Extracting Java Library Subsets for Deployment on Embedded Systems

Derek Rayside
Kostas Kontogiannis

ABSTRACT
Embedded systems provide means for enhancing the functionality delivered by small-sized electronic devices such as hand-held computers and cellular phones. Java is a programming language which incorporates a number of features that are useful for developing such embedded systems. However, the size and the complexity of the Java language and its libraries have slowed its adoption for embedded systems, due to the processing power and storage space limitations. A common approach to address storage space limitations is for the vendor to offer special versions of the libraries with reduced functionality and size to meet the constraints of embedded systems. However, such an approach will severely limit the type of applications that can be deployed. This paper presents a technique that is used for selecting, on an as needed basis, the subset of library entities that is exactly required for a given Java application to run. This subset can then be down-loaded to the device for execution on an as needed basis. The advantage of this approach is that the developer can use arbitrary libraries, instead of being restricted to those which have been adapted for embedded systems by the vendors. A prototype system, that builds library subsets on per application basis, has been built and tested on several mid-size Java applications with encouraging results.

Keywords Java, embedded systems, library extraction, application extraction, static dependency graph, call graph construction

1 Introduction

Embedded systems are now an important part of modern programming activities, and will by all estimations become more so in the next few years. It has been estimated that the market of embedded PC, and "soft" PC devices will exceed US $1 billion by the year 2001 [Lee98]. Examples of such embedded systems include hand-held terminals, cellular phones with

*This is an expanded and revised version of a paper with the same title that appeared in *Proceedings of the 3rd IEEE Conference on Software Maintenance and Re-engineering* (CSMR'99, Amsterdam), edited by Paolo Nesi and Chris Verhoef [RK99].

Internet and World-Wide-Web capabilities, and other industrial or household control devices. However, due to limitations on size, processing power and, storage capabilities, embedded systems pose a number of additional requirements on software application development.

Java^{TM} was originally developed for consumer electronics devices. However, it has evolved over the recent years, more as a programming language for workstations and mainframes than a language for embedded systems. This is partly due to the features of the language which are inherently difficult to implement in embedded systems. These features include multithreading and, the overall size of the standard Java class libraries (JDK). For example, JDK 1.1 is about 10MB in size, and has grown significantly with the version 1.2 release [RKK98].

Current commercial efforts to re-target Java to embedded systems, such as Sun's Java 2 Micro Edition and KVM and IBM's VisualAge Micro Edition, usually define restricted subsets of the Java libraries for use on embedded systems. Defining restricted library subsets in this *a priori* fashion limits the functionality available to application developers and is inflexible.

In this paper an alternate solution to defining subsets of a class library is proposed. The main idea is to determine, on an as needed basis, which parts of a library are required for each given application, instead of *a priori* limiting the capabilities of the language by excluding whole portions of the library. Thus, the basic idea is to identify and extract the subset of the libraries that is needed for the specific application. The motivation is based on the observation that applications that use the entirety of a class library are rare. In most cases only a portion of the library is required for any given application.

The proposed solution is more flexible than one that limits the language features by excluding *a priori* a large number of standard Java libraries. The proposed technique allows application programmers to use the functionality they deem necessary from arbitrary libraries. The tool presented here extracts the code needed to run an application from a set of standard JDK libraries and is based on the analysis of the dependencies between a given application and its supporting libraries. The dependencies are revealed by parsing the Java byte-code and building an entity-relationship dependency graph. The relations are drawn from a Java domain model developed for this purpose, and are discussed later in the paper. Once a dependency graph has been built, the selection of the required library subset is based on traversing the graph and extracting only the nodes that correspond to a library entity that are accessible by the given application. Java byte-code (class files) are used to build the dependency graph which the analysis presented here is based on. Experimental results show that constructing and traversing the dependency graph are both fast and scalable operations.

Organization

This paper is organized in eight sections. The first section (this one) is the introduction, and the second section provides an overview of important features of the Java programming language as well as other related work. The most important related work is that on call graph construction for programs written in object-oriented languages.

Section three gives a detailed discussion of how we model dependencies in Java applications, including such distinctively object-oriented features as 'inheritance' and 'polymorphism' (and these are why the previous discussion on call graph construction is important). The fourth section identifies three different kinds of subsets according to the linking strategy used by the target virtual machine. Following that, the fifth section describes the process of extracting a subset from a library.

The seventh section presents some experimental results on medium size Java applications that make use of fairly large libraries. Section eight discusses some scenarios in which this technology may be useful, including embedded systems, distributed systems, native code compilers, and library re-factoring. Finally, section eight provides a summary of the work and concludes the paper.

2 Background and Related Work

2.1 Java Features

The Java language [GJS96] is designed to execute on the Java Virtual Machine [LY97], an abstract computer model that executes code contained in Java class files. These class files may be generated from source languages other than Java (such as Ada or Eiffel, etc.).

The Java Virtual Machine is a simple stack based computer model with no registers and a one-byte instruction set. There are less than two hundred opcodes currently defined in the machines instruction set [LY97].

The class file is similar to object files created by traditional compilers in the sense that it contains symbolic references to all external code, and is not bound with that code. In Java, binding occurs in the virtual machine at run-time.

Each class file contains information about the class, such as the compiler version that created it, the super-classes of the defined class, and all literal constants and references to external code. The class file also contains the fields and methods declared by the class; fields and methods declared in super-classes are stored in the file for the super-class, unless overridden.

Java code (source code and byte-code) is organized into "packages". A Java package is essentially a directory where the code is kept. Classes in the same package have "friend" status with each other. There is a stan-

dard naming convention for packages which ensures that code from different organizations will not have naming conflicts. The fully qualified name of a class is its proper name prepended with its package name (e.g. `java.lang.Object`).

The Java `import` statement has different semantics from the C `include` statement which are important to highlight in the context of this work. The `import` statement is a syntactic device to allow the programmer to reference classes in other packages by their proper names instead of fully qualified names: it does not imply "friend" status, nor does it affect the byte-code representation in any way.

Types in Java are organized into two main categories: *primitive* and *reference*. The primitive types include things such as `int` and `float`. The reference types are further sub-divided into *arrays* and *ClassOrInterfaces*, which in turn are divided into *classes* and *interfaces*. The Java notion of 'interface' is a special kind of abstract class that has slightly different typing rules associated with it than regular classes do (to allow for 'multiple inheritance'). This organization is depicted in figure 1. Programmer's may only define ClassOrInterface types, and so these are generally of primary interest. Understanding the subtleties of how the type system is organized is crucial for constructing accurate dependency graphs of Java programs.

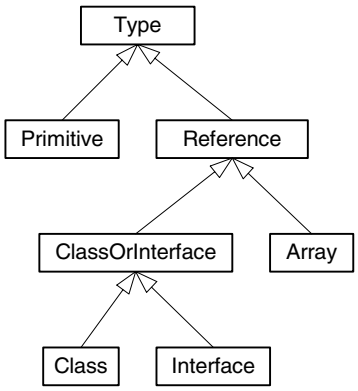

FIGURE 1. Organization of the Java Type System

Bruce Eckel's book *Thinking In Java* provides a clear and in-depth discussion of the Java programming language [Eck98] with comparisons to C and C++. Other articles on the Java virtual machine and class file format, can be found at the Java World website (`www.javaworld.com`) [Ven96a, Ven96b, Ven96c]. Moreover, both the Java Language Specification [GJS96] and the Java Virtual Machine Specification [LY97] provide a detailed view of Java's features.

