A STUDY OF SIDE-EFFECT ANALYSES FOR JAVA

by
Chrislain Razafimahefa

School of Computer Science
McGill University, Montreal

December 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

In the context of an object-oriented programming language such as Java, the ubiquitous use of instance fields and the presence of pointers (references to objects) make it difficult to understand a program's memory usage. Understanding memory usage is a key element for optimizing programs and *side-effect analysis*, an analysis to estimate variables that are inspected or altered by a computation, is of prime importance for understanding and optimizing Java programs.

In this thesis, we study the impact of a set of practical side-effect analyses on optimizing Java programs. The main contribution of the thesis is the design and implementation of two groups of analyses, namely *type-based analysis* and *refers-to analysis*. These analyses use different strategies to capture the effect of an instruction on variables present in the program. The former uses type information whereas the second models the program's storage shape graph. In both type-based analysis and refers-to analysis, two variations of the analysis are presented, one which considers the internal structure of objects such as fields as a key element of the analysis, and another which considers objects as aggregates and ignores their internal structure.

In order to assess the effectiveness of each analysis, we have implemented *loop-invariant removal*, an optimization which moves invariant computation out of loops. We used the Soot framework to implement the analyses and the optimization. We present both static and dynamic results based on experiments performed on a set of Java benchmarks of various sizes. Our results suggests that side-effect analysis is beneficial. Furthermore, we also notice that analyses that exploit the internal structure of objects, as a key element of the analysis, performs significantly better than their counterparts which do not.

Résumé

Dans les langages de programmation orientés-objets tels que Java, l'utilisation intensive des variables d'instance et la présence des pointeurs rendent difficile la compréhension de l'usage de la mémoire par les programmes. Mieux cerner l'utilisation de la mémoire est un élément clé pour l'optimisation de programmes. L'*analyse des effets de bord*, qui consiste à détecter les variables qui sont lues ou modifiées par une instruction, est de première importance pour comprendre et optimiser les programmes Java.

Cette thèse porte sur l'étude de la capicité d'un ensemble d'analyse d'effets de bord à optimiser des programmes Java. La principale contribution de la thèse est la conception et la réalisation de deux groupes d'analyses, *les analyses basées sur le typage* et *les analyses basées sur les références*. Ces analyses utilisent différentes stratégies pour capturer l'effet de chaque instruction sur les variables du programme. Le premier groupe utilise le typage pour parvenir à ces fins, alors que le second modélise un graphe des relations entre objects en mémoire. Deux variations de chacune de ces analyses sont présentées, l'une qui considère les structures internes aux objets (les variables d'instance) comme un élément clé de l'analyse et l'autre qui considère les objets comme opaques et ainsi ignore les structures internes.

Pour évaluer l'impact de chaque analyse, nous les avons associées à une optimisation appelée *suppression des invariants dans les boucles* qui consiste à déplacer hors des boucles les instructions dont le résultat ne varie pas d'une itération à l'autre. Les analyses ainsi que l'optimisation ont été développées dans le cadre du projet *Soot*. Nous présentons à la fois les résultats statiques et les résultats dynamiques collectés lors des analyses et lors des exécutions des programmes optimisés. Nos expériences ont été réalisées sur un ensemble de programmes étalon de tailles variables. Il résulte de nos observations que d'une part, l'analyse des effets de bord est bénéfique, et d'autre part que les analyses qui exploitent les structures internes des objets comme

élément clé de l'analyse donnent des résultats bien supérieurs à ceux qui ne le font pas.

Acknowledgments

First, I would like to thank my advisor Professor Laurie Hendren for her constant help, patience and kindness. She was always there to listen. I have learned a lot under her guidance.

I am also grateful to the *Sable Reasearch Group* members, Etienne Gagnon, Patrick Lam, Laleh Tajrobehkar, Vijay Sundaresan, who is a good friend and with whom I had a lot of technical discussions, Raja Vallée-Rai, who is at the heart of the Soot project and all other people who contributed to the project. Without them, this thesis would not have been possible.

I would like to say a special thank you to all my great friends met throughout the course of my study at McGill University, in particular, Ian Garton, Prasad Kakulavarapu, Tallman Nkgau and Mike Soss. They were always there to help and made my stay in Montreal really enjoyable.

I am also thankful to Professor Denis Tsichritzis, Ciàran Bryce, Dimitri Konstantas, Jean-Henry Morin, Manuel Oriol, Thomi Piloura, Karim Taha, Dimitri Thanos and Jan Vitek, who are members of the *Object Systems Group* in Geneva, for being such good friends and for their infinite patience and understanding when I was finishing this thesis.

Finally, I would like to thank my parents for their infinite support.

Dedicated to my family, Roland, Jeanine and Fabrice

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Java's popularity has grown rapidly during the last few years [GJS97]. This is mainly due to a clean design of the language and the presence of many features such as object-orientedness, strong typing, multithreading support, exception handling and automatic garbage collection. But certainly the most attractive aspect of Java is its platform independence where a program written on one platform can run transparently on another one. This is made possible by the intermediate representation called bytecode [LY97] generated by Java compilers. Bytecodes are instructions targeted for a virtual machine and executed by an implementation of the virtual machine on each platform.

Originally, for the first version of Java, bytecodes were interpreted. However, interpretation comes with a cost, namely lack of performance. To overcome this problem, most of Java implementations in use today use a JIT (Just In Time) compiler [ATCL+98]. JITs work by dynamically translating bytecode into native code. That is, at runtime when the virtual machine is about to execute a program method for the first time, the method is first translated into native code and this native code is executed instead of the bytecode. JITs have improved the situation but still, since producing native code at runtime consumes execution time, JITs do not have enough time to produce code of high quality. Consequently Java's performance is still lagging behind those of other popular languages such as C and C++ and therefore performance remains the main drawback of the Java paradigm. We should also mention here that another technique which consists in translating bytecodes directly to C code exists and then to compile the C code into assembly using existing optimizing C compilers. A tool like TowerJ [Tow], an ahead of time compiler, exploits this strategy.

The advantage is that performance can be improved using whole program analysis and exploiting the benefit of the C compilers advanced analyses. is the same as for programs written in C. However, Java-like portability is lost since in its assembly form the code is restricted to an architecture and in its C form, the code has to be recompiled on each new target.

1.1 Side-effect analysis

In this thesis, our goal is to improve runtime performance of Java programs. Our focus is on static analysis and optimization of Java bytecode. Analyzing the code at the bytecode level provides certain advantages. First of all, the Java compilation model, which keeps the compiler and the bytecode independent of the platform on which the code is executed, is not breached. Second, independence from the source code is preserved, and finally, optimization of bytecode produced by other languages than Java is also possible. However, independence from the virtual machine means also that low level optimizations dependent of the platform is not possible.

Typically, optimizing compilers strive to make programs run faster by, for example, eliminating redundant computation, reducing memory traffic and removing invariant instructions from loops. A key component of an optimizing compiler is the computation of read/write sets i.e., determining memory locations that are inspected (read) and/or altered (written) by a given statement or a group of statements in a program. This is also known as *side-effect analysis* [Ban79][CK88] and it is the emphasis of this thesis. By applying side-effect analysis, an optimizing compiler gains more knowledge about the behavior of a program because static disambiguation of the memory usage is obtained. More precisely, it is possible to know whether two given memory references do not interfere i.e., to know if they do not access the same memory locations. This knowledge enables the compiler to perform optimizations such as rearranging instructions or removing invariant computations out of loops and therefore impact on the runtime performance of programs being analyzed.

To illustrate the importance of side-effect analysis, let us consider figures 1.1 and 1.2 where two loops are presented. Suppose that after applying side-effect analysis the read/write sets of each statement are those given on the right side of each statement. First consider figure 1.1. Since none of the statements in the loop defines (writes) both variables (`o1.potentialAlias` and `y`) used in statement (1), the latter is therefore invariant to the loop since it always evaluates to the same value and it can be moved out of the loop by the optimizing compiler. This transformation therefore leads to an improvement in the program running-time, since fewer instructions are executed through the iterations of the loop. In figure 1.2, the same conclusion can be drawn since neither `o1.potentialAlias` nor `y` are written inside the loop. Therefore statement (1) in figure 1.2 can also be moved out of the loop.

```
void m1( O o1, O o2, int x, int y )
{
   for(  int i=0 ; i < 100 ; i++ )
   {
    (1):    x = o1.potentialAlias + y;  // Read { o1, o1.potentialAlias, y }
                                        // Write { x }

    (2):    o2.potentialAlias = i + x;  // Read { o2, i, x }
                                        // Write { o2.potentialAlias }
   }
}
```

Figure 1.1: simple read/write sets example

As opposed to above, let us examine the same examples again but this time without the availability of side-effect information. In this case for figure 1.1 since the information stating that defining (writing) field `o1.potentialAlias` is not equivalent to defining `o2.potentialAlias` is not available, the optimizing compiler must not move statement (1) out of the loop since `o1` and `o2` could potentially refer to the same object and if so, writing `o2.potentialAlias` in statement (2) is the same as writing `o1.potentialAlias` which also means that statement (1) is no longer invariant. Regarding the loop in figure 1.2, without the side-effect information the optimizing compiler can not conclude that the first statement is invariant since the call to `foo(...)` on the second statement could potentially write the field `o1.potentialAlias` and if so, moving the first statement out of the loop would be illegal.

The previous two examples expose two problems that have to be solved in order to obtain an effective side-effect analysis , namely:

- detection of aliases
- accurate estimation of the potential effect of method calls.

Indeed, the presence of aliases renders the collection of side-effect information more difficult in object-oriented languages such as Java. We say that two or more

```
void m2( O o1, O o2, int x, int y )
{
   do
   {
    (1):    x = o1.potentialAlias + y;  // Read { o1, o1.potentialAlias, y }
                                        // Write { x }

    (2):    foo(o1 , o2 , x);          // Read { o1, o1.potentialAlias, o }
                                       // Write { o2.potentialAlias }
   }
   while( x < 100 );
}

void foo( O o1, O o2, int x )
{
    o2.potentialAlias = o1.potentialAlias + x ;
}
```

Figure 1.2: read/write sets example with method call

expressions are *aliases* [Wei80][LR92][DMM98] if they represent exactly the same location in memory. This definition will be used for the rest of this document. In Java, aliases arise due to assignments (either explicit in the case of an assignment statement or implicit in the case of actual to formal parameters binding occurring during method calls). Even though all types of variables (locals, instance fields, class variables) participates in the creation of aliases, *only array elements and instance fields can be aliased in Java* (notice that it is impossible in Java for two locals to be aliases since there is no mechanism that allows another variable to refer/point to a local). We illustrates this point in figure 1.3. In this figure, locals `o1` and `o2` in statement (1) are the source of aliasing since after the assignment they are referring to the same object, but `o1` and `o2` are not considered to be aliases since they are stored at different memory locations. It is rather both expressions, `o1.potentialAlias` and `o2.potentialAlias`, in statements (2) and (3) that have to be considered as aliases since they are represented by the same memory location.

From figure 1.2, we have seen that in order to be less conservative, side-effect analysis must accurately estimate the potential effect of method calls. This is specially true, knowing that size of Java methods are expected to be small and the frequency of method calls is expected to be rather high. Consequently, the use of intraprocedural analyses is limited (an analysis is intraprocedural when it does not consider the effect

```
{
    (1)   o1 = o2;
    (2)   o1.potentialAlias = ...;
    (3)   ... = o2.potentialAlias;
}
```

Figure 1.3: aliasing illustration

of method calls). Therefore, we have chosen to implement all our analyses in an inter-procedural way.

In this thesis, we present a set of flow and context insensitive side-effect analyses. An analysis is *flow insensitive* when its result is independent from the program control flow. That is if a statement `a` is executed before another statement `b`, the result of the analysis is the same as if the statement `b` were executed before statement `a`. In the same vein, suppose that two calls `a` and `b` to the same method `m` exists in a program, an analysis is said to be *context insensitive* if the result of the analysis of method `m` is the same for both call sites `a` and `b`. The main advantage of being context and flow insensitive is that the analyses scale well and consequently the analysis of larger programs is possible.

Our main goal in this thesis is to assess the effectiveness of a set of side-effect analyses in optimizing Java programs and how they compare to each other. We consider two main kinds of analyses: *type-based analyses* [Cla97][DMM98], which exploit type information available from the program to perform the analysis and *refers-to analyses* which are our adaptation for Java of Steensgaard's points-to analysis [Ste96b] [Ste96a]. Each group contains two analyses of varying granularity. The two type-based analyses are called *class-based* and *field-based* whereas the two refers-to analyses are called *aggregate* and *with-fields*. The difference for each analysis in each category resides in the way they handle high level information such as object fields and array elements present in the program. The first analysis of each group (class-based and aggregate analysis) considers objects as opaque and therefore ignores the effect of fields, whereas the second (field-based and with-fields) considers each field of an object as a separate entity. This thesis describes a working implementation of these different side-effect analyses as well as an evaluation of their effectiveness through the implementation of a standard code optimization, namely loop-invariant code motion. Loop-invariant code motion detects statements that remain invariant i.e., that always evaluate to the same value during the execution of the loop, and move these computations out of the loop. In figure 1.4, we can see the result of applying loop-invariant code motion on the loop in figure 1.2.

```
void m2( O o1, O o2, int x, int y )
{
   x = o1.potentialAlias + y;

   do
   {
      foo(o1 , o2 , x);
   }
   while( x < 100 );
}
```

Figure 1.4: loop-invariant code motion example

`Soot` is the framework under which this thesis was developed [SOO]. It will be described in more detail in chapter 2. However, it should be noted that Soot provides program optimization such as *method inlining* [AGS97][Sun99]. Method inlining consists in replacing in a given method a call site by the code of its callee. Method inlining provides two major advantages. First, it reduces the cost of method calls since calls are replaced by the code being called and, second, it augments optimization opportunities for other analysis such as loop invariant removal. Therefore, another goal of this thesis is to know to what extent method inlining can improve optimizations based on side-effect analysis.

1.2 Thesis contributions

In this thesis our focus is on assessing the effectiveness of a set of cheap and practical side-effect analyses on improving runtime performance of large programs translated to Java bytecode. Our contributions include:

Design and implementation of refers-to analyses: We say that a variable `a` refers to an object `o` if `a` contains the address of `o`. Refers-to analysis produces a graph which summarizes all of the refers-to relationships in the program under analysis. We use the term refers-to instead of points-to since in Java references are used rather than pointers. Based on the refers-to relationships, we are able to compute the read/write sets (side-effect analysis) of a statement or group of statements. Refers-to analysis come in two versions of varying granularity. The first one, aggregate refers-to analysis, considers objects as opaque whereas the second, with-fields refers-to analysis, considers fields inside objects. Aggregate refers-to is more efficient both in terms of space and time but is less precise than with-fields. We provide both the rules defining the analyses as well as their implementation.

Implementation of type-based analyses: In order to compare the effectiveness of cheap side-effect analyses we have implemented a different approach than refers-to analysis known as type-based analyses [Cla97]. There are also two variants of this analysis and they use the typing information available for each variable in the program to conservatively approximate the aliasing relationship that holds between variables in the program.

Implementation of a client of the analyses: To evaluate the impact of the different side-effect analyses, we have implemented loop-invariant code motion as a client.

Empirical studies: In addition to providing static measurements on the precision of each side-effect analysis we also provide their impact on runtime performance of large Java benchmarks. Moreover, we present results on the effect of method inlining on loop-invariant code motion based on side-effect analysis.

1.3 Thesis organization

In chapter 2, we first describe Soot, the framework under which this work has been realized as well as the interprocedural analysis framework. Chapter 3 gives a more detailed description of the different analyses and chapter 4 describes the client of the analysis, ie. loop-invariant code motion. In chapter 5, we discuss the experimental results and finally related work is described in chapter 6. Chapter 7 gives conclusions as well as future work.

