Program Analysis Techniques for Method Call Devirtualization in Object-Oriented Languages

Sander Sõnajalg

December 7, 2009

Abstract

In an object-oriented language, a call to an instance method generally has to be implemented as virtual, as the compiler doesn't know which implementation of the method should the call be forwarded to. Virtual invocation means a lookup routine is performed at runtime each time that call site is reached, degrading the performance. At many call sites, virtual method invocations can be replaced with more optimal static calls after applying some program analysis methods to discover that a call site actually always calls the same implementation of the method. Class hierarchy analysis and rapid type analysis are the simplest and most classical devirtualization methods, while later more sophisticated analyses like variable-type analysis have also been proposed.

1 Introduction

With the emergence of the object-oriented programming paradigm, a lot of new space was created for applying additional compiler-level optimizations. Class inheritance is the cornerstone of object-oriented abstraction. Large and complicated inheritance hierarchies promote good application design and code reuse, but degrade performance. The most obvious problem is the virtual method invocation with its runtime lookup overhead.

The exact type of a variable can be any subclass of the declared type. Methods can be overridden at any point in the inheritance hierarchy. Various types can show up at one call site, and they can each define their own implementation of the method at hand by overriding the inherited versions. This ends in the compiler not knowing which implementation of the method should the call site be bound to, and the invocation has to be implemented as a virtual method call. Virtual call sites delay the choosing of the correct method implementation until runtime, performing a lookup routine each time this call site is reached. Compared to static method invocations, this lookup routine is an obvious extra overhead. Many of these virtual call sites can actually be resolved and replaced with faster static method invocations by providing the compiler

with extra meta-information about the call sites recovered from analysing the source code.

This writing gives an overview of the classical program analysis techniques for method devirtualization like class hierarchy analysis and rapid type analysis. One of the more sophisticated methods, variable-type analysis, is also presented. The rest of this paper is structured as follows. Section 2 covers some of the necessary background to understand the topic (virtual method calls, devirtualization, call graphs, etc.). Sections 3, 4 and 5 cover the techniques of class hierarchy analysis, rapid type analysis and variable-type analysis, respectively. Section 6 discusses some aspects of the applicability of the reviewed methods for real-life environments, focusing on the Java language. Section 7 references some evaluation results, leaving section 8 for some final remarks and conclusions.

2 Background

2.1 Virtual Method Calls and Devirtualization

In this writing, we'll be having **Java** as an example language of the interpreted (*just-in-time*-compiled) virtual machine languages, and **C++** of the compiled languages. The examples are Java-based, as it is widely known and used.

To better explain the concept of virtual method calls, let's view the code from listing 2.1. When a program gets compiled, the compiler tries to bind as many call sites directly to the called procedures as possible (i.e. implement the invocations as *static procedure calls*). Call sites 1 and 3 (a call to a static method and a call to a private instance method, correspondingly) can be statically bound to corresponding methods of the class `ExampleClass` without any hesitation, as neither static methods nor private instance methods can be overridden in subclasses, in the straight sense. The location of code that is to be executed is always statically known.

Things are a bit more complicated when invoking non-private instance methods (call sites 2 and 4). Even though the *declared type* of the object is statically known, the actual *runtime type* can be the declared type itself or any of its subclasses (when the declared type is an interface, then any implementing class of that interface or its subinterfaces, or any of their subclasses). The runtime type class can override the non-private instance method, meaning that the actual procedure that needs to be called can vary from one runtime type to another. As it's not statically known which procedure needs to be called, the compiler can't bind it to any certain implementation but rather has to insert a lookup routine to find the correct target at runtime. These dynamically bound calls are called **virtual method calls** (sometimes also referred to as *dynamic message sends*).

This is exemplified by the behaviour of the example code, where visually the same call `ec.normalInstanceMethod()` will result in invoking the method `ExampleClass#normalInstanceMethod()` at call site 2 and `SubClass#normalInstanceMethod()` at call site 4.

```
public class ExampleClass {
  public ExampleClass() {}
  public static void main(String[] args) {
    ExampleClass.staticMethod();       // call site 1

    ExampleClass ec = new ExampleClass();
    ec.normalInstanceMethod();         // call site 2
    ec.privateInstanceMethod();        // call site 3

    ec = new SubClass();
    ec.normalInstanceMethod();         // call site 4
  }
  public static void staticMethod() {
    System.out.println( - static method called);
  }

  public void normalInstanceMethod() {
    System.out.println( - normal instance method called);
  }

  private void privateInstanceMethod() {
    System.out.println( - private instance method called);
  }
}

public class SubClass extends ExampleClass {
  public SubClass() { super(); }
  public void normalInstanceMethod() {
    System.out.println( - OVERRIDDEN instance method called);
  }
}
```

Listing 2.1: A snippet of Java code that demonstrates various types of method calls in Java.