2.2 Trends

Over the past year a growing demand for 'information appliances' such as 3Com's PalmPilot is observed. By some estimates, the market for these devices is expected to grow to US $ 4.2 billion by the end of 2002, when it will surpass the demand for home PCs [Lee98] [Ham98]. It can be expected that a large proportion of these will be networked in some way, and that they will also be running Java.

The spin-off and the demand for embedded systems using Java has also grown to a point that standards initiatives have been formed. In [Wor] a Compact HTML for Small Information Appliances has been proposed. On the same trend the Handheld Device Markup Language Specification standard has been proposed in [Wor97].

2.3 Bytecode Compression

Compressing byte code is an area of work that is complimentary to this, and that has received a fair amount of attention in recent years.

Ernst *et al* [EEF+97] draw a distinction between compressed code formats that can be directly interpreted and those that are intended primarily for efficient transmission ('wire' formats). Most work in Java bytecode compression has focused on wire formats, such as CLAZZ, JAZZ, and Pugh [Pug99]. CLAZZ [Cor94, HC98] explores the application of standard compression techniques, including ZIP, to particular sections of the Java class file. The results reported in [Cor94, BHV98] show that CLAZZ achieves compression that is up to 25% better than ZIP alone. JAZZ [BHV98] also uses standard compression techniques (in a slightly different fashion than CLAZZ), but focuses on JAR files instead of individual class files. Pugh has explored a number of compression techniques and is able to produce better results than JAZZ [Pug99]. These so-called 'wire' formats are acceptable for systems where the target machine has enough memory and processing power to decompress the code. However, this is often not the case for embedded systems with limited resources.

There has also been some work on directly interpretable formats (i.e. that do not need decompression) targeted specifically at embedded systems. Rayside *et al* [RMH99] propose a new structure for the class file that reduces the overall size by 25% and does not require decompression. Their study focuses on the constant pool (symbol table), but also examines more efficient encodings for the opcodes. There is another study [CSCM00] that focuses solely on compressing Java opcodes for embedded systems, but it disregards the constant pool.

Both of these approaches to byte-code compression can be complimentary to this work: the focus of this work is to identify and extract a subset of a library; the focus of the compression work is to represent that subset in a more compact fashion.

2.4 Bytecode Packaging

In the past, tools similar to the one that we have developed have been referred to as 'packagers' (especially within the Smalltalk community).

IBM Research has independently produced a tool named JAX (Java Application Extractor) [TLSS99] that is similar to ours, although their's performs some more exotic transformations such as class hierarchy specialization [TS97] and class hierarchy slicing [TCFR96], in addition to the subset extraction. The IBM VisualAge^{TM} Micro Edition integrated development environment also includes a 'SmartLinker' tool with similar functionality [IBMb].

Sun Microsystems produces two tools that are also related to our work: JavaFilter and JavaCodeCompact [Sun]. JavaFilter is closest to our work; JavaCodeCompact translates Java to C that is targetted for embedded systems. There has also been some collaborative research between Sun and Stanford in this area, focusing on dynamically typed languages [Age95].

The Secure Internet Programming Group at Princeton University [Sec98] has developed another tool called JavaFilter (not related to Sun's JavaFilter) which can be used for preventing applets that originate from a restricted site to be executed in a web-browser.

2.5 Call Graph's for Object-Oriented Languages

Devising cost-efficient algorithms for constructing an object-oriented program's call graph from a static analysis of the source code has been an active area of research for the last few years. This research is usually carried out in the context of compiler optimization, as many conventional optimizations such as in-lining cannot be performed without a call graph. A good discussion of the problem is given by Grove et al in [GDDC97]. In this section we will explain three of the most common approaches to solving this problem by constructing the call graph for `foo()` (all code examples are written in a Java-like syntax).

```
static void foo(Shape s) {
   s.draw();
}
```

The target of the invocation `s.draw()` depends on the actual type of the object that is bound to the formal parameter `s` each time `foo()` is executed. The *declared* type is `Shape`, but the *actual* type may be any sub-type of `Shape`. Furthermore, the actual type may 'inherit' the implementation from the *implementing* type, which may be any super-type of the actual type (including super-types of the declared type). Suppose `foo()` is written with reference to the following code:

```
abstract class Shape {
```

```
    abstract void draw();
}

class Circle extends Shape {
    void draw() { printf("circle");}
}

class Triangle extends Shape {
    void draw() { printf("triangle");}
}

class Rectangle extends Shape {
    void draw() { printf("rectangle");}
}

class Square extends Rectangle {}
```

Naive

A simple and inaccurate solution to this problem is to assume that the actual and implementing types are the same as the declared type. In the terms of this example, to assume that because `s` is declared to be a `Shape`, `s.draw()` always resolves to `Shape.draw()`. This is the result recorded in the byte-code by every Java compiler, and the one used in the static analysis of regular function calls in procedural languages.

The benefits of this solution are that it requires no extra analysis, is sufficient for the purposes of a non-optimizing compiler, and is very simple. However, its accuracy leaves something to be desired. In the given example `Shape.draw()` is `abstract`, and so the `s.draw()` invocation could not actually branch there: there is no code to branch to. This is a somewhat less than desirable solution for re-engineering tasks that require a reasonably accurate call graph, such as automatic clustering.

Class Hierarchy Analysis

Class Hierarchy Analysis (CHA) [DGC95, DMM96] is a *whole program analysis* that determines the actual and implementing types for each method invocation based on the type structure of the program. The whole program is not always available for analysis, due to features such as reflection and remote method invocation. However, for many practical reverse engineering tasks it is sufficient to analyze the code that is available for analysis (this may not be conservative enough for the purposes of compiler optimization).

In the above example, Class Hierarchy Analysis would construct *three* invocation arcs from `s.draw()`, to `Circle.draw()`, `Triangle.draw()`, and `Rectangle.draw()`. CHA would not produce an invocation arc to `Shape.draw()`, as it is `abstract`. This result is a significant improvement over the naive approach, which produced only one arc that could not possibly be traversed during execution.

Class Hierarchy Analysis is flow and context insensitive, and consequently is efficient in both time and space.

Rapid Type Analysis

Rapid Type Analysis (RTA) [Bac97, BS96] uses extra information from the program to eliminate spurious invocation arcs from the graph produced by CHA. This extra information is the set of instantiated (used) types: clearly `Triangle.draw()` can never be invoked if `Triangle` is never used in the program. This analysis is particularly effective when a program is only using a small portion of a large library, which is often the case in Java.

RTA begins at all program entry points and traverses over the program, building the call graph and the set of instantiated types as it goes. Consider the following `main()` as an entry point for the example program:

```
static void main(String[] args) {
   foo( new Square() );
}
```

Now it can be seen that the only sub-type of `Shape` instantiated in the program is `Square`, and so this must be the *actual* type of `s` in `foo()`. Note, however, that the *implementing* type is `Rectangle`: that is, `Square` 'inherits' the implementation of `draw()` from `Rectangle`.

