Sealed Calls in Java Packages

Ayal Zaks
IBM Research Laboratory in Haifa, Israel
zaks@il.ibm.com

Vitaly Feldman
IBM Research Laboratory in Haifa, Israel
felvit@cs.technion.ac.il

Nava Aizikowitz
IBM Research Laboratory in Haifa, Israel
aizik@il.ibm.com

ABSTRACT

Determining the potential targets of virtual method invocations is essential for inter-procedural optimizations of object-oriented programs. It is generally hard to determine such targets accurately. The problem is especially difficult for dynamic languages such as Java, because additional targets of virtual calls may appear at runtime. Current mechanisms that enable inter-procedural optimizations for dynamic languages, repeatedly validate the optimizations at runtime. This paper addresses this predicament by proposing a novel technique for conservative devirtualization analysis, which applies to a significant number of virtual calls in Java programs. Unlike previous work, our technique requires neither whole program analysis nor runtime information, and incurs no runtime overhead. Our solution is very efficient to compute and is based on a newly introduced, seemingly unrelated security feature of Java file archives. On average, our analysis "seals" (safely devirtualizes) about 39% of the virtual calls (to non-final methods) that appear in SPECjvm98 programs, and about 29% of the calls invoked while executing these programs. In the runtime library rt.jar, about 10% of the packages contain a significant percentage (20–60%) of sealed calls, with a total average of about 8.5%. Most of these calls are also shown to be monomorphic, a fact which can be safely exploited by aggressive inter-procedural optimizations such as direct inlining. These results indicate that our technique has a strong potential for enhancing the analysis and optimization of Java programs.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Java, Object-oriented Programming*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization, Run-time environments*

General Terms

Languages, Performance

Keywords

Java, object-oriented programming, inter-procedural analysis, call devirtualization, method inlining, class hierarchy graph, call graph, sealed package

1. INTRODUCTION

Inter-procedural optimizations of object-oriented programs are based on the determination of the potential targets of virtual method invocations. This is a challenging problem especially for dynamic languages such as Java$^{\text{TM}}$ [1], because additional targets of virtual calls may appear at runtime. In certain languages the programmer can prohibit methods from being overridden, thereby resolving calls to these methods. Calls to such "final" methods are not considered to be virtual calls in this paper. Existing techniques that determine the targets of virtual calls in dynamic languages either restrict dynamic class loading (by relying on the user [36] or by applying *whole-program analysis* [4]), or inflict a runtime overhead. In addition, some solutions imply restrictions on further optimizations and some require special runtime mechanisms. An ideal solution should have no runtime overhead and restrictions. So far, all techniques fall short of this ideal.

Our goal is to overcome all these drawbacks by identifying a new type of calls in Java whose set of potential targets can be determined completely and explicitly in advance, even prior to runtime. More specifically, we say that a call is a *sealed call* if it appears inside a *sealed Java package* [34], and has the property that all its targets are guaranteed to belong to the same package. Sealed calls can be identified by analyzing single Java packages, without relying on whole-program analysis. Our analysis can be viewed as an application of *almost whole program compilation* [7].

Identifying sealed calls and their potential targets facilitates aggressive inter-procedural intra-package optimizations. For example, if a sealed call has only one potential target, it can be inlined safely without any preconditions. In addition to the immediate application to devirtualization and inlining, our analysis clearly contributes to other inter-procedural analyses and optimizations such as immutability and escape analyses [6, 10, 13, 39], and object inlining [7, 14]. Our technique can be implemented efficiently and can support static compilers, bytecode-to-bytecode transformers, or dynamic compilers by using bytecode annotations [3, 36]. Since our

[1]Java is a trademark of Sun Microsystems, Inc.

analysis is applied to Java bytecode, it can be used for a wide variety of languages that produce such bytecode.

The results obtained by our technique indicate a significant potential for practical improvements in analyses and optimizations. In about half of the packages in the runtime library rt.jar (whose packages are known to be sealed except for two), 5–60% of the virtual calls (to non- `final` methods) are sealed calls according to our analysis. On average, our algorithm identifies nearly 39% of all virtual calls that appear in packages of SPECjvm98^{TM} [32] benchmarks and candidates as sealed calls (assuming packages are sealed), with half of the packages scoring between 50% and 95%. Almost 29% of all virtual calls invoked while running these benchmarks were identified as sealed calls. For many packages, all sealed calls are shown to have only one possible target, which makes them good candidates for direct inlining.

The rest of this paper is organized as follows. Section 2 provides a background and describes related work on devirtualization of calls in dynamic languages. Section 3 defines sealed calls, and Section 4 describes an algorithm for identifying sealed calls and their possible targets. Section 5 provides data on the extent of prevalence of sealed calls. Conclusions and directions for future work are presented in Section 6.

2. BACKGROUND

2.1 The Devirtualization Problem

The execution time of programs can be reduced considerably by using compiler optimizations which better exploit hardware resources [27]. Programs that consist of many short procedures require inter-procedural optimizations in order to obtain efficient optimized code. One of the most relevant inter-procedural optimization is inlining, which can improve performance by reducing the overhead of calls and by increasing opportunities for other optimizations. In order to perform inlining and other inter-procedural optimizations it is essential to identify the targets of calls.

