Proc. 2nd International Workshop on State Of the Art in Java Program analysis (SOAP '13)—Seattle, WA

Soot Class Loading in the ROOTBEER GPU Compiler

Philip C. Pratt-Szeliga
Syracuse University
Syracuse, NY, USA
pcpratts@syr.edu

Marc-André Laverdière
TCS Innovation Labs,
Tata Consultancy Services, Ltd.
&
École Polytechnique de Montréal
Montréal, Canada
marc-andre.laverdiere-papineau@polymtl.ca

Ettore Merlo
École Polytechnique de Montréal
Montréal, Canada
ettore.merlo@polymtl.ca

James W. Fawcett
Syracuse University
Syracuse, NY, USA
jfawcett@twcny.rr.com

Roy D. Welch
Syracuse University
Syracuse, NY, USA
rowelch@syr.edu

Abstract

One of the first activities of the Soot program analysis framework is to load the classes for analysis. With the current class loader, more classes are loaded than necessary. The overhead in memory of these classes can make whole-program analysis of large binaries infeasible on systems with limited memory. This paper describes new algorithms and data structures to efficiently load Java Bytecode classes for whole program analysis in Soot. Our method uses a modified version of Rapid Type Analysis (RTA) to determine what classes, methods and fields would be reachable during program execution. This enables us to load significantly less information in memory to enable program analyses. We implemented our approach for loading Java bytecode in the Soot-based ROOTBEER compiler. The new class loader loaded a Scene that had 58% to 64% less classes, representing memory savings of 44% to 82%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – code generation—compilers

General Terms Algorithms, Performance, Design

Keywords class loading; Soot; Java Bytecode

1. Introduction

Class loading in Soot [3, 10] can require significant time and memory when using whole-program mode. The current algorithm loads all classes from the process path and any class transitively referenced from loaded classes. All methods in a class are loaded whether or not they exist in any possible program traces. The number of loaded types starts out small, but as each class is loaded, more and more unnecessary types are added to the `Scene`. This results in a large memory footprint for analyzing even small programs, making some analyses impossible on lower-end systems.

To illustrate this issue, we introduce a running example in Listing 1.

```
public class A {
  public static void main(String[] args){
    B b = new B();
    b.foo();
  }
}
public class B {
  public void foo(){
    C c = new C();
    c.abc();
  }
  public void bar(){
    D d = new D();
    d.abc();
  }
}
public class C {
  public void abc(){
    System.out.println("In class C");}
}
public class D {
  public void abc(){
    System.out.println("In class D");}
}
```

Listing 1. Running example

When loading this program in Soot in whole-program mode, without exclusions, the class loader fully loads 2031 `SootClass` objects. With the ROOTBEER class loader it loads 712 `SootClass` objects, representing an approximate reduction of 114 Mb in memory usage.

This paper describes a new class loading algorithm and the required data structures to only load the set of classes that would be needed during program execution. Our strategy is to gather information from the Coffi bytecode representation without creating the Soot internal objects, while we build an over-approximate call

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOAP'13 June 20, 2013, Seattle, Washington, USA
Copyright © 2013 ACM ISBN 978-1-4503-2201-0/13/06...$15.00

Resolving Level	Associated Information
Dangling	Nothing is known about the class, except its fully-qualified name.
Hierarchy	The class, its superclass and interfaces are loaded.
Signatures	Hierarchy information, as well as types declared in method signatures and fields.
Bodies	Signatures information, as well as the classes referenced in the method bodies.

Table 1. Resolving levels in Soot

graph. Once all of the reachable classes have been discovered, the types are numbered according to the class hierarchy and we load a minimal Scene.

We have implemented this scheme in the ROOTBEER GPU Compiler [5].

The contents are organized as follows. In section 2, we will first describe the current Soot class loading algorithm and why it has problems with memory usage. Then, in section 3, we will describe our methodology and describe the new class loading algorithm in detail. In section 4, we describe the new API added to Soot. We then compare the performance of our method with Soot's class loading in ROOTBEER in section 5. Finally, we conclude in section 6.

2. Soot Whole-Program Class Loading

The original Soot class loading algorithm has three basic resolving levels: Hierarchy, Signatures and Bodies. As the level of a class progresses from Hierarchy to Bodies, additional details are loaded from the class files to memory. Table 1 summarizes the information added at each resolving level.

