RC22178 (W0109-033) September 19, 2001
Computer Science

IBM Research Report

Practical Extraction Techniques for Java

Frank Tip, Peter F. Sweeney
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Chris Laffra, Aldo Eisma
Object Technology International

David Streeter
IBM Toronto Laboratory

Practical Extraction Techniques for Java

Frank Tip and Peter F. Sweeney
IBM T.J. Watson Research Center

Chris Laffra and Aldo Eisma
Object Technology International

David Streeter
IBM Toronto Laboratory

Abstract

Reducing application size is important for software that is distributed via the internet, in order to keep download times manageable, and for software that is deployed on embedded devices where memory is a scarce resource. This paper is concerned with the use of program transformations such as the removal of dead methods and fields, inlining of method calls, and transformation of the class hierarchy for reducing application size. We implemented a number of these techniques in *Jax*, an application extractor for Java, and evaluate their effectiveness on a set of applications ranging from 45 to 2,326 classes (with archives ranging from 55,765 to 3,810,120 bytes). We measured an average size reduction of 54.2% on these benchmarks. Modeling dynamic language features such as reflection, and extracting software distributions other than complete applications requires additional user input. We present a uniform approach for supplying this input in the form of MEL, a modular specification language for specifying extraction. We also study the domain-specific issues that arise when extracting embedded systems applications.

KEYWORDS: Application extraction, call graph construction, class hierarchy transformation, whole-program analysis, packaging

1 Introduction

Application size is an important limiting factor for distribution of applications over the internet, and for deploying applications on embedded devices. The internet is rapidly becoming the preferred medium for software distribution because one can avoid the cost of creating and distributing physical media such as CD-ROMs by having users download applications from internet sites. However, the time required to download an application is proportional to the application's size, and download times can be quite long, especially over slow modem connections. In the embedded systems domain, where memory is a scarce resource and where applications are often stored in Read-Only or Flash memory instead of on disk, it is also desirable to keep applications as small as possible, primarily to keep costs low. The increasing use of third-party class libraries and components in both PC-based and embedded applications is in direct conflict with these size requirements. This is the case because one often cannot assume that the user has installed (the correct versions of) these libraries, and the entire library/component is therefore shipped along with the application, resulting in a significant size increase.

This paper evaluates the effectiveness of a number of compiler-optimization and program transformation techniques for reducing the size of Java[1] [18] applications, and other kinds of Java distributions such as libraries and components. Java programs are typically distributed as compressed class file archives (i.e., zip files and jar files), a format that is much more amenable to program analysis than traditionally used representations such as object code. We

[1] Java and all Java-based marks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

have implemented a number of size-reducing techniques in the context of *Jax*, an application extractor for Java. *Jax* reads in the class files [27] that constitute a Java application (or other software distribution), and performs a static analysis to determine the components (e.g., classes, methods, and fields) of the application that must be retained in order to preserve program behavior. *Jax* then applies several size-reducing techniques, and writes out a class file archive containing the extracted application. Transformations incorporated in *Jax* include removal of redundant methods and fields, inlining of method calls in cases where this reduces program size, transformation of the class hierarchy, and renaming of packages, classes, methods and fields. *Jax* has been available from IBM's alphaWorks web site[2] since June 1998, and is one of the most popular tools there, having received over 30,000 downloads. Several commercial software products (developed both inside and outside IBM) have been processed with *Jax* before being shipped.

We have applied *Jax* to a set of realistic benchmark applications ranging from 45 to 2,326 classes (with corresponding class file archives of 55,765 to 3,810,120 bytes), and measured an average archive size reduction of 54.2%. More dramatic size reductions upwards of 70% are not uncommon for large, library-based applications, because such applications tend to use only a small part of the library functionality, and much unneeded functionality is pruned away.