Like CHA, RTA is flow and context insensitive, and consequently is efficient in both space and time. Again like CHA, RTA also requires the whole program for analysis. However, RTA is more sensitive to the use of reflection: the analyst must inform the algorithm if reflection is used to instantiate any class, otherwise the algorithm may incorrectly eliminate some arcs from the call graph. CHA is not as sensitive to the use of reflection, as long as the whole program is available for analysis.

Summary

In summary, for this example, the naive approach produces a single impossible arc, CHA produces three possible arcs, and RTA narrows these three down to a single target. Most studies have shown that RTA is a significant improvement over CHA, often resolving more than 80% of the polymorphic invocations to a single target [BS96, PMS98, Ray01]. Furthermore, RTA is an extremely fast analysis: in our experience it can usually be computed in a matter of seconds, even for very large programs. RTA does require the results of CHA, which can usually be computed in a minute or two. Both of these analyses combined take less than 10% of the time required to parse the program's bytecode.

RTA is implemented in the Jax [TLSS99] and Toad [PMS98] tools from IBM Research, both available on the IBM alphaWorks website [IBMa], as well as the front end of the IBM VisualAge C++ compiler [Kar98]. For this

study we used JPack, which is a research version of the `jport` tool, which was originally developed as a part of IBM VisualAge for Java, Enterprise Toolkit 390. We have used this research tool in a previous studies on impact analysis [RKK98] and automatic clustering [RRHK00].

Rapid Type Analysis is currently considered to be the best practical algorithm for call graph construction in object-oriented languages because it produces good results very inexpensively. There are a number of groups researching algorithms that produce better results than RTA for similar cost (e.g. [SHR+00, TP00]).

3 Program Representation (Domain Model)

In order to conduct our research and implement the tools discussed in this paper, we first had to be able to represent the source code at a higher level of abstraction. For this work we have chosen the Rigi Standard Form (RSF) which allows for entity-relationship tuples to be defined in a straightforward way. RSF tuples are of the form `<class defines method>` or `<method stores field>`. The tuples are emitted from a custom made byte-code parser. The relations used conform with a domain model.

We have constructed a domain model for Java byte-code in the object-oriented data model: it has entities, relations and attributes; some entities may be generalizations of other entities. Our domain model is illustrated by a number of UML class diagrams, which are described in overview here:

3.1 Entities and Attributes

Figure 2 shows the different types of entities, their attributes, and the sub-typing relations between them. There are categories of entities to represent types, fields, methods, packages, and polymorphic method invocations (discussed in more detail below).

Polymorphic Choice Vertices

We use a *polymorphic choice vertex* to represent polymorphic method invocations, similar to the one presented in [LJH96]. An important distinction between this work and the work in [LJH96] is that their slicing work is flow sensitive, while this work is flow in-sensitive (similar to [Bac97, BS96]).

This form of representation improves the efficiency of our analysis. Each polymorphic choice vertex may represent an arbitrarily large number of program statements that all have the same static invocation target. Instead of analyzing every single invocation we only have to analyze the polymorphic choice vertices. Note that this efficiency is gained at the expense of flow-sensitivity.

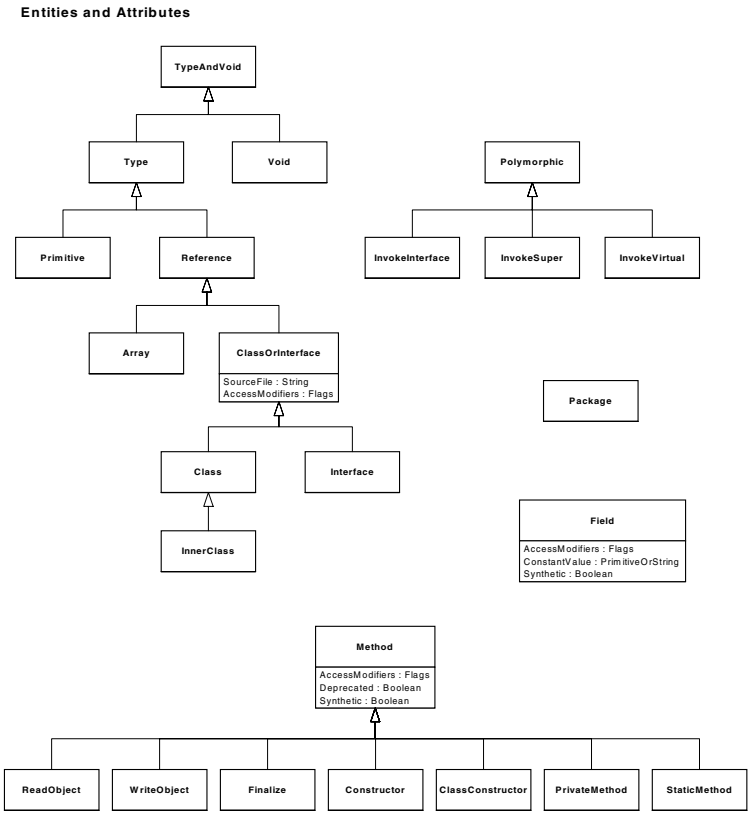

FIGURE 2. Entities and Attributes

Our domain model contains three types of polymorphic choice vertices, to represent the three types of dynamic dispatch that occur in Java: InvokeVirtual, InvokeInterface and InvokeSuper. InvokeVirtual is the normal case for most method invocations. Methods may also be invoked directly using the `invokestatic` opcode, which does not require a polymorphic choice vertex.

Structural and Representational Entities

Katz [Kat90] makes a distinction between *structural* and *representational* entities. Entities such as packages are structural: they contain other entities (classes or interfaces). Entities such as fields are representational: they represent the system under examination.

We consider that ClassOrInterfaces are *both* structural and representa-

tional entities, as they serve the dual roles of structuring the fields and methods in the program and representing part of the program in themselves. Given this representational facet, we consider that the interface of a ClassOrInterface is *not* the aggregate of the fields and methods it declares (since this would be considering ClassOrInterfaces as only structural entities). For our model, the interface of a ClassOrInterface is defined in terms of its type (Class or Interface), access modifiers, and so on. For example, certain arcs such as `instanceof` and `checkcast` may terminate at ClassOrInterface nodes, and in these cases the interface of the ClassOrInterface itself is important and the fields and methods it declares are not.

Signatures

Our domain model defines the signatures of various entities as follows:

ClassOrInterface the fully qualified name (i.e. including the Package name).

Field `Signature` of the `ClassOrInterface` that declares it appended with its proper name and the `Signature` of its type.

Method `Signature` of the `ClassOrInterface` that declares it appended with its proper name and the `Signatures` of its parameters and return type.

Note that is important that method signatures include parameter types due to overloading. Also note that it is important to include the return type in the method signature, even though the language specification does not allow overloading on return type, most VM implementations do and this is exploited by some experimental Java compilers that support generic types (and is likely to become an explicit part of the specification in the future).

3.2 Declaration Arcs

Figure 3 shows arcs that represent the declaration statements in the bytecode. Essentially, classes declare fields and various types of methods.