Object-oriented programs promote the use of short methods to encapsulate functionality and as a means of modularity and abstraction. Such methods are usually virtual, a fact which complicates inter-procedural optimizations. Each time a call to a virtual method is executed, the declared method or one of several methods which override it is actually invoked, depending on the dynamic type of the receiver object. The ability to identify in advance the potential targets of a virtual call, known as *call devirtualization*, is crucial for optimizing object-oriented programs and has received much attention in recent years (see, for example, [1, 11, 12, 21, 20, 29, 30, 31]).

Determining the precise set of potential targets of virtual calls is related to the problem of object-oriented type analysis, which is known to be difficult [18]. A set of potential targets can be determined, at least conservatively, by performing *whole-program analysis* on the entire application (see [4, 12, 20, 29, 35, 36, 38]). One simple approximation of the targets of a call is the set of all methods that override the called method. This set can be constructed efficiently using *class hierarchy analysis* [11, 24, 31].

Various techniques have been developed in recent years to refine the set of possible targets of a virtual call, producing subsets of the set of all overriding methods. Liveness (*rapid type*) analysis [4, 31] was used to exempt *dead methods* (methods guaranteed never to be invoked during execution of the program) from being potential target candidates. Variable type analysis was used by the Sable research group [35, 38] to obtain more accurate results than rapid type analysis. A variety of flow-sensitive and flow-insensitive type analyses are described by Grove [21]. All these techniques including the basic “all-overridings” approximation rely on having all the classes that might participate at runtime, fixed and available at analysis time.

2.2 Dynamic Languages

In dynamic languages, the complete set of classes that will participate in the execution of an application may not be known in advance. Dynamically loaded classes may be encountered only at runtime, when they are referenced for the first time. Inter-class analysis such as call devirtualization may be invalidated when new unanticipated classes are loaded. In several languages such as Dylan [16], Java [19] and Trellis [28], the programmer can prohibit classes from having subclasses or methods from being overridden. This enables related calls to be devirtualized safely, without the risk of future invalidation, but lays a burden on the programmer and limits potential extension and reuse of the code. This paper deals with calls which are not explicitly restricted to a single target by the programer.

There are several approaches that enable inter-class analysis and optimizations for dynamic languages such as Java. One approach is to assume that all relevant classes and interfaces are supplied by the user in advance and revert to traditional static compilation techniques. Such an approach was adopted by the Jove^{TM} optimizing compiler [26] which analyzes and compiles entire Java applications. The compiler inside the TowerJ3.0^{TM} deployment environment [37] also relies on having all relevant classes available for its final step. The major drawback of such a whole-program optimization approach is that the requirement of having all relevant classes available in advance cannot always be satisfied.

A second approach is to assume that new classes may appear at runtime and prepare mechanisms for dealing with such unanticipated classes. The simplest of such mechanisms is a precondition that checks the actual type of the receiver object before executing the inlined code. The Jalapeño dynamic compiler uses such guards [8]. A similar but more robust *method test* precondition was recently suggested [13]. Preconditions are easily used by both static and dynamic compilers, but they prevent other optimizations from taking full advantage of the increased method size provided by inlining.

More sophisticated mechanisms for dealing with unanticipated classes detect invalidated optimizations at runtime and correct them using dynamic recompilation. The Java Hotspot^{TM} compiler [23] uses such a dynamic *deoptimization* mechanism. Such mechanisms can discover if a newly loaded class invalidates any existing inlinings and correct every such inlining immediately by recompiling the appropriate methods. A problem can occur if a method which needs to be recompiled is currently executing, a fact which greatly complicates the recompilation process. A special mechanism such as *on stack replacement* in the Self system [22] can solve this problem.

Detlefs and Agesen [13] recently identified a family of situations (called *preexistence*) which do not require on-stack replacement. In such situations currently-executing methods that contain invalidated inlinings are allowed to continue executing the original code until they exit. Only subsequent invocations of these methods are to execute the recompiled code, a fact which simplifies the recompilation process.

Ishizaki et al.[24] correct invalidated inlinings by replacing a single instruction (a direct call or the first inlined instruction) with a jump to the original virtual call. This saves recompilation time but increases the footprint since both the original call and the inlined code are present at all times, and it also limits the opportunities for other optimizations since the inlined code must be kept contiguous.

3. SEALED CALLS

Existing techniques for determining the potential targets of virtual calls either restrict dynamic class-loading (by relying on the user or by applying whole-program analysis), or inflict a runtime overhead. Our technique overcomes these drawbacks. The main obstacle that prevents static devirtualization is the possibility that a dynamically loaded class will override the called method and thereby provide a new target for the call. This obstacle can be overcome by making sure that no new subclass of the called class can be loaded, or that the called method cannot be overridden by new subclasses. In Java, one trivial way to ensure this is to declare the called class or the called method `final`, but this is not always desired or applicable.

Our technique addresses calls to methods that are not declared `final` (and whose class is not `final`). In such cases, additional features are needed to enable static devirtualization. Two key ingredients can be combined to achieve our goal: the first restricts the freedom to dynamically load new classes and the second restricts the ability to subclass and override the called class and method. The general idea is to ensure that (i) all the targets of a virtual call must belong to a certain subset of classes and that (ii) no new class can be added to this subset without violating the standard rules of the language. In our case these subsets correspond to Java packages which are *sealed*, as explained below. When dealing with a sealed package it is safe to assume that all the relevant classes (i.e., those that belong to the package) are available for analysis in advance.

