Java Method Call Devirtualization with Soot

Brown Farinholt Kristen Wetts Rizwan Ahmad
{*bfarinho, kwetts, rmahmad*}*@eng.ucsd.edu*
Department of Computer Science and Engineering
University of California, San Diego

Abstract

Java, being an object-oriented language, provides support for polymorphism and inheritance, and thus virtual method calls. Though a boon for programmers, virtual method calls negatively impact the execution time of Java programs due to dynamic dispatch, wherein virtual method call resolution is delayed until run-time, as the type of object calling the virtual method is unknown beforehand. In this paper, we use a Java static optimization framework to implement multiple ways of resolving virtual method calls at compile-time, and compare their effectiveness.

1 Introduction

One of the main concepts in object-oriented programming is inheritance. When a derived class inherits from a base class, an object of this derived class can be referenced through a pointer of the base class's type. For example, the following is a legal way to reference a Wolf object in Java, given that Wolf inherits from Animal:

```
Animal w = new Wolf();
```

An important facet of inheritance is polymorphism. The derived classes of a base class may implement the same method differently, both from each other and from the base class[2]. Drawing on our previous example, Animal may implement the method eat() differently from its derived classes Wolf and Dog, such that each of the following calls will result in different output:

```
Animal a = new Animal();
a.eat(); // calls Animal's eat method
a = new Wolf();
a.eat(); // calls Wolf's eat method
a = new Dog();
a.eat(); // calls Dog's eat method
```

This is a particularly helpful coding construct, since it allows the programmer to reference a collection of objects of derived class types using the single base class type, and to use a universal (virtual) method call to evoke the specific behavior of each object.

However, polymorphism and virtual method calls entail significant performance issues for Java programs. Polymorphic/virtual method calls are resolved at run-time rather than compile-time, largely as a result of the fact that standard Java compilation employs no contextual or dynamic optimizations[5]. As a result, each time a polymorphic method is called in a Java program, the run-time must check the type of object calling the method as well as query and fetch its specific method code, all of which results in significant run-time overhead.

Thus, while inheritance and polymorphism are beneficial aspects of object-oriented programming for the programmer, they add overhead to the program itself. Therein lies the motivation for devirtualizing polymorphic method calls at compile-time.

There are preexisting ways to resolve procedure calls in Java at compile-time, namely inlining and static method binding. These methods will be discussed in more detail in Section 2, but it is important to understand that these methods are already employed by the standard Java compiler in situations where the resolution of a method call is unambiguous regardless of context. For example, the following method call to eat() would be resolved by any Java compiler, provided that Wolf is not extended by any other classes:

```
Wolf w = new Wolf();
w.eat();
```

While techniques for resolving polymorphic method calls exist and are used in unambiguous situations, applying them to more complicated or flow/context-sensitive code requires a significant amount of computation, namely the performing of points-to analyses, and is out of the scope of a compiler like javac. However, Java optimization frameworks do exist to this end, and even employ various points-to analysis algorithms, all of which will be discussed in Section 2.

In this paper, we test the various methods of method call resolution in combination with the various points-to

analysis algorithms. In Section 2, we discuss the details of these algorithms and methods in more detail. In Section 3, we outline our methodology. In Sections 4 and 5, we display and discuss our results.

2 Background

2.1 Optimization Frameworks

We used several preexisting frameworks to facilitate our experimentation. We describe them in detail below.

2.1.1 Soot

Soot is a highly versatile Java optimization framework[8]. Developed and actively maintained by researchers at McGill University, Soot can be used to analyze, optimize, and visualize Java programs and Android applications.

Of its many features, we were particularly interested in Soot's ability to facilitate a variety of points-to analyses. Soot also allows for the custom resolution of Java method calls through its use of user-manipulable, intermediate representations of the program during optimization[1][6]. Soot is also highly extensible, as we demonstrate and discuss in the next subsection.

2.1.2 Spark

One of Soot's major limitations in the scope of this project is that it can only perform points-to analyses and optimizations based on static class hierarchy analysis. Unfortunately, static class hierarchy analysis does not help to resolve polymorphic method calls any more than a standard, non-contextual compiler.