Chapter 2

Environment: Soot and the interprocedural analysis framework

In this chapter, we first outline *Soot*, the environment under which our analyses and optimizations were designed and implemented. Then, we present the interprocedural analysis infrastructure.

Soot is a framework aimed at bytecode manipulation [SOO]. Soot was developed to ease analysis and optimization of bytecode. To do so, it provides a set of intermediate representations called Baf, Jimple and Grimp.

Baf is a low-level representation, which mimics bytecode and makes it easier to manipulate bytecode by hiding the encoding issues related to the constant pool. Baf is useful when the need for manipulating bytecode as stack code arises. Therefore Baf is well-suited for low-level optimizations such as peephole transformations (peephole transformations consists in examining short sequence of instructions and replacing these instructions by a shorter or faster sequence).

The second representation offered by Soot is called *Jimple* [VRH98]. It is a three-address code representation [ASU86] of the bytecode. Three-address code is a sequence of the form:

```
x := y op z
```

where `x, y, z` are constants or variables (instance fields, static variables, array accesses or locals) and `op` is a logical or arithmetic operator. Obviously not all Java instructions can fit into this format and, therefore, Jimple provides additional instructions for method invocation, branching, synchronization and exception handling. The main advantage of Jimple is that it is much easier to do code analysis on such a stackless representation, rather than on stack oriented one such a bytecode. The main problem with a stack resides in the difficulty to track dataflow information for values on the stack. For instance, keeping track of variable definitions is greatly facilitated in three-address code compared to a stack-based representation. For this reason, most optimizations in the literature apply to code representation that uses variables and registers instead of a stack. Another advantage of Jimple over bytecode is that locals and parameters are typed. The typing is inferred from the bytecode [GH99].

Grimp is the third representation available in Soot. It is similar to Jimple except that it represents expressions as trees. Grimp is used for bytecode regeneration after its manipulation in one of the other Soot representations. Having larger agregated expressions simplifies this task. The use of Grimp is not limited to code generation, its ultimate goal is to become the basis of a Java decompiler.

Figure 2.1 shows the overall structure of the Soot framework whereas figure 2.2 shows a Java method and its equivalent representations in Baf, Grimp and Jimple. Notice that at the beginning of each method, Soot introduces identity statements which maps the implicit `this` and the other method parameters to local variables (eg.: `r0 := @this`). These variables are subsequently used inside the method body instead of the parameters.

We have chosen to implement all our analyses using the Jimple representation. The reasons for this choice are the following:

- First, three-address code compared to stack-oriented code simplifies the analyses as already mentioned above.
- Second, Jimple is typed. This is mandatory, as their names suggest, for type-based analyses described in this thesis.
- Third, since the Jimple statement format is rather simple and the number of statement kinds is limited, it is simpler and less cumbersome to define rules for the different analyses.

In the next section, we describe the construction of the class graph and the call graph which serves as a basis for our interprocedural analyses.

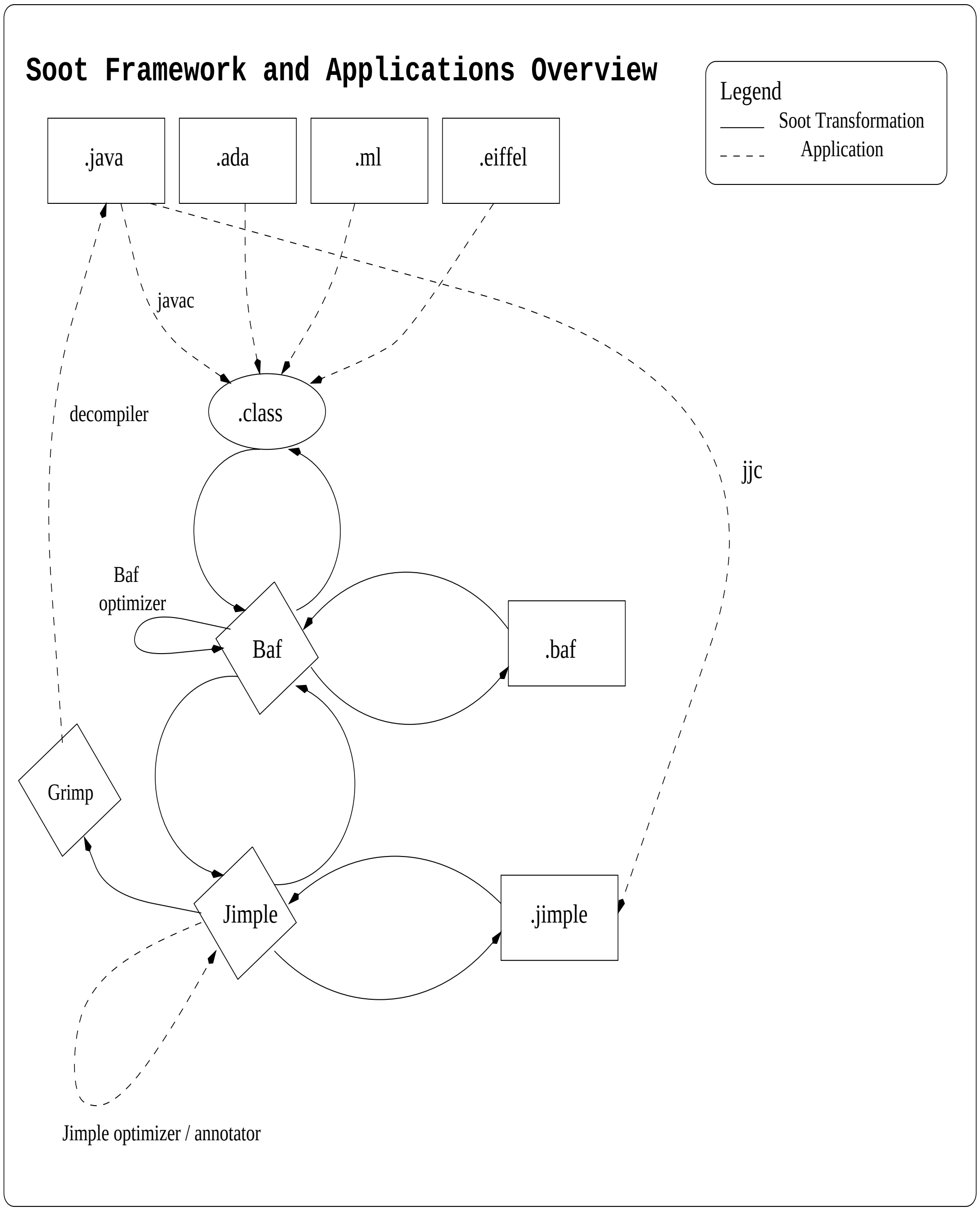


Figure 2.1: Soot framework

```
public A foo(B b) {
    A a = new A();

    if( b.value != 0 )
    {
      a.setValue(0);
    }

    return a;
}
```

(a) Java source

```
public A foo(B) {
   A r0, r3;
   B r1;

   r0 := @this;
   r1 := @parameter0;
   r3 = new A();
   if r1.value == 0 goto label0;
   r3.setValue(0);

 label0:
   return r3;
}
```

(b) Grimp representation

```
public A foo(B) {
   word r0, r1, $r2, r3, $i0

   r0 := @this
   r1 := @parameter0
   new A
   store.r $r2
   load.r $r2
   specialinvoke <A: void <init>()>
   load.r $r2
   store.r r3
   load.r r1
   fieldget <A: int value>
   store.i $i0
   load.i $i0
   ifeq label0
   load.r r3
   push 0
   virtualinvoke
        <A: void setValue(int)>
 label0:
   load.r r3
   return.r
}
```

(c) Baf representation

```
public A foo(B)  {
   A r0, $r2, r3;
   B r1;
   int $i0;

   r0 := @this;
   r1 := @parameter0;
   $r2 = new A;
   specialinvoke $r2.<A: void <init>()>();
   r3 = $r2;
   $i0 = r1.<A: int value>;
   if $i0 == 0 goto label0;

   virtualinvoke r3.
        <A: void setValue(int)>(0);

 label0:
   return r3;
}
```

(d) Jimple representation

Figure 2.2: the different representations in Soot

2.1 Class Graph

We use Jimple as the basis for the implementation of the different analyses, but it serves also for the call graph and class graph construction. The *class graph* represents the inheritance hierarchy of a program. The hierarchy is a key element in the construction of the call graph. Indeed, the class graph is used by the call graph to resolve the target of virtual and interface method calls as we will see in the next section. It contains all classes and interfaces present in the program as well as their superclasses and superinterfaces. A node in the graph represents a class or an interface. An edge is drawn between a class (or interface) and respectively its unique superclass and the interfaces it implements.

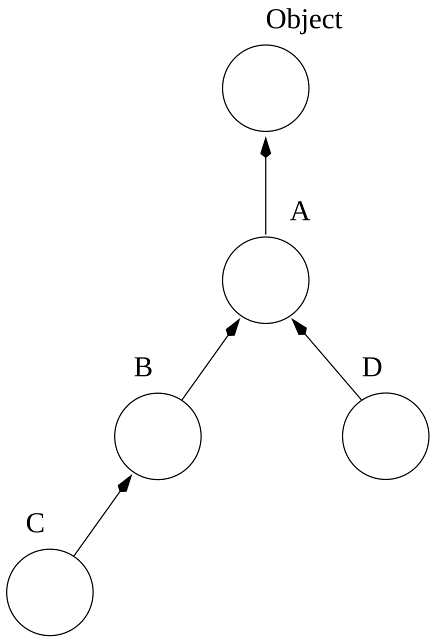
The construction of the class graph starts by the analysis of the class containing the `main` method as well as classes containing an `init`, `start`, `run` or `clinit` method. Their bytecode is read and translated into the Jimple representation. Going through each method of these classes, all the other classes they reference are collected. Each of these classes is then added to the class graph and is in turn analyzed for classes they use. This recursive step is performed until the transitive closure is reached. Once classes directly used by the program are collected, the second step consists in collecting all of their superclasses. Each class gets its superclass added to class graph (if it is not already present) and a link is drawn between them. Then for each superclass, its superclass is also added to the class graph. Once again a recursive step is applied until either the superclass is already present in the graph or the class `java.lang.Object`, the root of the hierarchy, is reached. Figure 2.3 shows an example of a class graph. This superclass collection step is required by the presence in Java of virtual and interfaces calls. Indeed an object might call a method which is defined in an object's superclass but that superclass is not used (referred) in the program, therefore we still need to add this superclass to the class graph. Finally, the same procedure is followed for interfaces. That is, we first collect the interfaces directly referenced by classes in the program, then add them to the class graph and finally recursively collect all their superinterfaces.

2.2 Call Graph

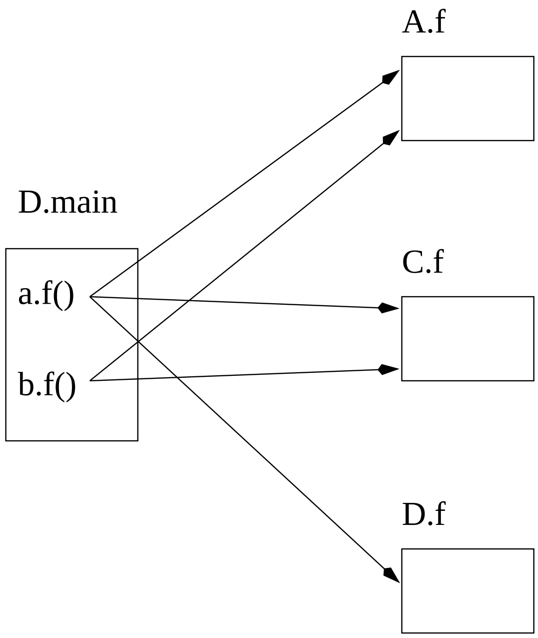
A call graph [CGDD97] is a directed graph that captures the calling relationship between methods in a program. Each method in the program is represented by a node

```
class A
{
   int a = 0;

   int f()
   {
      return a++;
   }
}

class B extends A
{

}

class C extends B
{
   int f(){ ... }
}

class D extends A
{
   int f(){ ... }

   public static void main()
   {
      A a;
      B b;
      int g;

      ...
      g = a.f();
      ...
      g = b.f();
      ...
   }
}
```

(a) example program



(b) class hierarchy



(c) call graph

Figure 2.3: class hierarchy and call graph illustration

in the call graph. Each node contains a set of indexed call sites. From each call site, zero or more edges are drawn to method nodes (callees) that are targets of the call. Typically for object-oriented languages, a dynamically dispatched message generates multiple edges leading from the call site source of the call to callees. This is because the receiver of the call might be bound to objects of different types at runtime. An example of a call graph is given in figure 2.3.

We start out our construction of the call graph by the main method. For multithreaded programs and applets, `init`, `start`, `run` and `clinit` methods are also considered as entry points. Our implementation first builds a conservative call graph using class hierarchy analysis [DGC95], i.e. it might contain more edges and nodes than the call graph we would have obtained by a runtime trace. Our implementation builds the call graph by using a worklist strategy. Each entry point (`main, start, run, init, clinit`) is added to the work list and the first method on the list is retrieved and analyzed. Call sites inside this method are located and for each of these call sites, one of the case below is applied according to its type. There are four different method call types in Java (and in Jimple). The goal is to safely approximate at compile time the set of method(s) that might be reached from a given call site at runtime.

- *invokestatic*: In the case of a static invoke, the target method is already statically resolved by the compiler since only one method can be reached from a static call. Thus, we draw an edge in our call graph from the call site to the method node target of the call in the call graph.
- *invokevirtual*: In the case of a virtual invoke, there might be more than one edge drawn from a call site since the variable receiver of the call in the invoke expression might be bound to objects of different type at runtime. To statically resolve the set of method(s) that might be reached from a virtual call site, we apply the following algorithm:

  Given a call site `o.m(...)` where `C` is the declared type of `o`, we first check whether class `C` declares a method with the same signature as `m`. If so, then an edge is added from the call site `o.m(...)` to the node corresponding to `C.m` in the call graph. Otherwise if `C` does not declare a method `m`, using the class graph, we look up the class hierarchy until we eventually reach a superclass of `C` that does declare a method with the same signature as `m`. If so, then an edge is drawn between the call site and the method `m` declared in the superclass of

C. Finally, we look down the hierarchy for subclasses of C. For each subclass C', we add an edge between the call site and the method node C'.m, if C' defines a method m with the same signature as the method m declared at the call site. Figure 2.3 gives an example of a call graph with the presence of virtual method calls.

- *invokeinterface*: In Java, interfaces declare methods but let the responsability of defining the methods to concrete classes implementing the interface. And as for classes which can be subclassed, interfaces can have subinterfaces too. In presence of an interface invoke expression, as for the invokevirtual case above, there might be more than one edge drawn from a call site to callees since there might not be an unique type bound to the receiver of the call at runtime.

  To resolve the set of methods that might possibly be called from an interface invoke expression, we collect all the concrete classes that implement the interface as well as those implementing the subinterfaces. For each of these classes, we apply exactly the same procedure as the one described for invokevirtual. That is, we check if the class defines a method with the same signature as the one in the invoke expression, if so we add an edge, if not we look up the class hierarchy and add an edge if a superclass defines a method with the same signature. Finally, we look down the class hierarchy for classes that also define the method and draw an edge between these method nodes and the call site.

- *invokespecial*: In Java, invokespecial is used to invoke instance initialization methods as well as private methods and methods defined in the superclass of the current class. Regarding the call graph construction, in the case of an invokespecial where the method being called is a private method or an instance initialization, we just have to draw one edge between the method node declared in the invoke expression and the call site since only one method can be reached by such calls. In the case of a call directed to a superclass of the current class, we need to look up the class hierarchy until we find the method that corresponds to the method signature in the invoke expression.