One could claim that for our example, it isn't exactly true that it's statically unknown which exact method implementations should be called at each of those call sites. If we take a more intelligent, open-minded look at the code, we'll easily notice that in our particular case, both of these call sites obviously always call the same target procedures, and could therefore still be statically bound. In fact, this is exactly what the program analysis methods for devirtualization aim for. **Devirtualization** is the replacement of virtual method calls with static procedure calls at call sites always pointing to the same procedure. Devirtualization can be applied if the compiler can *prove* that no multiple target procedures are possible at a particular call site, using various program analysis methods like class hierarchy analysis (section 3) and rapid type analysis (section 4). The main goal is to find the set of all possible runtime types at a call site as accurately as possible. If this set contains only one type, or all the contained types inherit the same implementation of the called method, the call site can be devirtualized by binding it to this implementation. This set always

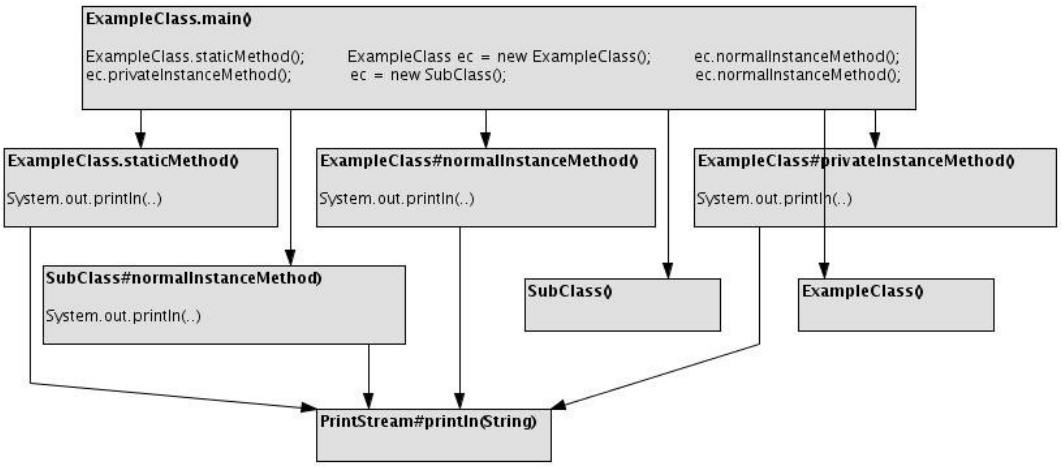

Figure 1: A call graph for code on listing 2.1

has to be found *conservatively*: if you *can't prove* that a certain type will never reach the given call site, you *can't leave it out*, as it might lead to erroneous devirtualization.

Devirtualization plays an important role in optimizing compilers mainly due to two reasons. Firstly, replacing virtual calls with static ones avoids performing the costly method lookup routine on each method invocation. This routine involves examining the inheritance hierarchy of loaded classes and finding the correct implementation by applying the inheritance rules. Secondly, statically binding a call site to the invoked method enables the use of the **method inlining optimization**. Rather than invoking the method, the body of that method is copied straight into the calling method (*inlined*), thus eliminating the runtime overhead of passing arguments and return values [2].

2.2 Call Graphs

As call graphs are the key concept of the classical devirtualization techniques like class hierarchy analysis and rapid type analysis, a brief description is hereby given.

Call graph is a directed graph which includes one node for each reachable method of the represented program. An edge is drawn from method A to method B, if a call site in method A calls or possibly calls method B. In the context of devirtualization, call graphs have to be constructed *conservatively*: if the unreachability of a method is not proven, the corresponding node has to be included, and if the absence of calls from call site to a method is not sure, an edge has to be included. From that follows that the more accurate call graph is the one containing *less* nodes and edges. It is also obvious that the less edges the call graph contains, the more call sites can potentially be devirtualized.

