwise )
( rightparamnode = C$this if r == this
  rightparamnode = DeclaredType(r) otherwise )

22.      p = q.method(c);             for ( i = 1; i < N+1; i++ )
                                      {
                                       add an edge from rightbasenode to Ci$this;

                                       add an edge from signature(methodi)$return to
                                       DeclaredType(p);
                                       add an edge from DeclaredType(p) to
                                       signature(methodi)$return if
                                       ( p is of array type );

                                       add the type of c, say C to the set InstanceTypes of
                                       DeclaredType(p1);
                                      }

( Ci is the class in which the ith method attached to this callsite ( methodi )
 in the conservative call graph is declared.
 There are N methods in all attached to this callsite.
 p1 is the first formal parameter in the definition of methodi )
( rightbasenode = C$this if q == this
  rightbasenode = DeclaredType(q) otherwise )
```

Figure B.3: Rules for Declared Type Analysis (continued)

Bibliography

[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://SmallEiffel.loria.fr/.

[5] URL: http://research.persimmon.co.uk/mlj/.

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

[7] Ole Agesen. Constraint-based type inference and parametric polymorphism. In Baudouin Le 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 September 1994.

[8] Ole Agesen. The Cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In Walter 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 August 1995. Springer.

[9] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In Pierre 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 July 1996. Springer.

[10] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In David S. Wise, editor, *Proceedings of the SIGPLAN '88 Conference on Programming Lanugage Design and Implementation (SIGPLAN '88)*, pages 241–249, Atlanta, GE, USA, June 1988. ACM Press.

[11] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 134–145, New York, June 15–18 1997. ACM Press.

[12] David F. Bacon and Peter 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, October 6–10 1996. ACM Press.

[13] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *21st Symposium on Principles of Programming Languages*, pages 397–408, January 1994.

[14] Paul R. Carini. Automatic inlining. Technical Report RC 20286, IBM T.J. Watson Research Centre, IBM Research Division, November 1995.

[15] Craig Chambers, David Grove, Greg DeFouw, and Jeffrey 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, October 5–9 1997. ACM Press.

[16] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–369, May 1992.

[17] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, April 1993.

[18] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software Practice and Experience*, 21(6):581–601, June 1991.

[19] Keith D. Cooper, Mary W. Hall, and Linda Torczon. Unexpected side effects of inline substitution: a case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, March 1992.

[20] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software Practice and Experience*, 18(8):775–790, August 1988.

[21] J. Dean and C. Chambers. Training compilers to make better inlining decisions. Technical Report TR 93-05-05, University of Washington, 1993.

[22] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig 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, October 1996.

[23] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter 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 August 1995. Springer.

[24] Amer Diwan, J. Eliot B. Moss, and Kathryn 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, October 6–10 1996. ACM Press.

[25] Mary F. Fernández. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 103–115, La Jolla, California, June 18–21, 1995. *SIGPLAN Notices,* 30(6), June 1995.

[26] Etienne Gagnon and Laurie J. Hendren. Intra-procedural inference of static types for java bytecode. Technical Report 1999-1, Sable Research Group, School of Computer Science, McGill University, March 1999.

[27] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[28] Anne M. Holler. *A Study of the Effects on Subprogram Inlining.* PhD thesis, University of Virginia, Charlottesville, Virginia, USA, March 1991. Computer Science Report No. TR-91-06.

[29] Urs Hölzle and David 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, June 1994. ACM Press.

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

[31] Wen mei W. Hwu and Pohua P. Chang. Inline function expansion for compiling C programs. In Bruce Knobe, editor, *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (SIGPLAN '89)*, pages 246–257, Portland, OR, USA, June 1989. ACM Press.

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

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

[34] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis vs. procedure integration. *Information Processing Letters*, 32(3):137–142, August 1989.

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

[36] Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, Atlanta, Georgia, June 22–24, 1988. *SIGPLAN Notices,* 23(7), July 1988.

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

[38] Tucker Taft. Programming the Internet in Ada 95. In Alfred Strohmeier, editor, *Reliable software technologies, Ada-Europe '96: 1996 Ada-Europe International Conference on Reliable Software Technologies, Montreux, Switzerland, June 10–14, 1996: proceedings*, volume 1088, pages 1–16, 1996.