There are several important differences between our technique and the utilization of the `final` modifier. The `final` modifier is part of the Java programming language, while *sealing* is external to the language definition. The `final` keyword disables further extension of a class or method, while *sealing* forces such extensions to belong to other packages. The `final` modifier can be used to resolve monomorphic calls to class methods, while *sealing* can devirtualize polymorphic calls to classes and interfaces. In general, it is not surprising that `final` reduces polymorphism, but it is quite unexpected that a security feature applied during class-loading has such an effect.

The usage of the `final` modifier leaves enough room for other devirtualization techniques. Specifically, our approach leads to a significant amount of additional safe devirtualization. Henceforth, the term *virtual call* denotes calls to virtual non-`final` methods.

3.1 Sealed Packages

In Java, every class belongs to one specific *package*. Each package can have, as members, several classes which are usually logically and functionally related. The classes of packages can be aggregated into one JAR (Java ARchive) file, together with additional information. Several JAR files and directories usually apply when a Java program is executed, as specified by the "class-path". During execution, when a class is referenced for the first time, the Java virtual machine (JVM) searches the applicable JAR files and directories for the desired class.

Starting with version 1.2 of the Java Software Development Kit (SDK 1.2.2 [34]), Java packages that reside inside JAR files can be *sealed*. If a package is sealed, all classes defined in that package must originate from the same JAR file, otherwise an exception is thrown ("java.lang.SecurityException").

When a package is sealed inside a JAR file we are certain that every application will either load all the classes that belong to this package from this JAR file, or not load any. It is not unreasonable to expect that many packages will be sealed. All standard core Java packages in the Java 2 runtime library rt.jar are sealed except for two [34]. The original motivation to seal packages was to help enforce security and consistency within a version. Grouping together sets of class files is also very important for inter-class analysis. To ensure the persistence of such analysis it is important to detect changes made to the package (e.g., modification and removal of existing classes, insertion of new classes, unsealing the package). This is provided for in the form of JAR-file signing and versioning, which are also available in SDK 1.2.2.

Currently Java provides sealing only of individual packages. It is possible to seal an entire JAR file, thereby sealing all its packages (unless stated otherwise). However, nothing binds two sealed packages together, even if they belong to the same JAR file. An application can load all classes of one sealed package p_1 from a certain JAR file and load none from another sealed package p_2, if p_2 is available at a location earlier in the class-path. Providing the ability to seal several packages together, may increase the potential for safe devirtualization in the future.

Sealing a package provides the first ingredient of identifying sealed calls, by defining a set of classes to which no additional class can be included. The other ingredient, proving that all the targets of a virtual call must belong to such a subset, is presented in the next subsection.

3.2 The Default (package) Modifier

The access restrictions imposed by the default (package) access modifier of classes, interfaces and methods can be used to prove that all the targets of a virtual call must belong to one specific package. A call to a virtual method m of class $c1$ (denoted by $c1::m$) can only target methods that override method m. Method $c2::m$ can override method $c1::m$ only if $c2$ extends $c1$ and has access to $c1::m$. The access and extension can be either *direct* or *indirect*. There is one exception to the above rule described later, where $c2::m$ overrides $c1::m$ even though $c2$ does not extend $c1$.

Java has several class and method modifiers, which are relevant to our analysis. In this document, the term "class" refers to both classes and interfaces, unless stated otherwise. A class can be declared `public`, in which case it can be directly extended by classes from any package (provided it is not declared `final`). Classes which are not declared `public` can be directly extended only by classes of the same package, and will be referred to as *packaged* classes. The methods of a class can be declared `public`, `protected`, or `private`; the methods of an interface are implicitly declared `public`. Public and protected methods can be accessed from outside the package. Private methods can be accessed only from within the same class and cannot be overridden (they are implicitly `final`). Finally, a method that is not declared `public`, `protected`, or `private`, is directly accessible only from within its package, and will be referred to as a *packaged* method.

Packaged classes and methods can be directly extended and accessed only from within their package, but they might be extended and accessed *indirectly* from other packages through transitivity. For example, a packaged class $c1$ can have a direct `public` subclass $c2$ within the same package. It is now possible for a class $c3$ from a different package to directly extend class $c2$, thereby indirectly extending class $c1$. Similar scenarios enable indirect access to packaged methods from outside their package. The potential of such indirect extension and access enables the targets of a call to a packaged method to belong to different packages.

There is one special case in Java where a method of a class $c2::m$ can override a method of an interface $c1::m$ although $c2$ does not implement $c1$ (see [19, pages 166–167]). This happens when another "combining" class ($c3$) implements interface $c1$, extends class $c2$, and inherits method $c2::m$ as an overriding implementation for $c1::m$. If $c1$ is a packaged interface and $c2$ belongs to a different package, we obtain another case of inter-package overriding.

It is thus possible to find all the methods that override method $c1::m$ by scanning the subclasses of $c1$, and occasionally examining superclasses of such subclasses. The algorithm presented in the next section identifies sealed calls by checking if the called method is accessible and if its class can be extended, either directly or indirectly from outside their package.

