1

Extracting Java Library Subsets for Deployment on Embedded Systems

Derek Rayside
Systems Design Engineering
University of Waterloo
drayside@swen.uwaterloo.ca

Kostas Kontogiannis
Electrical & Computer Engineering
University of Waterloo
kostas@swen.uwaterloo.ca

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 found in these systems. 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. This paper presents a technique that is used for dynamically 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. 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 dynamically builds library subsets on an as needed per application basis, has been built and tested on several mid-size Java applications with positive results.

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[1] by the year 2001 [10]. Examples of such embedded systems include hand-held terminals, cellular phones with 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 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 inherent features inherent to the language which are difficult to implement in embedded systems. These features include multi-threading 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 [15].

Efforts, attempting to re-target Java at embedded systems by defining standardized subsets of the Java language and class libraries are under way. Examples include Sun's PersonalJava, EmbeddedJava and, PicoJava environments. These JDK subsets limit the functionality of Java by removing certain language features such as multi-threading. This reduces both the processing power and storage requirements for executing simple Java applications.

In this paper an alternate solution to defining a standardized subset of a class library is proposed. The main idea is to dynamically 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.

*This work was funded by IBM Canada Ltd. Laboratory: High Performance Java Group and Centre for Advanced Studies. Special thanks to Scott Kerr of Object Technology International Inc. for his assistance in untieing the knot.

[1] This is just over one half of one percent of the estimated cost of fixing the Y2K problem[8].

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 bytecode (class files) are used to build the dependency graph which the analysis presented here is based on.

The process of building and traversing the dependency graph is very fast and is promising for use in Web-enabled embedded systems with limited processing power.

This paper is organized in nine sections. Section two provides an overview of Java features that are related to this work. Sections three and four provide pointers to related work and, an overview of the Java domain model respectively. Section five defines the library subsets to be extracted with respect to the Java language. Section six discusses the library subset extraction process and section seven provides and discusses our experimental results. Section eight discusses different usage scenaria for this technology. Finally section nine provides the conclusion of the paper.

2 Java Features

The Java language[3] is designed to execute on the Java Virtual Machine[11], 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).

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 [11].

Th JDK compiler (`javac`) translates Java source code to the class file format. `javac` has two options of interest for this work: `-O` for optimization, and `-g` to include debug information in the class files. `-O` creates slightly larger class files by inlining some fields and methods. `-g` makes class files significantly larger by including debug information such as line number tables in the class files.

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 parents of the 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 parent classes are kept in the parent class files, unless overridden.

Java code (source code and bytecode) 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 standard 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 (i.e. `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 bytecode representation in any way.

Bruce Eckel's book *Thinking In Java* provides a clear and in-depth discussion of the Java programming language[2] 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`)[22, 21, 20]. Moreover, both the Java Language Specification[3] and the Java Virtual Machine Specification[11] provide a detailed view of Java's features.

3 Related Work

The approach presented in this paper is based on a dependency analysis between the application code and library programming entities. The Java domain model from [15] has been adapted for these purposes.

3.1 Java on Embedded Systems

There are two main approaches for adapting the Java Virtual Machine architecture to embedded systems.

The first one focuses on porting a subset of the desktop Java VM directly into the Real Time Operating System (RTOS) environment. In [18] a core and standard extension (optional) API designed specifically for resource limited environments, with the addition of specific features required by consumer applications is defined. Similarly, in [19] a more restricted subset of Java than PersonalJava, designed specifically for severely resource constrained environments,

is proposed. Both EmbeddedJava and PersonalJava, are derived from the Java API and therefore, are upward compatible. Although these subsets have been optimized for small memory footprints and various visual displays, they usually result in a large (for embedded systems standards) end product [10]. In [5] a similar to Sun's version of small memory Java is specified.

The second approach is focusing on real-time deeply embedded systems. In this approach a Java Virtual machine version is specifically designed and implemented to satisfy real-time execution requirements. Native Java compilers have also been proposed by HP, IBM and other companies.

3.2 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[10] [4]. 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 [24] 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 [25].