To solve this problem, we use Spark, an experimental points-to analysis framework that runs on top of Soot and provides access to a variety of points-to analysis algorithms, including rapid type analysis, variable type analysis, and geometric points-to analysis among others[4].

2.2 Method Call Resolution

There are multiple ways that method calls can be resolved in Java, both during compile-time and during run-time. We describe the three methods with which we experimented in the following subsections.

2.2.1 Inlining

Inlining is the process of replacing a resolved method call with the code referenced by that method call. Consider the following code:

```
public class Wolf {
    public void eat() {
        System.out.println("Wolf Eat");
    }
}
```

```
Wolf w = new Wolf();
w.eat();
```

As the call to eat() can be resolved with certainty, inlining would result in eat() being replaced by its corresponding code, as shown below:

```
Wolf w = new Wolf();
System.out.println("Wolf Eat");
```

As this example demonstrates, inlining can completely eliminate the run-time overhead of method call resolution by removing method calls entirely. However, inlining can be problematic in certain situations, and can lead to an explosion in code size if the imported functions are large or if the calls are made very frequently.

2.2.2 Static Method Binding

Static method binding describes the linking of method calls to their corresponding function code at compile-time[3]. This linking, or binding, is performed when a method call encountered during compilation can be attributed to a single type of object without ambiguity. Static method binding normally only occurs on monomorphic methods calls and polymorphic method calls wherein the object calling them is of a non-extensible class; any call which could not possibly be attributed to more than one type of object at compile-time is usually resolved using static method binding. Static method binding is not used, however, for calls which could be attributed to multiple objects, as these calls are considered ambiguous. An optimization framework like Soot, though, can disambiguate these method calls through various contextual code inspection techniques such as points-to analysis and thus allow static method binding for these ordinarily unattributable calls.

2.2.3 Dynamic Method Binding

Dynamic method binding is the term describing the standard run-time method call resolution technique inherent to Java[3]. When a virtual method call is encountered at run-time, the type of its object is determined based on the run-time context, and the call is bound (linked) to the resolved method's location, soon to be fetched and executed.

2.3 Points-To Analysis

In order to determine how specifically method calls may be resolved, points-to analyses are often required. We describe the various points-to analysis algorithms with which we experimented below.

2.3.1 Class Hierarchy Analysis

Class hierarchy analysis (CHA) is the most conservative form of analysis we tested. It utilizes a call graph constructed from the declared types of each object in the Java program to prune the collection of total possible method calls[10][9]. CHA, however, leaves much to be desired as it is still incredibly conservative and lacks the ability to resolve method calls made by parent pointers. To illustrate this, in the following example, the first eat() call would be resolved by CHA, while the latter would not be:

```
Wolf w = new Wolf();
Animal a = new Wolf();
w.eat(); // resolved
a.eat(); // NOT resolved
```

2.3.2 Rapid Type Analysis

Rapid type analysis (RTA) is slightly less conservative than class hierarchy analysis, and is actually considered an extension of CHA. It constructs a call graph identical to that constructed using CHA, but then removes any edges from this graph that refer to object types that are not instantiated in the program[10][9]. In our previous example, this would result in both eat() calls being resolved, as the call graph could only contain edges pertaining to the Wolf class:

```
Wolf w = new Wolf();
Animal a = new Wolf();
w.eat(); // resolved to Wolf eat()
a.eat(); // resolved to Wolf eat()
```

An example where RTA fails is trivial to construct. In this example, a new class, Dog, which extends Animal is used:

```
Wolf w = new Wolf();
Animal a = new Dog();
a.eat(); // could refer to Wolf or Dog eat()
```

2.3.3 Variable Type Analysis

Variable-type analysis is a much more fine-grained analysis method than both class hierarchy analysis and rapid type analysis. For each variable in the program, it attempts to construct a set of all possible types that could reach this variable[10][9]. The resulting graph is called a type propagation graph, as opposed to a simple call graph. This graph is constructed using all object initializations in the program as entry points, and propagating the set of all reachable types for each variable via union through the program from the top down.