Jax also supports the extraction of software distributions other than complete applications such as extensible class libraries, and applications that use dynamic features such as reflection. Additional user input is required in these case, and is supplied to *Jax* using a small modular specification language named MEL. We will present a small case study in which a number of extraction scenarios is applied to one of our benchmarks.

A number of the optimizations and program transformations performed by *Jax* have also been implemented in SmartLinker, a packaging tool that is incorporated into IBM's VisualAge® Micro Edition (VAME), an environment for developing Java applications for embedded systems. SmartLinker shares a significant amount of infrastructure with *Jax*, but also incorporates functionality that is specific to the embedded systems domain, such as the conversion to device-specific program representations, and several optimizations not performed by *Jax* such as feedback-directed inlining and ahead-of-time compilation. The paper studies several of these domain-specific issues in some detail.

[2]See `www.alphaWorks.ibm.com/tech/jax`.

1.1 Distribution Scenarios

Figure 1 shows several distribution scenarios that frequently arise in practice, and will be used to give a high-level overview of the significant issues that arise when extracting different kinds of software distributions. The figure shows a library vendor l responsible for creating and distributing a class library L, an application vendor a responsible for creating and distributing an L-based application A, and two users of application A named u and v, respectively.

1.1.1 Scenario 1: Extract a library without assumptions about its clients

In general, library vendor l will want to make library L as small as possible. Hence, l creates an *extracted version* L_{ext} of L, and distributes L_{ext} instead of L. Clearly, L_{ext} should offer the same functionality as L, since l cannot make assumptions about the way in which L is used by applications. In this case, it has to be assumed that any `public` and `protected` method[3] in L may be called by client applications. However, size-reducing transformations may still be applied to parts of L that are not directly exposed to users. In particular, `private` methods that cannot be called (directly or indirectly) by any `public` or `protected` method may be removed.

Another issue that arises when extracting libraries is whether or not the client should be able to create subclasses of the library classes. As we will see in Section 4, this will impact certain optimizations such as devirtualization and inlining.

[3]This assumes that the client application is not in the same **package** as the library. If this assumption is false, methods with package-level access have to be taken into account as well.

