Averroes: Whole-Program Analysis without the Whole Program

Karim Ali and Ondřej Lhoták

David R. Cheriton School of Computer Science
University of Waterloo, Canada
{karim,olhotak}@uwaterloo.ca

Abstract. Call graph construction for object-oriented programs is often difficult and expensive. Most sound and precise algorithms analyze the whole program including all library dependencies. The *separate compilation assumption* makes it possible to generate sound and reasonably precise call graphs without analyzing libraries. We investigate whether the separate compilation assumption can be encoded universally in Java bytecode, such that all existing whole-program analysis frameworks can easily take advantage of it. We present and evaluate Averroes, a tool that generates a placeholder library that overapproximates the possible behaviour of an original library. The placeholder library can be constructed quickly without analyzing the whole program, and is typically in the order of 80 kB of class files (comparatively, the Java standard library is 25 MB). Any existing whole-program call graph construction framework can use the placeholder library as a replacement for the actual libraries to efficiently construct a sound and precise application call graph. Averroes improves the analysis time of whole-program call graph construction by a factor of 4.3x to 12x, and reduces memory requirements by a factor of 8.4x to 13x. In addition, Averroes makes it easier for whole-program frameworks to handle reflection soundly in two ways: it is based on a conservative assumption about all behaviour within the library, including reflection, and it provides analyses and tools to model reflection in the application. The call graphs built with Averroes and existing whole-program frameworks are as precise and sound as those built with Cgc. While Cgc is a specific implementation of the separate compilation assumption in the Doop framework, Averroes is universal to all Java program analysis frameworks.

1 Introduction

Constructing sound and precise call graphs for object-oriented programs is often difficult and expensive. The key reason is dynamic dispatch: the target of a call depends on the runtime type of the receiver. One approach, Class Hierarchy Analysis (CHA) [8], is to conservatively assume that the receiver could be any object admitted by the statically declared type of the receiver. Because call graphs constructed with this assumption are imprecise, most call graph construction algorithms attempt to track the flow of potential receivers through the program [1, 6, 12, 13, 24]. Since the receiver might be created anywhere in the program, these algorithms generally analyze the whole program. However, modern programs have large library dependencies (e.g., the Java standard library). This makes it very expensive to construct a call graph even for a small program. Even if the algorithm itself is cheap, just reading all of the library dependencies of a program takes a long time. Moreover, in many cases, the whole program may not even be available for analysis.

Previously, we defined and evaluated the *separate compilation assumption* [2], which enables a sound and reasonably precise call graph to be constructed for a program without analyzing its library dependencies. In the rest of this paper, we will use the singular "library" to mean all of the libraries that a program depends on. The assumption states that the library is developed and can be compiled without the client program that uses it. This is true of most real programs and their dependencies. The properties that follow from the assumption and from the Java type system effectively limit the imprecision that would otherwise result from conservatively assuming arbitrary behaviour for the unanalyzed library code. We have evaluated the separate compilation assumption in Cgc [2], a prototype implementation in Datalog based on the Doop framework [6]. Our experiments have shown that with the separate compilation assumption, the sound call graphs constructed without analyzing the library can be nearly as precise as those constructed by whole-program analysis. However, implementing the constraints that follow from the assumption in popular analysis frameworks such as Doop [6], Soot [24], and Wala [12] is difficult, and would complicate the frameworks significantly and make them more difficult to maintain.

In this paper, we investigate whether the constraints that follow from the separate compilation assumption can be encoded in a form that is universal to all Java program analysis frameworks, the Java bytecode. Our goal is to enable any existing whole-program analysis framework to take advantage of the separate compilation assumption without modifications to the framework. To accomplish this, we present Averroes, a Java bytecode generator that, for a given program, generates a replacement for the program's library that embodies the constraints that follow from the separate compilation assumption. An existing, unmodified whole-program analysis framework needs only to read the replacement library instead of the original library to automatically gain the benefits of the separate compilation assumption. For example, instead of going through all of the work that was necessary to implement Cgc, one can now achieve the same effect automatically by running Averroes followed by Doop. Moreover, the same adaptation can be applied automatically not only to Doop, but to any other whole-program call graph construction framework.

We evaluate the performance improvements that the use of Averroes enables over the whole-program analysis frameworks Spark and Doop. The improvements are very significant because the replacement library is much smaller than the original library: for example, even version 1.4 of the Java standard library contains 25 MB of class files, whereas the Averroes replacement library contains in the order of only 80 kB of class files. Depending on the size of the analyzed client program, Averroes improves the running time of Spark and Doop by a factor of 4.7x and 3.7x, respectively, and reduces memory requirements by a factor of 13x and 8.4x, respectively.

Averroes also enables other benefits in addition to performance. One such benefit is generality. For example, many whole-program analysis frameworks are designed to soundly model some specific version of the Java standard library. However, the replacement library constructed by Averroes soundly overapproximates all possible implementations of the library that have the interface used by the client application. Therefore, Averroes makes any existing whole-program analysis framework independent of the Java standard library version. A related benefit is the handling of difficult features such as reflection and native methods. A whole-program analysis must correctly model in detail all such unanalyzable behaviour within the library in order to maintain soundness. On the other hand, Averroes is automatically sound for such behaviour because it already assumes that the library could "do anything". That said, the generated library must still model reflective effects of the library on the client application (e.g., reflective instantiation of classes of the application). However, this issue is also made easier by Averroes. Any tools or analyses that provide information about such reflective effects (e.g., analysis of strings passed to reflection methods or dynamic traces summarizing actual reflective behaviour) can be implemented once and for all in Averroes. Whole-program analysis frameworks can then take advantage of these effects without modification.

The rest of this paper is organized as follows. Section 2 provides background information about call graph construction and the separate compilation assumption. Section 3 describes how Averroes encodes the constraints that follow from the separate compilation assumption in the Java bytecode. Section 4 discusses the performance improvements gained by using the placeholder library generated by Averroes instead of the original library code. Section 5 presents related work, and Section 6 concludes this paper.

2 Background

2.1 Call Graph Construction

A static call graph is an overapproximation of the method calls that may occur in a program at run time. For every method invocation instruction in the program (a *call site*), the call graph contains an *edge* to every *target* method that might be invoked by that instruction. In Java, as well as other object-oriented languages, the targets of method calls are selected using the run-time type of the receiver object. Therefore, call graph construction requires a combination of two static analyses: calculating the sets of possible receiver types (i.e., points-to analysis), and determining the targets of method calls. The two analyses are inter-related: receiver types decide the targets of calls, and the calling relationships between methods determine how objects of specific types flow through the program to the call sites. A precise call graph construction algorithm computes these two

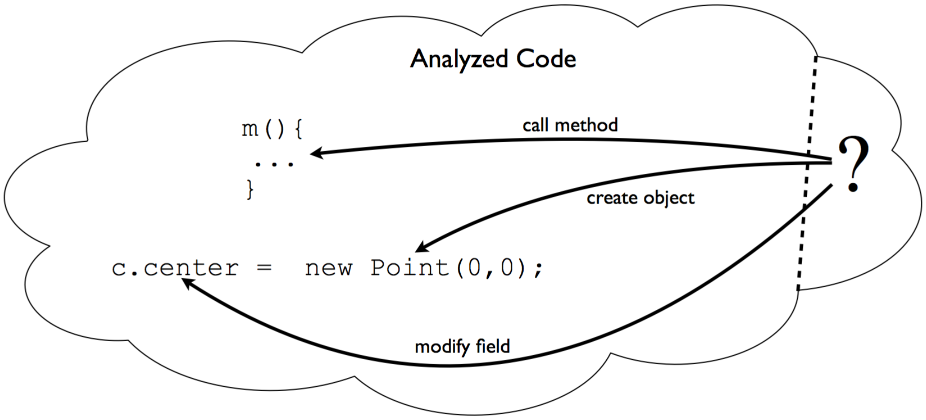


Fig. 1. Conservative assumptions that a sound partial-program analysis must make.

inter-dependent analyses concurrently until it reaches a mutual least fixed point. This is sometimes called *on-the-fly* call graph construction.