Chapter 3

Side-effect analysis

In this chapter we present the four flow and context intensive side-effect analyses developed for this thesis. We first start our presentation by describing the generic side-effect algorithm followed by an overview of each specific analysis, namely type-based analysis and refers-to analysis. Finally, we give a more detailed description of the analyses.

3.1 Side-Effect Analysis: the generic algorithm

Side-effect analysis [Ban79][CK88][Cla97] is an analysis aimed at determining memory locations that are altered or inspected by computations. This knowledge permits a better understanding of the program's behavior by disambiguating memory usage. Therefore it renders an optimizing compiler less conservative in the transformations it might apply, thus increasing the impact on run-time performance. In this section we present the generic side-effect algorithm which serves as a basis for all other side-effect analyses presented in this thesis.

The algorithm is at a first glance rather simple. We simply visit each statement of each method and compute their read/write sets. A statement's read set is defined as the set containing all variables whose values are inspected (read) by the statement. Similarly, a statement's write set is made of variables altered (written) by the statement. The only statements of interest for the analysis are those that read and/or write a field or an array since fields and array accesses are the only entities that can be aliased in Java. When such an access occurs in a statement, side-effect analysis returns all the aliases of the access and inserts them into the corresponding set according to the nature of the access ie., a read or a write. How these aliases are computed depends on the type of the side-effect analysis under consideration. This is the subject of the following sections where we present each analysis in more detail. Coming back to our general algorithm, once the read/write set of a statement is computed it is attached to the statement being visited. When read/write sets of all the statements of a method are computed, we summarize the method's effect by respectively merging read and write sets resulting from each of the method statements. Note that using the same principle we can summarize the effect of a region in a program, such as a loop body, for instance.

We now illustrate the algorithm with the aid of a few examples. We assume that no variables are aliased in the following examples and while computing read/write sets, we are not interested by local variables since they cannot be aliased. Consider figure 3.1 which shows a method with two statements. The algorithm proceeds by first visiting statement (1) where it collects its read/write sets. The result is shown in the figure on the right hand side of the statement. Then, statement (2) is visited and computing its read/write sets leads to the set given on its right hand side. At this point, all statements of the method are visited and a summary of the method effect is obtained by merging the read/write sets of each statement in the method. This summary is given in the figure on the right hand side of the method signature.

```
class A
{
   f()                       // Read {b.y , c.z}      Write {a.x , c.z}
   {
    (1): a.x = b.y + c.z;    // Read {b.y , c.z}      Write {a.x}

    (2): c.z = 15;           //                       Write {c.z}
   }
}
```

Figure 3.1: side-effect analysis in the absence of method calls

The example in figure 3.1 shows a rather simple case where no method calls occur. In the presence of method calls and assuming that no recursive calls are present in the call graph, the read/write sets of a caller is the union of the read/write sets of each callee that can be reached from the caller according to the call graph. Consider figure 3.2 where `Class B` overrides method `f` of `class A` defined in the previous example. At statement (3), the call to `f()` can lead to either method `f` in `class A` or `f` in `class`

```
class B extends A
{
   e(A  a)                  // Read {b.y , c.z}   Write {d.y , a.x , c.z}
   {
    (3):   a.f();           // Read {b.y , c.z}   Write {d.y , a.x , c.z}
   }


   f()                      // Read {c.z}         Write {d.y}
   {
    (4):   d.y = c.z        // Read {c.z}         Write {d.y}
   }
}
```

Figure 3.2: side-effect analysis with the presence of method calls

B depending on the type of the object bound to the receiver `a` at runtime. Statically, the read/write sets of the call expression `a.f()` in statement (3) must therefore be the result of merging both read/write sets of method `f` in `A` and method `f` in `B`. The result is shown on the right hand side of statement (3).

The previous example shows that in the absence of recursive calls, it is sufficient to proceed by a depth-first search (DFS) traversal of the call graph propagating side-effect information (read/write sets) from callees to callers to complete the analysis.

However in the presence of recursive calls, strongly connected components (SCC) [Tar81] exist in the call graph and applying a simple DFS traversal can lead to erroneous results as will be shown below. In a graph, an SCC is the largest set of nodes that satisfy the following criteria: from each node in the SCC, it is possible to reach any other node in the SCC. In terms of the call graph, this means that it is possible to call any method in an SCC from any other method in the SCC. Regarding side-effect analysis, this implies that read/write sets of a method present in an SCC must contain the read/write sets of all other methods in the SCC. In other words, the expected final read/write sets of each method in an SCC must be identical. In figure 3.3, we illustrate why simply applying a DFS traversal as described above might not satisfy this property. The figure shows an SCC made of three methods. Assume that `h`'s write set is the only non empty one. As mentioned above, the expected solution is that eventually all three methods will have the same read/write set. In this example, the final solution for all three methods will be `h`'s write set. Now consider a DFS traversal that starts by visiting `f`. Then, suppose that the visit is followed by a visit of `g`. The analysis would then definitively conclude that `g`'s final write set is empty since no non-empty read/write sets have been encountered so far. This result is unfortunately wrong since it is possible to call `h` from `g` and consequently `g`'s write set must include `h`'s write set. This example therefore shows that special care must be taken in presence of recursive calls in the call graph.

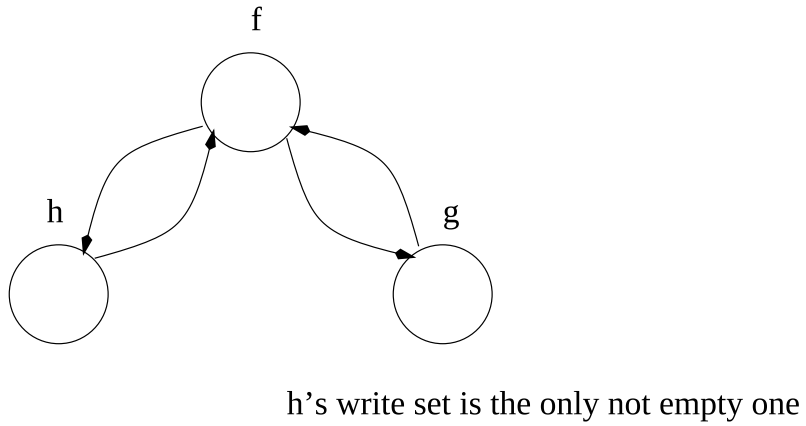


Figure 3.3: side-effect analysis in the presence of recursive calls

The solve to the problem caused by the presence of recursive calls in the call graph, we replace an SCC by a virtual node. This virtual node will serve as a summary of the SCC. To compute the overall effect of the SCC, each method in the SCC is handled separately. The same procedure as described above is applied to each method ie., visit of each statement and computation of its read/write sets and at call sites, visit of callees and propagation of read/write sets from callees to callers. However an important difference remains when visiting a call site. If one of the call site's callees is a method that also belongs to the SCC, then the callee must not be visited.

Once the previously described procedure has been applied to all methods in the SCC, then their respective read/write sets are merged and this merge results in a read/write set which is the summary of the SCC overall effect. This summary set is then attached to the virtual node representing the SCC in the call graph. For those call sites in the SCC whose callees are also member of the SCC, their read/write sets are merged with the summary set as well.

Once SCCs are removed from the call graph, a simple DFS traversal as described above is therefore sufficient to complete the analysis.

In the next section, we first provide an insight into the different side-effect analyses presented in this thesis before going in subsequent sections into a more detailed explanation as well as a precise definition of the rules that govern each analysis.

3.2 Type-Based and Refers-To Analyses: an overview

In the previous section, we have described the generic side-effect analysis algorithm but we have not yet mentioned how aliases are captured by the analysis. Indeed, since Java allows objects to be shared, one or more expressions can denote the same object. Modification of an object state through one of these expressions will also be visible to all other expressions referring to the same object. Consequently, when an expression in a statement reads or writes an object state, all other expressions denoting the same object must also be included into the statement read/write set. In this section, we give an informal presentation of the two different approaches for side-effect analysis investigated for this thesis, namely type-based analysis and refers-to analysis. These analyses are based on work done by Clausen [Cla97] and by Steensgaard [Ste96b][Ste96a]. A motivating example is presented in order to give a first intuition and to show the differences and trade-offs existing between the analyses.

When computing read/write sets of a statement, the goal is to be as precise as possible in the collection of aliases. Yet this collection is always conservative since alias analysis is undecidable [Lan92]. Type-based analysis is one such approach to approximate the alias relationship. In this thesis, we consider two variations of type-based analysis: class-based and field-based. These two analyses work by conservatively assuming that when the state of an object is read and/or written then states of all other objects of the same type as the original object are also read and/or written. Their difference resides in the fact that in field-based, the state of the object (a field) being inspected is clearly specified whereas in class-based, as soon as a field is read or modified then all other fields of the same object are also considered as being read or written.

To better understand how type-based analysis works and how field-based differs from class-based, let us consider figure 3.4. When a field of an object `o` is accessed (modified or inspected), class-based analysis considers that all other fields of all other objects that belong to the same class as `o` are also accessed. This can be seen in column 1 from figure 3.4, where in statements (1) and (2), even though two different fields of the same object `o1` are accessed, the write sets for class-based still contains the same element, that is type `O`. Type `O` is for the analysis the representative of all objects of type `O`. In the same vein, even though in statement (1) and statement (3) the accessed objects are different, class-based analysis still considers that the same object is written (by setting the write set to contain type `O`). Now, consider statement (4), where an object is being read. This object is also of type `O`. Class-based analysis therefore concludes that dependences exist between statements (1)(2)(3) and (4) since statements (1)(2)(3) write the same entity as the one read by statement (4).

Field-based analysis, on the other hand, takes field names into account when computing read/write sets. As can be seen from column 3 in figure 3.4, since fields accessed by both statements (1) and (2) are different, they are considered to be non-aliases. However, regarding statements (1) and (3) where the accessed objects are different but the fields are the same, field-based analysis is still conservative since it considers that the same object is written. Nevertheless, with field-based analysis, we can see an increase in precision compared to class-based since in this case, dependences exist only for statements (1)(3) and (4) and no longer for statements (2) and (4).

statements	class-based	field-based	aggregate	with-fields
O o1 = new O();				
O o2 = new O();				
(1) o1.f = ...;	write O	write O.f	write o1	write o1.f
(2) o1.g = ...;	write O	write O.g	write o1	write o1.g
(3) o2.f = ...;	write O	write O.f	write o2	write o2.f
(4) ... = o1.f;	read O	read O.f	read o1	write o1.f

Figure 3.4: differences in precision between the analyses

What should be noted from the example described above is that the granularity of type-based analyses is at the class level, ie. in class-based, it is not possible to distinguish between two object instances derived from the same class and similarly, in field-based, for two different instances accessing the same field, it is not possible to consider them as non-aliases. Based on this observation, we designed refers-to analyses. The requirement was to lower the granularity down to the object level.

To better understand how refers-to analysis works and how it differs from type-based analysis, consider figure 3.4 again. It shows in its third column, the result of applying aggregate analysis, the first of the two refers-to analyses, on the code in the first column of the figure. As for class-based analysis, aggregate analysis does not use field name information as a key component of the analysis. This results in loss of precision when as in statements (1) and (2), the accessed object is the same but the fields are different. However in statements (1) and (3), as opposed to class-based, aggregate considers that both accesses to field `f` are not aliases since the objects at the basis of the access are different. Therefore, in this example, aggregate analysis is more precise than class-based analysis. This can also be seen by looking at the dependences between statements resulting from aggregate analysis. We can see from column four that for aggregate analysis, dependences exist only for statements (1)(2) and (4) whereas in addition to the previous dependences, a dependence also exists between statements (3) and (4) for class-based.

We now turn our attention to the most precise side-effect analysis developed for this thesis, that is with-fields analysis. Not only is the analysis granularity at the object level but moreover, with-fields, as its name suggests, uses field name information as a key element of the analysis. Consider statements (1) and (2), even though the underlying object is the same, the analysis still concludes that both accesses are not aliases since the accessed fields are different. This is better than the result obtained from aggregate. Furthermore, as can be seen from column five, only one dependence exists between statements in the example. This illustrates well why with-fields is the most precise from the set of analyses.

In this section, we have presented an informal presentation of the different analyses in order to give a first intuition on their respective strengths and weaknesses. In the next section, we define more precisely rules that govern the analyses.

3.3 Type-Based Analysis

In side-effect analysis, the goal is to compute read/write sets at each program statement. Because of the presence of aliasing, while visiting statements, if the analysis detects that an expression reads or writes an object state, all other expressions that might represent the same object state must also be included into the read/write set. A subsidiary goal of the analysis is therefore to refine the set of expressions that might represent the same variable. Without the analysis, the compiler must be very conservative and must assume that all expressions can access any object state.

In Jimple, the intermediate form used in this thesis, complex expressions are broken down into simpler expressions and represented into three-address code format. Further, there are only two kinds of expressions that can access an object state. These accesses are either a field access or an array access and they are represented as follows in Jimple:

- `o.f`, for a field access where `o` is the reference to the object whose state is being accessed and `f` is the field (the state) being accessed.
- `o.[i]`, for an array access where `o` is the reference to the array elements which represent the array state and `i` is the element being accessed.

In this section, we precisely describe how type-based analysis, namely class-based and field-based, refines the set of expressions that might refer to the same object state. The approach is based on the observation that first, on a type safe language such as Java, an expression can access an object state only if the expression and the object are type-compatible and second, two field access expressions can represent the same object state only if the accessed fields are *declared* in the same class. This is what we call class-based analysis. Given a field access expression `o.f` where `O` is the compile-time type of `o`, we say that `f` is declared in class `O` if there exists an explicit definition of instance field `f` in class `O`. Otherwise, we look up the class hierarchy until we find a superclass of `O` defining field `f`. Class-based analysis considers as aliases all Jimple field access expressions whose accessed fields are declared in the same class `C`. Class `C` becomes afterwards the representative of these expressions. Note that as part of the data structure for field accesses, Jimple already provides the class where the field is declared. Also note that it is not possible to override fields in Java [GJS97]. However it is still possible for an object to possess two or more fields with the same name due to inheritance. In this case, it is the compile-time type of the object whose field is being inspected that decides the actual field being accessed. Therefore, at compile-time, it is always possible for type-based analysis to unambiguously decide the actual field being accessed for each Jimple field access expression.

An even smaller set of aliases can be estimated if we observe that in order to access the same object state, fields accessed by two field access expressions must have the same name. Field-based analysis is the analysis that exploits this refinement. In this case, the analysis uses an encoding of the class where the field is declared as well as the field name as representative for all expressions that might refer to the same object state. Regarding implementation details, while performing the analyses, instead of carrying around actual expressions inside read/write sets, we rather use representatives. This has the advantage of saving space.