We now describe the class loading algorithm with more details, without including information related to phantom reference support[1]. The Soot resolver will work incrementally in the Scene, by means of a worklist[2], by loading each class and its dependencies, one at a time.

Note that the steps below are repeated for every class in the process path and every basic class. Please note also that every step has an early termination feature if the class is determined to be at the given level already.

The algorithm is initialized in the `resolveClass` method with an empty `SootClass` container object at the Dangling level. This empty container is then added to the desired resolving level.

In the first phase, Soot loads the class to the hierarchy level. First, the class source (Coffi, Jimple, etc.) is loaded into the `SootClass` object. The `SootClass` objects of the outer class, the super class and the interfaces are added to the worklist. The corresponding `Type` instances are created and stored for later use.

In the second phase, Soot loads the class to the signature level. It does so by examining the method signatures and the fields. All classes identified in the fields and signatures (including the return type, the arguments and the declared exceptions) are added to the worklist.

In the third phase, Soot resolves the class at the bodies level. All types found in every method from all soot classes are loaded.

The weakness of this algorithm is its all-or-nothing nature. It loads all the classes that are reachable from the basic classes and the processable classes, irrespective of whether they would be used in the program or not.

[1] Phantom references are classes that are not resolved, either because they are not found in the class path, or have been excluded.

[2] There are four worklists, but their behavior is identical in whole-program mode, so we merge them as one for simplicity

For our running example (c.f. Listing 1), this algorithm loads the classes A,B,C and D, as well as a large amount of JDK classes.

3. Class Loading using RTA

Our class loading strategy is to keep class names, method bodies as well as method and field signatures as strings for as long as possible. The `SootClass`, `SootMethod`, `SootField` and `Body` objects are only created once the types are numbered. Another important part of our strategy is that we load classes that are found on a depth first search walk from the entry points.

For our running example (c.f. Listing 1), this algorithm loads the classes A,B and C, as well as a restricted set of JDK classes.

We explain the initial class loading step in section 3.1, followed by the construction of the call graph in section 3.2. Then, we explain some special features of ROOTBEER and how we accommodate them in our class loading scheme in section 3.3. Afterwards, we explain the class numbering algorithm in section 3.4 and how the `Scene` object is populated in section 3.5.

3.1 Load and Wrap Classes

The first step of our algorithm uses the list of classes in the process path and loads them in memory using Coffi. We convert the constant pool indices for the current class, super class and interfaces into a string representation. We put all this information into a wrapper class named `HierarchyClass` and store them in a map (with the class name as key) for fast lookups.

3.2 Building a Call Graph

In order to know which classes are reachable in a normal program execution, we build a call graph of the program using a bidirectional variation of the optimistic Rapid Type Analysis (RTA) algorithm [1, 7].

We chose RTA as our base because it is a very fast algorithm that doesn't require us to perform any flow analyses, making it suited for the low-information phase we are in.

The original optimistic RTA algorithm uses Class Hierarchy Analysis (CHA) [2] to bootstrap a traversal of the program from the entry point. While the program is traversed, the class instantiations detected are recorded, and only the call targets that correspond to the found classes are preserved. This algorithm executes as a fixed point over the number of new class invocations.

However, this algorithm needs to be modified to fit our needs. First, RTA presumes a single entry point at the root of the program execution. In our case, however, we allow multiple entry points, which may not be at the start of the program execution.

Note that this traversal can be bounded, resulting in a truncated call graph. While it is unsound, it is often useful in practice. The mechanism for evaluating whether to visit a node is called the *don't follow* set, which is described in section 4.

3.2.1 Entry Points

The first step in building a call graph is to determine the entry points. Our class loader provides a flexible entry point mechanism. Soot users who analyze software that does not have a `main` method need not generate one for the sake of analysis, but simply need to create an appropriate `MethodTester` class and register it with the `RootbeerClassLoader.addEntryMethodTester` method. All the details of this API is provided in section 4.

Our class loader submits all the methods found in the previous steps to the registered method testers. If any of the testers return true, then this method is considered as an entry point.

3.2.2 Forward Traversal

During the forward traversal, we start at the entry points and traverse program using a breadth-first search. Along the way, we record all constructor invocations. Since we do not load instructions to Jimple, we use the `HierarchyValueSwitch` API described in section 4. When a constructor call to a new class is found, its virtual methods are placed on the call graph and the virtual method cache.