3.3 Tools

Sun Microsystems produces two tools that are related to this work: JavaFilter and JavaCodeCompact [17]. JavaFilter allows for configuring the environment (i.e. selecting the necessary libraries) based on specific functionality required (i.e. for a given application). So far, we have not been able to obtain a license for JavaFilter for further evaluation. JavaCodeCompact is less related, as its primary purpose is to translate bytecode to platform independent C source code. Princeton University [16] has developed another tool also 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 client. Princeton's JavaFilter differs from the approach discussed in this paper, in the sense that it does not compute dependencies between applications and libraries. It basicaly performs pattern matching between the applet's originating URL and the URLs of restricted sites that are stored in its local database.

The IBM Research Lab in Haifa has independently developed a similar tool for the static analysis of Java software, although their focus is on compiler optimization[14]. Their tool will be posted on IBM's *alphaWorks* site[6] as a part of `Toad`.

A joint effort between the University of Victoria and the University of Geneva has produced a tool for the compression of Java class files[1]. Their tool performs better than standard compression algorithms by exploiting the known structure of class files. For optimal results, their tool would be applied to the class files after the process described in this paper.

Finally, the dependency analysis presented in this paper, is also related to dependency analysis proposed for analyzing the dependencies in C preprocessor statements [12], [23].

4 Domain Model

The domain model for Java software systems presented here is derived from [15], and consists of *entities*, *relations* and *attributes*. The purpose of this model is to represent components of Java software system and their interrelationships at a higher level of abstraction. Instances of this model are extracted from Java bytecode and expressed in Rigi Standard Form (RSF) [13].

4.1 Entities

This model has only one category of entities: *components*. Components have attributes that describe their interface and implementation, and relations to other components. Components in this model are at a finer level of granularity than the word component often implies.

Most of the components in the model are instances of Java data types. The Java data types are organized in a hierarchical fashion. There are two main categories: `Primitive` and `Reference`. The `Reference` types are subdivided into `Array` and `ClassOrInterface`, which is made up of the `Class` and `Interface` types [3]. The Java notion of an `Interface` is a special type of abstract class; it is distinct from the general concept of interface. The difference between a Java Class and a Java Interface is so small for our purposes that it makes sense to model both as `ClassOrInterface`. This is the approach adopted in [3] and [11]. For the sake of linguistic simplicity, we will use the term *class* to denote `ClassOrInterface` in the rest of this paper (after this section).

This model does not represent Packages explicitly: a Package is a mechanism used in Java to partition the `ClassOrInterface` namespace in a hierarchical fashion. A Package correlates to a directory where the source code and bytecode are kept, and is included in the *fully qualified name* [11] of a `ClassOrInterface`. In the terminology of [9], a Package is a *structural* entity.

The common notion of interface, as it applies to a `ClassOrInterface`, is the aggregate interface of the declared `Fields` and `Methods`. This is contrary to the no-

tion employed here: a `ClassOrInterface` is recognized as a component unto itself, not just an aggregation of other components (as a Package is). For example, a non-public `ClassOrInterface` cannot be used for casting by methods declared in other Packages. This example illustrates that a `ClassOrInterface` has an interface that is completely independent of its declared `Fields` and `Methods`. Similarly, the implementation of a `ClassOrInterface` is considered independently of its declared `Fields` and `Methods`. In [9]'s terminology, a `ClassOrInterface` is both a *structural* and a *representational* entity.

The following simple application is used as an illustrative example of the domain model. The fully qualified `ClassOrInterface` names are not used here in the interest of clarity: the fully qualified name of `Object` is `java.lang.Object`.

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

The bytecode representation of the above source is parsed to produce a dependency graph, which is expressed in RSF. A simplified visual representation of the graph for this example is shown in Figure 1. Sample RSF for this simple example may be found in [15].

The bytecode presents a more detailed representation of the software than the source code. Information not explicit in the source code can be found in bytecode and therefore facilitate a more accurate dependency analysis. For example, the graph in Fig. 1 contains `Object` and its default constructor, even though these are not explicitly referenced in the source code. There are other cases where the compiler will put insert bytecode in the class file that is not apparent in the source code [11].

4.2 Relations

The model presented here contains two categories of relations: *definition* and *use*. All relations represent a dependency between two components, and the direction of the relation is defined by the dependency [13]. Therefore, these relations define a *dependency graph* of the entities.

The direct use of a `ClassOrInterface` component is represented by such relations as `new`, `checkcast` and `instanceof`.

Note that the relations implicitly represented in a method signature, such as `ParameterType`, are explicitly represented in the model.