Regarding arrays, class-based and field-based generate more than one representative when an expression reading or writing an array is encountered. The reason is that since it is possible to subclass an array in Java, when an array expression accesses an element then any arrays which are subclasses of the original array might be read or written as well. Table 3.1 gives the rules for deducing representatives of access expressions for both class-based and field-based analysis.

expression	class-based	field-based
o.f	*declaringClass*(f)	*declaringClass*(f) + *fieldName*(f)
o[i]	$\{t \mid t \in subtype(type(o))\}$	

Table 3.1: rules for computing alias representative in type-based analysis

3.4 Refers-to Analysis

As for type-based analysis, the goal in refers-to analysis is to refine the set of expressions that might represent the same object state. In order to do so, refers-to analysis aims at approximating the refers-to relationship that holds in a Java program. More precisely, given a reference variable `v`, we would like to know what other reference variables might possibly refer to objects referred to by `v` during any execution of the program. To achieve its goal, refers-to analysis builds the storage shape graph of the program [CWZ90]. The graph is made of two kinds of nodes:

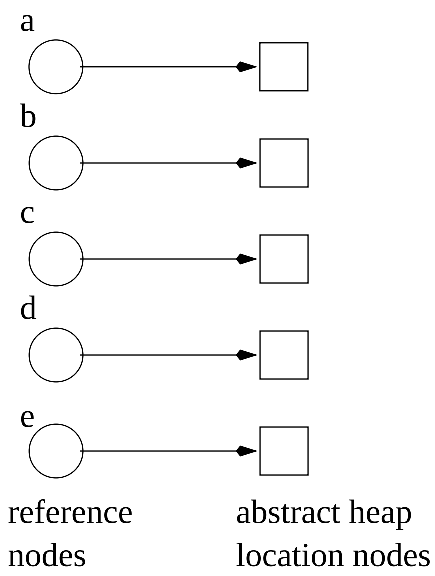
- *reference nodes*
- *abstract heap location nodes*

Intuitively, a reference node in the graph represents a reference variable present in the program whereas abstract heap location nodes represent objects that reside on the heap during the execution of the program. A directed edge in the graph always bind a reference node to a unique abstract heap location node meaning that the variable associated with the reference node might refer to the given object in memory.

For instance, when the graph shows two reference nodes referring to the same abstract location node, we can conclude that the variables in the program represented

```
public void foo( Object a )
{
   Object  b , c , d , e;
   if (b == a)
      c = d;
   else
      e = b;
}
```

(a) example program



(b) initial refers-to relationship

Figure 3.5: initial refers-to relationship

by the references nodes in the graph might possibly refer-to the same object (heap location) during an execution of the program and consequently, if the state of the object is accessed through one of the variables, then this access will be visible through the other variable as well. Figure 3.5 shows some reference and abstract heap location nodes and illustrates their relation with variables used in the program. In the next two sections, we present both variants of refers-to analysis as well as explaining their differences and we also illustrate how the graph in figure 3.5 evolves.

3.5 Aggregate Refers-to Analysis

The first variant of refers-to analysis presented here is called aggregate analysis. As already mentioned, the goal is to compute a conservative storage shape graph of the dynamic storage shape graph. The algorithm works as follows.

In an initial step, each reference variable (locals, fields, parameters) in the program is associated with a unique reference node in the storage graph and each of these reference nodes is in turn mapped to a unique and distinct abstract location node.

Figure 3.5 (b) shows this initial setup for the given example.

Once the initial step is performed, a second phase takes place: each statement in each method of the call graph is visited once and at implicit or explicit assignments (implicit assignments occurs during method calls when mapping actual to formal parameters and during method returns), for each pair of variables involved in a assignment, we merge their respective abstract heap location nodes. Figure 3.7 describes the merging rules for Jimple statements where an assignment might occur. The idea behind merging is to union both abstract heap location nodes to make them appear as a unique one. As a consequence, both reference nodes which were previously pointing to two different nodes are now pointing to the same node, meaning that the two reference variables represented by both reference nodes in the graph might refer to the same object on the heap. The union (merging) of abstract location nodes is done using the fast set-union find algorithm developed by Tarjan [Tar81]. The algorithm works as follows: at the beginning of the analysis, each abstract heap location node is considered as a set, with the node as unique member. Each set has a unique representative called the equivalent class representative (ECR), which initially is the abstract heap location node itself. Then as the algorithm proceeds, when an assignment between two variables is encountered, their respective sets are merged through their respective ECRs and one of these ECRs is chosen to be the new representative of the newly created set obtained after merging. Figure 3.6 illustrates the algorithm based on the example in Figure 3.5.

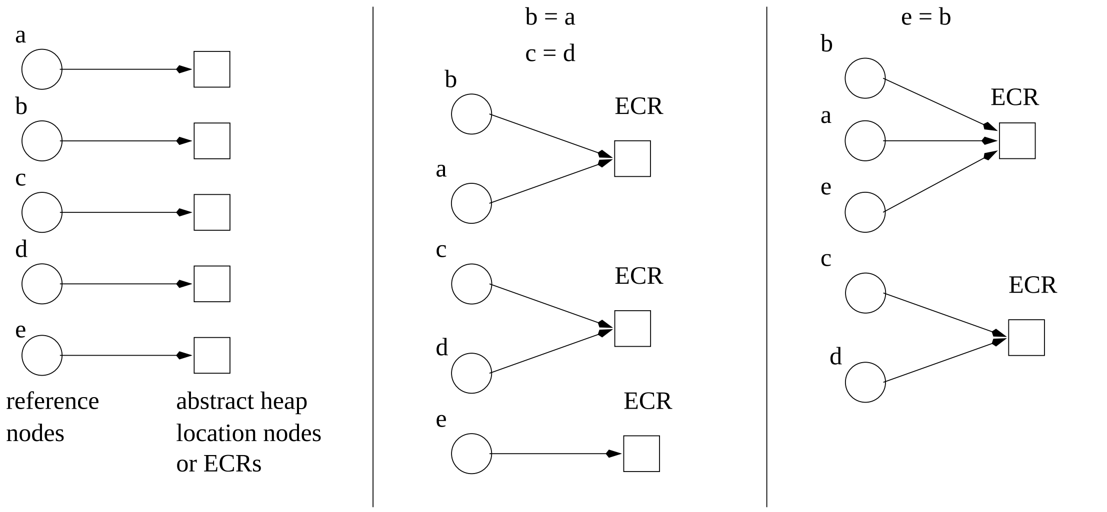


Figure 3.6: illustration of merges

The main advantage of using the fast-union algorithm is that the running time of the analysis is almost linear in the size of the program. Merging two ECRs requires a constant time whereas the cost of N find operations (finding the ECR of a set) takes on average $O(N\alpha(N,N))$, where *alpha* is the inverse Ackermann's function [Tar83], a very slowly increasing function. Therefore the running time is also almost linear. The space cost of the analysis is also linear since the number of ECRs created during the analysis is proportional to the size of the program and union of ECRs requires a constant space.

`a = b`	Merge `a`'s ECR with `b`'s ECR.
`a :=` *parameter*	Jimple provides these kind of special assignments at the beginning of each method for each parameter where *parameter* is either the implicit `this` or a formal parameter of the method. In this case, an ECR is created for each parameter and a merge of `a`'s ECR and *parameter*'s ECR is performed.
`return a`	An ECR is created for representing values returned by the method and this ECR is merged with `a`'s ECR.
`a = p.f(c,d)`	A method invocation requires three steps: 1) merge `p`'s ECR with the ECR of the implicit `this` of each class that the call might reach. The methods reachable from the call site are given by the conservative call graph. Notice that when the call is a static method invocation then this step is avoided. 2) merge the ECRs of actual parameters (in this example `c` and `d`) with each corresponding formal parameter's ECR. Repeat this process for all methods that might be reached by the call. 3) merge `a`'s ECR with the ECRs that represent return values of each method that might be reached by the call.

Figure 3.7: rules for aggregate analysis

In Figure 3.7, rules for merging ECRs are described using scalar reference variables a,b,c and d but these rules can be generalized for field or array accesses. In the case of an instance field access of the form a.i appearing in a Jimple statement, since in aggregate analysis objects are considered as aggregates, we merge a's ECR (a being the representative of the entire object) with ECRs of other variables involved in the assignment according to the rule. Similarly for array accesses of the form a[x], since in aggregate arrays are handled as a whole, we merge a's ECR (a being the array representative) with other variables representatives involved in the assignment.

3.6 With-Fields Refers-To Analysis

We now consider with-fields analysis, which leads to a gain in accuracy by keeping different fields inside the same object as separate entities. Whereas aggregate analysis considers abstract locations as totally opaque, with-fields analysis includes a set of reference nodes inside abstract location nodes which in turn are able to refer to other abstract location nodes. Nodes inside an abstract location node represent all the fields used by all objects whose representative is the abstract location. The initialization step is for with-fields analysis the same as for aggregate analysis except when an instance field reference of the form a.i is encountered. In this case, a reference node for field i is added to a's ECR if it does not already exist. Figure 3.8 shows the difference between aggregate analysis and with-fields analysis. In aggregate analysis, merging two abstract locations is straightforward. However for with-fields, since fields are now present in abstract locations, more work needs to be done. When two abstract locations are merged, the idea remains the same as before, that is both abstract location nodes must appear as a unique node after merging. To do so, one of them is promoted as their new representative. The role of the representative is to summarize the information contained in both nodes. That is, the new representative must afterwards refer to locations where both were referring to before merging. Therefore, the following steps occur when abstract locations are merged in with-fields analysis:

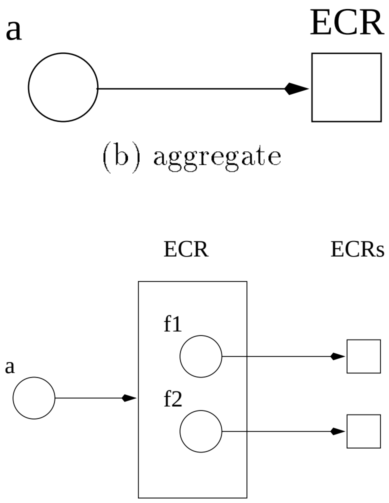
- First, both abstract locations are merged in the same way as in aggregate, then one of them then becomes the new representative.
- The next step is to transfer all the fields of the non-representative one into the representative one. However at this stage, it might happen that both of them possess some fields in common. In this case, we merge the ECRs of the common

fields. This is a recursive step because the merges ones might also have common fields.

- Finally, fields that exist only in the non-representative node are moved to the representative one.

```
class A
{
   Object f1;
   Object f2;

   void foo()
   {
      a = new A();
      a.f1 = new Object();
      a.f2 = new Object();
   }
}
```

(a) example



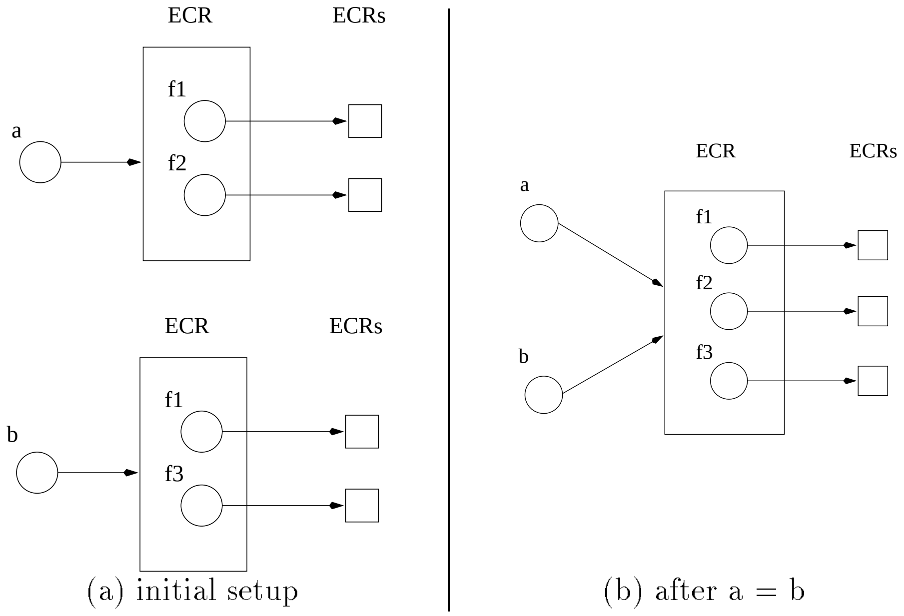
(b) aggregate

(c) with-fields

Figure 3.8: difference between aggregate and with-fields

Figure 3.9 illustrates the merging of abstract locations in the presence of fields. All the rules defined in aggregate remain valid for with-fields although three major differences exist. First when the rule specifies that two ECRs have to be merged, it is the above procedure that has to be applied. Second, when a merge involves an instance field reference of the form `a.i`, instead of merging `a`'s ECR, we now have to merge `a.i`'s ECR with the corresponding ECR(s) according to the rule under consideration. To get `a.i`'s ECR, we first get `a`'s ECR and then look for field `i`'s ECR inside `a`'s ECR.

Finally, arrays are also handled in a special way. As opposed to aggregate analysis where a reference to an array and the array elements are considered as a whole, in with-fields this is not the case. That is, array elements are considered as one special field called `arrayElements`. Therefore, all ECRs representing arrays contain a reference node named `arrayElements`. Consequently, merging ECRs involving arrays is handled in the same way as merging instance fields.



(a) initial setup (b) after a = b

Figure 3.9: nodes merging in with-fields

To determine the worst-case running-time of refers-to analysis, consider an ECR with F fields, where F is the maximum number of fields in an object. Merging this ECR with another ECR requires the union of their internal fields. The union of a pair of fields can produce at most $O(N\alpha(N, N))$ merges, where N is the number of ECRs in the program. Since the number of fields might grow linearly with the program size, there are at most N merges to be done inside an ECR. This means that the union of two ECRs, in with-fields analysis, can take up to $O(N^2\alpha(N, N))$ operations. And since, in the worst case, an ECR can be visited only N times, the overall worst case running-time of the algorithm is $O(N^3\alpha(N, N))$. However we believe that in practice the number of fields inside objects tends to be constant due to the tendency of programmers to create new classes when structures become too complex. Therefore we expect the running-time of the analysis to be quadratic. However we noticed during our experiments that, in practice, the analysis tends to scale well i.e., linearly.

Once we obtain the storage shape graph computed by refers-to analysis, it is possible to infer all expressions in the program altered or inspected by a statement. For instance, assume that we obtain the storage shape graph of figure 3.9 (b) after applying with-fields refers-to analysis and suppose that a Jimple statement defines `a.f1`, the write set of this statement must therefore contain `a.f1` and `b.f1` since according to the analysis `a` and `b` might refer to the same object.

Regarding implementation details, note that as for type-based analysis, elements carried around in read/write sets are representatives (ECRs) instead of actual expressions. This has the advantage of saving space and also, it avoids a traversal of the storage shape graph for collection of aliases. Indeed, if we carry around fields and arrays identifiers instead of their ECRs, then if, for instance, we want to check whether there is dependence between two statements which respectively read and write a variable, then we would have to collect all the aliases of the variables first and then make sure they do not interfere.

3.7 Limitations

Our analyses and optimization are performed under the closed world assumption. That is, we require that all the bytecodes of an application be present during analysis and optimization. Consequently, applications dynamically loading classes which are not part of the statically analyzed set of bytecodes are excluded. The same holds for applications with inherited classes added after analysis and optimization of the initial set of bytecode. However, in this case, a new iteration of the analysis and optimization can be performed to avoid the inconvenience. Despite this limitation, we believe that there is still a lot of Java programs that do not use Java's dynamic class loading mechanism and therefore are good candidates for being optimized by our solution; for instance, we can mention compilers and server side applications.

Now that we know how to compute side-effect information, in the next chapter, we show how this knowledge can be used by optimizing compilers to improve runtime performance of Java programs. The optimization on which we focus is loop-invariant removal.

Chapter 4

Loop Invariant Removal

To study the impact of side-effect on improving Java program performance, we have implemented *loop-invariant removal* [ASU86]. We have chosen this optimization because it is generally widely applicable. Further, Sun suggests a list of optimizations [Sun97] that should be applied manually by programmers on Sun's JDK1.1.3. Loop-invariant removal is the first one on this list. Even though we did not run our experiments on Sun's JIT but on the Linux JIT distribution from Blackdown [Bla], we believe that the same suggestion should hold. Further, since JITs do not have enough time to perform sophisticated analyses such as side-effect and transformations based on these analyses such as loop-invariant removal, there is still room for Java program compile-time analysis and optimization such as the ones presented in this thesis. In this chapter, we present our implementation of loop-invariant removal.