Figure 1 shows a call graph constructed for the code on listing 2.1.

3 Class Hierarchy Analysis

Class hierarchy analysis (CHA) is a static analysis[1] that constructs the class inheritance graph and combines this information with the *declared* types of target objects at each call site [3]. By doing so, a set of all possible target types is estimated and a conservative call graph of the program can be constructed. If this analysis yields only one possible target type for some call site, this call site can be *resolved*, replacing the dynamic method invocation with a more optimal static procedure call of the only possible candidate. To illustrate this, let's investigate the class hierarchy on figure 2 and the corresponding client code on listing 3.1.[2]

It is obvious that without performing any program analysis at all, the compiler would have to implement all calls to instance methods on any target object as dynamic method invocations (dynamic message sends), as any object's real type can always be a subclass of the declared type, and the called method could have been overridden in the subclass. So a reference point can be taken that the method invocations to `hasRightToVote()` are dynamic at both call sites.

For the first call site, the declared type is `EUCitizen`, and when CHA matches it with the inheritance hierarchy, it finds that its actual runtime type is one of `{EUCitizen, Estonian, Latvian, MaleEstonian, FemaleEstonian}`. As this set includes subclasses overriding the called method `hasRightToVote()` (namely `Estonian` and `Latvian`), CHA is unable to determine which implementation of the method should be invoked and the call site remains unsolved.

The declared type of the receiver is `Estonian` at the second call site. CHA finds the set of possible receiver types to be `{Estonian, MaleEstonian, FemaleEstonian}`. As none of the subclasses override `hasRightToVote()`, the dynamic method invocation can be replaced with a static procedure call to `Estonian#hasRightToVote()`. Note that if the class hierarchy had been as shown on figure 3, it wouldn't have been possible to know whether to bind the call to `Estonian#hasRightToVote()` or `PersonWithRestrictedRights #hasRightToVote()`.

Notice that in the example program, the *implementations* of methods `getCitizen()` and `getEstonian()` are such that they obviously *can't* return instances of types other than `Estonian`. If this would be taken into account for, both call sites could always be resolved, no matter how the class inheritance hierarchy would look like. This detail is something that remains completely uncaptured by the CHA, so there is definitely room for improvement (this example will be revised in the rapid type analysis section).

[1] It is traditionally a static analysis. Actually, the implementation of CHA on a dynamic class loading enabled environment can't be static any longer (see the next footnote and section 6.2.2).

[2] Please notice that sections 3, 4 and 5 use examples written in the Java language but without paying any attention to the effect of JVM's dynamic class loading capability. This effect is separately discussed in section 6.2.2. Dynamic class loading severely complicates the implementation of these optimizations, but once they are properly implemented, they'd still behave in the same way, so the examples remain valid.

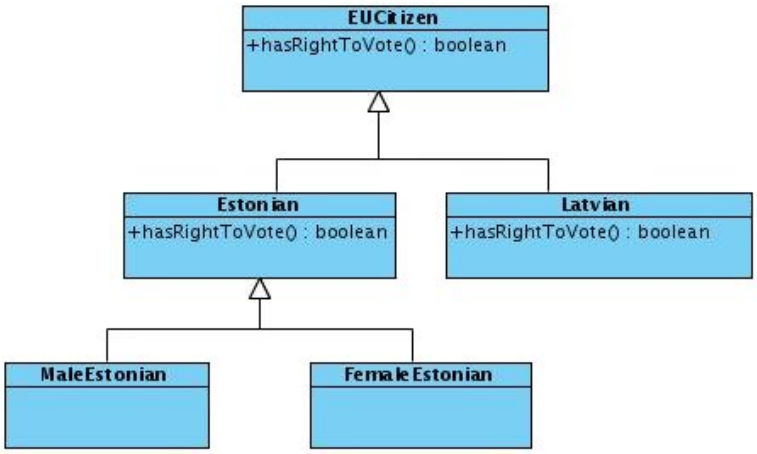

Figure 2: The class inheritance hierarchy used by the example program on listing 3.1