To make the backward traversal rapid, the forward traversal maintains the *reachable set*, which is the set of all reachable methods. This set is seeded with the entry points and we add the signature of every method that is found reachable.

3.2.3 Backward Traversal

If an entry point is not a main function, all of the methods on a depth first search walk from a main to the entry point need to be loaded in order to accurately represent the runtime behavior.

This is handled by a backward traversal from the entry points. When traversing the methods, we see if a method body makes a call into the *reachable set*. If it does, the edge is added to the call graph and the new method is added to the forward traversal queue.

3.2.4 Convergence

Since our algorithm is monotonic (only new virtual call targets are added) and the number of classes loaded is finite, we are guaranteed to converge. Note that we use the number of classes created with `new` to determine the fixed point.

3.3 ROOTBEER-Specific Overrides

The ROOTBEER GPU Compiler needs additional information in the loaded `Scene` that would not be present in the classes loaded previously.

Firstly, ROOTBEER needs a mechanism to handle entry points determined by reflection, which ROOTBEER uses. These are called *follow methods* and *follow classes*. Note that *follow methods* could also be used to model other non-explicit entry points, such as `Thread.run` entry points. These are added to the set of entry points when the call graph is constructed.

Secondly, ROOTBEER also needs its runtime classes to be loaded to signatures (i.e. with no method bodies) so that the generated Jimple code can refer to the runtime methods. The set of *to-signature classes* and *methods* allows this.

All the overrides mentioned are configured with `MethodTesters` and `ClassTesters`. The *follow testers* and *to-signature testers* are evaluated once to save time.

3.4 Numbering Types

After the call graph is created, the types are numbered as follows. The number starts at one for the root Java `Object`. Afterwards, we number the interfaces in reverse topological order. Then, we number the remaining classes using a breadth-first traversal on the hierarchy.

We use topological sorting for the interfaces because interfaces can inherit from multiple interfaces, unlike objects. A pure breadth-first numbering has problems where an interface appended to an interface hierarchy can be numbered incorrectly.

3.5 Load Scene

The final stage of the algorithm is to load the Scene.

The loading happens in four steps: loading the hierarchy, loading reachable fields, loading reachable methods, and finally the method bodies are loaded from Coffi.

These steps are necessary because the `SootClass`, `SootField` and `SootMethod` instances reachable from the bodies must exist before the body can be resolved.

The last two steps are straight-forward, so we will only explain the loading of the hierarchy and of the reachable fields.

To load the hierarchy of `SootClasss`, the numbered types are visited from smallest to largest and empty `SootClass` objects are created. At each empty `SootClass` object the super class and interfaces are filled in. The super class and interfaces are retrieved from the `Scene` by calling `Scene.getSootClass`. Since the numbering is according to the class hierarchy, the object classes are guaranteed to already exist in the `Scene`.

To load the fields, we traverse the program to find all the field references. We parse the string-based information with `FieldSignatureUtil` and construct the appropriate `SootField` object. This field object is then added to the declaring `SootClass`.

Note that this approach results in a subset of the classes loaded, and a subset of the class details being converted to Jimple. Since our call graph is sound, no information is missing that would be required during program execution. However, if unknown library classes are linked with our output at runtime, unknown method exceptions can occur.

In the case of our running example, we would have a lighter scene loaded, as illustrated in Listing 2. Please note that JDK library classes were not added in. We notice that the class `B` does not have the `bar()` method, nor is the class `D` loaded.

```
public class A {
  public static void main(String[] args){
    B b = new B();
    b.foo();
  }
}
public class B {
  public void foo(){
    C c = new C();
    c.abc();
  }
}
public class C {
  public void abc(){
    System.out.println("In class C");}
}
```

Listing 2. Loaded Scene for Running example

4. Implementation

To create the ROOTBEER class loader, new classes and data structures were needed that supported keeping the information as strings. This section gives a summary of the classes included with the ROOTBEER class loader.

Note that analyses in Soot will not need to refer to these APIs, as the class loader eventually brings the Coffi data to the familiar classes, and that the `jb` or `sb` phases will load the bodies in Jimple [9] or Shimple [8], respectively.

The `RootbeerClassLoader` is a singleton with the entry point to class loading (`loadNecessaryClasses`). You can configure the entry points by adding a `MethodTester`. Please refer to Figure 1 for the details of the methods offered.