Loop-invariant removal aims at moving out of loops, expressions that always evaluate to the same value throughout all iterations of the loop. By moving statements or expressions out of loops, the dynamic number of instructions executed by the program is reduced. Further, when moved expressions are made of accesses to an object state on the heap, pointer dereferences can be saved. Finally, these accesses are typically assigned to temporary variables when moved to preheader, making them good candidate for register allocation, thus decreasing memory traffic.

Loop-invariant removal proceeds in three steps: first, loops are detected, then invariants inside loops are located and finally invariants are moved out of loops. In the next sections, we will cover each of these steps in detail.

4.1 Loop detection

Before being able to optimize loops, obviously, loops have to be detected first. Before going into the details of the algorithm, we first start by introducing the notion of a control flow graph, which will be used throughout this chapter. A control flow graph is a graph whose nodes are, in our case, program statements and which contains an edge between two nodes, if the control can flow from one node (statement) to the other. Successors of a node `n` are the nodes that are targets of edges starting at `n` and predecessors of a node `n` are those nodes that possess an edge leading to `n`.

In order to find loops inside a method, we use the algorithm described in [ASU86]. Basically, the idea comes from the observation that there is only one entry to a loop, this entry is called the header of the loop, and that as soon as an edge in the control flow graph comes back to the header, then a loop has been detected. The edge coming back to the header is called a back-edge. To detect back-edges, the algorithm uses the dominator relation. We say that a statement `D` dominates statement `S`, if all paths leading from the entry of the control flow graph to statement `S` go through `D`. We will just give the key idea of the algorithm computing the dominator relation here, that is, a statement `A` dominates a statement `B` only if all predecessors of `B` are dominated by `A`. More details on the algorithm can be found in [ASU86]. A back-edge, and consequently a loop, exists in the control flow graph, if a statement `S` in the control graph is dominated by another statement `D` and if there exists an edge from `S` to `D`. Figure 4.1 illustrates these notions. All statements located between `D` and `S` are statements inside of the loop. These statements are collected starting from `S` and by recursively collecting predecessors of `S` until `D` is reached.

As we will see in section 4.3, to fully benefit from loop-invariant removal, we need to transform while loops into repeat ones. More details on the rationale behind this transformation will be given in section 4.3 but for now, since this transformation implies that we need to distinguish between while and repeat loops, we will just give the criteria that allows one to differentiate between a while loop and a repeat one. Given the structure of loops in bytecode (as well as in Jimple) as presented in figure 4.1, a loop is a while loop, if the source of its back-edge (statement `S` in the figure) is directly followed by the header (the target of the back-edge, statement `D`) in the list of bytecode (Jimple) instructions. Figure 4.1(a) illustrates this case. In any other cases, the loop is a repeat one (figure 4.1(b)).

In our implementation once a loop has been detected, we compute its type ie.,

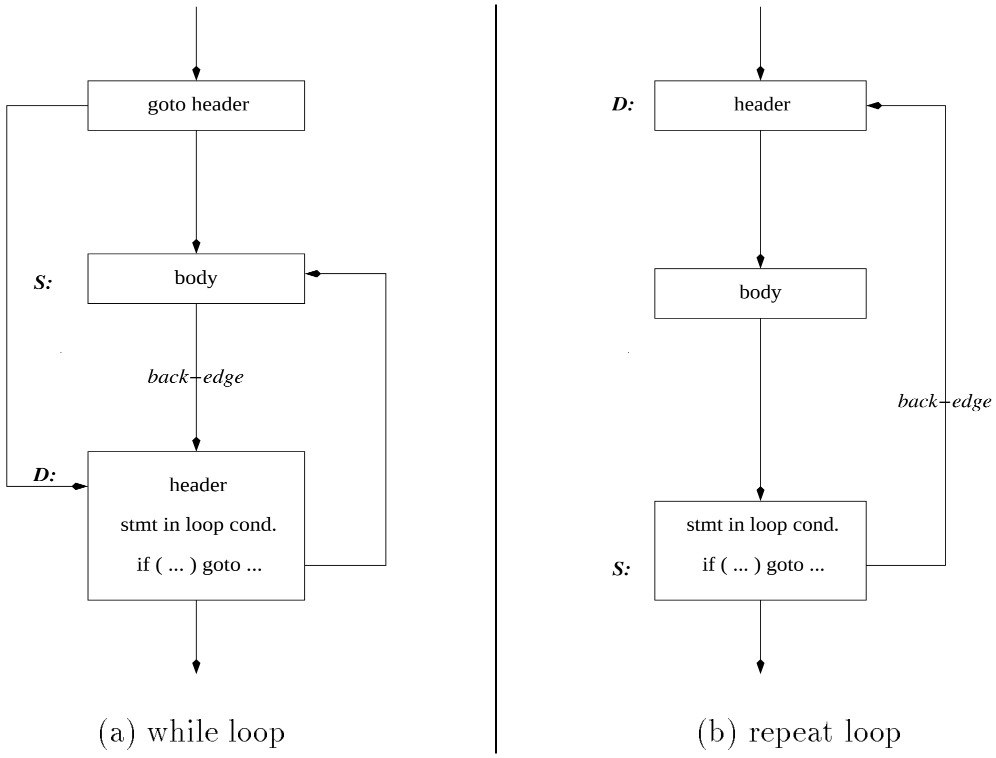


Figure 4.1: structure of loops in bytecode

whether it is a while or a repeat loop, as well as other properties that are needed by the other algorithms used in performing loop-invariant removal. These properties include detection of the exits of a loop as well as computation of variables that are read and written by the loop according to our side-effect analyses. Exits of a loop are statements inside the loop that have at least one of its successors out of the loop. To compute the set of locations read or written by the loop, we simply merge the read/write sets of each statement in the loop.

4.2 Detection of invariants

We now turn our attention to the problem of detecting invariant expressions inside loops. As we will see below, detection of invariant requires computation of ud-chains, and moving invariant statement of loops requires computation of du-chains. Therefore before going into the details of these algorithms, we first introduce the notion of ud/du-chains.

4.2.1 ud/du-chain

Ud-chains are computed using a standard iterative dataflow analysis [KU76] whose goal is to associate a use of a variable with all the definitions that might reach that use. A variable is used when its value is read. Soot already provides a version of ud-chain, but it works only for scalar variables ie., it will only associate a use of a local variable to statements that might define it, but it is unable to handle statements involving uses or definitions of field or array accesses. For this reason, we have extended the original ud-chains routine in Soot to benefit from side-effect information. This allows the routine to overcome the previous limitation. Intuitively, the idea behind the algorithm computing ud-chains is to let definitions of variables flow along the control flow graph to reach uses of variables they define. However, when a definition reaches another definition which define the same variable, then the former definition is killed and it is the latter that keeps flowing along the control flow graph until it reaches its uses or another definition. In which case, it is killed and the process can start again. In figure 4.2, we just mention the equation that drives the analysis. These equations have to be solved at each program point (statement) until a fixed point is reached. For more details on dataflow analysis, we let the reader refer to [ASU86] for their full coverage.

$$in[S] = \bigcup_{p \in Preds(S)} (out(p))$$
$$out[S] = gen[S] \cup (in[S] - kill[S])$$

$gen[S]$ is the set of definitions that arise in statement S. A definition is a pair (v, S), where v is the variable defined in statement S.

$kill[S]$ is the set of definitions in $in[S]$ that define the same variables as the ones defined in S.

$in[S]$ is the set of definitions that reach statement S.

$out[S]$ is the set of definitions that are valid after S.

Figure 4.2: equations for ud-chains

The main difference between our version of ud/du-chains computation and the scalar one already available in Jimple resides in $gen[S]$, the set of definitions that arise in statement S. In the scalar version, only statements defining a local variable are considered and there is possibly only one variable defined by these statements ie., the local. Whereas in our version, we have to consider statements defining an instance field or an array as well. Moreover, due to aliasing, there might be more than one variable defined at a given statement in our case. Consequently, when an instance field or an array element is defined in a statement, we will define $gen[S]$ as the set of all pairs (v, S), where v is a variable written by S according to side-effect analysis.

In the same vein, du-chains, which will be used later for the algorithm that moves invariant statements out of loops, is also another iterative dataflow analysis. Du-chains are exactly the inverse of ud-chains. That is, in du-chains, we want to know what are the uses of a variable that a given definition of the variable might reach. As for ud-chain, the idea behind the algorithm is to let information flow along the control flow graph, but this time, instead of definitions, uses are propagated backwards towards their definitions. The equation for solving the dataflow problem is given in figure 4.3. Du-chains require side-effect information in the same way as ud-chains did.

$$out[S] = \bigcup_{succ \in Succs(S)} (in(succ))$$
$$in[S] = gen[S] \cup (out[S] - kill[S])$$

$gen[S]$ is the set of pairs (v, S), where v is a variable used in statement S.

$kill[S]$ is the set of pairs (v, S) in $out[S]$, where v is a variable defined in S.

$in[S]$ is the set of pairs (v, S) (uses) that are propagated backward to statements predecessors of S.

$out[S]$ is the set of pairs (v, S) (uses) that reaches statement S.

Figure 4.3: equations for du-chains

4.2.2 Invariant detection

We now come back to our initial problem, that is detection of loop invariants. We are interested only by two kinds of Jimple statements:

- `lhs = rhs` (assignment statement)
- `invoke statement`

where `lhs` can either be a local variable and a field or an array access, whereas `rhs` can be a local, a field or an array access, a binary expression whose operands are locals or an invoke expression.

A statement in a loop is *invariant*, if for each of the variables (local, field or array access) it uses (reads):

- the variable is either constant or,
- all the definitions reaching the variable are located outside the loop, or
- there is exactly one definition inside the loop that reaches the variable and that definition is an instruction which has already been detected as invariant.

Furthermore, we can consider an invoke expression as invariant if two conditions are satisfied; first, it does not write any field or array variables and second it does not use (read) a variable (local, fields or arrays variables) which is written by another statement in the loop. These two conditions hold especially on access methods in Java since these methods are just used for returning an object's state.

As for ud-chains, if no side-effect information is available, loop-invariant removal would be restricted to statements which use and define local variables only. On the other hand, with read/write sets provided by either type-based analysis or refers-to analysis, loop-invariant removal can not only be extended to statements with accesses to an object state on the heap, but also to statements with method calls.

We now outline the contribution of side-effect analysis in the detection of loop-invariant statement described above. First, side-effect information is needed when getting all the uses of a statement. For instance, in a Jimple assignment statement, if `rhs` is a field access such as `o.f`, then uses of the statement must not only include `o.f` itself, but also any other field accesses aliased to `o.f`, because otherwise, the algorithm for detecting invariant statement would be wrong. Note that if `lhs` is a field access such as `o.f`, then `o` and only `o` needs to be added to the set of uses, since `o` is a read variable (whereas `o.f` is written, thus not used) and also because it is a local variable and consequently can not be aliased. Second, side-effect information is also needed when looking for definitions of a statement's uses. Detecting definitions of a use is exactly the role of ud-chains and as we have already seen they require side-effect information. Finally, when verifying whether an invoke expression is invariant, read/write sets provided by either refers-to or type-based analysis are not only required for obtaining all the variables that are read and written by the invoked method but also for all variables that are written by the loop.

Our implementation of the algorithm for loop-invariant detection works as follows:

1. Keep track of the order with which instructions are detected as invariant in the following two steps.
2. Visit each statement in the loop and verify, according to the definition of invariance described above, whether the statement is invariant.
3. Repeat step 2 until no more invariants are found during an iteration.

4.3 Invariant code motion

Once loop-invariant statements are detected, one could think that they can be directly moved to the preheader of the loop (the preheader is the region just before the header of the loop). Unfortunately, this is not the case. An invariant statement is truly movable only if satisfies the following two conditions. That is, it must:

- dominate the exits of the loop and,
- dominate all the uses of variables it defines.

Consider code fragment in figure 4.4 (a) where statement `n = 2` is invariant. If this statement was directly moved to the preheader of the loop, then the value printed at statement `print(n)` would always be `2`. But in the original program, if the loop body is never executed, then this statement would print the value `1`. This example illustrates why an invariant statement must always dominate loop exits in order to be movable.

<pre>n = 1;
while(...)
{
   if(x > 0)
      n = 2;   // invariant
}
print(n);</pre>
	<pre>x=0
	n=1
	do
	{
	   x += n; // use of n
	   n = 2; // invariant
	}
	while(...)</pre>

(a) invariant does not dominate loop exits	(b) invariant does not dominate its use

Figure 4.4: issues in moving invariant statements

Similarly, consider figure 4.4 (b). In this case, statement `n = 2` is also invariant and further, it dominates the exit of the loop. However, because it does not dominate the use of its variable `n` in statement `x += n`, if it was moved to the preheader, then the value assigned to `x` in the first iteration of the loop would be `2`, which is wrong because in the original program, the value assigned to `x` during the first iteration would be `1`.

Note that to obtain all the uses of a defined variable, we use du-chains defined in section 4.2.1. Du-chains also require side-effect information as we have already mentioned.

While checking whether a statement is movable to the loop preheader, the order with which invariant statements are checked against the above two conditions is the order with which they were found invariant by the algorithm for detecting invariant statement. If an invariant statement `S` satisfies both conditions, then it is removed from its location in the loop body and inserted at the end of the loop preheader. In figure 4.5, we give an example of a statement moved to the preheader. Part (a) is the original loop and part (b) shows the motion of statement (1) in part (a) to the preheader.

We do not always blindly apply the previously described procedure when a statement is found to be movable. If `S` is an invariant statement which satisfies both conditions and there already exists another invariant assignment statement `O` in the preheader whose `rhs` is equivalent to `S`'s `rhs`, then we do not add `S` to the preheader but we rather replace all uses of the variable defined by `S` (`S`'s `lhs`) in the loop body by `O`'s `lhs`. This way, we decrease register pressure that would have increased, if we had let two different variables storing the same value close to each other in the preheader. Another benefit from this transformation is that it decreases program size, even though the impact should be minimal. Figure 4.5 shows that the invariant statement at line (2) in part (a) does no longer appear in the program in part (b) because there is already a statement with the same `rhs` in preheader. Further, all the uses of `y`, the variable defined by the invariant statement , have been replaced by `x` in part (b).

```
do
{
(1)    x = o.f;             // invariant
(2)    y = o.f;             // invariant
       if( ... )
(3)       x = y.foo();
       else
(4)       y.bar();
}
while( ... )
```

(a) original program

```
x = o.f;
do
{
   if( ... )
      x = x.foo();
   else
      x.bar();
}
while( ... )
```

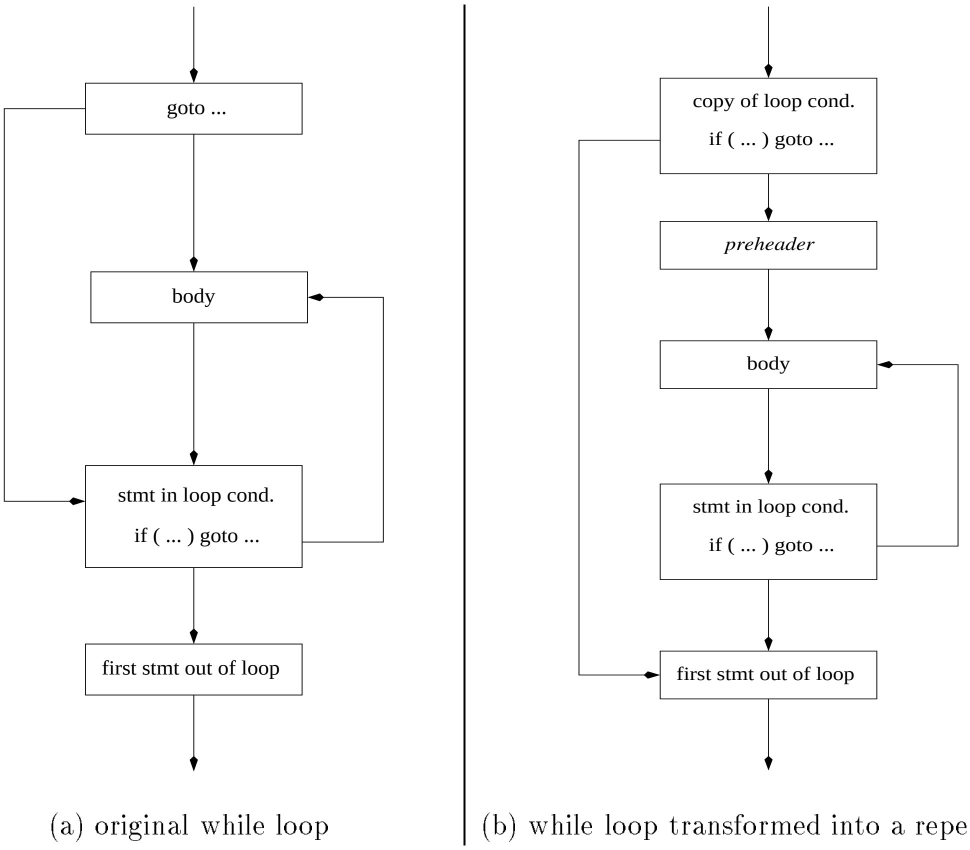
(b) after invariant code motion

Figure 4.5: moving invariant statements out of loop

4.3.1 Augmenting opportunities for optimization

In the previous subsection, we have presented the two conditions that must be satisfied by invariant statements in order to be movable. However, if a statement turns out to be non-movable, then there still exists ways to get around these limitations. In this subsection, we describe two techniques that augments opportunities for loop-invariant removal.