In our previous example, VTA would recognize the call to eat() as specific to Dog, as the set of reachable types for variable a is {Dog}.

```
Wolf w = new Wolf();
Animal a = new Dog();
a.eat(); // resolved
```

The following, however, would not be resolved, as VTA is still context insensitive with regards to run-time variable-type bindings:

```
Animal a = new Dog();
a = new Wolf();
a.eat(); // could refer to Wolf or Dog
```

Though obvious to the reader, VTA considers the reaching set of variable a to be {Dog, Wolf} and does not distinguish between the two due to its context insensitivity. This is still something that VTA would leave to run-time resolution.

2.3.4 Geometric Points-To Analysis

Geometric points-to analysis is particularly interesting, as it is a context-sensitive points-to analysis algorithm[11]. It uses a complex encoding scheme to generate a geometric graph of object instantiations and type castings, and performs a variable (set by the user) number of iterations through this exponentially-sized graph, eliminating unreachable edges in each round. GeomPTA, being context sensitive, offers significant improvement over the aforementioned analysis algorithms with regards to virtual method call resolution, but it is also more costly in terms of compilation time relative to the other algorithms.

3 Methodology

As this project was focused on dynamic dispatch optimization specifically in Java, the aforementioned Soot framework and Spark package were used extensively in this project. While both Soot and Spark did not necessarily have a large contributing community and had a rather steep learning curve, both were very effective at accomplishing their tasks.

3.1 Setup

While Soot is distributed as a JAR and can be referenced like any other JAR from the command line, its recommended use is as a plugin to Eclipse. Given this recommendation and the amount of materials that provide support for Soot as an Eclipse plugin, we chose to run Soot through Eclipse. This aided in reducing Soot's learning curve, and further allowed us to focus on the various optimization methods and their effects.

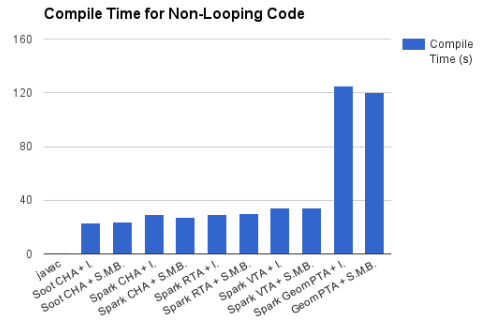

Figure 1: Compilation Time in Non-Looping Code

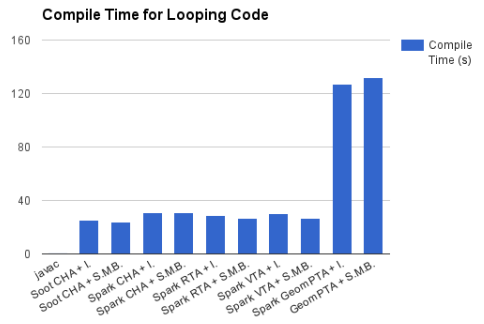

Figure 2: Compilation Time in Looping Code

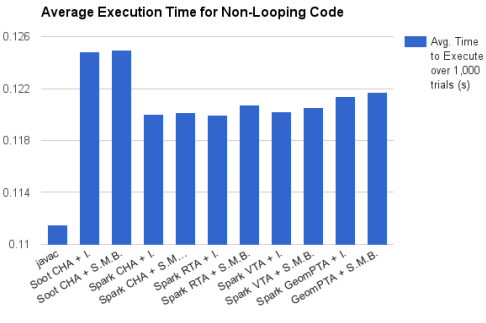

Figure 3: Average Execution Time in Non-Looping Code

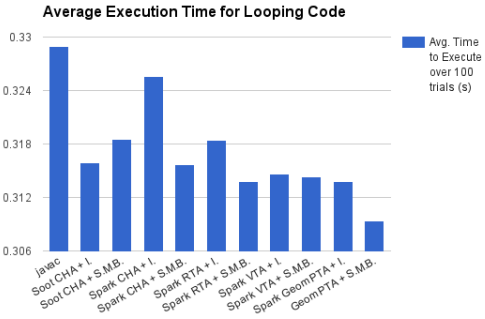

Figure 4: Average Execution Time in Looping Code

3.1.1 System Specifications

The experiments were run on a Macbook Pro running OS X Yosemite, with a 2.4 GHz Intel Core i7 processor and 16GB 1600 MHz DDR3 memory.