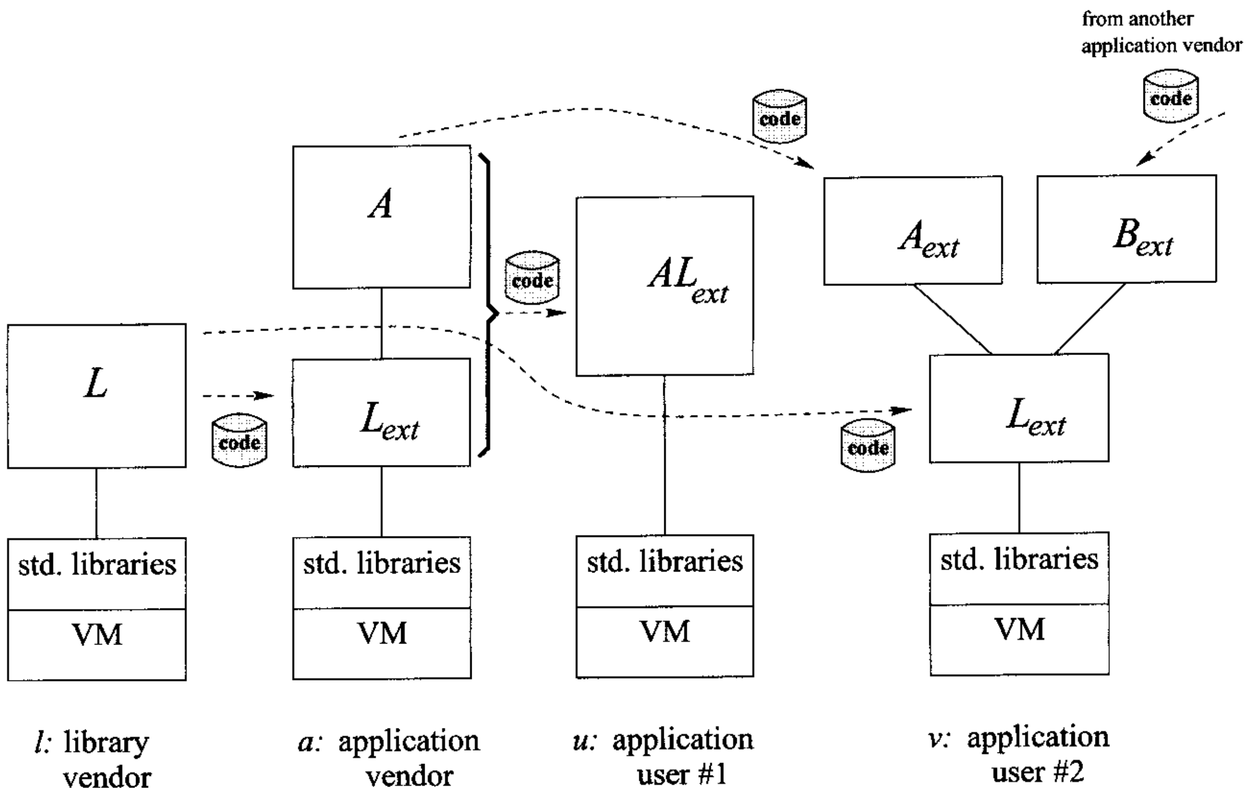


Figure 1: Illustration of different distribution scenarios.

1.1.2 Scenario 2: Extract a library in the context of a specific application

Application vendor a downloads L_{ext} for use during development of application A. When application A is ready for distribution, there are two options, depending on whether or not a user already has the prerequisite library L installed. Scenario 2 is the case where one cannot assume that the user of application A has (the correct version of) L installed. In such cases, it is desirable to extract L under the assumption that A is the *only* application that uses L. In this case, we can safely remove from L any method that is not transitively reachable from A's `main()` method.

The closed-world assumptions that we can make in this scenario will enable several other optimizations, such as the renaming of classes, methods, and fields, transformations of the class hierarchy, and devirtualization and inlining of method calls. It should be pointed out, however, that even in this case, there are limits to the kinds of transformations that can be applied. For example, classes accessed using reflection cannot be renamed, and it is generally the case that no platform-specific assumptions should be made based on a specific implementation of the standard Java libraries. These issues will be studied in greater detail in Sections 2 and 4.

1.1.3 Scenario 3: Extract a client without assumptions about its libraries

There is yet another case to consider: we may want to extract a client application A without making assumptions about the code in library L. This situation may occur when we want to ship A to a party that has a *different implementation* of L, or to a user who wants to *share* L between different L-based applications. In this case, determining the set of reachable methods in A requires that worst-case assumptions be made in cases where classes in A override methods defined in L.

Figure 1 shows another user v of application A, who has downloaded L_{ext} directly from l, because he is planning to deploy multiple applications that rely on the library. Because v already has L_{ext}, he only needs to download the application itself from vendor a. To this end, a creates an extracted version A_{ext} of A that can be downloaded by v.

1.2 Organization of this paper

Section 2 will present the optimizations and program transformations that are incorporated into *Jax*, and describe how they can be applied to extract complete applications (Scenario 2 above). Section 3 presents the results we gathered by applying *Jax* to a set of real-life benchmark applications. Section 4 is concerned with extraction scenarios other than complete applications (including Scenarios 1 and 3 above), and presents a solution in the form of a modular specification language that allows one to specify the extraction of various kinds of software distributions. Section 5 analyzes the issues that arise in the domain of embedded systems, and presents an overview of the techniques implemented in SmartLinker. Section 6 discusses related work. Section 7 presents conclusions, and outlines ongoing activities and plans for future work.

2 Overview of approach

This section presents a high-level overview of the transformations and optimizations that *Jax* applies to complete applications. Section 4 discusses how to adapt these transformations to accommodate other kinds of software distributions.