First, let us recall the first condition that a statement must satisfy in order to be movable. This condition states that the statement must dominate loop exits. If we consider while loops, then it is clear that no invariant statements can be moved out of while loops since a possible exit of a while loop is its condition and this exit will always dominates all other statements in the while loop. Furthermore, loops in bytecode, and consequently in Jimple, tend to be while loops. To overcome this problem, we apply a loop transformation that converts a while loop into a repeat one. As can be seen



(a) original while loop

(b) while loop transformed into a repeat one

Figure 4.6: transforming a while loop

from figure 4.6, the idea is to replace the `goto` that leads to the loop condition in the original while loop by a copy of the loop condition. This conditional checks whether the loop is entered and if not, flow continues directly at the first statement out of the loop. The consequence of this transformation is that some invariant statements that were not movable in the original while loop, might now be moved to the preheader, as shown by figure 4.7. Note that after this transformation, if no invariant statements are moved to the preheader, the loop executes exactly the same statements as the original while loop. Note also that a gain is obtained by applying the loop transformation, since the execution of the `goto` that leads to the loop condition in the original while loop is no longer required after the transformation.

Second, even if a statement is loop-invariant but it can not be moved to the preheader because it violates one of the two conditions for motion, then we can still

```
while( n > 0 )
{
   x = o.f;   // invariant but
              // does not dominate
              // exit.
   if( x > 0 )
     ...

   n--;
}
```

(a) invariant does not dominate exit

```
if( n > 0 )
{
   x = o.f;    // invariant moved
               // to preheader

   do
   {
      if( x > 0 )
        ...

      n--;
   }
   while( n > 0 )
}
```

(b) after transformation, invariant is moved

Figure 4.7: consequence of while loops transformation

have a chance to perform an optimization if the following condition holds: if the invariant statement, say `S`, is an assignment statement and if there exists already another invariant statement `O` moved to preheader and `O`'s `rhs` is equivalent to `S`'s `rhs`, then we can replace `S`'s `rhs` with `O`'s `lhs`. Figure 4.8 illustrates this optimization.

```
x = o.f;    // invariant already
            // in preheader
do
{
   if( ... )
      y = o.f;  // invariant but
                // not movable
   ...
}
while( ... )
```

(a) invariant but not movable

```
x = o.f;
do
{
   if( ... )
      y = x;  // rhs substitution
   ...
}
while( ... )
```

(b) rhs substituted

Figure 4.8: substitution of an invariant rhs

Finally, if a statement is invariant but it can not be moved to the preheader, nor can its `rhs` be substituted, then we do not try to further optimize it. However, possible remaining optimizations would be, for instance, to still move a statement's `rhs` to the preheader even if another statement with the same `rhs` does not already exist there. The problem with this approach is that the program running-time might increase because the computation associated with the `rhs` moved to preheader will now be executed in any case, whereas originally, this computation might have never been executed by the loop. Another option would be to unroll the loop once and to assign the `rhs` to a temporary variable just before the invariant statement and to use the temporary instead of the `rhs` anywhere else in the loop. Even though this solution will not suffer from the problem of the previous solution, it does, however, substantially increase the program size and it is not clear whether the gain will compensate for the time spent in dynamic compilation of a larger piece of code in environment such as JITs. Note that in our approach, we do not move code that might not be executed by the loop to preheader. However, while converting while loops into repeat ones, there is a small increase in the program size by the copy of the loop condition to the preheader. But we believe that it is not significant, because the number of statements forming the loop condition tend to be rather small in practice.

In the next chapter, we present our experimental results based on loop-invariant removal.

Chapter 5

Experimental results

In this chapter, we present and discuss both static and dynamic results collected during our experiments.

5.1 Platform and environments

Experiments were run on a 400MHz Pentium II running Linux. We measured runtime performance of our set of benchmarks under Blackdown's implementation of Java 2 for Linux [Bla] using both environments, the JIT and the interpreter. All running times are reported in seconds. None of the environments above performs optimizations on field or array accesses. Therefore, they will best reveal the impact of our side-effect analysis.

5.2 Benchmarks

We applied the analyses on a set of benchmarks with various sizes as outlined in Table 5.1. The benchmarks come from the `SPECjvm` [SPE] benchmark suite. `Compress` is a compression program based on a modified Lempel-Ziv-Welch method. `Db` is database emulator and `raytrace` is a graphic raytracer. Finally, our last benchmark is `javac`, the java compiler from Sun.

Figure 5.1 presents some characteristics of the benchmarks. The second column labeled `Jimple stmts` gives the number of Jimple statements and the third column gives the number of classes and interfaces present in the whole program.

benchmark	jimple stmts	classes	hierarchy depth	refs		time (sec.)	
				aggregate	with-fields	aggregate	with-fields
compress	26968	327	6	14539	16131	23	35
raytrace	29774	340	6	16141	17828	32	42
db	27721	321	6	14958	16498	27	38
javac	49492	486	8	28036	30155	93	115

Table 5.1: benchmarks characteristics

To confirm our assumption on space and time consumption of refers-to analysis (aggregate and with-fields), we give in columns labeled `refs` and `time`, respectively the number of reference nodes initially created by both analyses (this is good indication on the space requirement of the analyses), and the time required to perform each analysis (in seconds). As we can see, the number of reference nodes grows linearly with the program size and the time required to perform the analysis is in turn linear to the number of nodes created initially. Note that, in the table, we do not present numbers regarding type-based analysis, since it is clear that the number of representatives (class names or class and field names) used by the analysis and the time spent for visiting statements and collecting read/write sets are linear with the program size.

5.3 Static measurements

Before discussing the impact of the analyses on runtime performance which is the subject of the next section, we first present results collected at compile-time. These results comprise measurements done on the original versions of the benchmarks as well as the inlined versions. Inlining is a feature offered by the Soot framework [Sun99].

Table 5.2 part (a) shows for each benchmark and for each analysis, the number of loop-invariant statements (labeled `invariant` in the table) and among these, the number of statements fully moved to the preheader (`fully moved`) as well as those statements that could not be fully moved to preheader (because they did not dominate an exit of the loop or did not dominate all their uses), but whose `rhs` could still be replaced because another statement with the same `rhs` already exists in the preheader (`rhs replaced`). Part (b) of table 5.2 presents the same information as part (a) except that inlining has been performed on the original program.

The first trend exhibited by table 5.2 shows that on our set of benchmarks, fields matter. Indeed, both analyses (field-based and with-fields) exploiting high level information, such as field names, find significantly more loop-invariant statements than analyses that do not. However, number of invariant statements alone is not a good indication of expected runtime performance improvements. It is rather the number of statements fully and partially moved that does. But even for these statements, as we can see from the table, our observation on fields is still valid. This is especially true for refers-to analysis where we can see that moved statements more than double from aggregate to with-fields on each benchmark.

The second observation we can infer from the tables is that field-based analysis is the analysis of choice. Indeed over all the benchmarks, field-based analysis detects at least as many optimizable statements than any other analysis present in our experiments. This can be explained by the fact that refers-to analysis is rather deceiving. Indeed even though, with-fields analysis compares generally well with field-based (its corresponding analysis in type-based analysis), it may happen that it is just as good as class-based. This is the case for instance on `javac`. This bad behavior of refers-to analysis is mainly caused by the following two problems. First, refers-to analysis is transitive. That is, if reference variable `a` refers to the same object as reference variable `b` and if later on, `b` is found to refer to the same object as another reference variable `c`, then according to the analysis `a`, `b` and `c` refers to the same object, whereas in reality, `b` might now simply refer to same object as `c` and `a` is referring to a distinct object. Second, another important reason explaining the bad behavior of refers-to is the parameters merging step occurring at a call site. Since a method invocation triggers the union of actual parameters to formal ones (including the implicit this), two different calls whose targets (receivers) are initially pointing to distinct objects will after the call refer to the same object if the called method is the same because of the merging of the implicit parameter `this`. Similarly, if two variables referring to two different objects are passed as arguments to the same method call, then, according to the analysis, the variables end up referring to the same object because of parameters merging.

As can be seen from the tables, aggregate points-to analysis is the worst analysis on our set of benchmarks. Besides the problems mentioned above regarding refers-to, we explain the gap between results from aggregate analysis and class-based analysis in the way they handle arrays. Indeed, in class-based analysis when a statement reads

		refers-to		type-based	
		aggregate	with-fields	class-based	field-based
compress	invariant	14	67	25	73
	fully moved	4	24	17	26
	rhs replaced	0	2	0	3
raytrace	invariant	28	90	69	95
	fully moved	7	29	23	29
	rhs replaced	0	19	15	19
db	invariant	21	44	44	64
	fully moved	5	11	11	12
	rhs replaced	1	5	5	5
javac	invariant	180	276	269	294
	fully moved	43	116	119	127
	rhs replaced	6	16	17	19

(a) no inlining

		refers-to		type-based	
		aggregate	with-fields	class-based	field-based
compress	invariant	24	94	41	103
	fully moved	4	31	21	33
	rhs replaced	0	3	0	6
raytrace	invariant	30	108	90	113
	fully moved	4	38	31	38
	rhs replaced	0	19	15	19
db	invariant	42	77	76	124
	fully moved	5	11	20	21
	rhs replaced	1	5	5	5
javac	invariant	314	445	460	511
	fully moved	43	136	154	170
	rhs replaced	6	17	18	20

(b) inlining

Table 5.2: loop-invariant statements

or writes an element of an array, the analysis does not consider that the reference variable referring to the array (the base of the array) is read or written. In aggregate analysis, however, it is the case. For instance, consider `a` as the base of an array whose elements are reference variables and suppose that aggregate encounters the following Jimple statement: `a[i] = r`. Since the base of the array and the array elements are considered as the same entity in aggregate analysis, we would obtain that `a`, `a[i]` and `r` represent the same object. Therefore, writing an element of the array is now equivalent to writing `r`'s state.

We now turn our attention more specifically to results obtained after inlining (the results are presented in table 5.2 part (b)), we can see that as expected, inlining provides more opportunities for optimization. The number of loop-invariant statements increases by nearly a factor of two in all cases. Regarding the number of statements fully or partially moved to preheader, there is always a significant increase for all the analyses except for aggregate and with-fields on `db`.

Another result obtained after inlining, which at first sight might be surprising, appears on aggregate for `raytrace`. Compared to the non-inlined version, the number of statements fully moved decreases from seven to four after inlining. In fact, the three statements that are missing are invoke statements present in the non-inlined version that have been moved to the preheader by our optimizations. However the inliner can inline these three statements. Therefore, these three statements do not appear in the inlined version and obviously can not be moved to the preheader.

In table 5.3 and table 5.4, we respectively give the decomposition of statements fully moved to preheader and statements whose `rhs` have been replaced (we will also use the term partially moved to refer to this kind of statements in the rest of this document) into statements that use at least one non-scalar variable (`non-scalar`) - these statements can be detected only by side-effect analysis, and statements which use scalar variables exclusively (`scalar`). The tables also show the number of statements involving a method call moved to preheader (`invoke`). Detecting these statements also requires side-effect analysis.

These tables clearly demonstrate the contribution of side-effect analysis to loop-invariant removal since, as we can see, the proportion of statements detected as invariants based on side-effect analysis is rather high compared to statements that use scalar variables exclusively (except for aggregate analysis on `db` and `javac`, but this is due to the analysis relative weakness as we have seen earlier). This therefore suggests that expected runtime improvements from our optimization are mostly due

		refers-to		type-based	
		aggregate	with-fields	class-based	field-based
compress	non-scalar fully moved	4	24	17	26
	scalar fully moved	0	0	0	0
	invoke fully moved	0	0	0	0
	non-scalar rhs replaced	0	2	0	3
	scalar rhs replaced	0	0	0	0
	invoke rhs replaced	0	0	0	0
raytrace	non-scalar fully moved	6	28	22	28
	scalar fully moved	1	1	1	1
	invoke fully moved	3	12	9	12
	non-scalar rhs replaced	0	19	15	19
	scalar rhs replaced	0	0	0	0
	invoke rhs replaced	0	6	5	6
db	non-scalar fully moved	2	8	8	9
	scalar fully moved	3	3	3	3
	invoke fully moved	0	0	0	0
	non-scalar rhs replaced	0	4	4	4
	scalar rhs replaced	1	1	1	1
	invoke rhs replaced	0	0	0	0
javac	non-scalar fully moved	19	80	83	91
	scalar fully moved	24	36	36	36
	invoke fully moved	0	1	2	2
	non-scalar rhs replaced	5	15	16	18
	scalar rhs replaced	1	1	1	1
	invoke rhs replaced	0	0	0	0

Table 5.3: details on statements fully and partially moved without inlining

to these statements detected by side-effect analysis.

Note that after inlining, the side-effect analysis contribution is even greater since the number of statements involving non-scalar variables and method calls increases compared to the non-inlined version.