3.2 Experiments

We constructed a Java program with a four-level class hierarchy for testing purposes, excerpts of which are shown in the Sample Code addendum. The code has a high-iteration loop and makes virtual method calls in the loop body. We then compiled this program using the following analysis techniques: javac (standard compilation), Soot CHA, Spark CHA, Spark RTA, Spark VTA, and Spark GeomPTA. For each analysis technique except javac, we compiled the program using both static method binding and inlining to test the difference between the two techniques in each context.

4 Results

The data presented in Figures 1 and 2 is the time, in seconds, the code took to compile. The two figures serve to show that all compilation techniques have a relatively similar time cost, except for the expensive GeomPTA analysis, as expected. Note that the standard compilation with javac is essentially free compared to any of the other techniques.

Figure 3 shows the average execution time of the non-looping test code over 1,000 trials. Two artifacts are prominent in this graph. First, the various points-to analyses show no distinct benefits over one another, save that they all outperform the standard Soot CHA analysis without Spark. Second, the javac-compiled code dramatically outperforms all of the compilation techniques we employed. We discuss this in the Analysis Section.

Figure 4 shows the average execution time of the looping code over 100 trials. Note that, as opposed to the previous graph, the javac-compiled code under-performs all other compilation techniques, and the points-to analyses offer fairly distinct improvements. We discuss this further in the Analysis Section.

5 Analysis

The compilation times we recorded were as expected. The javac standard compiler was essentially cost-free, while the GeomPTA analysis was the most expensive by far. CHA costs slightly less than RTA, and RTA costs slightly less than VTA, all of which make sense given the VTA is the most fine-grained of the set, while RTA is

a slightly more complicated CHA procedure.

The average execution times we recorded were much more interesting. Firstly, we note that javac-compiled code outperforms all other analyses when a 10,000 iteration loop is not included in the executed program. While this was initially offputting, we attribute it to the fact that, without the loop, the code is so brief that the overhead added by the advanced compilers outweighs any method call time reductions. The file size of each of the Java programs compiled with the advanced analysis techniques always increased beyond that of the javac-compiled program, supporting this claim.

The ultimate set of tests which we ran on a program with a 10,000 iteration loop included offers more telling results. The javac-compiled code is outperformed by all subsequent techniques. CHA is outperformed by RTA and VTA (themselves offering similar improvements), while GeomPTA offers the most improvement. These were our expected results. Static method binding outperformed inlining in almost all cases, which makes sense given a relatively small class hierarchy and looping being involved; these conditions are more favorable for static method binding over inlining.

These results were heavily influenced by the small size of the program, the program structure consisting mainly of a large loop with no threading, and the uncomplicated class hierarchy. As we will discuss in the Future Work Section, testing on a larger code base would yield more useful results, but Soot imposed an unexpected amount of limitations on us as an Eclipse plugin.

6 Future Work

As was seen in the results from GeomPTA, context sensitive analyses seem to show the most significant runtime improvements. As such, one of the main avenues for future work we would like to investigate is the use of another Soot package, Paddle, which is a prototypical points-to analysis engine that provides a collection of exclusively context-sensitive points-to analysis algorithms[7]. Using these algorithms, we could further see the effects of different context-sensitive analyses.

Expanding our sample code base so as to better test the benefits of virtual method call resolution would be another interesting area for future experimentation. Our current code base was rather small and simplistic due to the limitations of Soot and Spark. Being able to perform analyses on larger code bases, and potentially even real programs would provide very interesting insight into the efficacy of these algorithms and whether the run-time efficiency benefits outweigh the increased compilation times.