2.1 Loading the application

Jax begins by reading in the application from the original archive(s), and constructing an in-memory representation of the class files in the archive that the application refers to. *Jax* only loads classes that contain the application's entry points, and any classes that are directly or indirectly referenced from those classes.

Java provides a mechanism (in the Java literature referred to as *dynamic loading*) that allows an application to load a class (and create an object of that type) by providing the class name as a string. Because these strings are run-time values, it is in general not possible for a static analysis to determine which classes are dynamically loaded by an application. Therefore, *Jax* relies on the user to specify all classes that are dynamically loaded (the specification language that is used to provide this information will be presented in Section 4). Dynamically loaded classes are treated as additional entry points for the class loading process.

2.2 Removal of redundant class file attributes

Bytecode attributes that are unnecessary for *executing* the program such as local variable name tables and line number tables are discarded during loading.

2.3 Call graph construction

In general, only a subset of the methods in the loaded classes are required by an application. Unreachable methods may arise for many reasons. As systems get larger, programmers tend to lose track of the methods that are used. More importantly, breaking up applications into components that are designed and implemented separately leads to unreachable methods, because the creators of a component cannot always anticipate what functionality is needed. The scenario where applications use general-purpose class libraries that are developed elsewhere is a well-known example of this situation.

In order to determine which methods are reachable, *Jax* constructs a call graph. The key step in call graph construction is to conservatively approximating the "target" methods that can be invoked by a dynamic dispatch. Various algorithms have been proposed to determine the potential targets of a dynamic dispatch, including Class Hierarchy Analysis (CHA) [13], 0-CFA [38, 19], VTA [41], and algorithms for alias or points-to analysis [28, 37, 39]. *Jax* initially used Rapid Type Analysis (RTA) [6, 5] to resolve virtual calls. Later, we adopted the more precise XTA algorithm [46], which uses one set of types per method, to approximate the behavior of virtual method calls.

The example program of Figure 2 will be used to illustrate the CHA, RTA, and XTA call graph con-

```
interface I {
  public void foo();
}
class A implements I {
  int x, y, z;
  public void foo(){ int j=x; }
  public void bar(){ y=0; }
}
class B extends A {
  public void foo(){ y=0; z=0; int j=z; }
  public void bar(){ y=1; }
}
class C extends A {
  public void foo(){ this.bar(); }
}
public class Example {
  public static void main(String argv[]){
    f();
    g();
  }
  public static void f(){
    I i1 = new B();
    i1.foo();
  }
  public static void g(){
    I i2 = new C();
    i2.foo();
  }
}
```

Figure 2: Example program.

struction algorithms (for a more in-depth discussion and comparison of these algorithms, the reader is referred to [46]). The program contains classes `A`, `B`, and `C` and an interface `I` such that `A` implements `I`, and `B` and `C` inherit from `A`. Interface `I` declares a method `foo` that is overridden in `A`, `B`, and `C`, and class `A` defines a method `bar()` that is overridden in class `B`. Moreover, class `A` declares three fields `x`, `y`, and `z`. The program contains two direct call sites (the calls to methods `f()` and `g()`), and three dynamically dispatched call sites (the calls `i1.foo()`, `i2.foo()`, and `this.bar()`). Observe also that the program creates objects of types `B` and `C`, but not of type `A`. What are the reachable methods in this program?

- CHA determines that `A.foo()`, `B.foo()`, and `C.foo()` can be reached from call sites `i1.foo()` and `i2.foo()`, because classes `A`, `B`, and `C` each provide an overriding definition of `I.foo()`. Following a similar argument, CHA determines that both `A.bar()` and `B.bar()` can be reached from call site `this.bar()`.