If the whole program is not available for analysis, a sound call graph construction algorithm must conservatively assume that the unanalyzed code could do “anything”. In particular, the unanalyzed code could call any method, assign any value to any field, and create new objects of any type, as summarized in Figure 1. Due to the mutual dependencies between the two analyses that make up any call graph construction algorithm, imprecise results in one analysis can quickly pollute the other. Therefore, without any assumptions about the unanalyzed part of the code, a sound algorithm generates a call graph that is so imprecise that it is useless. However, it is frequently the case that the unanalyzed code is a library that is developed separately and can be compiled without access to the rest of the program. This *separate compilation assumption* [2] enables the construction of a precise and sound call graph for the part of the program that is analyzed (the *application*) without analyzing the library.

2.2 The Separate Compilation Assumption

The key assumption underlying both Cgc [2] and Averroes is that the library can be compiled separately without the client application program. From this assumption, more specific constraints are inferred that bound the possible behaviours of the unanalyzed library code. In Cgc, the call graph construction algorithm is extended to conservatively assume that the library can have any behaviour that satisfies the constraints that follow from the separate compilation assumption. Averroes, on the other hand, constructs a placeholder library that exercises all those behaviours. Any unmodified whole-program call graph analysis can then analyze the application with the placeholder library to achieve a similar result as Cgc. The rest of this section briefly summarizes the constraints that follow from the separate compilation assumption and that underlie

Averroes. A thorough discussion of the justification of each of these constraints is found in [2].

Constraint 1 [*class hierarchy*]

A library class cannot extend or implement an application class or interface.

Constraint 2 [*class instantiation*]

An allocation site in a library method can instantiate an object whose run-time type is:

- a library class, or
- an application class whose name is known to the library (i.e., through reflection).

Constraint 3 [*local variables*]

Local variables in the library can point to the following objects:

- objects instantiated by the library,
- objects instantiated by the application and passed to the library due to interprocedural assignments,
- objects stored in fields accessible by the library code, or
- objects whose run-time type is a subtype of `java.lang.Throwable`.

Constraint 4 [*method calls*]

A call site in the library can invoke:

- any method in any library class visible at this call site, or
- a method m in an application class c, but only if:
  1. m is non-static and overrides a (possibly abstract) method of a library class, and
  2. a local variable in the library points to an object of type c or a subclass of c.

Constraint 5 [*field access*]

A statement in the library can access (i.e., read or modify):

- any field in any library class visible at this statement, or
- a field f of an object o of class c created in the application code, if:
  1. f is originally declared in a library class, and
  2. a local variable in the library points to the object o.

In the case of a field write, the object being stored into the field must also be pointed to by a local variable in the library.

Constraint 6 [*array access*]

The library can only access array objects pointed to by its local variables. If the library has access to an array, it can access any of its elements through its index. Similar to field writes, objects written into an array element must be pointed to by a local variable in the library.

Constraint 7 [*static initialization*]

The library causes the loading and static initialization (i.e., execution of the method `<clinit>()`) of classes that it instantiates (according to the class instantiation constraint).

Constraint 8 [*exception handling*]

The library can throw an exception object e if:

- e is instantiated by the library, or
- e is instantiated by the application and passed to the library (as discussed in the local variables constraint).

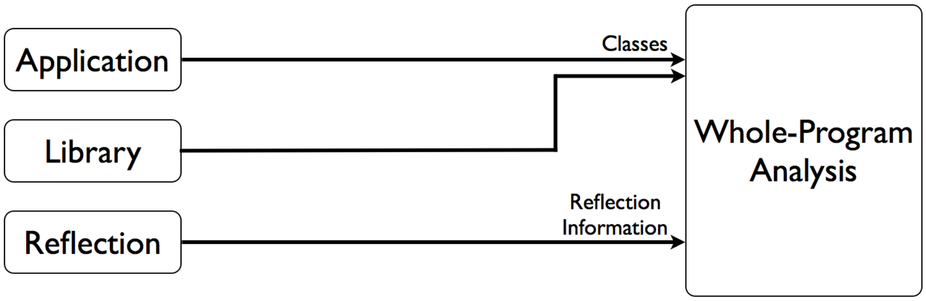


Fig. 2. The usual context of a whole-program analysis.

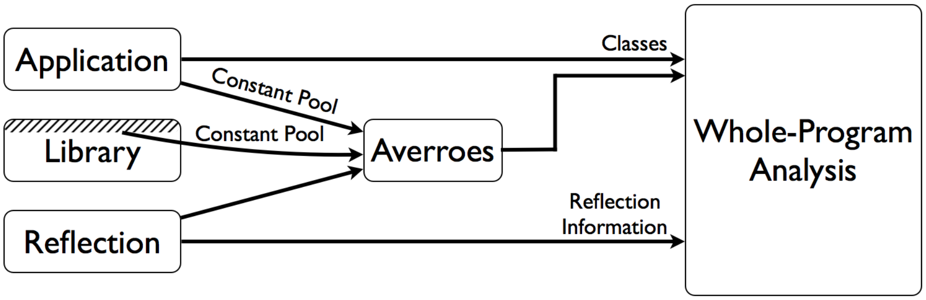


Fig. 3. The context of a whole-program analysis using Averroes.

3 Averroes Overview

This section presents the context in which Averroes is used, and defines in detail the contents of the placeholder library that it generates.

The usual context of a whole-program analysis tool is depicted in Figure 2. The tool expects to analyze all of the classes of the program, including any libraries that it uses. The tool does not necessarily distinguish between application and library classes. Optionally, the tool may also make use of additional information about the uses of reflection in the program. This information could be provided by the user or collected during execution of the program being analyzed with a tool such as TamiFlex [5].