The `ClassHierarchy` (see Figure 2) is reachable from the `RootbeerClassLoader` singleton. It allows a developer to retrieve the virtual methods for a signature, get a `HierarchyGraph` for a class name, get a `HierarchyClass` and get the `NumberedTypes`.

The `MethodTester` interface is designed to be implemented by classes that can tell if a `HierarchyMethod` meets a criteria. It has a single method, `boolean test(HierarchyMethod sm)`, which returns true if the criterion is met.

The `ClassTester` interface is similar to `MethodTester`, except that it can be used to see if a `HierarchyClass` meets a class or package criterion. It has a single method, `boolean test(HierarchyClass sm)`. The `RootbeerClassLoader` calls

Method	**Description**
`void loadNecessaryClasses()`	Load the Scene
`ClassHierarchy getClassHierarchy()`	Get the class hierarchy data structure
`List<SootMethod> getEntryPoints()`	Get the `SootMethod` entry points that were specified using the MethodTesters
`void addEntryMethodTester (MethodTester mt)`	Add a MethodTester that will be used to define entry points
`void addFollowMethodTester (MethodTester mt)`	Add a MethodTester that will be used to include methods in the reachable walk
`void addFollowClassTester (ClassTester ct)`	Add a ClassTester that will be used to include classes in the reachable walk
`void addDontFollowMethodTester (MethodTeter mt)`	Add a MethodTester that will stop a reachable walk
`void addDontFollowClassTester (ClassTester ct)`	Add a ClassTester that will stop a reachable walk
`void addToSignaturesMethodTester (MethodTester mt)`	Add a MethodTester that will define methods to be raised to signatures
`void addToSignaturesClassTester (ClassTester ct)`	Add a ClassTester that will define classes to be raised to signatures

Figure 1. `RootbeerClassLoader` API

Method	**Description**
`boolean containsClass(String name)`	Returns true if the class name is loaded
`List<String> getAllVirtualMethods(String signature)`	Return the virtual method signatures with the same covariant subsignature for all classes where new has been invoked on the string call graph
`HierarchyGraph getHierarchyGraph(String className)`	Gets a HierarchyGraph for a class name
`HierarchyClass getHierarchyClass(String className)`	Gets a HierarchyClass for a class name
`List<NumberedType> getNumberedTypes()`	Get the numbered types in ascending order
`Set<String> getAllClasses()`	Get all of the classes in the class hierarchy

Figure 2. `ClassHierarchy` API

the `ClassTester` once per class and the `MethodTester` once per method.

The `HierarchyClass` keeps the class name, super class name and interfaces as strings. It can return all of the `HierarchyMethods` or find them by name or sub-signature. Its methods are described in Figure 3.

The `HierarchyMethod` keeps the method name, return type, parameter types and exception types as strings. It can return the bytecode instructions of a method as `HierarchyInstructions`. The `HierarchyInstructions` are string based representations of the bytecode. The method signature and sub-signature can be obtained. The class is described in Figure 4.

Method	**Description**
`List<String> getExceptionTypes()`	Returns the exceptions
`HierarchyClass getHierarchyClass()`	Get the declaring HierarchyClass
`List <HierarchyInstruction> getInstructions()`	Get the Instructions of the method in a string based format
`String getName()`	Get the name of the method
`List<String> getParameterTypes()`	Get the types of the parameters
`String getReturnType()`	Get the return type
`String getSignature()`	Get the signature of the method
`String getSubSignature()`	Get the subsignature of the method
`String getCovarientSignature()`	Get the covariant subsignature of the method

Figure 4. `HierarchyMethod` API

The `HierarchyInstruction` has two methods. The first is `String getName()`, which keeps track of the name of the instruction (like "new"). The second is `List<Operand> getOperands()`, representing the string-based `Operand`.

The `Operand` class keeps track of the instruction operand as converted to a string using the class constant pool. It keeps track of a string-based type that allows easy searching for class_refs, method_refs and field_refs. Its description is available in Figure 5.