- RTA uses the fact that no object of type `A` is created *anywhere in the program* to rule out `A.foo()` as a potential target, and determines that only `B.foo()` and `C.foo()` can be reached. RTA is unable to detect that method `B.bar()` is unreached, because it computes that both `A.bar()` and `B.bar()` may be reached from call site `this.bar()`.

- XTA finds that the `B`-object created in method `f()` cannot reach method `g()`, by analyzing how objects are moved around in an application via parameter passing and reads/writes to fields. Likewise, XTA determines that the `C`-object created in `g` cannot reach `f()`. Hence, XTA finds that the call site `i1.foo()` in method `f()` can only invoke `B.foo()`, and that the call site `i2.foo()` in `g()` can only invoke `C.foo()`. Moreover, XTA determines that no objects of type `B` reach call site `this.bar()`, and finds that method `B.bar()` is unreached.

Unfortunately, not all unreachable methods can simply be removed. Method `B.bar()` can be removed without any problem, but removing `A.foo()` would lead to an invalid class file, because class `A` promises to implement interface `I`, but it would define method `foo()`, which is declared in `I`. In such cases, *Jax* removes the *body* of the method and replace it with a return statement.

Implementing a call graph construction algorithm for the full Java language requires that a number of pragmatic issues be addressed. Class initializer methods are executed upon the first active use of a class (i.e., when the class is instantiated, or when a member in the class is accessed). We have approximated this in our analysis by treating an initializer method as a reached method when we see an active use of the class. The use of class libraries further complicates the task of determining reachable methods. For the sake of this discussion, we will make a distinction between *application* libraries that are shipped along with the application and for which full information is available, and *external* libraries that are outside the scope of our analysis, and for which we only know the signatures of methods that can be overridden. Consider a situation where a class C in the application inherits from a class L in an external library, and suppose that C provides an overriding definition for a method $L.f()$. Then, a virtual dispatch inside L can resolve to method $C.f$ in the application's code. The crucial issue is that the code for the library is unavailable, so our analysis may never see a call to any method f. We conservatively approximate calls from within libraries by assuming that any overridden library method may be called. However, by keeping track of the types of objects that have been passed into the libraries, we can determine that certain methods f in the application's code cannot be reached [46].

Java's reflection mechanism [4] further complicates the task of determining reachable methods, because it essentially allows one to invoke a method by specifying as a string (computed at run-time) its name and signature. Because it is in general undecidable to detect such method calls by static analysis, *Jax* relies on the user to inform it of such method invocations, and treats them as entry points for reachable method analysis. Default constructors of dynamically loaded classes are treated as entry points as well.

2.4 Redundant field elimination

The elimination of methods can lead to the elimination of fields. In our example, `x` is only accessed from `A.foo()`, and because `A.foo()` is unreachable, `x` may be removed from the application without affecting program behavior. Fields that are only written to (but not read) can also be removed because their value cannot affect the program's behavior [42]. In addition to removing the field itself, this involves removal of the instructions that store the value in the field. In the example of Figure 2, field `A.y` is write-only, and can be eliminated.

Figure 3 depicts a source-to-source view of the successive transformations performed by *Jax* (in reality, all these operations are performed at the class file level). Figure 3(a) shows the original program. Figure 3(b) shows the program after removing unreachable method `B.bar()`, removing the body of unreachable method `A.foo()`, removing unaccessed field `A.x`, and removing write-only field `A.y`.

2.5 Class hierarchy transformations

Jax applies a number of semantics-preserving transformations to the class hierarchy. These transformations reduce archive size by eliminating classes entirely, and by merging adjacent classes in the hierarchy. The benefits of merging classes are:

- merging classes may enable the transformation of virtual method calls into direct method calls, and

- merging classes reduces the duplication of literals across the constant pools of different classes (this will be discussed in more detail in Section 2.8).

The class hierarchy transformations used in *Jax* are an adaptation of the ones in [47, 48], where they serve as a simplification phase after the generation of specialized class hierarchies (the relationship with this work will be discussed in Section 6).