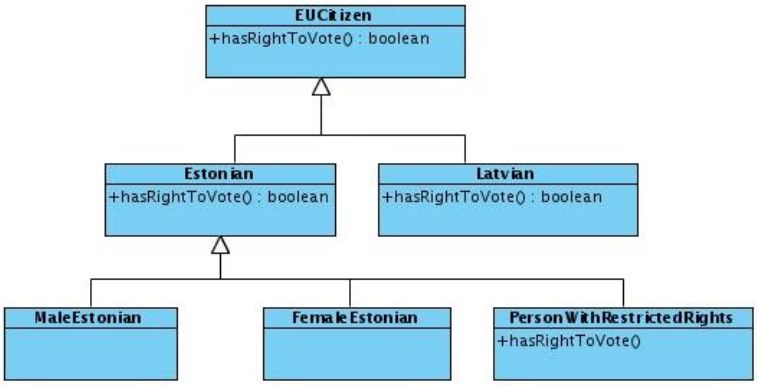

Figure 3: The modification of the class inheritance hierarchy of figure 2.

```
public class MyProgram {
  public static void main(String[] args) {
    EUCitizen citizen = getCitizen();
    citizen.hasRightToVote();          // Call site 1

    Estonian estonian = getEstonian();
    estonian.hasRightToVote();         // Call site 2
  }

  private static EUCitizen getCitizen() {
    return new Estonian();
  }

  private static Estonian getEstonian() {
    return new Estonian();
  }
}
```

Listing 3.1: Methods `getCitizen()` and `getEstonian()` are returning instances with declared types `EUCitizen` and `Estonian` correspondingly. The exact runtime types returned are unknown by looking at just the method signatures, but obvious when investigating also the method bodies.

4 Rapid Type Analysis

Rapid type analysis (RTA) is a simple extension to CHA that further reduces the set of possible target types by investigating which types are instantiated overall in the given program ([1] , [9]). The set of instantiated types is built optimistically by traversing the call graph built by the CHA. For each call site, the global set of instantiated types is intersected with the local set of possible call targets found by CHA, hopefully further reducing the number of candidate target types.

For the example program on listing 3.1, RTA finds the global set of instantiated types to be `{Estonian}` (assuming that the constructor `Estonian()` doesn't do anything weird). Intersecting it with the results of CHA at call site 2 for example yields:

$$\{Estonian\} \cap \{Estonian, MaleEstonian, FemaleEstonian\} = \{Estonian\}$$

The result will be the same for call site 2 after the modification in the inheritance hierarchy shown on figure 3 or for call site 1. In all of the cases, applying RTA will resolve those call sites.

RTA is a simple extension to the CHA that comes at a low cost, but also fails quite easily. Notice that if we replace the implementation of `getCitizen()` as shown on listing 4.1, the global list of instantiated types will become $\{Estonian, Latvian\}$, making the result of the intersection for call site 1 also to be $\{Estonian, Latvian\}$. Resolving call site 1 would now fail, though it is still quite obvious that actually the only possible target type is still `Estonian`.

More sophisticated (and more expensive) flow-sensitive analyses could be applied to reveal this additional piece of information.

```
private static EUCitizen getCitizen() {
  Citizen c = new Latvian();
  return new Estonian();
}
```

Listing 4.1: A method body that would confuse RTA, but actually always returns an instance of the same type.

It has to be noticed that the object instantiation has to be analysed more thoroughly than just looking for calls to constructors in the source code. Consider the reimplementation of method `getEstonian()` shown on listing 4.2. Combined with the dynamic class loading capability of the JVM platform, it now really *is* impossible to estimate which types are instantiated during the runtime. There are still ways to work around this though, for example, the JIT compiler of the virtual machine could memorize all the types of objects which it has *actually* instantiated. This would also remove the need to traverse the call graph of the program for finding the set of instantiated types.

```
private static Estonian getEstonian() {
  try {
    BufferedReader reader =
      new BufferedReader(new FileReader(new File("input.txt")));
    String newClassName = reader.readLine();
    Class c = Class.forName(newClassName);
    return (Estonian) c.getConstructor().newInstance();

  } catch (Exception e) {
    return new Estonian();
  }
}
```

Listing 4.2: A method implementation with truly no way to statically know what the exact return type will be.

5 Variable-Type Analysis

Both CHA and RTA are very coarse-grained and simple static analyses. They provide a valuable gain at a very low cost, but sometimes, more accurate (yet more expensive) analyses would be preferable.