4. IDENTIFYING SEALED CALLS

This section presents an algorithm for identifying sealed calls, and for providing a complete (conservative) set of targets for each sealed call. Aggressive inter-procedural analysis and optimization can be applied safely to such calls.

Calls are categorized by the algorithm as being sealed calls based on information related only to the called method. Thus, the algorithm actually identifies *sealed methods* — methods that can be called only from within the same package, where each such call is guaranteed to be a sealed call. It is possible to mark non-overridden sealed methods `final`, or suggest such declarations to the programmer, as proposed by Jax [36].

A detailed description of the algorithm is given in Subsection 4.1. Our algorithm is based on a class hierarchy graph of a single sealed package, without taking into account any out-of-package information, as explained in Subsection 4.2.

4.1 The Algorithm

Consider a call to method m of class or interface c, denoted by $c::m$. The following two steps determine whether or not the call is a sealed call. First, the classes and interfaces which belong to the package of c are analyzed and their hierarchical inheritance relationships are recorded in the form of a *Class Hierarchy Graph* (CHG). Next, a standard search for all methods overriding method $c::m$ is performed within the package (based on the CHG) to determine if method $c::m$ can be overridden by methods from other packages. The call is a sealed call if and only if we have verified that all overriding methods of $c::m$ must be confined to c's package.

The subclasses of c that can potentially override method m or inherit such an overriding implementation from a superclass are traversed along this search. These classes are the subclasses d that extend class c (or implement interface c) directly or indirectly, with the exception that if a class declares method m as `final` all its subclasses are exempted. The search is aborted if a `public` non-`final` class d is found that redeclares method m as `public` or `protected` and not `final`, or inherits such a declaration from a superclass. In this case, the original call to $c::m$ is not a sealed call because class d can be extended by a class e from another package and e will be able to override $c::m$. If no such class d is found, the call to $c::m$ is a sealed call.

Figure 1 presents an implementation for such an algorithm that determines whether a call to $c::m$ is a sealed call or not. Gathering the targets of a sealed call is a simple part of our algorithm, and has been omitted for clarity. The two `methodIsSealed` functions handle the cases where c is a class or an interface. The function `methodIsExposed` recursively scans the inheritance tree rooted at the given class, and determines if the given method can be overridden from a different package. Code related to the `checkRoots` flag is explained in the next subsection.

4.2 Package Based Class Hierarchy Graph

Our underlying assumption is that the classes of a single sealed package are available for analysis, and nothing can be assumed about classes of other packages. Therefore the analysis for building the CHG of a package must be based only on information internal to the package. However, it may be necessary to examine classes of other packages in order to detect certain inheritance relations in the CHG. For instance, suppose a class of package p extends a class from another package, which in turn extends a class of package p. This way the former class indirectly extends the latter

Figure 1: Algorithm for Identifying Sealed Methods

```
boolean methodIsSealed(class $c$, method $m$) {

  // 1. Check if $c$ inherits $m$ from another package
  class $super \leftarrow c$
  while $super$ does not declare $m$ {
    $super \leftarrow$ the super class of $super$
    if $super$ does not belong to analyzed package
      return false
  }

  // 2. Check if $c$ inherits $m$ as final
  if $super$ declares $m$ final
    return true

  // 3. Check recursively if $c$ or a subclass of $c$ exposes $m$
  global boolean $checkRoots \leftarrow$ false
  boolean $mIsPublicOrProtected \leftarrow$ ($super$ declares $m$ public
                                              or protected)
  if methodIsExposed($c$, $m$, $mIsPublicOrProtected$)
    return false

  // 4. If needed check all classes recursively starting from roots
  if $checkRoots$
    foreach class $root$ whose superclass is not in package do
      if methodIsExposed($root$, $m$, false)
        return false

  return true
}


boolean methodIsSealed(interface $i$, method $m$) {

  if $i$ is a public interface    // $m$ is implicitly a public method
    return false

  foreach class (interface) $c$ implementing (extending) $i$ do
    if not methodIsSealed($c$, $m$)
      return false

  return true
}


boolean methodIsExposed(class $c$, method $m$,
                        boolean $mIsPublicOrProtected$) {

  if $c$ is a final class or declares $m$ final
    return false

  if $c$ declares $m$
    $mIsPublicOrProtected \leftarrow$ ($c$ declares $m$ public or protected)

  if $c$ is a public class $\wedge$ $mIsPublicOrProtected$
    return true

  foreach $subclass$ directly extending $c$ do
    if methodIsExposed($subclass$, $m$, $mIsPublicOrProtected$)
      return true

  if $c$ is a public class    // and $m$ is a packaged method
    $checkRoots \leftarrow$ true

  return false
}
```

class and both belong to package p, even though no inheritance relationship is visible by looking only at classes inside package p. One must examine the intermediate class which belongs to the other package in order to completely determine the inheritance relationship. A conservative way to cope with this deficiency is to assume that any two classes in the package that can possibly extend one another through cross-package inheritance, do so.