The benchmark `raytrace` presents an interesting result. The number of statements with method calls moved or replaced is quite high since they represent more than half of all the statements that were optimized. Looking at the nature of these invoke statements shows that they are accessors methods ie., small methods that just

		refers-to		type-based	
		aggregate	with-fields	class-based	field-based
compress	non-scalar fully moved	4	31	21	33
	scalar fully moved	0	0	0	0
	invoke fully moved	0	0	0	0
	non-scalar rhs replaced	0	3	0	6
	scalar rhs replaced	0	0	0	0
	invoke rhs replaced	0	0	0	0
raytrace	non-scalar fully moved	3	37	30	37
	scalar fully moved	1	1	1	1
	invoke fully moved	0	0	0	0
	non-scalar rhs replaced	0	19	15	19
	scalar rhs replaced	0	0	0	0
	invoke rhs replaced	0	0	0	0
db	non-scalar fully moved	2	8	17	18
	scalar fully moved	3	3	3	3
	invoke fully moved	0	0	0	0
	non-scalar rhs replaced	0	4	4	4
	scalar rhs replaced	1	1	1	1
	invoke rhs replaced	0	0	0	0
javac	non-scalar fully moved	19	101	119	135
	scalar fully moved	24	35	35	35
	invoke fully moved	0	0	0	0
	non-scalar rhs replaced	5	16	17	19
	scalar rhs replaced	1	1	1	1
	invoke rhs replaced	0	0	0	0

Table 5.4: details on statements fully and partially moved with inlining

returns an object state. If these statements were located at critical regions in term of running-time, then moving them out of a loop would be really beneficial since the cost of a method call is no longer paid. In fact, this transformation would have the same impact as inlining. Also, it should be noted that on our set of benchmarks, as we can see from table 5.4 where no more invoke statements are detected as movable, all statements with method calls detected as movable before inlining are, afterwards, inlined. Note also that these tables confirms what we mentioned earlier about the three invoke statements that were fully moved to preheader on aggregate analysis for `raytrace`. That is, before inlining, they were moved to preheader and after inlining, they no longer exist since they are inlined.

5.4 Dynamic results

In this section, we discuss results obtained at runtime on our set of benchmarks. Our methodology for measuring runtime behavior was to run the benchmark five times for each level of optimization and to take the minimum of the sum of the user and the system time on a completely unloaded machine.

We present one graph for each benchmark as can be seen from figure 5.1. These graphs are drawn according to the results presented in table 5.5. Each graph is made of four groups of measurements. The first two groups present running-times under the JIT, respectively for the non-inlined and the inlined version and the last two groups, on the right side of a graph, present running-time under the interpreter, also for both the inlined and the non-inlined version. Each group of results is made of six bars representing six different levels of optimization. All our measurements are normalized against the Jimple version. Reporting the results in this way exposes the impact of each of our analysis. The first bar on the non-inlined version presents the normalized time for the Jimple version. Its value is always 0. For the inlined version, the first bar represents the running-time of the benchmark after inlining and it is labeled `inlined`. The second bar, valid for all versions and labeled `repeat loops`, represents the running-time after transforming all while loops in the benchmark into repeat ones, as explained in chapter 4, and before applying loop invariant removal. We present this result because the loop transformation by itself can improve benchmarks performance. The reason is that the goto instruction in figure 4.6 (a) is no longer executed after the transformation as can be seen from figure 4.6. Finally, our graphs present results obtained after applying loop-invariant removal based on our set of side-effect analysis starting from the more conservative analyses ie., aggregate analysis and class-based analysis, to the more powerful ones, with-fields analysis and field-based analysis.

We now discuss in more details the results obtained from each benchmark.

As can be seen from figure 5.1, our optimizations are rather efficient on `compress`. As expected, the better the analysis, the better the runtime improvement. Looking specifically at the results for the JIT shows that without inlining, our best analysis (with-fields) yields a significant speedup of around 0.10 over the Jimple version. Furthermore, after inlining, field-based and with-fields give an even better improvement of

0.20 on the benchmark, provided that inlining, alone, already improves running-time by a factor of 0.20. As expected from our static measurements, inlining has provided more opportunities for optimization and this explains the gain in term of speed-up over the Jimple version. Overall, it is interesting to note that the combination of inlining, loop-invariant removal and side-effect analysis provide a very significant speedup of more than 0.40 on this benchmark. Finally, also note that aggregate analysis gives a small speedup of 0.03, whereas class-based analysis performs significantly better. Class-based analysis, however, is not as effective as field-based analysis and with-fields analysis. These results match with our static measurements. `Compress` demonstrates that loop-invariant removal associated with our set of side-effect analyses can be very effective in optimizing Java programs.

Profiling `compress` showed that it spends most of its time (68 %) in two methods, namely `compress()` and `decompress()`. In figure 5.2, we present excerpts from the non-inlined version of method `decompress()`. Part (a) shows a very hot while loop located inside method `decompress()` in its original Jimple representation. Part (b) presents the same loop after applying loop-invariant removal based on aggregate analysis, whereas part (c) and part (d) shows the transformation after respectively class-based and both field-based and with-fields analyses. As we can see, aggregate analysis is not able to detect any invariants inside method `decompress()` and only transformation of the while loop into a repeat one was performed. On the other hand, class-based analysis moves four assignment statements to the preheader and both analyses that take field information into account (with-fields analysis and field-based analysis) are able to detect one more movable statement. As can be seen, there is a direct correlation between runtime performance and number of optimized statements on this benchmark. Therefore, this example clearly illustrates the impact of the different analyses on runtime performance. Note that even though optimizations performed inside method `decompress()` amounts for a great part of the improvement we see at runtime, there are other parts of the benchmark which also contribute to the overall improvement. This is for instance the case for method `compress()`, but we do not show them here.

Looking at results from the interpreter shows that the trend is similar to the JIT.

Finally, the last observation we can make out of `compress` is that even though field-based analysis moves and replaces more invariant expressions than with-fields analysis, the running-time of the benchmark is slower after inlining on the interpreter and without inlining on the JIT. After spending some time investigating the source of this problem, our conclusion is that it finds its root at the architecture level.

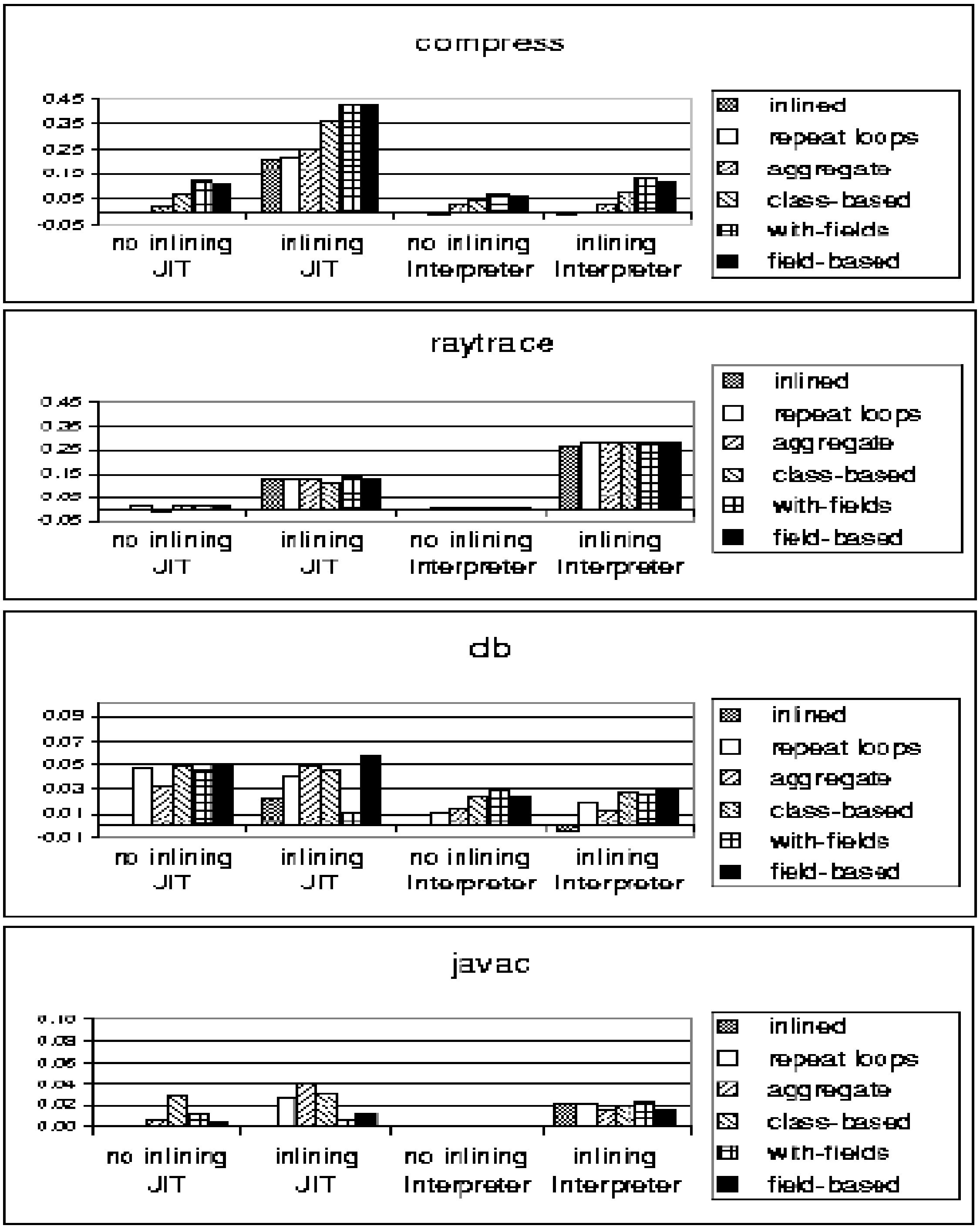


Figure 5.1: speed-up

```
    ...
    goto label4;
label3:
    r6 = r0.de_stack;
    r7 = r0.tab_suffix;
    r8 = r7.tab;
    i11 = r8[i2];
    r9 = r6.tab;
    r10 = r6;
    i12 = r10.index;
    r10.index = i12 + 1;
    r9[i12] = i11;
    r11 = r0.tab_prefix;
    i2 = r11.of(i2);
label4:
    if i2 >= 256 goto label3;
    ...
```

(a) original

```
    ...
label3:
    if i7 < 256 goto label5;
label4:
    r5 = r0.de_stack;
    r6 = r5.tab;
    i11 = r5.index;
    r5.index = i11 + 1;
    r7 = r0.tab_suffix;
    r8 = r7.tab;
    r6[i11] = r8[i7];
    r9 = r0.tab_prefix;
    i7 = r9.of(i7);
    if i7 >= 256 goto label4;
label5:
    ...
```

(b) aggregate

```
    ...
label3:
    if i7 < 256 goto label5;

    r6 = r0.tab_suffix;
    r2 = r6.tab;
    r7 = r0.de_stack;
    r8 = r0.tab_prefix;
label4:
    r9 = r7.tab;
    i11 = r7.index;
    r7.index = i11 + 1;
    r9[i11] = r2[i7];
    i7 = r8.of(i7);
    if i7 >= 256 goto label4;
label5:
    ...
```

(c) class-based

```
    ...
label3:
    if i7 < 256 goto label5;

    r7 = r0.tab_prefix;
    r8 = r0.de_stack;
    r9 = r0.tab_suffix;
    r2 = r9.tab;
    r10 = r8.tab;
label4:
    i11 = r8.index;
    r8.index = i11 + 1;
    r10[i11] = r2[i7];
    i7 = r7.of(i7);
    if i7 >= 256 goto label4;
label5:
    ...
```

(d) with-fields and field-based

Figure 5.2: different levels of optimization in method `decompress`

		JIT		interpreter	
		no inlining	inlining	no inlining	inlining
compress	jimple	66.74	55.26	439.58	442.66
	repeat loops	66.67	54.76	443.06	441.70
	aggregate	65.52	53.40	426.92	428.67
	class-based	62.33	49.25	419.93	408.66
	with-fields	59.54	46.81	412.48	387.85
	field-based	60.09	46.79	413.25	394.13
raytrace	jimple	54.14	47.89	124.11	98.22
	repeat loops	53.40	48.09	123.74	97.09
	aggregate	54.56	47.86	123.76	97.21
	class-based	53.38	48.65	122.99	97.24
	with-fields	53.37	47.62	123.48	97.25
	field-based	53.28	47.89	123.66	97.16
db	jimple	150.72	147.48	256.64	258.05
	repeat loops	143.97	144.95	253.76	251.85
	aggregate	145.93	143.72	253.02	253.33
	class-based	143.67	144.12	250.63	249.92
	with-fields	144.28	149.14	249.30	250.07
	field-based	143.71	142.46	250.67	249.02
javac	jimple	76.10	76.22	141.18	138.15
	repeat loops	76.40	74.09	141.68	138.16
	aggregate	75.54	73.19	141.39	139.09
	class-based	73.97	73.86	141.35	138.52
	with-fields	75.22	75.63	141.86	138.09
	field-based	75.73	75.23	141.77	139.04

Table 5.5: running-time

More precisely, we believe that the slow down is likely due to a bad behavior of the instruction cache.

Turning our attention to the benchmark `raytrace`, we can observe that loop-invariant removal has little effect on improving the benchmark performance. Note that we have changed the scale of the graph to better reflect variations between the different analysis results. On `raytrace`, there is only a minimal speedup of around 0.02. This has to be accounted mainly for transforming while loops into repeat ones. This suggests that invariants moved to preheader by our analyses are, in this case, not located at "hot spots" in the benchmark. However, notice that inlining is quite effective on this benchmark.

Let us now consider the results for the benchmark `db` shown on figure 5.1. The interesting point is that we get speedups ranging from 0.02 to 0.05 over the Jimple and the inlined version. However, this benchmark seems to also suffer from the bad behavior of the instruction cache mentioned earlier for `compress`.

The same conclusion as for `db` is valid for the benchmark `Javac`. The best improvement we can get is a speedup of 0.04 on aggregate analysis with the JIT and in the presence of inlining. However, on the JIT, it seems that the more optimization there is, the slower the benchmark is. This, we believe, is again a reminiscence of the instruction cache problem. Note that on both graphs for `db` and `Javac`, the scale has been changed again but this time, it is important to realize that for these two benchmarks, a gap which may appear as very significant on the graphs, might only represent a difference of 0.01 in term of speedup. Therefore, gaps between the different results should be interpreted carefully on these graphs.

Chapter 6

Related work

In this section, we present work related to the different side-effect analyses presented in this thesis. We first look at work done on side-effect analysis for languages without pointers and where aliasing can only occur through parameter passing. Then, we describe a side-effect analysis developed specifically for Java, and finally, we turn our attention to work related to points-to analysis which can serve as a basis for side-effect analysis as we have shown in this thesis.

6.1 Side-effect analysis for language without pointers

One of the first who thoroughly covered side-effect analysis was Banning [Ban79]. His work addresses languages without pointers and where aliasing can only occur through parameter passing. His solution decomposes the problem of computing $MOD(s)$, the set of variables that might be altered by a computation s, into the resolution of the following three subproblems. Aliasing is ignored until late in the computation.

$IMOD(p)$ is the set of variables immediately modified by procedure p without taking into account effects of procedure calls occurring inside p.

$GMOD(p)$ is the set of variables altered in procedure p as a result of an invocation of p. This includes variables in p, modified as a consequence of procedure calls originating from p.

$DMOD(s)$ is the set of variables that might be modified by statement s ignoring any aliasing effects. When s is a procedure call, $DMOD$ is simply a specialization of $GMOD$ on call site s.

Banning's approach for computing MOD starts by first computing $IMOD$ for each procedure. Because aliasing and effects of calls occurring inside p are ignored, it is straightforward to deduce variables in $IMOD(p)$.

Given that $GMOD(p)$ is defined as the union of variables in $IMOD(p)$ and variables that might be altered by calls originating from p, and since $IMOD(p)$ is already known, $GMOD$ computation reduces to the problem of detecting the effect of procedure calls inside p. Assuming that there exists a call to procedure g inside p, the effect of g on p is the set made up of global variables possibly modified by g plus actual variables in p that are bound to formal parameters of g where these formals are written. As we can see, computation of $GMOD(p)$ is based on $IMOD(p)$ and on $GMOD$s of all procedures called from p. By using a call graph and propagating information from leaves to root, it is therefore possible to compute $GMOD$ for each procedure. In presence of recursion, a fix-point analysis has to be performed.

Given $GMOD(p)$, $DMOD(s)$, the set of variables that might be altered by statement s, is simply the result of mapping variables in $GMOD(p)$ to actual parameters of s in case where statement s is a call to procedure p. The resulting set must also contain global variables modified by p. In case where s is not a procedure call, since aliasing is not taken into account, it is obvious to deduce $DMOD(s)$.