Method	**Description**
`String getValue()`	Returns, as a string, the operand parsed from the constant pool
`String getType()`	Returns a type of the value. Can be one of the following: **basic types** byte, int, long, float, double, string **descriptor types** class_ref, method_ref, field_ref

Figure 5. `Operand` API

The `HierarchyValueSwitch` takes a method signature and processes each `HierarchyInstruction`. It records types, method and field references and new invokes. It is described in Figure 6

There is one `HierarchyGraph` instance per class name. It allows the developer to retrieve all classes in the hierarchy, the direct children of a parent class and the parents of a child class. It is described in Figure 7.

The `StringCallGraph` (see Figure 8) keeps a mapping of edges going out of a method signature and methods hitting a method signature. It also can return all signatures and tell if a signature is on the call graph.

Method	**Description**
`HierarchyMethod getMethodByName(String name)`	Searches for a method by name
`HierarchyMethod getMethodBySubSignature(String subSig)`	Searches for a method by subsignature
`List<String> getInterfaces()`	Get the interfaces as a string
`List<HierarchyMethod> getMethods()`	Get the methods
`HierarchyMethod getMethodCovarient(String css)`	Get the HierarchyMethod matching the input covariant subsignature
`String getName()`	Get the name of the class
`String getSuperClass()`	Get the name of the super class
`boolean hasSuperClass()`	Returns true if the class has a super class

Figure 3. `HierarchyClass` API

Method	**Description**
`Set<String> getRefTypes()`	Get reference types
`Set<String> getAllTypes()`	Get all types
`Set<String> getArrayTypes()`	Get array types
`Set<String> getMethodRefs()`	Get method refs
`Set<String> getFieldRefs()`	Get field refs
`Set<String> getNewInvokes()`	Get new invokes

Figure 6. `HierarchyValueSwitch` API

Method	**Description**
`List<String> getAllClasses()`	Get all the classes in the hierarchy graph
`List<String> getChildren (String parent)`	Get the direct children of a parent class or interface
`List<String> getParents (String child)`	Get the direct parents of a child class or interface

Figure 7. `HierarchyGraph` API

Method	**Description**
`Set<String> getAllSignatures()`	Get all reachable method signatures in the call graph
`Set<String> getForwardEdges(String sourceSig)`	Get the method signatures that the sourceSig method calls directly
`Set<String> getReverseEdges(String destSig)`	Get the method signatures that call the destSig directly
`boolean isReachable(String signature)`	Returns true if the method signature is in the call graph

Figure 8. `StringCallGraph` API

There are two remaining useful classes in the `soot.rbclassload` package. `MethodSignatureUtil` can parse a method signature and return its name and types as strings. Any part of a method signature can be modified and a `SootMethod` can be returned. `FieldSignatureUtil` can parse a field signature and return its parts. Also any part of a field signature can be modified and a `SootField` can be returned.

5. Evaluation

We have implemented our algorithm inside the ROOTBEER-specific fork of Soot. We currently only load from Java bytecode. The code is available online[3] and we hope to integrate it in Soot in the near future.

We have executed the test cases on a 4 core Intel Xeon Processor (E5405) running at 2.00GHz with 16 Gb of RAM, running Linux 2.6.32-5-amd64 and JDK 1.6.0_45 64-bit.

Our test cases include our running example, and the SPECjvm2008 benchmark [6]. We measured the execution time with Soot's built-in feature, and used the Classmexer[4] agent to measure the memory used by the loaded `SootClass` objects. The overall memory usage was calculated using the JVM's global memory usage feature.

Note that we configured our program to run the SPARK [4] points-to-analysis engine with default settings afterwards, in order to validate that our method did not remove any necessary dependency for analysis.

Test Case	**Classes Loaded**	**Memory Used (Mb)**	`SootClass` **Memory (Mb)**	**Time (s)**
Running example (Soot)	2031	885	139	91
Running example (RBC)	712	1357	25	105
SPEC jvm 2008 (Soot)	6539	3411	333	469
SPEC jvm 2008 (RBC)	2737	3733	186	337

Table 2. Experimental results

We compare our implementation against the Soot class loader by loading programs and taking measurements in the wjpp phase in Table 2.

While we could have measured the overall memory used by the Java virtual machine at any given point, we chose not to because of the garbage collection's impredictability as well as the fact that newer garbage collectors typically work continuously in a separate thread. These factors make it essentially impossible to know what is the amount of memory really used in the program at a given point. This is the reason why we measured the `SootClass` usage separately.