Here is a scenario involving cross-package inheritance which is relevant to identifying sealed calls. Suppose a package p contains two `public` non-`final` classes $c1$ and $c3$, with no inheritance relations visible within p. Suppose $c1$ declares a *packaged* method m, and $c3$ declares a similar method m as `public` or `protected`. Now it is possible for a class $c2$ from a package other than p to extend $c1$ and be extended by $c3$. This enables $c3::m$ to "smuggle" $c1::m$ out of package p, since it overrides $c1::m$ and can be overridden from outside p. This is however the only relevant scenario: $c1$ must be `public` in order to be extensible by a class $c2$ from outside p and $c1::m$ must originally be *packaged* for otherwise it can be overridden directly by methods of other packages. Being confined to package p, our analysis conservatively assumes that such a class $c2$ always exists. Methods of interfaces cannot participate in such scenarios since they are all `public`.

The algorithms presented in Figure 1 cope with potential cross-package inheritance in the conservative manner explained above. Classes whose direct superclasses do not belong to the analyzed package are identified as *root classes*. The search for all methods overriding $c::m$ signals the potential for cross-package extension when it encounters a `public` class $c1$ which declares m as *packaged* and non-`final` or inherits such a declaration. If such a signal is raised, all root classes (except for the root which is a superclass of c) are considered as indirect extensions ($c3$) of c (see code related to the `checkRoots` flag in Figure 1).

Regarding the complexity of our algorithm, first note that the CHG of a package can be constructed in time $O(N+M)$ where N is the number of classes and interfaces, and M is the number of inheritance edges. Given the CHG of the package, our algorithm visits each class and interface once at the most (usually only a "shallow" inheritance tree is visited), taking a constant time per visit, in order to identify a sealed method. Overall, per method, the execution time of the algorithm is at most linear in the size of the CHG of the analyzed package.

5. EXPERIMENTS

The algorithm for identifying sealed calls was implemented and tested on several benchmarks. Calls that are identified as sealed calls can be devirtualized safely, for instance by converting `invokevirtual` to `invokespecial` as suggested by Jax [36]. Sealed calls to methods of interfaces may benefit from a similar strength-reduction optimization, by converting `invokeinterface` to `invokevirtual` as proposed by the Sable research group [17], however only few such candidates were found in our experiments. The potential performance improvements of eliminating virtual function calls in C++ programs have been studied by several researchers [9, 2, 15, 30]. In addition to reducing the overhead of dynamic dispatch by devirtualization, monomorphic sealed calls can be safely inlined without a guard. The potential benefit of such direct inlining in Java is reported by Agesen and Detlefs [13, Table 6].

In this section we present experimental results showing that a significant percentage of the virtual calls that reside inside certain library and application packages are identified as sealed calls by our algorithm. An interesting observation is that a very high percentage of the calls that were found to be sealed have exactly one possible target, and are therefore good candidates for safe direct inlining. These results indicate the strong potential of using sealed calls to enhance analysis and optimization of Java programs.

Subsection 5.1 describes the benchmarks that were used. Subsection 5.2 reports the number of sealed call sites found in the benchmark packages, and Subsection 5.3 presents the number of times sealed calls in application packages were executed while running the benchmarks, according to profiling data. Remarks regarding the experiments appear in Subsection 5.4.

5.1 Benchmarks

Table 1 describes the Java programs used in the experiments. In each benchmark we considered all the packages that contain a significant amount (100) of virtual calls to non-`final` methods (for Jigsaw the threshold was 300). Only such calls are relevant to devirtualization analysis.

The runtime library rt.jar from Java 2 JDK release 1.2 was chosen because all its packages except for two are known to be sealed, and because it is fairly large, diverse, and highly reusable. Optimization improvements made to rt.jar have great potential impact on the performance of other Java programs.

Jigsaw [25] and pBob [5] are two large, multi-packaged server applications. Jigsaw is an object-oriented web server of W3C implemented in Java. Portable business object benchmark (pBOB) is a kernel of business logic inspired by the TPC-C benchmark specification[2]. SPECjvm98 [32] candidates and benchmarks were chosen to represent client-oriented programs. The analysis of application benchmarks assumes that all packages are sealed. Note that this is a valid assumption which does not cause security exceptions for these applications.

5.2 Static Counts

Tables 2, 3, and 4 present static measurements regarding the calls that reside inside rt.jar and the application packages tested. For each package, we counted the number of virtual call sites to non-`final` methods that appear in the package (column V_s). We also counted how many of these virtual calls were intra-package calls, where both caller and the original callee belong to the same package; only such calls are candidates to be sealed calls (column P_s). The percentage of virtual calls that were identified as sealed, and that were also identified as monomorphic (according to our algorithm) are presented next (columns S_s, M_s respectively). The tables are sorted according to the percentage of sealed calls (column S_s), in descending order.

[2]In accordance with the TPC's fair use policy we note that pBOB deviates from the TPC-C specification and is not comparable to any official TPC result.