Finally, $MOD(s)$ is simply $DMOD(s)$ plus any variables that might be aliased to a member of $DMOD(s)$.

Notice that the solution for USE, the set of variables whose values might be read by computation s, is obtained in a way similar to MOD, therefore it will not be described any further here.

We now briefly summarize Banning's approach to alias computation. It stems from the observation that inside a procedure, aliases can only occur between reference parameters or between global variables and reference parameters. His algorithm is based on a fix-point iteration whose aim is to collect all pairs of aliases. Initially, each variable is considered as an alias of itself. Then, given a procedure p with a call site s whose target is a procedure g, if two actual parameters x and y of s are aliased inside p, then inside procedure g, both formal parameters x' and y' mapped to x and y are also aliases. Similarly, if at call site s a global variable v is aliased with actual parameter x, then in procedure g, the formal parameter x' mapped to x is also aliased with v.

The main problem with Banning's approach is that it requires solving a global dataflow analysis whose running-time is $O(NE)$ bit vector steps, where N and E are respectively the nodes and the edges of the underlying flow graph used for solving the dataflow problem and where sets are represented as bit vectors. Since the size of bit vectors and the number of nodes in the flow graph are linear with the program size, the worst case running-time of Banning's solution is $O(N^2E)$.

Cooper and Kennedy [CK88] proposed an improvement over Banning's work. Their main contribution stems from the observation that the side effect problem can be subdivided into two subproblems: side-effects to parameters passed by reference and side-effects to global variables.

In the first subproblem, the goal is to discover all variables that are either directly modified inside a procedure p or passed by reference to another procedure and modified as a side effect of an invocation of that procedure. Solving the reference parameter problem requires the use of what they call the *binding multi-graph.* A program binding multi-graph captures relations between the program's formal parameters. A node in the graph represents a program formal parameter whereas an edge between formal parameters f_1 and f_2 exists if there is a call site where f_1 is passed as the i-th argument to a procedure which declares f_2 as its i-th formal parameter. From the binding multi-graph, one can deduce whether formal parameter f of procedure p is modified as a result of an invocation of p. If there is no strongly connected component (SCC) in the graph, then f is modified either if it is altered locally in p or if any formal reference parameters successors of f in the graph are modified locally in their declaring procedure. In the presence of strongly connected components, a formal parameter f member of the SCC is modified if one of the SCC's members is modified. Given the fact that a formal parameter might be modified as a result of an invocation of its declaring procedure, it is then possible to deduce all variables that might be modified after being passed as a parameter. It suffices to check whether the variable has been bound to a modified formal parameter. Solving this subproblem takes no more than $O(N + E)$ where N and E are the number of nodes and edges in the binding multi-graph. These nodes and edges are bounded by the number of nodes and edges in the program's call graph.

The second subproblem is the one related to global variables, Cooper and Kennedy noticed that a global variable is modified by a procedure p only if it is modified locally in p or if it is modified through a call chain starting at p. An adaptation of Tarjan's SCC detection algorithm is sufficient for solving this problem. The use of the SCC algorithm is similar to our approach presented in this thesis where the effect of the elements composing the SCC is summarized by a representative node. As above, this step takes no more than $O(N + E)$ bit vector steps which means an overall running time of $O(N^2 + NE)$.

In Cooper and Kennedy's solution, $GMOD$ computation only differs from Banning's solution. It is obtained by the union of both solutions to the formal parameter problem and to the global variable problem. Otherwise, $DMOD$ and MOD computations are similar to Banning's approach.

As for our side-effect analyses for Java described in this thesis, Banning's and Copper and Kennedy's analyses solutions are flow and context insensitive. However, our approach differs from their works in that Java is an object-oriented language which implies that object-oriented features such as fields or virtual method calls need to be modeled by our side-effect analysis. But above all, the main difference resides in the fact that Java has references which further complicates the analysis. Finally the last difference between their work and ours resides in the fact that they do not show the possible impact of their analysis. No insight into the precision of the analysis is given nor do they present subsequent analysis or optimization which could exploit the information provided by side-effect analysis.

6.2 Side-effect analysis for Java

Not many side-effect analyses have been proposed for Java. Clausen [Cla97] developed an inter-procedural side-effect analysis which is the basis of our class-based analysis and field-based analysis. His method assigns different levels of purity to bytecode instructions. The purity of an instruction for an object field or an array element present in a program, is

- $pure$ if the instruction does not alter or inspect the field or array element,
- $read - only$ if the instruction reads but does not write the element,
- $write - only$ if it writes but does not read the element and finally,
- $read/write$ if it both reads and writes the field or array element.

These different types of purity form a partial ordering where $pure \sqsubseteq read-only \sqsubseteq read/write$ and $pure \sqsubseteq write-only \sqsubseteq read/write$. The purity of a group of instructions, such as a method body or a loop, on a field or array element can be inferred by computing the least upper bound of the corresponding purities of each instruction.

Clausen proposes four variations of the analyses with different granularity. Fields and arrays are grouped by representatives and the analysis precision varies according to the way fields and arrays representatives are chosen. The first analysis, called *simple purity analysis*, assumes that if a field or array is read or written by an instruction, then all other fields and arrays are also read and written. Here only one representative is used to classify fields and arrays. This is a very conservative analysis. The second one, *subclass-based purity analysis*, uses the least class of which all classes with affected fields are subclasses. The third one, *class-based purity analysis*, uses the declared class of the affected field as representative. Finally, *field-based purity analysis*, the less conservative approach, uses the declared class as well as the field identifier as a representative of the affected field. Our class-based and field-based analyses are very similar to Clausen's class-based and field-based purity analyses. However, a difference exists and resides in the way arrays are handled. Clausen uses different representatives only for arrays known in the bytecode (Integer, Long, Float, Double and Object), whereas our analyses uses a distinct representative not only for these arrays, but also for any user defined arrays. Our analyses are therefore more precise. Clausen's analyses were applied to two program optimizations, namely dead code elimination and loop-invariant removal. His results shows that loop-invariant removal benefits from the analysis and that on one of his benchmarks, a running-time speed-up of 25% was possible.

6.3 Points-to analysis and side-effect analysis

Points-to analysis can serve as a basis for side effect analysis. In this section, we focus on Steensgaard's work on points-to analysis for C since it is the foundation of our refers-to analysis for Java.

6.3.1 Agreggate points-to

Steensgaard [Ste96b] developed a flow and context insensitive inter-procedural points-to analysis based on type-inference on a non-standard type system. The first variation of the analysis does not deal with objects with internal structure, ie. struct or union in C. Each location in a program is associated with a type. Each type has a component type which is used to give an indication on the usage of a location. That is, the component type ranges only over two values, either the location serves as a store for a pointer or it does not. In case of a location used as a pointer store, its component type is determined by the type associated with the pointed location, hence there exists a relation between types in the type system. This relation can be viewed as a model of a program's storage shape graph. A node in the graph would represent a type and an edge a relation between two types. This relation can be interpreted as a points-to relationship between locations represented by the node source of the edge and locations represented by its target. The graph models a conservative approximation of all the possible dynamic storage configurations.

The analysis is characterized by a set of typing rules specified for each C expression. These typing rules must ensure that if different locations might point to the same location then the component type of each type associated with these locations must be the same. Initially, each location is associated with a distinct type. As type inference proceeds, the typing rules impose constraints on types associated to locations. Two kinds of constraints are defined, namely equalities and inequalities. Equalities force two types to be equal. However, on an expression such as $x = y$, Steensgaard introduces the notion of inequality since it might happen that at the time when the expression is processed, it is not known whether y will store a pointer or not. Using inequality allows x's type to be "independent" from y's as long as y is not found to be a pointer store. However, if it is the case, x's type must then be equal to y's. Not forcing the types to be equal as long as it is not known whether y stores a pointer renders the analysis less conservative.

Our aggregate refers-to analysis is similar in its approach to the above description. However some differences exist. Using Steensgaard's terminology, our analysis could be described as follows: initially, each local reference variable is also associated with a unique and distinct type. These types could also be seen as representative of anonymous heap locations where the variables point to. As program statements are visited, constraints imposed by the typing rules must be satisfied. The property that these rules must enforce is that on reference variable assignment, type associated with both variables must be the same. In our analysis for Java, inequalities in typing rules are not necessary since we are sure that local reference variables will only store pointer values thanks to strong typing in Java.

The main advantage of Steensgaard's analysis and our aggregate refers-to analysis is that their running-time is almost linear, which means that analysis of large programs is possible.

6.3.2 Points-to with structures and unions

Since Steensgaard's almost linear points-to analysis does not consider elements of structured object in the analysis, its precision might suffer. Therefore Steensgaard proposed an extension which handles objects with internal structure [Ste96a]. The theoretical worst-case running-time of the new analysis is no longer almost linear but rather $O(N^2)$. Still, in practice it turns out that the analysis is linear. Type inference on a non standard type system is still used but this time, the type system is slightly different compared to the initial version described in the previous subsection. The set of types now includes *blank*, used initially for each location, *simple* for locations accessed as a whole, *struct* for locations accessed as structured objects and finally *object* for locations accessed in a way that is not covered by the other types. A lattice is formed out of these types and its partial ordering is as follows: $blank \sqsubseteq simple \sqsubseteq object$ and $blank \sqsubseteq struct \sqsubseteq object$. When a typing rule forces two types to be equal then the new type is the least upper bound of both types in the lattice.

An interesting point to notice is that there is a major difference in the way structured objects are handled between our approach in with-fields refers-to analysis and Steensgaard's points-to analysis with structures. Structures are composed of components accessed as a whole. When merging two structured objects in Steensgaard's solution, the new structure created out of them will be made of common components accessed as a whole appearing as a common prefix to both structured objects to be merged, plus another component accessed as a whole which is the representative of all components located after the common prefix. In with-fields refers-to analysis we do not proceed in a similar way. When two objects with internal structures (fields) are merged in our approach, the new representative will not only contain common elements but also includes all elements which appears exclusively in each object. Consequently, our solution is more precise. However, this gain in precision has a cost. Our with-fields analysis theoretical worst-case running time is $O(N^3)$, thus worse than Steensgaard's. Despite this, our experiments shows that our analysis is practical.

Chapter 7

Conclusion

Java is a fairly new language which, when compared to C, does not yet possess as sophisticated compilers with advanced optimizations. One of Java's characteristic, as for any object-oriented language, is the extensive use of instance fields. Further, the lifetime of fields spans multiple method invocations making it necessary to reason inter-procedurally about them. Good techniques that permit a better understanding of fields usage are of prime importance for optimizing Java bytecode.

In this thesis, we have focused on side-effect analysis, a technique aimed at determining variables that are inspected (read) or altered (written) by a computation. Side-effect analysis allows disambiguation of programs memory usage. The additional knowledge brought by the analysis increases opportunities for optimization.

The main contribution of this thesis is the design and implementation of two groups of context and flow insensitive side-effect analyses. The first one, called type-based analysis, exploits type and high level information such as field names to conduct the analysis. The goal of type-based analyses is to conservatively approximate the set of fields or array elements affected by the execution of an instruction. Two variations of the analysis are presented, class-based analysis which uses the declaring class of a field as a criteria for grouping aliased fields together and field-based analysis which uses the declaring class as well as the field name as the representative, and is thus less conservative.

The second group of analyses developed in this thesis is called refers-to analysis. These analyses have been designed to perform better than the type-based analyses. Indeed, the main disadvantage with type-based analyses is that the granularity of the analysis is at the class level. That is, any objects belonging to the same class will always end up being considered as aliases even though a simple analysis would show the contrary. Therefore, we have designed refers-to analysis to lower the analysis granularity down to the object level. Refers-to analysis is our adaptation for Java of Steensgaard's points-to analysis for C. Refers-to analysis aims at building a model of the storage shape graph capturing all the possible dynamic storage configurations. As for type-based analyses, two variations of the analysis were investigated, aggregate refers-to analysis and with-fields refers-to analysis. The main difference between the two resides in the fact that with-fields refers-to analysis uses objects internal structure (fields) as a key element of the analysis, hence a greater precision.

To assess the efficiency of the different side-effect analyses, we have implemented an optimization called loop-invariant removal. This optimization benefits from information collected by the different analyses. Loop-invariant removal, as its name suggests, moves expressions whose arguments are invariant out of loops. The optimization as well as the analyses were developed under Soot, a framework aimed at bytecode manipulation and optimization.

We have run our experiments on a set of Java benchmarks of various sizes and we have provided both static and dynamic numbers. From our set of static measurements, the main observation is that fields matter. Indeed, the number of invariant expressions and the number of statements moved out of loops increase significantly for analyses which use information about fields as a key element of the analysis. Another important observation is that refers-to analysis is rather deceiving. The gain expected from increasing the granularity of the analysis by including fields is diminished by merges occurring at call sites. To overcome this problem, a solution would be to add context-sensitivity to the analysis. We plan to investigate this possibility in the future. We have also applied the optimization without the help of any of our side-effect analyses. This shows that side-effect analysis has an non-negligible impact. Finally, since the Soot framework provides facility for inlining, we have also collected results about our analyses and optimizations with regards to inlining. We conclude that inlining augments opportunities for optimization.

Regarding our dynamic results, we observe that the combination of loop-invariant removal and side-effect analysis can give a speed-up of up to 20%. After inlining, this speed-up can raise as high as 40%. However it should be noted that some of our dynamic results are distorted because of a problem which takes its root at the architecture level. After careful analysis, we believe that this problem comes from a bad behavior of the instruction cache. We plan to further investigate this problem in the future.

As a conclusion, we have a demonstrated in this thesis that side-effect analyses is certainly a key element in optimizing Java programs.

Bibliography

[AGS97] Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 134–145, Las Vegas, Nevada, June 15–18, 1997. *SIGPLAN Notices,* 32(6), June 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[ATCL+98] Ali-Reza Adl-Tabatabai, MichałCierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* [PLD98], pages 280–290. *SIGPLAN Notices,* 33(6), June 1998.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, San Antonio, Texas, January 29–31, 1979. ACM SIGACT and SIGPLAN.

[Bla] URL: http://www.blackdown.org.

[CGDD97] 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.

[CK88] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, Atlanta, Georgia, June 22–24, 1988. *SIGPLAN Notices,* 23(7), July 1988.

[Cla97] Lars R. Clausen. A java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, November 1997.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, New York, June 20–22, 1990. *SIGPLAN Notices,* 25(6), June 1990.

[DGC95] 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.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* [PLD98], pages 106–117. *SIGPLAN Notices,* 33(6), June 1998.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices,* 29(6), June 1994.

[GH99] 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.

[GJS97] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1997.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, California, June 17–19, 1992. *SIGPLAN Notices,* 27(7), July 1992.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, USA, 1997.

[PLD98] *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montréal, Québec, June 17–19, 1998. *SIGPLAN Notices,* 33(6), June 1998.

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

[SPE] URL: http://www.spec.org/osg/jvm98/.

[Ste96a] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In Tibor Gyimothy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 136–150, Linköping, Sweden, 24–26 April 1996. Springer.

[Ste96b] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, Florida, January 21–24, 1996.

[Sun97] SunSoft. Java on solaris 2.6: A white paper. 1997.

[Sun99] Vijay Sundaresan. Practical techniques for virtual call resolution in java. Master's thesis, McGill University, Montreal, Canada, September 1999.

[Tar81] Robert E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

[Tow] URL: http://www.towerj.com.

[VRH98] Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. Sable Technical Report 4, Sable Research Group, McGill University, July 1998.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Las Vegas, Nevada, January 28–30, 1980. ACM SIGACT and SIGPLAN. Extended abstract.