CHA only takes account of the inheritance hierarchy information to find the set of all implementations of the called method. RTA further reduces this set by only keeping the types that are actually instantiated somewhere in the analysed

program. This is a definite improvement over CHA, but it is obviously also a very coarse-grained method. We'll briefly present **variable-type analysis**, a more fine-grained way to further reduce the set of possible runtime types at a call site, proposed by [9]. The same paper also proposes a cheaper but less-accurate alternative called **declare-type analysis** which is not going to be reviewed here.

Variable-type analysis (VTA) is a flow-insensitive inter-procedural full-program analysis. For each variable in the program, including local variables in method bodies, VTA aims to find the set of types that could possibly reach this variable. For that, a **type propagation graph** is constructed. Each node of the graph represents a variable, the encoding being roughly as follows:

- Node `C.a` denotes the instance variable `a` of class `C`.
- Node `C.m.p` denotes the parameter or the local variable `p` of method `m` of class `C`.
- Node `C.m.this` denotes the reference to the current instance inside instance method `m` of class `C`.
- Node `C.m.return` denotes the returned type of method `m` of class `C`.

Each node is associated with a `ReachingTypes` set that contains all the types that could reach the represented variable. The directed edges of the graph represent all kinds of assignments, the rules being roughly:

- For the assignment `x = y`, and edge is added from the node representing the variable `y` to the node representing the variable `x`.
- For any method invocation statement `o.m(x, y, ..)` to the method `C.m(arg1, arg2, ..)`, an edge is added from the node representing variable `x` to the node `C.m.arg1`, etc. Also, an edge is added from the node representing `o` to the node `C.m.this`.
- For the assignment `x = o.m(x, ..)`, an edge is added from the node `o.m.return` to the node representing the variable `x`.
- For the return statement `return x` in a method C.m, and edge is added from the node representing variable x to the node `C.m.return`.

As an example, the construction of a type propagation graph is shown for the code on listing 5.1. Class hierarchy from a previous example on figure 2 is reused as a model. The call graph construction starts off by identifying all the nodes and then all the edges between them. In the beginning, the `ReachingTypes` sets of all the nodes are empty. Then the type propagation graph is *initialized* by finding all object instantiation statements (e.g. `new MyClass()`) from the code. For each statement, the instantiated type is added to the `ReachingTypes` set of the node representing the variable that the new instance gets assigned to — see figure 4. Finally, the graph is processed, starting from sources (nodes with

```
public class C {
  private static EUCitizen a, b;

  public static void main(String[] args) {
    a = m1();
    b = m2();
    EUCitizen c = new FemaleEstonian();
    m3(c);
    a.hasRightToVote();          // call site 1
    b.hasRightToVote();          // call site 2
  }

  private static EUCitizen m1() {
    return new Latvian();
  }

  private static EUCitizen m2() {
    b = new Estonian();
    Estonian g = new MaleEstonian();
    return g;
  }

  private static void m3(EUCitizen p1) {
    b = p1;
    EUCitizen f = b;
    a = f;
  }
}
```

Listing 5.1: An example program for which the construction of the type propagation graph will be shown.

no incoming edges), and always continuing with nodes that have all their predecessors already processed. *Processing* of a node involves copying all the types from its `ReachingTypes` set to the `ReachingTypes` sets of all the nodes that have an incoming edge from the original node. The final type propagation graph of our example is shown on figure 5.

Using the `ReachingTypes` sets of the corresponding variables provides a much more accurate estimate for the sets of target types at call sites than CHA or RTA did. For example, the type propagation graph yields that call site 1 can't be devirtualized, as instance variable `a` is reachable by types `Estonian` and `Latvian` that both define the method `hasRightToVote()`. Call site 2, on the other hand, *can* be devirtualized, as although it is reachable by three types `Estonian`, `FemaleEstonian` and `MaleEstonian`, they all correspond to the same implementation of the method `hasRightToVote()`, namely `Estonian#hasRightToVote()`.