The complete model contains 7 distinct types of definition relations and 34 distinct types of use relations.

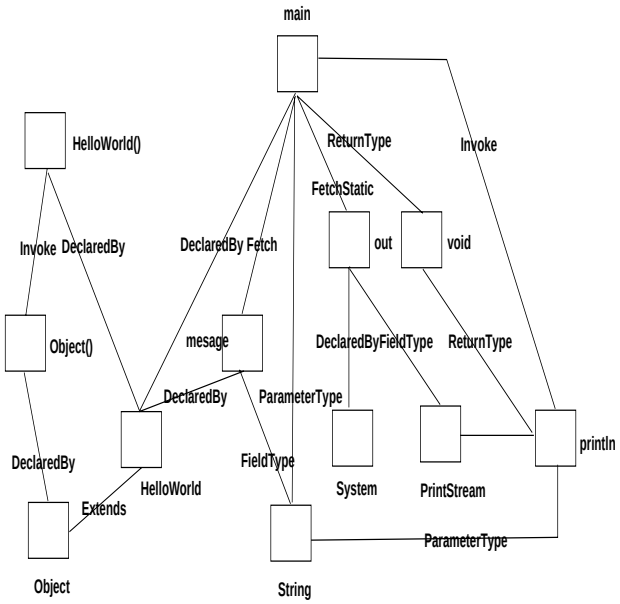

Figure 1. Dependency graph for "HelloWorld".

4.3 Attributes

Attributes describe the components in the model. All attributes take primitive values and are encoded in RSF. The notion of 'attribute' used in this model is separate and distinct from the notion of 'attribute' used in [11].

The component attributes are not as important in this usage of the domain model as they are in [15]. This usage is concerned primarily with the relations, and does not analyze the attributes in depth. The only important attribute for this analysis is the `Signature`. The `Signature` is the set of names which uniquely identifies each component.

The most interesting components are the `ClassOrInterface`, `Field` and `Method`, as they can be declared by the application developer. The `Signature` of these components are 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.

A more detailed version of the Java domain model used here can be found in [15].

5 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). Experimental results are discussed in section 7.

5.1 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[2] that originates in the application. 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 [11], 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.

Returning to 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.

5.2 Partial Space Optimized Subset

The partial 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.

[2] "Execution paths" includes event sequences generated by exception handling.

5.3 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).

6 Extraction Process

The extraction process has two main steps: *a)* identify the subset of the library[3] required for the given application, and *b)* extract the subset from the library. The input to this process is the bytecode 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 (see Figure 1). 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 compress 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 bytecode 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.

[3] 'Library' is used to mean one or more libraries.

Experiment	Application	Library	Reduced		Partially Optimized		Optimized	
	Size	Size	Lib. Size	% Impr.	Lib. Size	% Impr.	Lib. Size	% Impr.
HelloWorld	1	8693	535	94%	381	29%	328	14%
JTool	294	8693	550	94%	414	25%	361	13%
CDF Editor	56	8693 + 391 9084	1035 + 264 1299	88%, 32% 86%	967 + 222 1189	7%, 16% 8%	824 + 206 1030	15%, 7% 13%
Average Impr.				91%		21%		13%

Table 1. Space savings results. Percent improvement is with respect to previous column. Byte code sizes are given in KB.

7 Experiments

In this section, we present the results of experiments obtained by applying the proposed system to three Java applications. These results show that the majority of the JDK is not required for most applications, and that this technique is scalable.

7.1 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. The space savings results are illustrated in Table 1.

7.1.1 HelloWorld

The first experiment involved a small application that requires only a small part of the JDK library. A simple HelloWorld program, as shown above, 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.

This experiment showed that the `initializeSystem()` method in the `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.

7.1.2 JTool

The second experiment was conducted on the tool that performs the subset identification, 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.

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.

7.1.3 CDF Editor

The third experiment involved the CDF Editor which is a sample application that ships with IBM's XML4J XML parser[7]. 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 reduced subset was identified in 2580ms, and the optimized subset was identified in 4060ms.

This experiment showed 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`, `DateFormat ZoneData`, and the `LocaleElements` and `DateFormatZoneData` for one's particular geographic region. It is also useful to include `NoClassDef FoundError` 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 AWT requires `Container.layout()` and `LayoutManager` in `java.awt`.