3.3 Containment Arcs

Figure 4 shows the 'containment' arcs, which are generally inverses of the declaration arcs. For example, a field is contained by the class that declares it. Having explicit inverses is important for the traversal algorithm: it enables associating different mapping functions with each direction.

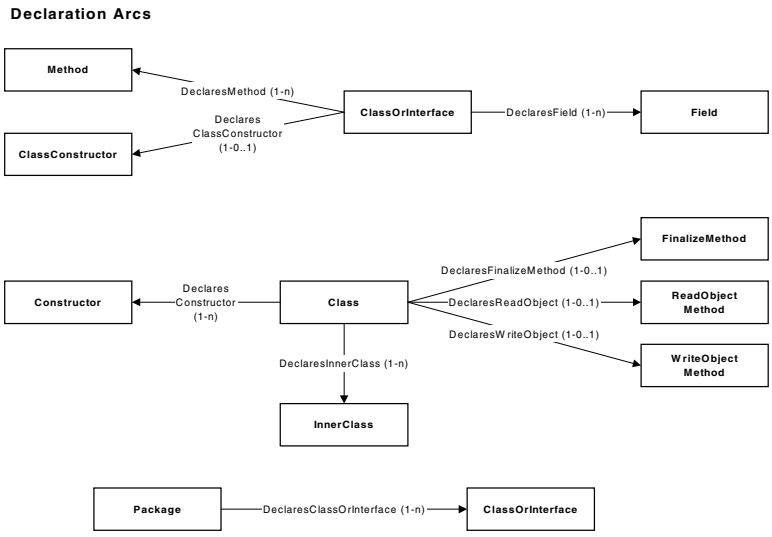

FIGURE 3. Declaration Arcs

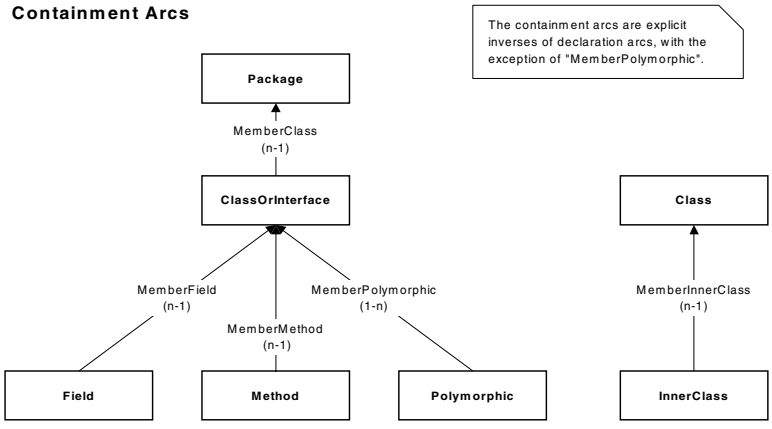

FIGURE 4. Containment Arcs

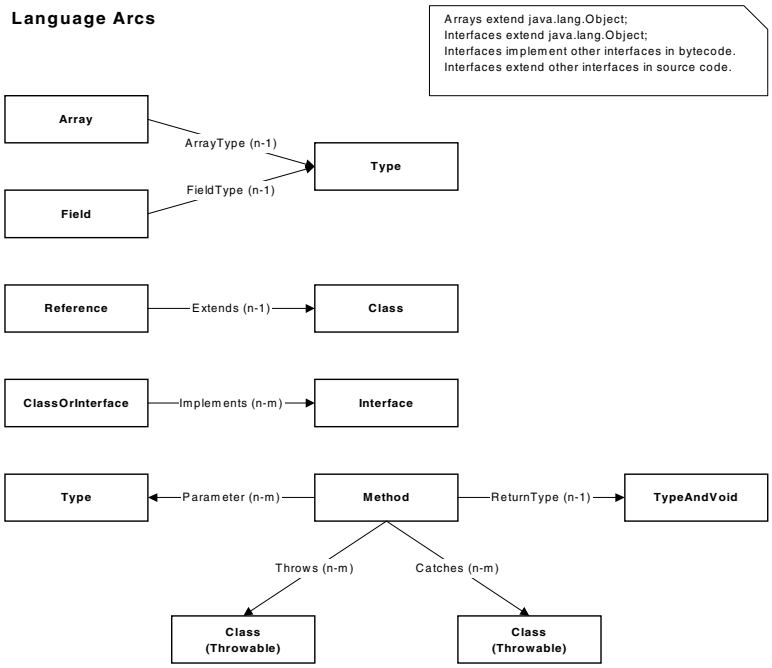

FIGURE 5. Language Arcs

3.4 Language Arcs

Figure 5 shows other structural arcs from the Java language that are neither declarative arcs nor containment arcs. Some examples are arcs for sub-typing and method parameters.

Modeling Sub-typing

As discussed in the previous section, Java supports only single 'inheritance' via the `extends` keyword. Java supports multiple interface 'inheritance' via the `implements` keyword. In this way, Java eliminates the ambiguity present in C++ when it is not always apparent which method implementation should be used when multiple super-types are present. Only one body for any method can be 'inherited', as the `extends` keywords supports only single 'inheritance'. These facts eliminate ambiguity from the static analysis of polymorphic method invocations in Java that may be present in other languages, such as C++.

It is worth noting that in the Java Language Specification an Interface is said to `extend` one or more other Interfaces. In the bytecode, this is

encoded as the `implements` relationship. In the bytecode, all Interfaces `extend java.lang.Object`. It is our opinion that the bytecode encoding of these relationships is more sensible, and these are the ones used in our domain model and algorithms.

In summary the model conforms with the following specification:

- Every class `extends` exactly one other class; `java.lang.Object` by default.
- A class may `implement` zero or more interfaces.
- Every interface `extends java.lang.Object`.
- An interface may `implement` zero or more other interfaces.
- Every array `extends java.lang.Object`.

3.5 Opcode Arcs

Figure 6 shows arcs that are generated from specific program statements (or *opcodes* in byte-code parlance). The `invokespecial` opcode is resolved to three distinct cases; this is important, because only one of these three cases uses dynamic dispatch (InvokeSuper). These arcs are generated from a **flow-insensitive** analysis, and duplicate arcs are eliminated (arcs are considered identical if they have the same source, target and type).

3.6 Class Hierarchy Analysis (CHA) Arcs

Figure 7 shows arcs that are computed by our graph construction algorithms, which are essentially *Class Hierarchy Analysis* (CHA) [DGC95, DMM96] algorithms adapted for the peculiarities of Java.

The polymorphic choice vertices are connected to method vertices by a whole program static analysis of the system (Class Hierarchy Analysis). If the method represented by the method vertex is abstract, a Signature arc is created; otherwise an Implementation arc is created from the polymorphic choice vertex to the method vertex. While computing these edges, we also compute (at no additional cost) an InheritedMethodImplementation arc. This arc connects a class vertex to a method body declared in some super-class, and is used when traversing for subsets.

InvokeVirtual and InvokeInterface from our perspective, signify different ways in which the sub-type structure of the program should be traversed looking for methods.