We have implemented Averroes, a tool that intends to provide the same input environment to the whole-program analysis, but without analyzing any actual code of the original library classes. We have made Averroes available at `http://plg.uwaterloo.ca/~karim/projects/averroes/`. Figure 3 depicts the context in which Averroes is used. Given any Java program, Averroes generates an alternative placeholder library that models the constraints that follow from the separate compilation assumption. To achieve that, Averroes uses Soot [24] to consult the classes of the input program. Unlike a whole-program analysis, Averroes does not inspect all classes, and does not analyze any Java bytecode instructions. For each application class, Averroes examines only the constant pool to find all references to library classes, methods, and fields. Among library classes, Averroes consults only the classes that are directly referenced by the application and their superclasses and superinterfaces. Within this restricted set of classes, Averroes examines only the constant pool. Averroes uses this information in order to build a model of the class hierarchy and the overriding relationships between methods in the program. Since Averroes examines only a small fraction of classes and only a small fraction of each class file, the execution of Averroes can be much faster than a whole-program analysis that reads and analyzes the code of the whole program. In addition, if the library code itself calls other dependent libraries, Averroes can process the library even if those dependencies are not available for analysis, assuming they are not directly referenced from the application code. Averroes also, optionally, reads in the reflection facts generated by TamiFlex [5] for the input program.

The output of Averroes is a placeholder library. Averroes again uses Soot to generate this library. Moreover, Averroes uses the Java bytecode verification tools offered by BCEL [7] to verify that the generated library satisfies the specifications of valid Java bytecode. The placeholder library contains stubs of all of the library classes, methods, and fields referenced by the application, so that the application could be compiled with the placeholder library instead of the original library. As a consequence, the application classes together with the generated placeholder library make up a self-contained whole program that can be given as input to any whole-program analysis. The placeholder library is designed to be as small as possible, while still being self-contained, such that the whole-program analysis can analyze it much more efficiently than the original library. In addition, the placeholder library overapproximates all possible behaviours of the original library, so that the call graph analysis produces a sound call graph when analyzing the placeholder library instead of the original library. The rest of this section defines in detail the contents of the generated placeholder library.

3.1 Library Classes

The Averroes placeholder library contains three kinds of classes: referenced library classes, concrete implementation classes, and the Averroes library class. We define the structure of these classes first, and we define the contents of their methods in Section 3.2.

Referenced Library Classes. The Averroes placeholder library contains every library class directly referenced by the application and their superclasses and superinterfaces. In addition, it contains a small fixed set of basic classes that are mentioned explicitly in the Java Language Specification [14] and expected by whole-program analyses (e.g., `java.lang.Object` and `java.lang.Throwable`).

Each such referenced class contains placeholders for all of the methods and fields that are referenced by the application. A method m is considered to be referenced by the application if:

- a reference to m appears in the constant pool of an application class,
- m is a constructor or a static initializer in the original library class, or
- a call to some method m' referenced by the application may resolve to m.

A field f of type t is considered to be referenced by the application if a reference to f appears in the constant pool of an application class. If an included library method is native in the original library, its placeholder is made non-native. This is because the generated placeholder library should stand alone and not depend on other code, including native code. Furthermore, the *throws* clause of a generated placeholder library method can only contain exception classes that are referenced by the application. To ensure that every library class has at least one accessible constructor, Averroes adds a default constructor (i.e., a public constructor that takes no arguments) to every included library class.

Concrete Implementation Classes. The class instantiation constraint of the separate compilation assumption states that the library code can create an object of any library type. This includes types that are not referenced by the application. The object of an unreferenced type could still be accessed by the application through one of its super-types. For the purpose of constructing an application-only call graph, the exact run-time type of the object is not important, since any calls on the object will just resolve to the library summary node. However, the call graph construction algorithm must be aware that an object of such an unknown type could be the receiver of a call.

Figure 4(a) shows a sample Java program that calls the method `java.util.Vector.elements()`. The return type of the method is `java.util.Enumeration`, which is an interface. If the application then calls a method such as `hasMoreElements()` or `nextElement()` on the value returned from `java.util.Vector.elements()`, the call should resolve to the library. Therefore, the call

```
class Main {
  void foo() {
    Vector v = new Vector();
     ...
    Enumeration e = v.elements();
    while(e.hasMoreElements()) {
     ...
    }
  }
}
```

(a)

```
class EnumerationConcrete
        implements Enumeration {
  boolean hasMoreElements() {
    return true;
  }

  Object nextElement() {
    return (Object) libraryPointsTo ;
  }
}
```

(b)

Fig. 4. An example illustrating the concept of concrete implementation classes in Averroes: (a) sample application Java code that uses the class `java.util.Enumeration`, (b) the concrete implementation class that Averroes creates for `java.util.Enumeration` in the placeholder library.

graph construction analysis must be aware that the receiver of the call could be some object that implements the `java.util.Enumeration` interface. However, if the application does not implement the interface itself, and if it does not reference any library class that implements it, then the whole-program analysis would not know about the existence of any concrete class that implements the interface. In this case, AVERROES adds to the placeholder library a concrete class that implements the interface, so that the call graph construction algorithm can resolve the call on this class. Figure 4(b) illustrates the contents of the concrete implementation class that AVERROES generates for `java.util.Enumeration` in the placeholder library.

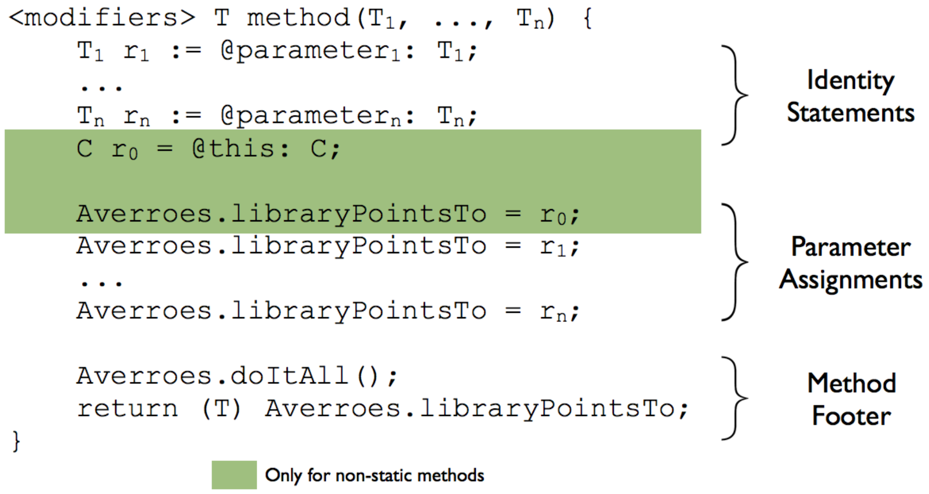
Specifically, AVERROES creates a concrete implementation class for each interface and abstract class in the library that is referenced by the application, but is not implemented by any concrete class already in the placeholder library. If the original library contains a concrete class implementing the given interface or abstract class, that concrete class would already be in the placeholder library only if the application references that concrete class. Each concrete implementation class contains implementations of all abstract methods in the interface or abstract class that caused the concrete implementation class to be created, including abstract methods inherited from superclasses and superinterfaces.

AVERROES **Library Class.** All of the conservative approximations of the possible behaviours of the library defined by the constraints listed in Section 2.2 are implemented in one class in the placeholder library, `AverroesLibraryClass`. In particular, this class models the following library behaviours: object instantiation, callbacks to application methods, array writes, and exception handling. The `AverroesLibraryClass` has two members:

1. The field `libraryPointsTo` is a public, static field of type `java.lang.Object`. It represents all local variables in the original library code. Every object that could be assigned to a local variable in the original library is assigned to this field. The points-to set of the `libraryPointsTo` field corresponds to the `LibraryPointsTo` set in CGC.
2. The method `doItAll()` is a public, static method. It is the main AVERROES method that models all of the potential side effects that the original library code could have.