7.2 Space Analysis

A summary of the space savings results are contained in Table 1. The Application and Library columns show the original size of the application and library bytecode. In the case of the CDF Editor experiment, sizes are shown for both libraries (JDK and XML4J). The last three columns display the results for each subset. The percent improvement columns are measured with respect to the previous column. The reduced percent improvement is measured against the original size, and the optimized percent improvement is measured against the partially optimized result.

The reduced subset showed an average of a 91% space savings over the original libraries. 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% 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 21% improvement over the reduced subsets. The optimized subsets showed an average improvement of 13% over the partially optimized subsets. The optimized subsets show an average 31% improvement over the reduced subsets (not shown in the table).

The optimized subset of the XML4J library showed a 47% improvement over the original library (not shown in table). 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.

7.3 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 2. 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 JTool 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 JTool 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 JTool). This observation is explained by the fact that the extra time is caused by the deep inheritance hierarchies present in JTool. This is an indication that, the time complexity of the selection tool is dominated by the height of the inheritance hierarchies in a given application.

8 Usage Scenaria

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

8.1 Embedded Systems

In this scenario, a software developer builds an application and then uses the tool to "trim" the libraries used by

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

Table 2. Time requirements for the tool, correlated to graph size.

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 bytecode compression tool in [1] would complement the tool developed here.

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

This scenario could be used for web enabled cellular phones, palm-top devices or corporate intranets. Again, the bytecode compression tool in [1] would complement the tool developed here.

8.3 Native Code Compilers

The third usage scenario is related to the use compilers which translate bytecode 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. The IBM High Performance Java S/390 Group has successfully used the tool for this purpose, as fast program load times are important in high speed transaction environments.

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 optimzed subset would be used for this.

8.4 Library Re-factoring

The usefulness 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 refactored. 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.

9 Conclusion

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

The selection is based on an entity-relation dependency graph that is dynamically created for each application. Nodes in the graph correspond to application and library entities, and arcs correspond to dependencies between those entities. A library subset contains only those nodes that correspond to library entities and on which a given application depends. The subset is built by traversing the application/library dependency graph and by collecting the library nodes that can be reached during the traversal.

A Java domain model has been built to facilitate the construction of such an entity-relation graph. The entity-relation graph is created by directly parsing the Java byte code.

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. The usefulness of these subsets for embedded systems, distributed systems and native code compilers has been illustrated.

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.

References

[1] Q. Bradley, R. N. Horspool, and J. Vitek. Jazz: An efficient compressed format for java archive files. In *Proc. of CASCON '98*, Toronto, December 1998.

[2] B. Eckel. *Thinking In Java*. Prentice Hall, 1998.

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

[4] A. Hamilton. Dial I for Internet. *Times Magazine*, 1998.

[5] Hewlett Packard. The Embedded Java VM Specification. http://www.hp.com/embeddedvm/.

[6] IBM. alphaWorks. http://www.alphaWorks.ibm.com.

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

[8] C. Jones. *The Year 2000 Software Problem — Quantifying the Costs and Assessing the Consequences*. Addison Wesley, 1998.

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

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

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[12] P. Livadas and D. Small. Understanding code containing preprocessor constructs. In *Proc. 3rd International Workshop on Program Comprehension*, pages pp. 89–97, Washington D.C., November 1994.

[13] H. A. Muller. Understanding software systems using reverse engineering technology perspectives from the Rigi project. In *Proc. CASCON '93*, pages pp. 217–226, Toronto, October 1993.

[14] S. Porat, B. Mendelson, and I. Shapira. Sharpening global static analysis to cope with java. In *Proc. of CASCON '98*, Toronto, December 1998.

[15] D. Rayside, S. Kerr, and K. Kontogiannis. Change and adaptive maintenance detection in Java software systems. In *Proc. 5th Working Conference on Reverse Engineering*, Honolulu, October 1998.

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

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

[18] Sun Microsystems Inc. PersonalJava . http://java.sun.com/products/personaljava/.

[19] Sun Microsystems Inc. The Embedded Java Specification. http://java.sun.com/products/embeddedjava/, February 1998.

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

[21] B. 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.

[22] B. 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.

[23] K. Vo and Y. Chen. Incl: A tool to analyze include files. In *Proc. USENIX Summer 1992*, pages pp. 199–208, San Antonio, 1992.

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

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