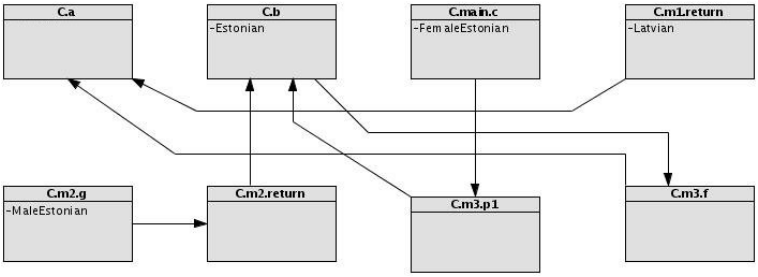

Figure 4: Type propagation graph *after* initialization and *before* type propagation.

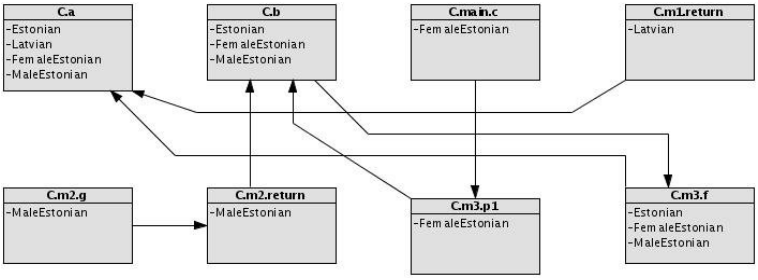

Figure 5: Type propagation graph after the types have been propagated.

6 Applicability of the Presented Techniques

Applying optimizations inside a compiler is always a matter of trading compilation speed against code quality [2]. If the compilation happens wholly on the software vendor's side, as is the case for C++ applications, the compilation time is less of an issue. If the compilation happens on the end-user's machine like for Java programs, and especially during the runtime, this trade-off is crucial. Pauses in the program's normal workflow due to the JIT-compiler performing some heavy optimizations have to be avoided. Therefore, in one environment, more sophisticated optimization techniques could be applied, while in another, more lightweight optimizations would be more appropriate.

6.1 Compiled Languages *à la* C++

For a compiled and linked language like C++, the implementation of techniques like CHA and RTA would be quite straight-forward, as the whole program is available for the compiler to analyse during the compilation. Also for C++-style languages, as native code generation doesn't happen during the runtime and won't thus affect the end-user, it is much more feasible to apply various expensive optimizations during compilation.

6.2 Interpreted Virtual Machine Languages *à la* Java

If we think about the presented analyses in the context of Java, we'll notice some important restrictions. Firstly, as JIT compilation happens during the runtime, it is not possible to run too time-consuming analyses, as the usability of an interactive program greatly suffers when program's response times grow (i.e. the user has to sit and wait while the JIT performs its sophisticated analyses). At least, the cost of the expensive analyses has to be distributed over a longer period of program execution. Secondly, the dynamic class loading feature of the JVM prevents the whole-program analysis [4]. Let's briefly discuss these two points.

6.2.1 Optimizing a JIT-Compiled Language

Languages running on high-level language virtual machines like the Java Virtual Machine (JVM)[5] have a more complicated compilation model. Further discussion of this family of languages uses Java and the JVM platform as an example. For other similar platforms like the .NET, and of course for other languages running on the JVM (Groovy, Scala, JRuby, Jython), the same still applies.

When Java source code (or the source code of any other JVM language) is compiled, the compiler produces JVM bytecode. JVM bytecode is a binary, platform-independent intermediate representation that can be executed by the JVM. By its language semantics, the JVM bytecode greatly resembles Java source code: it still has classes, objects, methods and other high-level concepts. This first compilation phase is thus very straight-forward and applies no interesting optimization techniques.

The more interesting things happen during the execution of the bytecode, when the JVM just-in-time (JIT) compiler translates the bytecode into native machine code. As the name suggests, the compilation of a fragment of code happens on demand, immediately before it actually needs to be executed for the first time. The compiler can perform the analyses and optimizations at the moment of first compilation, or come back to this code to optimize and recompile it later. It is a common technique to first compile all the demanded code with a fast compiler, then gather profiling information when the program is running, identify the most critical (most intensively used) methods, and later re-compile those methods with a slower optimizing compiler [2].