One of the simplest transformations is the removal of an uninstantiated class that does not have any derived classes, and that does not contain any live methods or fields. More interesting is the transformation where a base class X and a derived class Y are merged if there is no live non-**abstract** method f that occurs in both X and Y, and one of the following conditions holds:

1. X is uninstantiated, *or*

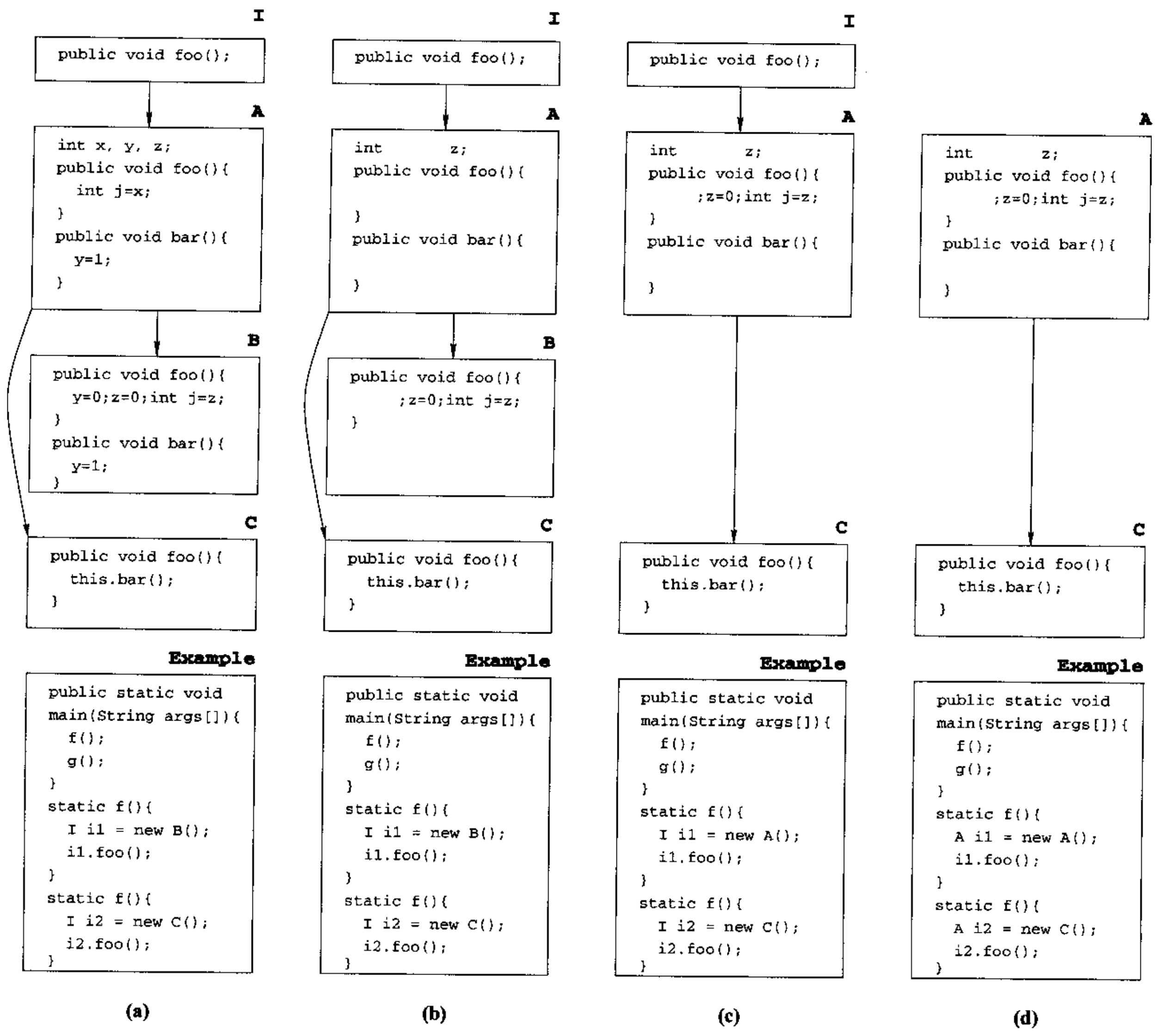
2. Y does not contain any live non-static fields,

By requiring that (1) or (2) hold, we ensure that no object created by the application becomes larger (i.e., contain more fields) as a result of the merge.