3.7 InvokeSuper

While relatively rare, InvokeSuper requires special attention for two reasons: First, it is one of three cases of the `invokespecial` opcode, and so

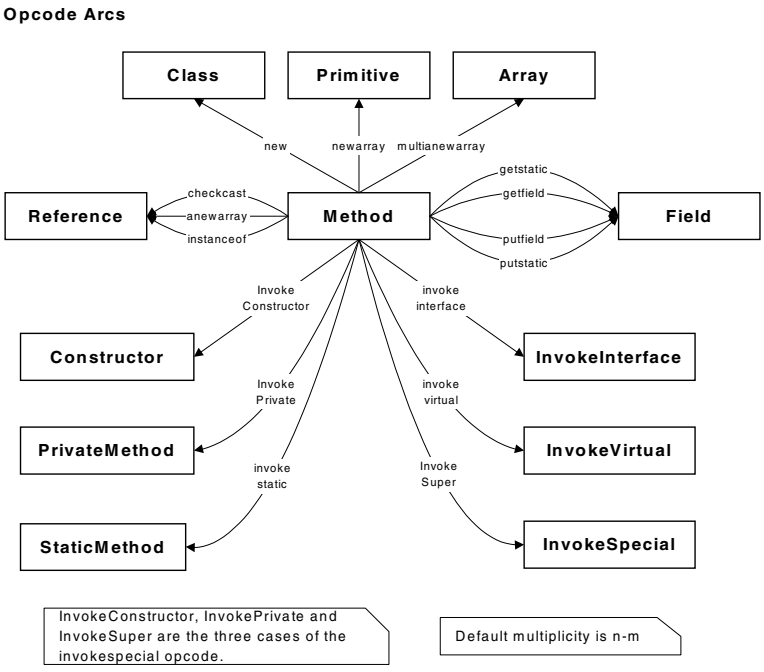

FIGURE 6. Opcode Arcs

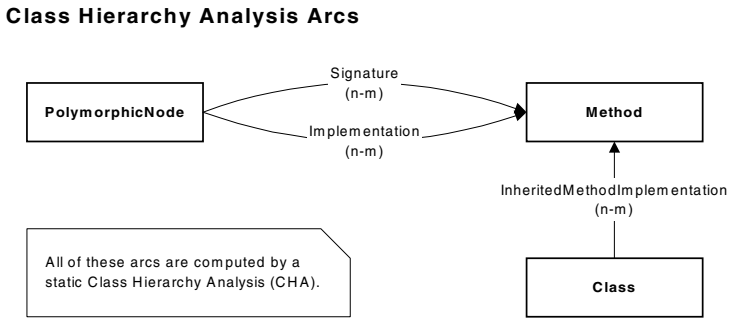

FIGURE 7. Class Hierarchy Analysis Arcs

is not obvious (almost all opcodes have only a single case). The other two cases are InvokeConstructor and InvokePrivate. Second, the `javac` compiler must be second-guessed to make an accurate analysis.

The InvokeSuper case is 'semi-polymorphic': it can always be statically resolved to a single target at link time, but because of the demands of *Release-to-Release Binary Compatibility* [GJS96], this single target may not be the one named at compile time. In other words, the super-class structure may have changed between the time when the sub-class was compiled and when it was linked. Moreover, the compiler does not name the immediate super-class but, rather, the first super-class in which the method is implemented.

Having said all that, the InvokeSuper situation does not occur frequently in practice.

3.8 Definition of Application and Library

In our model, only *use* arcs may cross the boundary from the application to the library. In other words, the application cannot define any components in the library and the library cannot define any components in the application. The *use* arcs include all of the *language* and *opcode* arcs; the *definition* arcs are the union of the *declaration* and *containment* arcs.

It is perhaps important to note that it is possible for the library to 'use' parts of the application due to polymorphism: a method signature may be defined in the library, invoked elsewhere in the library, but only actually implemented in the application. In this case an 'implementation' arc would be generated from the polymorphic choice vertex representing the invocation to the actual method in the application. This is the way that 'call-backs' must be implemented in Java, and is fairly common programming practice.

4 Java System Subsets

We define three library subsets (in increasing order of size) for a Java software system: *a)* space optimized, *b)* partially space optimized, and *c)* reduced (being the largest). Each of these is discussed in turn, below, with respect to the following "HelloWorld" example program. Experimental results are discussed in §6.

```
public class HelloWorld {
   public static void main(String[] args) {
      PrintStream p = System.out;
      p.println("Hello World!");
   }
}
```

4.1 Space Optimized Subset

The space optimized subset (herein the optimized subset) is composed of every class, field and method required for every possible execution path that originates in the application ('execution paths' includes event sequences generated by exception handling). This usually requires class files in the library to be modified by removing fields and methods that are not used by any execution path in the application. Since all execution paths in the optimized system originate in the application, the optimized subset is self contained.

The optimized subset is the subset that is actually loaded and resolved by a virtual machine that uses "lazy" resolution [LY97], with the additional constraint that every possible execution path originating in the application is exercised. The virtual machine that Sun distributes with the JDK uses "lazy" resolution. The optimized subset will also work in a virtual machine that uses "static" resolution.

In the HelloWorld example, `System.out` is obviously required, but `System.in` is not. Building the optimized subset would require removing the `in` field from the `System` class. This saves a significant amount of space, because the input functionality of the JDK is not used. In other words, the transitive closure of components required by the `in` field is large and unnecessary for this application.

4.2 Partial Space Optimized Subset

The partially space optimized subset (herein the partially optimized subset) is composed of all class files in the optimized subset, but without any modification. The partially optimized subset will work inside a virtual machine that uses "lazy" resolution, but may not work inside a virtual machine that uses "static" resolution.

In the case of HelloWorld, the `System` class would be included without modification, but none of the code that implements the input functionality would be included. A virtual machine that uses "static" resolution may complain that the code to implement `System.in` is not present.

Both the optimized subset and the partially optimized subset analyze the system using fields and methods as the atomic units. The partially optimized subset contains execution paths that will not work since it is not self contained. However, all execution paths that originate from the application will still work.

4.3 Space Reduced Subset

The space reduced subset (herein the reduced subset) is composed of all unmodified class files required by a virtual machine which uses "static" resolution to execute the system. The reduced subset calculation views the

class file as the atomic unit, as opposed to the field and method used in the optimized subsets.

All execution paths in the reduced system will work, including those that do not originate in the application. Therefore, the reduced subset is also self contained.

For the HelloWorld example, the reduced subset will include the `System` class unmodified, as well as all of the code necessary to implement in the input functionality (i.e. `InputStream`, etc.). An execution path that originates elsewhere in the JDK and uses `System.in` will work, although it is known that it cannot be exercised (by HelloWorld).

5 Extraction Process

The extraction process has two main steps: *a)* identify the subset of the library(ies) required for the given application, and *b)* extract the subset from the library. The input to this process is the byte-code for the system (application and library), a text file to specify the entry point of the application, and a switch to indicate which subset is to be extracted.

The subset is identified by first constructing a static dependency graph of the system according to the domain model specified above. The transitive closure of all elements required by the program entry point are identified by traversing this graph. The list of required components is then passed to the extractor.