Table 1: Description of benchmark programs

Type	Benchmark	V_t	Description
Library	rt.jar	52909	Java 2 JDK release 1.2
Server	Jigsaw	8322	W3C's web server, version 2.0.3
	pBob	1390	Transaction processing benchmark
SPECjvm98 Benchmarks	javac	2177	Java bytecode compiler
	jess	828	Java expert shell system
	jack	735	Parser generator
	check	238	Tests JVM features
	db	115	Search and modify a database
	mpegaudio	115	Decompress audio files
SPECjvm98 Candidates	nih	1684	Image manipulation
	raytrace	869	Graphics raytracer
	mpeg	260	MPEG video decoding
	si	208	Interpreter for a simple language
	cst	193	java/util class exerciser
	tmix	155	Dining philosophers
	richards	114	Threads running five OS simulator versions
	deltablue	105	Deltablue algorithm

V_t Total number of Virtual calls to non-`final` methods

Table 2: Static counts for "best" rt.jar packages

Package	V_s	P_s	S_s	M_s
sun/audio	324	65.7	**59.3**	55.2
sun/awt/Albert	1202	89.5	**45.5**	44.8
javax/swing/text/rtf	592	37.8	**37.8**	31.2
java/awt/datatransfer	114	50.0	**31.6**	31.6
sun/jdbc/odbc	1076	89.9	**27.0**	27.0
javax/swing/text/html/parser	377	73.2	**26.3**	26.3
javax/swing/tree	1113	73.9	**25.9**	25.9
sun/security/tools	1267	22.7	**22.7**	22.7
sun/applet	688	39.5	**20.5**	20.5

V_s	Number of Virtual calls
P_s	Percentage of calls in V_s to methods of same Package
S_s	Percentage of calls in V_s identified as Sealed
M_s	Percentage of sealed and Monomorphic calls in V_s

There are 64 packages in rt.jar with at least 100 virtual calls to non-`final` methods, of which all but one are known to be sealed. On average, 40% of the virtual calls in these packages are intra-package calls. More than half of these packages contain at least 5% sealed calls, where the total average of sealed calls is 8.5%. Furthermore, on average about 7.9% of the virtual calls are sealed and monomorphic according to our algorithm. Table 2 shows 9 of the packages that contain the highest percentage of sealed calls (at least 20%).

Regarding tables 3 and 4, roughly half of all the virtual calls that were analyzed are intra-package calls. Thus there is a large potential for sealing many virtual calls. In general, the results obtained by our algorithm show that there is a large variance in the number of sealed calls per package — in some packages nearly all (94.8%) virtual calls are sealed, whereas in other packages less than 1% of the virtual calls are sealed calls. For example, in package *richards/dai* the entire class hierarchy is packaged, except for the main

Table 3: Static counts in pBob and Jigsaw packages

Package	V_s	P_s	S_s	M_s
pBob (prefix = com/ibm/sf/)				
BOB/infra/Collections	138	97.1	**15.2**	15.2
BOB	1129	45.9	**13.8**	13.6
BOB/infra/Factory	123	77.2	**0.0**	0.0
Jigsaw (prefix = org/w3c/)				
cvs	393	48.1	**28.0**	27.7
www/http	735	77.0	**14.3**	14.3
tools/resources/store	362	31.5	**12.2**	12.2
www/protocol/http	407	45.0	**11.3**	10.8
jigadm/editors	1491	20.1	**8.5**	8.5
tools/widgets	491	25.9	**7.1**	6.7
jigsaw/frames	950	25.1	**3.7**	3.7
jigsaw/servlet	724	29.1	**3.2**	3.2
jigsaw/filters	430	12.3	**2.8**	2.8
jigsaw/admin	452	40.0	**2.4**	2.4
www/protocol/http/cache	502	36.9	**1.2**	1.2
jigsaw/http	642	33.2	**0.9**	0.9
tools/resources	743	78.6	**0.4**	0.4
Average		45.2	**7.8**	7.7

Table 4: Static counts in benchmark packages

Package	V_s	P_s	S_s	M_s
SPECjvm98 Benchmarks				
mpegaudio	115	65.2	**65.2**	49.6
check	238	53.8	**50.4**	50.0
jess/jess	828	84.7	**36.0**	36.0
javac	2177	77.4	**10.8**	6.2
jack	735	30.1	**9.4**	9.4
db	115	24.3	**1.7**	1.7
SPECjvm98 Candidates				
richards/dai	114	95.6	**94.7**	93.9
deltablue	105	86.7	**85.7**	50.5
tmix	155	62.6	**62.6**	38.7
si	208	64.4	**58.2**	58.2
nih	1684	57.1	**53.9**	43.2
cst	193	68.9	**12.4**	12.4
mpeg	260	74.6	**1.5**	1.5
raytrace	869	98.2	**0.0**	0.0
Average		67.4	**38.8**	32.2

(Richards) class which is `public`, resulting in a high percentage of sealed calls. On the other extreme, in packages *raytrace* and *com/ibm/sf/BOB/infra/Factory* all classes and methods are public, leaving no potential for sealed calls. In package *org/w3c/tools/resources*, all 56 classes and 407 methods are public except for two classes and two methods which are packaged, resulting in very few sealed calls.

For some rt.jar and SPECjvm98 packages, almost every virtual intra-package call is identified by our algorithm as a sealed call, leaving little prospect for more powerful algorithms. Overall, for most packages analyzed in both rt.jar and application benchmarks, nearly all sealed calls are shown to be monomorphic, a fact which makes them good candidates for aggressive optimizations such as direct inlining.

5.3 Dynamic Counts

Table 5 presents dynamic measurements regarding calls that were executed while running the SPECjvm98 candidates and benchmarks. These programs were executed with size '10' to produce dynamic profiles. Therefore, the results do not follow the official SPEC rules.