3.2 Library Methods

Referenced Library Method Bodies. Each placeholder method in the referenced library classes and in the concrete implementation classes is an entry point from the application into the library, and should conservatively implement the behaviours specified in Section 2.2. Most of these behaviours are implemented just by calling the `doItAll()` method of the `AverroesLibraryClass`. In addition, each placeholder method stores all of its parameters to the `libraryPointsTo` field. The return value of the method is also taken from `libraryPointsTo`.



```
<modifiers> T method(T1, ..., Tn) {
    T1 r1 := @parameter1: T1;
    ...
    Tn rn := @parametern: Tn;
    C r0 = @this: C;

    Averroes.libraryPointsTo = r0;
    Averroes.libraryPointsTo = r1;
    ...
    Averroes.libraryPointsTo = rn;

    Averroes.doItAll();
    return (T) Averroes.libraryPointsTo;
}
```

Fig. 5. The Jimple template used by Averroes to generate bodies for referenced library methods.

More specifically, the body of each placeholder method is constructed according to the template shown in Figure 5. The template is shown in the Jimple intermediate language of the Soot framework [24], which is used by Averroes to generate the placeholder library. The template has three code regions:

1. Identity statements define the variables that will hold the method parameters. Non-static methods have an additional identity statement for the implicit `this` parameter.
2. Parameter assignment statements assign the parameters to the `libraryPointsTo` field in order to model the interprocedural flow of objects from the application through parameters into the library (the local variables constraint).
3. The method footer contains two statements. The first statement is a call to the `doItAll()` method in the `AverroesLibraryClass` to model the side effects of the library. The second statement is the `return` statement of the method. The method can return any object from the library whose type is compatible with the return type of the method. This is modelled by reading the `libraryPointsTo` field and casting its value to the method return type. This completes the implementation of the local variables constraint. If the return type of the method is primitive, the constant value 1 is returned. Methods with return type `void` will just have an empty `return` statement.

The bodies of constructors of placeholder library classes are generated using the same Jimple template. However, a call to the default constructor of the direct superclass is generated before accessing the `this` parameter in the constructor body. Moreover, Averroes generates statements that initialize the instance fields of the declaring class. Each instance field is initialized by assigning it the value of the libraryPointsTo field after casting it to the appropriate type (the field access constraint).

The bodies of library static initializers are simpler. Since static initializers have no parameters or return value, no identity statements or parameter assignment statements are generated for them. In addition, they have an empty return statement (i.e., one that does not return any value). Moreover, for each static initializer, Averroes initializes the static fields of its declaring class with the value from the libraryPointsTo field through the appropriate cast (the field access constraint).

Averroes doItAll() **Method Body.** The doItAll() method implements most of the conservative approximation of the behaviour of the whole library. It is a static method with no parameters, and therefore does not have any identity statements. The body of the doItAll() method implements the following behaviours:

1. Class instantiation (Constraints 2 and 7): According to the class instantiation constraint, the library can create an object of any concrete class in the library or any application class that is instantiated by reflection. For each such class *c*, two statements are generated: a new instruction to allocate the object, and a special invocation instruction (corresponding to the invokespecial bytecode) to an accessible constructor of the class. Finally, if the class *c* declares a static initializer, Averroes generates a call to it.
2. Library callbacks (Constraints 3 and 4): Following the method calls constraint, the doItAll() method contains calls to all methods of the library that are overridden by some method of the application, since at run time, any such call could dispatch to the application method. In addition, the doItAll() method calls all application methods known to be invoked by reflection. The receiver of all of these calls is taken from the libraryPointsTo field, as are all arguments to the method. The values from the libraryPointsTo field are cast to the appropriate types as required by the method signature. Additionally, the local variables constraint states that objects may flow from the application to the library due to interprocedural assignments. Therefore, in Averroes, if the target method of a library call back has a non-primitive return type, its return value is assigned to the field libraryPointsTo.
3. Array element writes (Constraint 5): The library could store any object reference that it has into any element of any array to which it has a reference. Two statements are generated to simulate this. The first statement casts the value of the libraryPointsTo field to an array of java.lang.Object, which is a supertype of all arrays of non-primitive types. The second statement assigns the value of the libraryPointsTo field to element 0 of the array.
4. Exception handling (Constraint 8): The library code could throw any exception object to which it has a reference. To model this, Averroes generates

code that casts the value of the `libraryPointsTo` field to the type `java.lang.Throwable`, and throws the resulting value using the Jimple `throw` statement (which corresponds to the `athrow` bytecode instruction).

In the current implementation of Averroes, the `doItAll()` method is a single straight-line piece of code with no control flow. If Averroes were to be used with a flow-sensitive analysis, control flow instructions should be added to all library methods, including the `doItAll()` method. This allows the instructions to be executed in an arbitrary order for an arbitrary number of times. This enables a sound overapproximation for all possible control flow in the original library. Although this would be easy to implement, we have not done it because all of the call graph construction frameworks for Java that we are aware of mainly do flow-insensitive analysis.

Similarly, the `doItAll()` method writes only to element 0 of every array. If a framework attempts to distinguish different array elements, this should be changed to a loop that writes to all array elements. Again, we are not aware of any call graph construction frameworks for Java that distinguish different array elements.

3.3 Modelling Reflection

Averroes models reflective behaviour in the library in two ways. First, whenever a call site in the application calls a library method, Averroes assumes that any argument of the call that is a string constant could be the name of an application class that the library instantiates by reflection. For every such string constant that is the name of an application class, Averroes generates a `new` instruction and a call to the default constructor of the class in the `doItAll()` method.

Second, Averroes reads information about uses of reflection in the format of TamiFlex [5]. TamiFlex is a dynamic tool that observes the execution of a program and records the actual uses of reflection that occur. Averroes then generates the corresponding behaviour in the `doItAll()` method. Alternatively, a programmer who knows how reflection is used in the program could write a sound reflection specification by hand in the TamiFlex format. Averroes generates the following code in the `doItAll()` method to model the reflective behaviour recorded in the TamiFlex format:

1. For every class that the TamiFlex file specifies as instantiated by `java.lang.Class.newInstance()`, or even just loaded by `java.lang.Class.forName()`, the `doItAll()` method allocates an instance of the class using a `new` instruction, and calls its default constructor.
2. For every unique appearance of `java.lang.reflect.Constructor.newInstance()` in the TamiFlex file, the `doItAll()` method allocates an instance of the specified class and calls the specified constructor on it.
3. For every unique appearance of `java.lang.reflect.Array.newInstance()` in the TamiFlex file, the `doItAll()` method allocates an array of the specified type.

4. For every unique appearance of `java.lang.reflect.Method.invoke()` in the TamiFlex file, the `doItAll()` method contains an explicit invocation of the appropriate method.

Even though these behaviours are triggered by reflection in the original library, the Averroes placeholder library implements all of them explicitly (non-reflectively) using standard Java bytecode instructions. Therefore, even if the whole-program analysis that follows Averroes does not itself handle reflection, it will automatically soundly handle the reflective behaviour that Averroes knows about. This is because Averroes encodes the behaviour explicitly in the placeholder library using standard bytecode instructions known to every analysis framework.