6.2.2 Optimizing a Language with Dynamic Class Loading

For an interpreted language like Java, analyses like CHA or RTA can not be performed in a straight-forward manner, mainly because the dynamic class loading feature of the JVM platform can bring changes to the program's class hierarchy during program execution. Let's once again return to the example from section 3. For example, assume that the class `PersonWithRestrictedRights` was dynamically loaded *after* the compiler has performed the class hierarchy analysis, built the call graph and decided that call site 2 can be statically bound to

`Estonian#hasRightToVote()`. If `getEstonian()` should now return an instance of `PersonWithRestrictedRights`, the applied optimization would lead to an erroneous behaviour (this could of course never happen with the method body shown on the original listing, but might very well occur when the method body is replaced with the one on listing 4.2).

As this example shows, the implementation of CHA has to be much more sophisticated for environments with dynamic class loading capabilities like the JVM platform. It would be necessary to invalidate and rebuild parts of the call graph as changes to class hierarchy get loaded. Also, the optimized native code with devirtualized method calls and inlined methods would have to be replaced on-the-fly (which is something that the JIT compilers are good at, though) with the original virtual method invocations. For example, Microprocessor Research Lab Virtual Machine uses a technique called *dynamic inline patching* to revert the inlined methods that have turned invalid [2]. The compiler first makes assumptions that dynamic class loading will *not* happen, and inlines some of the virtual method calls that have only one implementation. If later on a new class is loaded dynamically that turns some of those method inlinings invalid, the call sites with inlined methods are patched with an additional *jump* instruction to fall back to virtual method invocation.

7 Evaluation

The usefulness of methods as naïve as the class hierarchy analysis or rapid type analysis can be doubted. In fact, as some benchmarking results show, they are surprisingly effective for their very low cost. In [9], CHA, RTA and VTA are compared, using 7 benchmark programs including the *javac* compiler. One measurement showed how big of a percentage of call sites that were always bound to one method during the actual program execution, had been resolved by those methods. CHA and RTA demonstrated almost equal results, devirtualizing 50-100% of the monomorphic call sites (i.e. call sites that were theoretically possible to devirtualize). VTA resolved 60-100% of monomorphic call sites and was the most effective for all benchmarks, but the difference from CHA and RTA wasn't actually that big ranging from 0-15%.

8 Conclusions

Devirtualization is one of the most important optimizations applied in the compilers of modern object-oriented programming languages. Traditional compiled languages like C++ can apply it more directly, while virtual machine platforms like the Java Virtual Machine have to implement it in a more sophisticated way, benefitting from the possibilities of the just-in-time compilation model.

It has to be emphasized that the three program analysis techniques reviewed in this writing are not the only ones around (though the first two — class hierarchy analysis and rapid type analysis — are the most classical ones). *Declared-*

type analysis is a more coarse-grained variant of the presented variable-type analysis ([9]). *Points-to analysis* is a more sophisticated static analysis technique that, as one of its applications, can also be useful in resolving virtual call sites ([7], [6]).

As opposed to languages like C++, languages with the dynamic class loading capability like Java are more complicated to analyse, because the *closed-world assumption* no longer holds and new classes may be loaded at any moment, changing the existing inheritance hierarchy. The traditional whole-program analysis methods have to be modified to correspond to these settings. In [8], an interesting general framework called *fragment class analysis* is presented that enables applying whole-program analyses to analysing only partial fragments of the program.

References

[1] D.F. Bacon and P.F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 324–341. ACM New York, NY, USA, 1996.

[2] M. Cierniak, G.Y. Lueh, and J.M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. *ACM SIGPLAN Notices*, 35(5):13–26, 2000.

[3] J. Dean, D. Grove, and C. Chambers. Optimization of Object-oriented Programs Using Static Class Hierarchy Analysis. *Lecture Notes in Computer Science*, 952(77-101):72, 1995.

[4] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. *ACM SIGPLAN Notices*, 35(10):294–310, 2000.

[5] T. Lindholm and F. Yellin. *Java Virtual Machine Specification, 2nd Edition*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[6] A. Milanova, A. Rountev, and B.G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[7] A. Rountev, A. Milanova, and B.G. Ryder. Points-To Analysis for Java Using Annotated Constraints. *ACM SIGPLAN Notices*, 36(11):55, 2001.

[8] A. Rountev, A. Milanova, and B.G. Ryder. Fragment Class Analysis for Testing of Polymorphism in Java Software. *IEEE Transactions on Software Engineering*, pages 372–387, 2004.

[9] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical Virtual Method Call Resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.