Table 5: Dynamic counts in benchmark packages

Benchmark	V_d	P_d	S_d	M_d
SPECjvm98 Benchmarks				
mpegaudio	26650	99.8	**6.2**	6.1
check	68	29.8	**1.2**	1.2
jess	20789	93.7	**5.9**	5.9
javac	3416	62.0	**14.4**	9.6
jack	3962	39.2	**17.5**	17.5
db	326	22.5	**0.0**	0.0
SPECjvm98 Candidates				
richards	88031	100.0	**100.0**	88.2
deltablue	46681	100.0	**100.0**	71.1
tmix	71834	99.1	**98.8**	76.0
si	10973	47.3	**33.0**	33.0
nih	5	18.0	**3.5**	3.5
cst	2124	21.6	**21.1**	21.1
mpeg	21987	67.0	**0.0**	0.0
raytrace	60051	100.0	**0.0**	0.0
Average		64.3	**28.7**	23.8

V_d	Number of invoked Virtual calls (in thousands)
P_d	Percentage of calls in V_d to methods of same Package
S_d	Percentage of calls in V_d identified as Sealed
M_d	Percentage of sealed and Monomorphic calls in V_d

Dynamic measurements are not presented for pBob and Jigsaw, due to their highly configurable and interactive nature. We considered all calls that originated from application packages, since they usually accounted for most of the executed calls. For each benchmark we counted how many virtual calls to non-`final` methods were executed (column V_d). Columns P_d, S_d and M_d are analogous to the static measurements (P_s, S_s, M_s) in Tables 2, 3 and 4.

Of all virtual calls executed, a high percentage (about 64%) were intra-package calls, and more than 28% were found to be sealed calls by our algorithm. On the average, while running a benchmark, our algorithm identified as sealed calls nearly 45% of the intra-package calls that are executed, with some benchmarks reaching close to 100%. As with the static counts (see Section 5.2), for many benchmarks all sealed calls are also shown to be monomorphic.

The variance in the dynamic results is huge. For three benchmarks (richards, deltablue and tmix) almost all (more than 98.8%) virtual calls are identified as sealed. On the other extreme, in four benchmarks (check, db, mpeg and raytrace) a very small percentage (less than 1.2%) of the virtual calls were identified as sealed.

Several benchmarks (check, nih, jack and db) executed a significant number of calls originating from JDK packages. In these cases, about 6.6% of the virtual calls from the JDK were identified as sealed and monomorphic.

5.4 Statistical Remarks

The profiling data used (the standard `java -prof` tool) reports the number of times a certain method calls another method[3]. If one method contains several call-sites, each potentially targeted at the same target method, then the distribution of the method-to-method frequency among these call-sites is not known to us. However, there was no prob-

[3]Note that for calls to or from native methods, the callee or caller were reported as unknown. Such cases were very scarce and had no significant effect on the statistics.

lem to determine the accurate invocation count for almost all sealed calls that appear in the benchmarks.

In order to build the CHG of a package we analyze the bytecodes (the `invokevirtual` and `invokeinterface` bytecodes in particular) to locate virtual call sites, and use the callee-class annotated at each call-site. There may be a difference between the callee class that appears in the Java source code, and the callee class annotated in the bytecode. This happens when the original callee class does not contain a declaration of the called method. In such cases, the annotated callee class is a superclass of the original callee class that contains the required declaration. It is conservative to use the annotated callee class (see [13, page 264] for further details).

6. CONCLUDING REMARKS

Using the default access permission of *packaged* classes, interfaces and methods together with the ability to *seal* Java packages, we are able to determine complete sets of targets for certain calls. In this paper we have proposed a type of Class-Hierarchy Analysis that identifies sealed calls, which for some cases identifies nearly all intra-package calls as sealed calls. In some SPECjvm98 packages, almost all intra-package calls are identified by our algorithm as sealed, and a significant number of sealed calls are invoked while running these benchmarks. In the widely used Java 2 rt.jar library, about 10% of the packages contain a significant percentage of sealed calls (20–60%), and in more than half of the packages, at least 5% of all virtual calls are sealed calls. For many packages all sealed calls are also shown to be monomorphic.

Our analysis supports inter-procedural analyses such as immutability and escape analysis, and enables aggressive optimizations such as direct inlining. It is efficient and static in nature — it can support both dynamic compilers by encoding the results as bytecode annotations, and static pre-runtime compilers or bytecode-to-bytecode transformers. The entire analysis and optimizations can be validated instantly at runtime when the package is first loaded from its JAR file by verifying the seal, version, and signature. There is no need for complex dependence models and mechanisms that check multiple files and timestamps, or for sophisticated recompilation techniques.

Our algorithm actually identifies *sealed methods* — methods that can be called only from within the same package, where each such call is guaranteed to be a sealed call. This analysis may be enhanced in several possible ways. Additional calls might be sealed by considering the specific context of each individual call site: data-flow analysis can be used to better determine the possible types of the receiver object. Recently, Sreedhar, Burke and Choi [33] presented a framework which addresses similar issues using dataflow analysis. Such techniques are significantly more complex than our proposed CHA-type algorithm. Moreover our experiments show that in many cases our algorithm is able to identify most of the calls that can potentially be sealed, leaving limited prospects for more powerful tools.