In addition, the placeholder library still contains the methods that implement reflection in the Java standard library. The `doItAll()` method also contains calls to `java.lang.Class.forName()` and `java.lang.Class.newInstance()` on the value of `libraryPointsTo` cast to `java.lang.String`. Therefore, if the whole-program framework knows about the special semantics of these reflection methods, or if it knows about some reflective behaviour that is unknown to Averroes, the whole-program framework can still model the additional reflective behaviour in the same way as if it were processing the original library instead of the Averroes placeholder library.

3.4 Code Verification

The placeholder library that is generated by Averroes is intended to be standard, verifiable Java bytecode that can be processed by any Java bytecode analysis tool. To guarantee this, Averroes verifies the generated placeholder library classes using the BCEL [7] verifier. BCEL closely follows the class file verification process defined in the Java Virtual Machine Specification [14, Section 4.9]. BCEL ensures the validity of the internal structure of the generated Java bytecode, the structure of each individual class, and the relationships between classes (e.g., the subclass hierarchy).

4 Evaluation

We evaluate how well Averroes achieves the goal of enabling whole-program analysis tools to construct sound and precise call graphs without analyzing the whole library. First, we quantify the improvements in performance when Averroes is used with both Spark and Doop. Second, we compare the resulting call graphs with dynamically observed call graphs to provide partial evidence that the static call graphs are sound. Third, we compare the call graphs constructed using Averroes to those constructed using Cgc to support the claim that Averroes enables existing whole-program analysis implementations to perform the kind of analysis that is prototyped in Cgc.

We conducted our experiments on two benchmark suites: the DaCapo benchmark programs version 2006-10-MR2 [4], and the SPEC JVM98 benchmark programs [20]. All of these programs are analyzed with the Java standard library from JDK 1.4 (jre1.4.2_11). We ran all of the experiments on a machine with four dual-core AMD Opteron 2.6 GHz CPUs (running in 64-bit mode) and 16 GB of RAM.

We created an artifact for the experiments that we conducted to evaluate Averroes. The artifact includes a tutorial with detailed instructions on how to use Averroes to generate the placeholder libraries for each program in our benchmark suites. It then shows how to reproduce all of the statistics we discuss in this section. We have made the artifact available at `http://plg.uwaterloo.ca/~karim/projects/averroes/tutorial.php`. The artifact has been successfully evaluated by the ECOOP Artifact Evaluation Committee and found to meet expectations.

4.1 Performance

To evaluate how much work a whole-program analysis saves by using Averroes, we first compare the size of the generated placeholder library with the size of the original Java standard library. We then measure the reductions in execution time and memory requirements of both Spark and Doop when using Averroes.

Averroes **Placeholder Library Size.** Over all of the benchmark programs that we have experimented with, the average size of the input library is 25 MB (min: 25 MB, max: 30 MB, geometric mean: 25 MB), while the average size of the generated Averroes library is only 80 kB (min: 20 kB, max: 370 kB, geometric mean: 80 kB). Additionally, the average number of methods in the original input library is 36,000 (min: 19,462, max: 48,610, geometric mean: 35,615), while the average number of methods in the generated Averroes library is only 600 (min: 137, max: 3,327, geometric mean: 570). That means that the number of methods in the placeholder library is smaller by a factor of 62x (min: 13x, max: 286x, geometric mean: 62x). As we will see, this reduction in the library size significantly reduces the time and memory required to do whole-program analysis.

Finding 1: The placeholder library generated by Averroes is very small compared to the original Java standard library.

Execution Time. We have compared the execution times of both Spark and Doop when analyzing each benchmark with the Averroes placeholder library and the original Java standard library. We break the total time required to construct a call graph into three components. First, the Averroes *library generation time* is the time required for Averroes to inspect the application for references to the library and to generate the placeholder library. Second, the *overhead time* is the time required for Spark or Doop to prepare for call graph

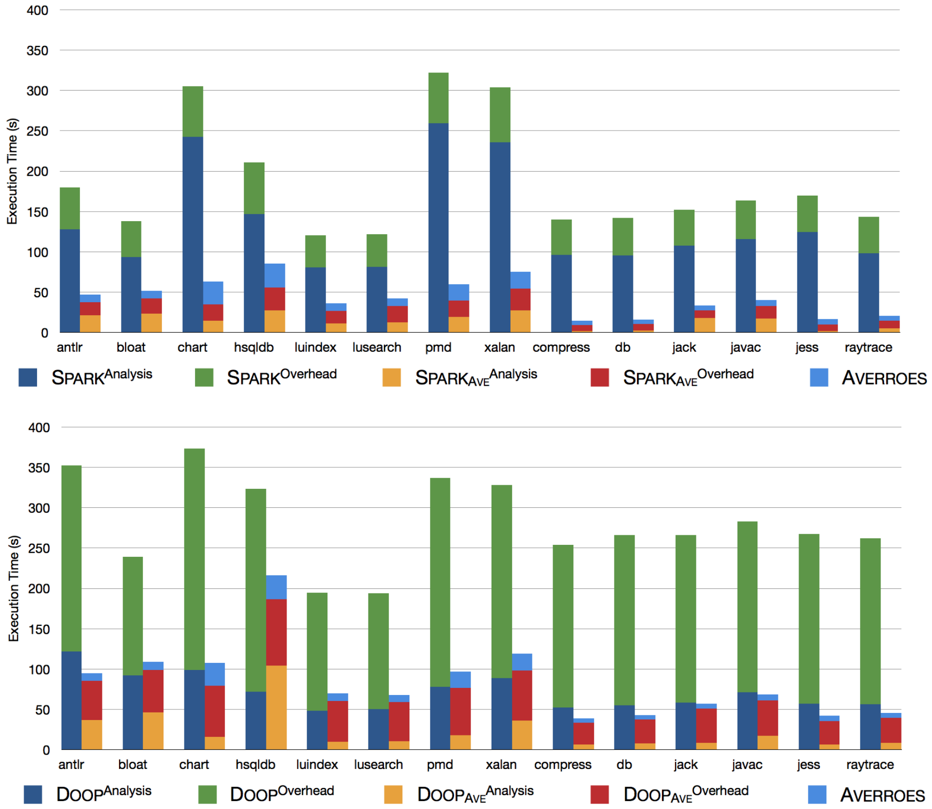


Fig. 6. The execution time of whole-program tools (Spark and Doop) compared to Averroes-based tools (Spark$_{\text{Ave}}$ and Doop$_{\text{Ave}}$).

construction analysis. In the case of Spark, this preparation includes reading the whole program from disk and constructing internal data structures. In the case of Doop, this preparation additionally includes generating the constraints required for the analysis and encoding them in Datalog relations. Third, the *analysis time* is the time required for Spark or Doop to solve the constraints and generate a call graph.