[3] `https://github.com/pcpratts/soot/tree/feature/rbcl`

[4] `http://www.javamex.com/classmexer/`

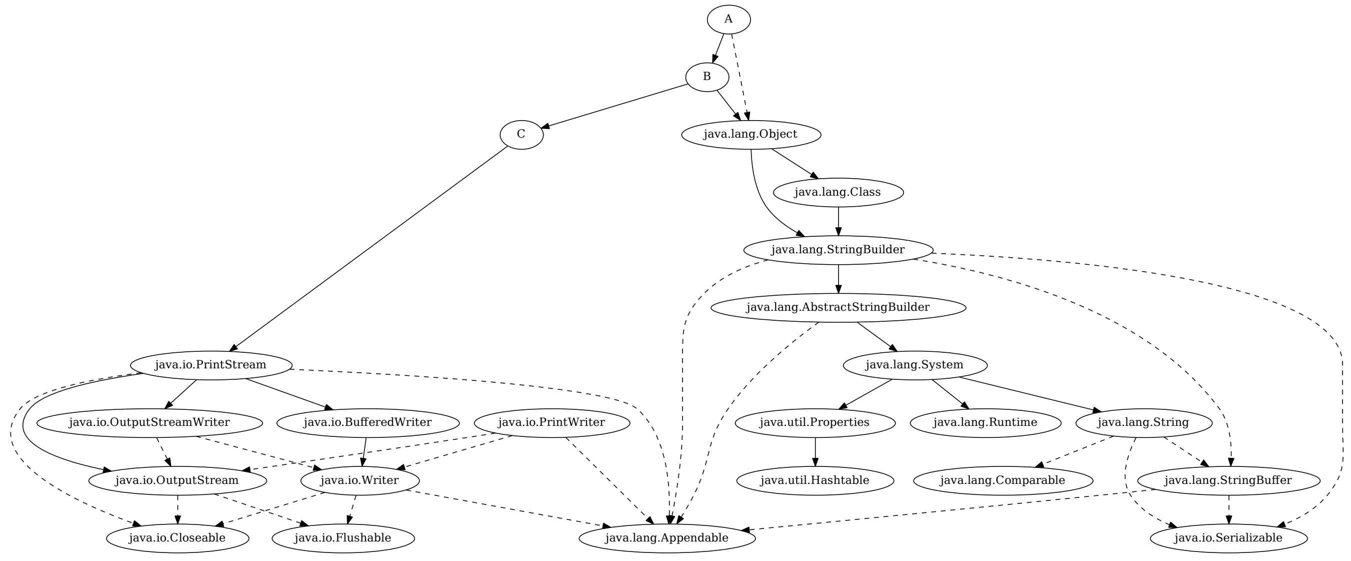

Figure 9. Subset of Class Dependencies for Listing 1

We notice a significant decrease in the number of classes loaded, with a decrease in processing time for the large test case. Our method requires significantly less information to be loaded from the disk. This explains why the class loading performance improves for larger test cases.

We include an illustration of a subset of the dependencies loaded with the current class loader for our running example in Figure 9. We show additional classes that would be loaded at the Signatures level by Soot's class loader with dashed edges. While significantly truncated, it shows that even trivial programs require a significant number of core Java classes to be loaded in memory.

6. Conclusion

We have shown data structures and algorithms to efficiently load a Soot Scene. The new class loader loaded a Scene that had 58% to 64% less classes, representing memory savings of 44% to 82%.

Acknowledgments

Philip C. Pratt-Szeliga was supported by Syracuse University and National Science Foundation grant number MCB-0746066 to R.D.W.

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*, OOPSLA '96, pages 324–341. ACM, 1996. ISBN 0-89791-788-X.

[2] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101. Springer-Verlag, 1995.

[3] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. Presentation at the CETUS Compiler Workshop, 2011.

[4] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[5] P. Pratt-Szeliga, J. Fawcett, and R. Welch. Rootbeer: Seamlessly using GPUs from Java. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 375–380, 2012.

[6] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 performance characterization. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35. Springer-Verlag, 2009. ISBN 978-3-540-93798-2.

[7] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. *SIGPLAN Not.*, 35(10):264–280, Oct. 2000. ISSN 0362-1340.

[8] N. Umanee. Shimple: An investigation of static single assignment form. Master's thesis, McGill University, 2006.

[9] R. Vallée-Rai and L. J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, McGill University, 1998.

[10] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.