Another way to try and enhance our analysis is to use liveness information (as in Rapid-Type Analysis [4]). For instance, packaged classes or classes which do not have `public` or `protected` constructors can be considered live only if they are instantiated within the package. There is however little hope of sealing additional calls this way, since a `public` class, that prevents a call from being sealed according to our algorithm, must be considered live (if it has a `public` or `protected` constructor). Liveness information can potentially reduce the number of targets a sealed call is known to have. However, our algorithm shows that for many SPECjvm98 and rt.jar packages, all sealed calls are monomorphic.

A related problem is to try and identify all possible callers of a given method, which is also very important for inter-procedural optimizations (e.g., inter-procedural redundancy elimination, dead method removal [36]). One condition that is sufficient and efficient for this purpose is to check if the method is a packaged method in a sealed and signed package. Sealed methods also belong to this category, since they can be called only from within their package (and from native code or via reflection).

7. ACKNOWLEDGEMENTS

The authors would like to thank Joseph (Yossi) Gil for his many constructive suggestions, Bilha Mendelson, Sara Porat and Daniel Hicks for helpful discussions, Chani Sacharen for useful editorial remarks, and the anonymous reviewers of OOPSLA and ECOOP for their feedback.

8. REFERENCES

[1] O. Agesen and U. Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 91–107, Oct 1995.

[2] G. Aigner and U. Hölzle. Eliminating virtual function calls in c++ programs. In *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP)*, pages 142–166, July 1996.

[3] A. Azevedo, A. Nicolau, and J. Hummel. Java annotation-aware just-in-time (ajit) compilation system. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 142–151, June 1999.

[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 324–341, Oct 1996.

[5] S. J. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, S. Munroe, R. Arora, , and R. Dimpsey. Java server benchmarks. *IBM Systems Journal*, 39(1):21–56, 2000.

[6] B. Blanchet. Escape analysis for object oriented languages, application to java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 20–34, Nov 1999.

[7] Z. Budimlić and K. Kennedy. Prospects for scientific computing in polymorphic, object-oriented style. In

Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, March 1999.

[8] M. G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 129–141, June 1999.

[9] B. Calder and D. Grunwald. Reducing indirect function call overhead in c++ programs. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 397–408, Jan 1994.

[10] J. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 1–19, Nov 1999.

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

[12] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 222–236, Jan 1998.

[13] D. Detlefs and O. Agesen. Inlining of virtual methods. In R. Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, pages 258–278. Springer-Verlag, June 1999.

[14] J. Dolby. Automatic inline allocation of objects. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 7–16, May 1997.

[15] K. Driesen and U. Hölzle. The direct cost of virtual function calls in c++. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 306–323, Oct 1996.

[16] N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *The Dylan Programming Book.* Addison-Wesley, 1997.

[17] E. Gagnon and L. Hendren. Intra-procedural inference of static types for java bytecode. Technical Report 1999-1, Sable Research Group, McGill University, March 1999.

[18] J. Gil and A. Itai. The complexity of type analysis of object oriented programs. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, pages 601–634, July 1998.

[19] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[20] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 108–124, Oct 1997.

[21] D. P. Grove. *Effective Interprocedural Optimization of Object-Oriented Languages.* PhD thesis, University of Washington, 1998.

[22] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, June 1992.

[23] The java hotspot performance engine architecture. Available at http://-www.javasoft.com/products/hotspot/whitepaper.html, April 1999.

[24] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 119–128, June 1999.

[25] Jigsaw - the w3c's web server. Available at http://www.w3c.org/Jigsaw.

[26] Jove super optimizing deployment environment for java. Available at http://www.instantiations.com/jove/jovereport.htm.

[27] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[28] P. D. O'brien, D. C. Halbert, and M. F. Kilian. The trellis programming environment. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 91–102, Oct 1987.

[29] H. D. Pande and B. G. Ryder. Static type determination for c++. In *Proceedings of the Sixth Usenix C++ Technical Conference*, pages 85–97, April 1994.

[30] S. Porat, D. Bernstein, Y. Fedorov, J. Rodrigue, and E. Yahav. Compiler optimization of c++ virtual function calls. In *Proceedings of the Second Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 3–14, Jun 1996.

[31] S. Porat, B. Mendelson, and I. Shapira. Sharpening global static analysis to cope with java. In *Proceedings of CASCON '98 Conference*, pages 303–316, Nov 1998.

[32] Spec jvm98 benchmarks. Available at http://www.spec.org/osg/jvm98, August 1998.

[33] V. C. Sreedhar, M. Burke, and J. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of the ACM*

SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI), pages 196–207, June 2000.

[34] Sun Microsystems. *Java 2 Software Development Kit version 1.2.2*, July 1999. Available at http://java.sun.com/products/jdk/1.2/, See there docs/guide/extensions/spec.html#sealing.

[35] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Valle-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. Technical Report 1999-4, Sable Research Group, McGill University, Nov 1999.

[36] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 292–305, Nov 1999.

[37] Towerj3 - a new generation native java compiler and runtime environment. Available at http://www.towerj.com/products/-whitepapergnj.shtml and also whitepapers3.shtml.

[38] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of CASCON '99 Conference*, Nov 1999.

[39] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 187–206, Nov 1999.