Figure 6 compares the times required for call graph construction by Spark and Doop with the original Java library (denoted Spark and Doop) and with the Averroes placeholder library (denoted Spark$_{\text{Ave}}$ and Doop$_{\text{Ave}}$). Averroes reduces the analysis time of Spark by a factor of 12x (min: 4x, max: 62x, geometric mean: 12x) and of Doop by a factor of 4.3x (min: 0.7x, max: 9.3x, geometric mean: 4.3x). In general, whole-program analysis is expensive not only because of the analysis itself, but also due to the overhead of reading a large whole program from disk and pre-processing it. Replacing the large Java library with the much smaller Averroes placeholder library reduces the time that

Spark executes (including overhead and analysis time) by a factor of 6.8x (min: 3.3x, max: 17x, geometric mean: 6.8x), and the time that Doop executes by a factor of 4.3x (min: 1.7x, max: 7.7x, geometric mean: 4.3x). When the Averroes library generation time is added to the time taken by Spark or Doop to finish, the total overall time to execute Spark$_{\text{Ave}}$ is faster than Spark by a factor of 4.7x (min: 2.5x, max: 10.3x, geometric mean: 4.7x), and the total overall time to execute Doop$_{\text{Ave}}$ is faster than Doop by a factor of 3.7x (min: 1.5x, max: 6.5x, geometric mean: 3.7x).

Finding 2: Averroes enables whole-program tools to construct application call graphs faster.

Memory Requirements. Spark and Doop store their intermediate results in different ways. Spark does all the calculations in memory, while Doop stores intermediate facts in a LogicBlox [15] database on disk. Therefore, we use different methods of calculating the memory requirements of each tool. We compare the maximum amount of heap space used during call graph construction by Spark$_{\text{Ave}}$ and Spark. On the other hand, we compare the on-disk size of the database of relations computed by Doop$_{\text{Ave}}$ and Doop.

Figure 7 compares the memory usage of Spark$_{\text{Ave}}$ against Spark, and Doop$_{\text{Ave}}$ against Doop. Overall, Spark$_{\text{Ave}}$ requires 13x less heap space than Spark (min: 4.8x, max: 35x, geometric mean: 13x), and Doop$_{\text{Ave}}$ uses 8.4x less disk space than Doop (min: 2.6x, max: 24, geometric mean: 8.4x).

Finding 3: Using Averroes reduces the memory requirements of whole-program analysis tools.

4.2 Call Graph Soundness

Static call graph construction is made difficult in Java by dynamic features such as reflection, and features that are difficult to analyze such as native methods. Averroes makes it much easier to construct a sound call graph in the presence of these features in two ways. First, whereas whole-program analysis frameworks try to model all behaviour of the whole program precisely, Averroes uses the conservative assumption that the library could have any behaviour consistent with the separate compilation assumption. Therefore, a whole-program analysis must model every detail of dynamic behaviour or risk becoming unsound. On the other hand, an analysis using Averroes remains sound without having to precisely reason about dynamic behaviour within the library. Second, Averroes contains analyses that model how the library affects the application using reflection. These analyses make use of information about strings passed into the library, as well as information about reflection generated by TamiFlex [5]. A whole-program analysis that uses Averroes can automatically benefit from the results of these analyses without having to implement the analyses themselves.

We have evaluated the soundness of static call graphs by comparing them against dynamic call graphs collected by *J [10]. Since a dynamic call graph

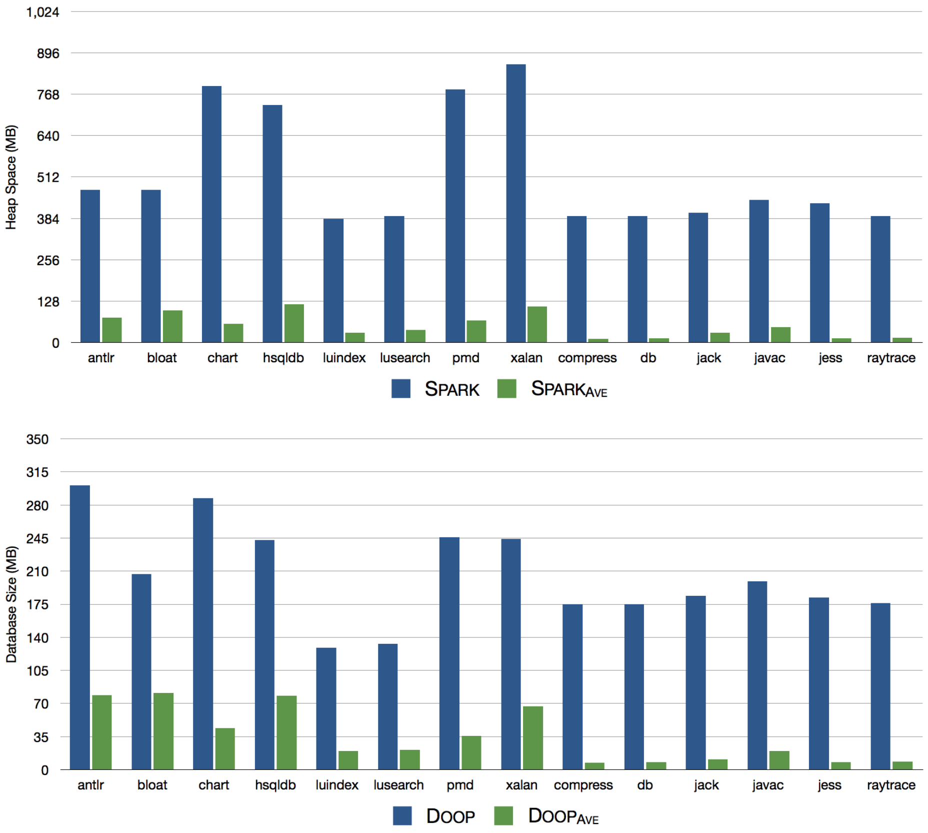


Fig. 7. The memory requirements of whole-program tools (Spark and Doop) compared to Averroes-based tools (Spark$_{\text{Ave}}$ and Doop$_{\text{Ave}}$).

results from only a single execution, it may miss edges that could execute in other executions. Therefore, such a comparison does not guarantee that the static call graph is sound for all executions. Nevertheless, the comparison can detect soundness violations, and the lack of detected violations provides at least partial assurance of soundness. The results of this comparison are shown in Table 1. The Dynamic line shows the number of call edges in the application portion of the dynamic call graph. The remaining lines show how many of these edges are missing in the static call graphs generated by Spark and Doop with and without using Averroes. When using Averroes, only two edges are missing from all of the call graphs. In `lusearch`, a `NullPointerException` is thrown and the dynamic call graph records a call edge from the virtual machine to the constructor of this exception class. This behaviour is not modeled by either Spark or Doop. In `xalan`, a call edge to `java.lang.ref.Finalizer.register()` from the application is missing from the call graph generated by Spark$_{\text{Ave}}$

Table 1. Comparing the soundness of Averroes-based tools to the whole-program tools with respect to the dynamic call graphs.

	antlr	**bloat**	**chart**	**hsqldb**	**luindex**	**lusearch**	**pmd**	**xalan**	**compress**	**db**	**jack**	**javac**	**jess**	**raytrace**
Dynamic	3,449	4,257	657	1,627	726	539	2,087	2,953	43	54	596	2,538	13	330
Dynamic-Spark	0	0	0	61	4	185	3	96	0	0	0	0	0	0
Dynamic-Spark$_{\text{Ave}}$	0	0	0	0	0	1	0	1	0	0	0	0	0	0
Dynamic-Doop	0	0	0	331	303	241	225	349	0	0	0	0	0	0
Dynamic-Doop$_{\text{Ave}}$	0	0	0	0	0	1	0	0	0	0	0	0	0	0

since Spark does not handle calls to this library method. On the other hand, the call graphs generated by Spark and Doop without using Averroes are missing a significant number of dynamically observed edges in benchmarks that make heavy use of reflection. This is despite the immense effort that has been expended to make these analysis frameworks handle reflection soundly.