7 Conclusion

Our experimentation with Java method call resolution has resulted in several findings. We observe that the benefits of compile-time method call resolution are only visible in larger programs, and can actually slow down smaller programs. We also find that Geometric Points-To Analysis is costly in terms of compile-time, even on small programs, though its benefits could be potentially significant in a situation where compile time is not an issue. Based on our results, RTA and VTA could be worth employing over basic CHA, given their similar compile times. Finally, we need to experiment on a larger code base to trust our empirical results before passing further judgement.

References

[1] EINARSSON, A., AND NIELSEN, J. D. A survivors guide to java program analysis with soot. *Web page: http://www.brics.dk/SootGuide/sootsurvivorsguide.pdf* (2008).

[2] JAVATPOINT. Polymorphism in Java. `http://www.javatpoint.com/runtime-polymorphism-in-java`.

[3] JAVATPOINT. Static and Dynamic Binding. `http://www.javatpoint.com/static-binding-and-dynamic-binding`.

[4] LHOTAK, O., AND HENDREN, L. Scaling java points-to analysis using spark. *Web page: https://plg.uwaterloo.ca/ olhotak/pubs/sable-tr-2002-9.pdf* (2002).

[5] ORACLE. javac - Java programming language compiler. `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html`, 1993.

[6] PATRICK LAM, ERIC BODDEN, O. L., AND HENDREN, L. The soot framework for java program analysis: a retrospective. *Web page: http://sable.github.io/soot/resources/lblh11soot.pdf* (2011).

[7] SABLE. Paddle: BDD-based context-sensitive interprocedural analysis of Java. `http://www.sable.mcgill.ca/paddle/#doc`, 1999.

[8] SABLE. Soot - A framework for analyzing and transforming Java and Android Applications. `http://sable.github.io/soot/`, 1999.

[9] SNAJALG, S. Program analysis techniques for method call devirtualization in object-oriented languages. *Web page: http://kodu.ut.ee/ varmo/seminar/sem09S/final/s6najalg.pdf* (2009).

[10] VIJAY SUNDARESAN, LAURIE HENDREN, C. R. R. V.-R. P. L. E. G., AND GODIN, C. Practical virtual method call resolution for java. *Web page: http://www.cs.ucla.edu/ palsberg/tba/papers/sundaresan-et-al-oopsla00.pdf* (2000).

[11] XIAO, X., AND ZHANG, C. Scaling Context Sensitive Points-to Analysis by Geometric Encoding. `http://www.cse.ust.hk/~richardxx/papers/geom-tr.pdf`, 2014.

Sample Code

```java
// Driver.java
package level1;

import level3.Canine;
import level4.Cat;
import level4.Dog;
import level4.Wolf;

public class Driver {
public static void main(String[] args) {
        Dog d = new Dog();
        Cat c = new Cat();
        Wolf w = new Wolf();
        Canine c1 = new Canine();
        Animal a = new Wolf();

        for (int i = 0; i < 10000; i++) {
        System.out.println(a.eat());
        a = c1;
        System.out.println(a.eat());
        a = w;
        System.out.println(a.eat());
        a = c;
        System.out.println(a.eat());
        a = d;
        System.out.println(a.eat());
}
    }
}
```

```java
// Animal.java
package level1;

public class Animal {
    public Animal() {}

public String eat() {
return "Animal Eat";
}

}
```

```java
// Mammal.java
package level2;

import level1.Animal;

public class Mammal extends Animal {
public Mammal() {}

public String eat() {
return "Mammal Eat";
}
}
```

```java
// Canine.java
package level3;

import level2.Mammal;

public class Canine extends Mammal {
    public Canine() {}

    public String speak() {
        return "Canine Speak";
    }

    public String eat() {
        return "Canine Eat";
    }
}
```

```java
// Wolf.java
package level4;

import level3.Canine;

public class Wolf extends Canine {
public Wolf() {}

    public String speak() {
        return "Wolf Speak";
    }

    public String eat() {
        return "Wolf Eat";
    }
}
```