The extractor is a fairly simple tool for the reduced and partially optimized subsets: it merely copies class files from the library to the target destination. The extraction tool for the optimized subset is significantly more complicated, and here we have only calculated an estimate of the space savings such a tool would generate. The optimized subset extraction tool must modify class files in order to remove unnecessary fields and methods, as well as re-pack the ConstantPool (symbol table) and remove debug information.

Difficulty may arise in the identification process if the program has execution entry points that cannot be identified through a static dependency analysis. This can occur when the byte-code interacts with 'native' code written in a language such as C, or through advanced usage of reflection. These problems can easily be worked around by specifying these extra entry points in the same text file that specifies the main entry point.

6 Experimental Results

In this section, we present the results of experiments obtained by applying the proposed system to three Java applications. These results indicate that

the majority of the JDK is not required for most applications, and that this technique is scalable.

6.1 Description of Experiments

The experiments were conducted on an IBM desktop computer with a 200Mhz Pentium processor and 64MB RAM using the JDK 1.1.5 for Windows 95. The tool which implements this technique is written in Java and runs inside the Java Virtual Machine. All extracted subsets were tested by executing the applications to ensure that they were still functional. The space savings between the partially optimized subset and the optimized subset is a minimal estimate: it is the space actually consumed by each method and field, it does not take into account the space that will be saved in the ConstantPool by removing these fields and methods.

A summary of the space savings results are contained in Table 1.1 — Table 1.4. The *Library Size* column illustrates the original size of the library bytecode. The *Relative Improvement* column indicates the percent space reduction with respect to the previous subset (row). The reduced relative percent improvement is measured against the original size, and the optimized relative percent improvement is measured against the partially optimized result. The *Absolute Improvement* indicates space reduction with respect to the original library size.

HelloWorld

The first experiment involved a small application that requires only a small part of the JDK library. A simple HelloWorld program, as illustrated in Table 1.1, does not require most of the JDK in order to execute: the reduced subset contains 178 files (535K), and the partially optimized subset contains 122 files (381K). The predicted optimized subset removes 979 fields and methods from the partially optimized subset and saves a further 53K. The reduced subset was identified in 760ms, and the optimized subset in 1100ms. The results are illustrated in Table 1.1.

This experiment indicated that the `initializeSystem()` method in the

HelloWorld *JDK subsets space savings results*	**Library Size (KB)**	**Relative Improvement (%)**	**Absolute Improvement (%)**
Original	8693	0	0
Reduced	535	93.1	93.1
Partially Optimized	381	28.7	95.6
Optimized	328	14.0	96.2

TABLE 1.1. JDK subsets space savings for *HelloWorld*.

`java.lang.System` class is executed by the native code in the VM when it is started up. It also showed that the `java.lang.ThreadDeath` class is referenced by the native code and required for execution.

JPack

The second experiment was conducted on the tool that performs the subset identification (JPack), as it is written completely in Java. The characteristics of this tool are similar to other tools: it reads input files, performs some processing, and writes the results to other files. The tool runs in a single thread and does not use any graphics. The reduced subset for this tool is comprised of 189 files (550K), which is just barely larger than the reduced subset for HelloWorld. The partially optimized subset was 142 files (414K). The predicted optimized subset is 974 fields and methods smaller than the partially optimized subset, for a further savings of 53K. The reduced subset was identified in 1380ms, and the optimized subset was identified in 5760ms. The results are illustrated in Table 1.2.

This experiment showed that the various character sets used by the JDK are referenced reflectively. The most common, ISO8859, is contained in the classes `sun.io.CharToByte8859_1` and `sun.io.ByteToChar8859_1`. The `sun.io` package contains classes for every character set supported by the base JDK.

JPack *JDK subsets space savings results*	**Library Size (KB)**	**Relative Improvement (%)**	**Absolute Improvement (%)**
Original Version	8693	0	0
Reduced Version	550	93.6	93.6
Partially Optimized	414	24.7	95.3
Optimized	361	12.8	96.2

TABLE 1.2. JDK subsets space savings for *JPack*.

CDF Editor

The third experiment involved the CDF Editor which is a sample application that ships with IBM's XML4J XML parser[IBMc]. CDF Editor is a GUI application for editing and viewing Channel Definition Format (CDF) files. This application was selected for two reasons: it uses two libraries (JDK and XML4J), and it indicates the overhead required to use XML and a GUI in an application.

The reduced subset requires 413 files from the JDK (1,035K) and 107 files from XML4J (264K). The partially optimized subset requires 368 files from the JDK (967K) and 90 files from XML4J (222K). The predicted

optimized subset does not require 2,575 fields and methods from the JDK (143K), nor does it require 390 fields and methods from XML4J (16K). Therefore, the approximate difference in size between the reduced subset and the optimized subset is 269K, approximately 20%. The results are illustrated in Table 1.3, and in Table 1.4.

The reduced subset was identified in 2580ms, and the optimized subset was identified in 4060ms.

This experiment indicated that a number of classes in `java.text.resources` are referenced either reflectively in the JDK or by the VM native code. Namely, `LocaleData`, `LocaleElements`, `DateFormatZoneData`, and the `LocaleElements` and `DateFormatZoneData` for one's particular geographic region. It is also useful to include `NoClassDefFoundError` and `ClassNotFound Exception` in the extracted JDK so that the VM can signal errors about missing code correctly.

The Abstract Window Toolkit (AWT) portion of the JDK also requires code that is referenced reflectively or through the VM native code. The class `java.awt.Event` is needed, as is the `initProperties` method of the `java.awt.Toolkit` class. The AWT is implemented differently behind the scenes for each platform, and this code is identified through the system property `awt.toolkit`. For the Windows version of the JDK the implementation requires the `WToolkit` and `WGraphics` classes in `sun.awt.windows`. The `font.properties` file also identifies `sun.awt.windows.CharToByteWingDings` and `sun.awt.CharToByteSymbol`. The layout manager of the

CDFEditor *JDK subsets space savings results*	**Library Size (KB)**	**Relative Improvement (%)**	**Absolute Improvement (%)**
Original	8693	0	0
Reduced	1035	88.0	88.0
Partially Optimized	967	6.5	88.8
Optimized	824	12.8	96.2

TABLE 1.3. JDK subsets space savings for *CDFEditor*.

CDFEditor *XML4J subsets space savings results*	**Library Size (KB)**	**Relative Improvement (%)**	**Absolute Improvement (%)**
Original	391	0	0
Reduced	264	32.4	32.4
Partially Optimized	222	15.9	43.2
Optimized	206	7.2	47.3

TABLE 1.4. XML4J subsets space savings for *CDFEditor*.

AWT requires `Container.layout()` and `LayoutManager` in `java.awt`.

6.2 Space Analysis

The reduced subset for the JDK showed an average of a 90.8% space savings over the original libraries (93.1%, 93.6%, and 88.0%; Tables 1.1, Table 1.2, and 1.3 respectively). The reason for this is evident when one examines the composition of the JDK: over two thirds of the library is consumed by international character sets and development tools, which are not used by most applications. The 32.4% (Table 1.4) improvement in the XML4J portion of the CDF Editor experiment is probably more typical of a regular library.