Finding 4: Averroes reduces the difficulty of constructing sound static call graphs in the presence of reflection.

4.3 Comparison with Cgc

We previously evaluated the soundness and precision of call graphs constructed by Cgc [2]. Our empirical evaluation showed that the static call graphs from Cgc are sound when compared against the corresponding dynamic call graphs. Our results also showed that Cgc generates precise static call graphs when compared against those generated by Spark and Doop, for most programs in our benchmark suite. However, there are spurious edges in the call graphs generated by Cgc. Further investigation showed that most of these spurious edges are due to spurious library callback edges. A library callback edge is an edge from a call site in the library back to a method in the application. Those spurious library callback edges eventually cause a small number of spurious call edges within the application and from the application to the library.

The design goal of Averroes is to enable existing whole-program analysis frameworks to build call graphs without analyzing the library in the manner of Cgc. We validate this claim by comparing the call graphs constructed by Cgc with those constructed by Doop with Averroes, since Cgc is more similar to Doop than to Spark. Since the separate compilation assumption overapproximates the targets of call sites in the library, we focus our comparison on the library callback edges in the call graph.

Table 2 shows the number of library callback edges in the call graph generated by Cgc but missing from the call graph generated by Doop$_{\text{Ave}}$ (denoted by Cgc-Doop$_{\text{Ave}}$), and vice versa (denoted by Doop$_{\text{Ave}}$-Cgc). The table also shows the percentage of those missing edges with respect to the total number of edges in the call graph from Doop$_{\text{Ave}}$. The biggest difference is 2% of the edges, in the `chart` benchmark. All of the edges missing in Doop$_{\text{Ave}}$ and present in Cgc (i.e., Doop$_{\text{Ave}}$-Cgc) are due to more precise handling of reflective constructor calls in Averroes than in Cgc. When the library reflectively creates an object

Table 2. Comparing Doop$_{\text{Ave}}$ with Cgc with respect to library callback edges.

	antlr	**bloat**	**chart**	**hsqldb**	**luindex**	**lusearch**	**pmd**	**xalan**	**compress**	**db**	**jack**	**javac**	**jess**	**raytrace**
Cgc-Doop$_{\text{Ave}}$	2	0	0	16	0	0	5	44	0	0	0	0	0	0
Cgc-Doop$_{\text{Ave}}$ (%)	0.03%	0%	0%	0.15%	0%	0%	0.1%	0.35%	0%	0%	0%	0%	0%	0%
Doop$_{\text{Ave}}$-Cgc	7	13	54	8	10	17	9	10	0	1	0	4	0	0
Doop$_{\text{Ave}}$-Cgc (%)	0.1%	0.08%	2.02%	0.08%	0.83%	0.68%	0.17%	0.08%	0%	1.33%	0%	0.05%	0%	0%

of an application class C, Cgc considers the library to potentially call all public constructors of C. On the other hand, Averroes generates a call edge only to the specific constructor that is actually invoked according to TamiFlex. Further investigation shows that some edges are missing in Cgc and present in Doop$_{\text{Ave}}$ (i.e., Cgc-Doop$_{\text{Ave}}$) due to calls of `java.lang.reflect.Constructor.newInstance()`. Whereas Cgc ignores these calls (and handles only calls to `java.lang.Class.newInstance()`), Averroes models calls to both `newInstance()` methods.

Finding 5: Averroes matches and slightly exceeds Cgc in both precision and soundness.

5 Related Work

5.1 Call Graph Construction

A distinguishing feature of different whole-program call graph construction algorithms is how they approximate the targets of dynamically dispatched method calls. This affects how they approximate the run-time types of the receivers of those calls.

Early work on call graph construction used simple approximations of run-time types. Dean et al. [8] formulated *class hierarchy analysis* (CHA), which uses the assumption that the run-time type of a receiver could be any subtype of its statically declared type at the call site. Thus, CHA uses only static type information, and does not maintain any points-to sets of the possible run-time types of objects. Bacon and Sweeney [3] defined *rapid type analysis* (RTA), which refines the results of CHA by restricting the possible run-time types only to classes that are instantiated in the reachable part of the program.

Diwan et al. [9] presented more precise call graph construction algorithms for Modula-3 that remain simple and fast. Rather than maintaining a single set of possible run-time types in the whole program, as in RTA, they compute separate sets of run-time types for individual local variables.

Sundaresan et al. [21] introduced *variable type analysis* (VTA). VTA generates subset constraints to model the possible assignments between variables within the program. It then propagates points-to sets of the specific run-time types of each variable along these constraints. Unlike the analyses of Diwan et al. [9], VTA computes these points-to sets for heap-allocated objects in addition to local variables.

Tip and Palsberg [22] studied a range of call graph construction algorithms in which the scope of the points-to sets was varied between a single set for the whole program (like RTA) and a separate set for each variable (like VTA). Their implementation was later used by Tip et al. [23] to implement Jax, a practical application extractor for Java.

Several static analysis frameworks for Java now include call graph construction implementations with a range of algorithms that can be configured for the desired trade-off between precision and analysis cost. The Soot [24], Wala [12], and Doop [6] frameworks all construct call graphs as prerequisites to the other interprocedural analyses that they perform. All three frameworks use whole-program analysis to build the call graph. However, Soot and Wala can be configured to ignore parts of the input program and generate an unsound partial call graph only for the part of the program that is analyzed. Averroes enables these and other whole-program frameworks to construct sound partial call graphs.

5.2 Partial-Program Analysis

The excessive cost of analyzing a whole program has motivated various efforts to construct call graphs while analyzing only part of the program.

The analysis of Tip and Palsberg [22] analyzes partial programs by defining a special points-to set, S_E. This set summarizes the objects passed into the unanalyzed external code (i.e., the library). Similar to Cgc [2] and Averroes, the analysis assumes that the external library code can call back an application method if: the application method overrides a library method; and the set S_E contains an object on which dynamic dispatch would resolve to that application method.

The main challenge of analyzing the application part of a program while ignoring the library is determining objects that may escape from the predefined application scope to the library, and vice versa. This directly affects the points-to sets of the local variables in the application and the library (or the summarized library points-to set in the case of partial-program analyses). Grothoff et al. [11] presented Kacheck/J, a tool that is capable of identifying accidental leaks of heap object abstractions by inferring the *confinement* property [25–27] for Java classes. Kacheck/J considers a Java class to be *confined* when objects of its type do not escape its defining package. A partial-program analysis then needs only to analyze the defining package of the input Java classes to infer their confinement property.

Rountev and Ryder [18] proposed a novel whole-program call graph construction analysis for C. Although the analysis requires the whole program, it analyzes each module of the program separately. A C program can take the address of a function, and later invoke it by dereferencing the resulting function pointer. Therefore, the function that is invoked depends on the target of the function pointer. The analysis proceeds in two steps. First, conservative assumptions are made about all possible applications that could use a given library. The analysis then builds up a set of constraints that model the precise behaviour of the library. Second, these constraints are used to model the library in an analysis of a specific application. Rountev et al. [17] then adapted the approach to Java. Like Averroes, their implementation encodes the constraints collected from the library in executable placeholder code. However, unlike Averroes, this placeholder code is a precise and detailed summary of the exact effects of the library, and its construction requires the entire library to be analyzed. Moreover, some of the constraints require changes to the application code in addition to the placeholder library. In contrast, the purpose of Averroes is to generate a minimal library stub that enables a sound analysis of the original application code.

Rountev et al. [16] applied a similar approach to summarize the precise effects of Java libraries for the purpose of the interprocedural finite distributive subset (IFDS) and interprocedural distributive environment (IDE) algorithms. Although these algorithms already inherently construct summaries of callees to use in analyzing callers, they had to be extended in order to deal with the library calling back into application code. This is done by splitting methods into the part before and after an unknown call. Summaries are then generated for each part rather than the whole method. When the target of the unknown call later becomes available, the partial summaries are composed. Rountev et al. [19] evaluated the approach on two instances of IDE, a points-to analysis and a system dependence graph construction analysis.

Averroes builds on our work on Cgc [2], which defined the separate compilation assumption and derived from it specific constraints that conservatively model all possible behaviours of the library. These constraints were implemented in Cgc as an extension of an existing whole-program call graph construction tool. Experimental results showed that the resulting call graphs are sound and quite precise compared to those constructed by a whole-program analysis that analyzes the whole library precisely. Whereas Cgc required significant implementation effort to extend the whole-program framework, Averroes enables the same approach to be implemented directly by any existing whole-program call graph construction framework without requiring extensions.

6 Conclusions

We have shown that the separate compilation assumption can be encoded in the form of standard Java bytecode. This enables any existing whole-program call graph construction framework to easily make use of it. Our Averroes generator, given an input program, automatically generates a conservative replacement for the program's library that embodies the constraints that follow from the separate compilation assumption (i.e., a placeholder library).

Constructing the placeholder library is fast and does not require analyzing the whole program. Moreover, the resulting placeholder library is very small, especially when compared to the size of the Java standard library. We have empirically shown that using Averroes with an existing whole-program analysis framework reduces the cost of call graph construction by a factor of 4.3x to 12x in analysis time and 8.4x to 13x in memory requirements. Averroes also makes it easier for a whole-program framework to soundly handle reflection. That is because AVERROES makes a conservative approximation of all library behaviour, including reflection. Additionally, AVERROES provides support for modelling uses of reflection in the application. Finally, we have shown that the call graphs generated with AVERROES and an existing, unmodified, whole-program framework are as precise and sound as those obtained by explicitly implementing the separate compilation assumption in some specific framework.

We plan to extend this work to generate placeholder libraries for various widely-used Java frameworks (e.g., Android, J2EE, Eclipse Plug-in) using AVERROES. We hope that this will lead to an easier means of analyzing client applications developed in these frameworks without the need to analyze the framework itself. Like a library, a framework typically satisfies the separate compilation assumption because it is developed without knowledge of the client applications that will be developed within it. One major difference is that in a framework, the main entry point to the program resides in the framework rather than in the client application. The application is then reflectively started by the framework code. We expect that with only minor changes, AVERROES will be applicable to these and other Java frameworks.

References

1. Agrawal, G., Li, J., Su, Q.: Evaluating a demand driven technique for call graph construction. In: 11th International Conference on Compiler Construction. pp. 29–45. CC '02 (2002)
2. Ali, K., Lhoták, O.: Application-only call graph construction. In: Proceedings of the 26th European conference on Object-Oriented Programming. pp. 688–712. ECOOP'12, Springer-Verlag, Berlin, Heidelberg (2012)
3. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 324–341. OOPSLA '96 (1996)
4. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications. pp. 169–190. OOPSLA '06 (Oct 2006)
5. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: 33rd International Conference on Software Engineering. pp. 241–250. ICSE '11 (2011)
6. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 243–262. OOPSLA '09 (2009)
7. Dahm, M., van Zyl, J., Haase, E.: The bytecode engineering library (BCEL). `http://commons.apache.org/bcel/` (November 2003)

8. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static Class Hierarchy Analysis. In: 9th European Conference on Object-Oriented Programming. pp. 77–101. ECOOP '95 (1995)
9. Diwan, A., Moss, J.E.B., McKinley, K.S.: Simple and effective analysis of statically-typed object-oriented programs. In: 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 292–305. OOPSLA '96, New York, NY, USA (1996)
10. Dufour, B., Hendren, L., Verbrugge, C.: *J: a tool for dynamic analysis of Java programs. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 306–307. OOPSLA '03 (2003)
11. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 241–255. OOPSLA '01 (2001)
12. IBM: T.J. Watson Libraries for Analysis WALA. http://wala.sourceforge.net/ (November 2012)
13. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Methodol. 18, 3:1–3:53 (October 2008)
14. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading, MA, USA, second edn. (1999)
15. LogicBlox Home Page: http://logicblox.com/ (April 2013)
16. Rountev, A., Kagan, S., Marlowe, T.: Interprocedural dataflow analysis in the presence of large libraries. In: 15th International Conference on Compiler Construction. pp. 2–16. CC'06, Berlin, Heidelberg (2006)
17. Rountev, A., Milanova, A., Ryder, B.G.: Fragment class analysis for testing of polymorphism in Java software. IEEE Trans. Softw. Eng. 30, 372–387 (June 2004)
18. Rountev, A., Ryder, B.G.: Points-to and side-effect analyses for programs built with precompiled libraries. In: 10th International Conference on Compiler Construction. pp. 20–36. CC '01, Springer-Verlag, London, UK, UK (2001)
19. Rountev, A., Sharp, M., Xu, G.: IDE dataflow analysis in the presence of large object-oriented libraries. In: the Joint European Conferences on Theory and Practice of Software and the 17th International Conference on Compiler construction. pp. 53–68. CC'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
20. Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks. http://www.spec.org/jvm98/ (May 2012)
21. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 264–280. OOPSLA '00 (2000)
22. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 281–293. OOPSLA '00 (2000)
23. Tip, F., Sweeney, P.F., Laffra, C., Eisma, A., Streeter, D.: Practical extraction techniques for Java. ACM Trans. Program. Lang. Syst. 24, 625–666 (November 2002)
24. Vallée-Rai, R., Gagnon, E., Hendren, L.J., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: 9th International Conference on Compiler Construction. pp. 18–34. CC '00 (2000)

25. Vitek, J., Bokowski, B.: Confined types. In: 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. pp. 82–96 (1999)
26. Vitek, J., Bokowski, B.: Confined types in Java. Softw., Pract. Exper. 31(6), 507–532 (2001)
27. Zhao, T., Palsberg, J., Vitek, J.: Type-based confinement. J. Funct. Program. 16(1), 83–128 (2006)