The partially optimized subsets showed an average 20.0% (28.7%, 24.7%, and 6.5%; Tables 1.1, 1.2, and 1.3, respectively) improvement over the reduced subsets. The optimized subsets showed an average improvement of 13.2% (14.0%, 12.8%, 12.8%; Tables 1.1, 1.2, and 1.3, respectively) over the partially optimized subsets. The optimized subsets show an average of almost 31% improvement over the reduced subsets (not shown in the table).

The optimized subset of the XML4J library indicated an approximately 47% improvement over the original library (Table 1.4). In other words, the CDF Editor uses about half the functionality available in the XML4J library. This result demonstrates the usefulness of this approach in allowing the developer to use arbitrary libraries without wasting storage space.

The JDK contains about 120 character sets, with an average size of approximately 40KB. However, almost all of these are either less than 20KB or greater than 100KB; the largest is almost 300KB. So, the optimized subset for an international HelloWorld may (in the very worst case) almost double in size. In many cases the growth will be less than 10% though.

6.3 Time Analysis

In each experiment it can be seen that the optimized subset is more difficult (time consuming) to identify than the reduced subset. This difference is to be expected, as the optimized subset deals with the system at a greater resolution. Results for the time analysis are shown in Table 1.5. Note that analysis time is the time required for the graph traversal, and does not include the time to parse the bytecode or write the output.

It is interesting to note that the reduced subset for JPack was identified almost as quickly as that for HelloWorld, but that the optimized subset was the longest computation. With respect to the reduced subset it can be seen that JPack does not use much more of the JDK than HelloWorld does, so the results are consistent. However, an interesting observation, is that the optimized subset takes longer to compute for our tool than for the CDF Editor (note that the code for the CDF Editor and XML4J are twice

as many files and bytes as that for JPack). This observation is explained by the fact that the extra time is caused by the deep inheritance hierarchies present in JPack. This is an indication that, the running of the selection process is dominated by the complexity of the sub-type hierarchies in a given application.

Measurement	HelloWorld	JPack	CDFEditor
# Graph Nodes	25,283	27,884	29,618
# Graph Arcs	174,729	190,421	203,626
Analysis Time for Reduced (ms)	760	1,380	2,580
Analysis Time for Optimized (ms)	1,100	5,760	4,060

TABLE 1.5. Analysis times compared to graph size.

7 Usage Scenarios

To illustrate the use of the system we provide four possible usage scenarios:

7.1 Embedded Systems

In this scenario, a software developer builds an application and then uses the tool to "trim" the libraries used by his or her application. If the subset is still too big for the constraints of the device that the system will be used in, the developer may modify the application and re-compute the subset until it meets the embedded system's requirements.

The optimized subset is the most useful for embedded systems because it is often known exactly what will be executed. The directly interpretable byte-code compression schemes discussed earlier ([RMH99, CSCM00]) would compliment the approach proposed here.

7.2 Distributed Systems

In this scenario the application and library subset is extracted and delivered to the end user "just in time" by a dedicated server. This server could keep track of the code that the client had previously downloaded and send only the delta. The partially optimized subset is useful here because the class files are not broken up: there is a balance between the configuration management difficulty and the amount of code transmitted.

This kind of application distribution can also be used to ensure that each client has the correct library version for the application in question.

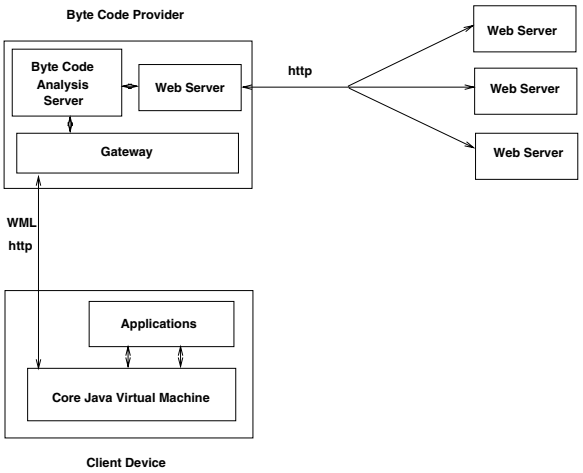

FIGURE 8. Distributed Systems Usage Architecture

All configuration management is done centrally; the clients merely request and execute applications.

The whole usage model is depicted in figure 8. In figure 8, the Client Device (CD) (i.e. a handheld computer, mobile phone, microPC) has only the core components of a JVM and is networked (wired or wireless) with a Byte Code Provider (BCP) that is composed of a Web Server, a Byte Code Analysis Server (BCAS) and a Gateway. The Byte Code Provider is running on a remote site and communicates with the Client Device through a Wireless Application (WAP) Gateway or the standard `http` protocol. All client requests are forwarded to the Internet domain by the Gateway. Once byte-code is send to the Client Device by an Internet Web server it is first passed by the Byte Code Analysis Server module that selects only the specific libraries required for the given downloaded application to run. The necessary libraries along with the application code are forwarded to the Gateway module that sends the only the absolutely necessary minimal size byte-code to the Client Device.

This scenario is applicable to web enabled cellular phones, palm-top devices or corporate intranets. Some of these systems may have the resources to decompress byte-code in a 'wire' format (e.g. [Cor94, BHV98, Pug99]), whereas others may be limited to directly interpretable formats (e.g. [RMH99, CSCM00]).

7.3 Native Code Compilers

The third usage scenario is related to the use compilers which translate byte-code to 'native' code for a particular platform. For example, using the

system discussed in this paper, library vendors can decide how to split their DLLs. In this case, the vendor would put all of the most commonly used code in the main DLL so that most applications would not have to load the entire library. The partially optimized subset is particularly important for this usage because the class files cannot be modified.

Alternatively, stand-alone EXEs may be created which do not depend on library DLLs. This is useful when distributing the application to those who may not have the appropriate library DLLs or runtime. The optimized subset would be used for this.

The IBM VisualAge C++ compiler front-end performs something similar to this [Kar98], and we have conducted some experimental work with the High Performance Java S/390 Group at the IBM Toronto Laboratory along these lines.

7.4 Library Re-factoring

The utility of this tool is predicated on library design with low coupling. If the library has extremely high coupling then it will not be possible to extract only a subset of it.

This tool can be used by the library vendor to identify poor coupling, which can be removed when the library is re-factored. Reports from application developers on the subsets that are being extracted for their applications can give the library vendors greater insight into how their code is being used. This information could be generated automatically if the application server discussed above were employed.

All three of the subsets presented here are useful for this task, and insight can be gained by comparing the results from each one.

8 Conclusion

This paper discussed a system that allows for the identification and extraction of software library subsets which are storage space optimized for a given application.

The selection is based on a dependency graph that is created for each application from the byte-code representation. A precise and accurate domain model (schema) has been presented for these graphs. Nodes in the graph correspond to application and library entities, and arcs correspond to dependencies between those entities. These dependencies may be divided into those computed by simply parsing the byte-code and those computed with *class hierarchy analysis* [DGC95, DMM96]. The dependencies computed with class hierarchy analysis are further refined with *rapid type analysis* [Bac97, BS96]. Both class hierarchy analysis and rapid type analysis are flow and context insensitive, and hence fairly fast to compute.

A library subset contains only those nodes that correspond to library entities and on which a given application depends (transitively). The subset is built by traversing the application/library dependency graph and by collecting the library nodes that can be reached during the traversal. Experimental results demonstrate that this traversal can be computed in a matter of seconds, even on graphs with tens of thousands of nodes and hundreds of thousands of arcs.

Three library subsets of interest have been identified in this paper. They are, in order of increasing size: the *optimized*, *partially optimized* and *reduced* subsets. These subsets have been defined in terms of the linking procedure used by the target virtual machine. The utility of these subsets for embedded systems, distributed systems and native code compilers has been discussed.

The selection algorithm is efficient and can be applied to large applications. Experimental results indicate that this system is scalable both with respect to time and space constraints, and can be a viable alternative to *a priori* library subsets.

Acknowledgments: We wish to thank Scott Kerr for his many helpful discussions in the early development of this work. Mike Fulton also contributed to the early development of this work. This work was funded in part by the IBM Centre for Advanced Studies and the High Performance Java S/390 Compiler Group, both at the IBM Toronto Laboratory.

9 References

[Age95] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, December 1995. Supervised by David Ungar. Appeared as Sun Microsystems Laboratories Technical Report SMLI TR-96-52.

[Bac97] David F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, University of California at Berkeley, December 1997. Supervised by Susan Graham. UCB/CSD-98-1017.

[BHV98] Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek. JAZZ: An efficient compressed format for Java archive files. In MacKay and Johnson [MJ98], pages 294–302.

[Blo97] Toby Bloom, editor. *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, Atlanta, Georgia, October 1997.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In Coplien [Cop96], pages 324 – 341.

[Cop96] James Coplien, editor. *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, San Jose, California, October 1996.

[Cor94] J. D. Corless. Compression of Java class files. Master's thesis, University of Victoria, 1994. Supervised by Nigel Horspool.

[CSCM00] Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, May 2000.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *ECOOP'95*, Århus, Denmark, August 1995. Springer-Verlag. LNCS 952.

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In Coplien [Cop96], pages 292–305.

[Eck98] Bruce Eckel. *Thinking In Java*. Prentice Hall, 1998.

[EEF+97] Jens Ernst, William Evans, Christopher Fraser, Steven Lucco, and Todd Proebsting. Code compression. In *PLDI'97*, June 1997.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In Bloom [Blo97].

[GJS96] James Gosling, Bill Joy, and Guy Steele Jr. *The Java Language Specification*. Addison Wesley, 1996.

[Ham98] Anita Hamilton. Dial I for Internet. *Times Magazine*, November 1998.

[HC98] R. Nigel Horspool and Jason Corless. Tailored compression of Java class files. *Software – Practice and Experience*, 28(12):1253 – 1268, October 1998.

[IBMa] IBM. alphaworks website. http://alphaworks.ibm.com.

[IBMb] IBM. VisualAge Micro Edition. http://www-4.ibm.com/software/ad/embedded/.

[IBMc] IBM. XML Parser for Java. http://alphaworks.ibm.com.

[Kar98] Michael Karasick. The architecture of montana: An open and extensible programming environment with an incremental C++ compiler. In *FSE'98*, pages 131 – 142, Orlando, Florida, November 1998.

[Kat90] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4), December 1990.

[Lea00] Doug Lea, editor. *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, Minneapolis, Minnesota, October 2000.

[Lee98] B. Lee. Internet embedded systems: Poised for takeoff. *IEEE Internet Computing*, 2(3):24–29, May 1998.

[LJH96] Loren Larsen and Mary Jean-Harrold. Slicing object-oriented software. In *ICSE'96*, pages 495 – 505, March 1996.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[MJ98] Stephen A. MacKay and J. Howard Johnson, editors. *Proceedings of the 8^{th} NRC/IBM Centre for Advanced Studies Conference (CASCON)*, Toronto, December 1998.

[PMS98] Sara Porat, Bilha Mendelson, and Irina Shapira. Sharpening global static analysis to cope with Java. In MacKay and Johnson [MJ98], pages 303 – 316.

[Pug99] William Pugh. Compressing Java class files. In *PLDI'99*, May 1999.

[Ray01] Derek Rayside. A generalized δ-wavefront algorithm for graph reachability problems with applications in software analysis. Master's thesis, University of Waterloo, 2001. Supervised by Kostas Kontogiannis.

[RK99] Derek Rayside and Kostas Kontogiannis. Extracting Java library subsets for deployment on embedded systems. In Paolo Nesi and Chris Verhoef, editors, *CSMR'99*, pages 102–110, Amsterdam, March 1999. Best Paper Award.

[RKK98] Derek Rayside, Scott Kerr, and Kostas Kontogiannis. Change and adaptive maintenance detection in Java software systems. In Michael Blaha, Alex Quilici, and Chris Verhoef, editors, *WCRE'98*, pages 10–19, Honolulu, October 1998.

[RMH99] Derek Rayside, Evan Mamas, and Erik Hons. Compact java binaries for embedded systems. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON'99*, pages 1–14, Toronto, November 1999. Best Paper Award.

[RRHK00] Derek Rayside, Steve Reuss, Erik Hedges, and Kostas Kontogiannis. The effect of call graph construction algorithms for object-oriented programs on automatic clustering. In Margaret-Anne Storey, Anneliese von Mayrhauser, and Harald Gall, editors, *IWPC'00*, pages 191–200, Limerick, Ireland, June 2000.

[Sec98] Secure Internet Programming Group. The Java Filter. Princeton University Dept. of Computer Science, 1998.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, and Etienne Gagnon. Practical virtual method call resolution for Java. In Lea [Lea00], pages 264 – 280.

[Sun] Sun Microsystems Inc. PersonalJava and EmbeddedJava Development Tools. http://java.sun.com/products/personaljava-/pjava_and_ejava_tools.html.

[TCFR96] Frank Tip, Jong-Deok Choi, John Field, and G. Ramalingam. Slicing class hierarchies in C++. In Coplien [Cop96], pages 179 – 197.

[TLSS99] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. In Linda Northrop, editor, *OOPSLA'99*, pages 292 – 305, Denver, Colorado, November 1999.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In Lea [Lea00], pages 281 – 293.

[TS97] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. In Bloom [Blo97], pages 271–285.

[Ven96a] Bill Venners. Under the hood: Bytecode basics. *Java World*, September 1996. http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html.

[Ven96b] Bill Venners. Under the hood: The Java class file lifestyle. *Java World*, July 1996. http://www.javaworld.com/javaworld/jw-07-1996/jw-07-classfile.html.

[Ven96c] Bill Venners. Under the hood: The lean, mean, virtual machine. *Java World*, June 1996. http://www.javaworld.com/-javaworld/jw-06-1996/jw-06-vm.html.

[Wor] World Wide Web Consortium. Compact html for small appliances. http://www.w3c.org/TR/1998/NOTE-compactHTML-19980209.

[Wor97] World Wide Web Consortium. Handheld device markup language 2.0. http://www.w3c.org/Submission/1997/5, May 1997.