Merging a base class X with a derived class Y involves a number of steps. All live methods and fields of Y are moved to X. Cases where X and Y have fields or static methods with identical names pose no problem, because we can simply rename any field or static method in cases where name conflicts occur. Constructors require special treatment because X and Y may have constructors with identical signatures, and constructors cannot be renamed. In such cases, a new signature for the constructor is synthesized by adding dummy arguments, and constructor calls are updated accordingly by pushing `null` elements on the stack. If X and Y both have static initializer methods, we turn the static initializer for X into an ordinary static method, and insert a call to that method at the beginning of the initializer for Y[4]. If X and Y both contain an instance method f, then at least one these methods must be **abstract** due to the preconditions stated above. If both methods f are abstract, $Y.f$ is simply removed. Otherwise, the non-abstract method is preferred over the abstract method. Finally, all references to Y, as well as methods and fields in Y are updated to reflect their new "location" in class X.

Other class hierarchy transformations include the merging of classes with interfaces, and are very similar to the transformation described above. For more details on class hierarchy transformation, we refer the reader to [48]. However, a few more issues pertaining to class merging should be mentioned. In order to allow the merging of classes across **package** boundaries, classes, methods and fields need to be made

[4]This assumes that programs do not rely on the execution order of static initializers of different classes.



(a)

```
I
public void foo();

A
int x, y, z;
public void foo(){
  int j=x;
}
public void bar(){
  y=1;
}

B
public void foo(){
  y=0;z=0;int j=z;
}
public void bar(){
  y=1;
}

C
public void foo(){
  this.bar();
}

Example
public static void
main(String args[]){
  f();
  g();
}
static f(){
  I i1 = new B();
  i1.foo();
}
static f(){
  I i2 = new C();
  i2.foo();
}
```

(b)

```
I
public void foo();

A
int       z;
public void foo(){

}
public void bar(){

}

B
public void foo(){
     ;z=0;int j=z;
}

C
public void foo(){
  this.bar();
}

Example
public static void
main(String args[]){
  f();
  g();
}
static f(){
  I i1 = new B();
  i1.foo();
}
static f(){
  I i2 = new C();
  i2.foo();
}
```

(c)

```
I
public void foo();

A
int       z;
public void foo(){
     ;z=0;int j=z;
}
public void bar(){

}

C
public void foo(){
  this.bar();
}

Example
public static void
main(String args[]){
  f();
  g();
}
static f(){
  I i1 = new A();
  i1.foo();
}
static f(){
  I i2 = new C();
  i2.foo();
}
```

(d)

```
A
int       z;
public void foo(){
     ;z=0;int j=z;
}
public void bar(){

}

C
public void foo(){
  this.bar();
}

Example
public static void
main(String args[]){
  f();
  g();
}
static f(){
  A i1 = new A();
  i1.foo();
}
static f(){
  A i2 = new C();
  i2.foo();
}
```

Figure 3: Successive steps in transformation of the example program of Figure 2 by *Jax*. **(a)** the original program, **(b)** the program after removing unreachable method `B.bar()`, removing the body of unreachable method `A.foo()`, removing unaccessed field `A.x`, and removing write-only field `A.y`. **(c)** the program after merging class `B` into class `A`. **(d)** the program after merging interface `I` into class `A`.

public. Finally, classes that are explicitly referenced using reflection cannot be merged with other classes.

Figure 3(c) shows the example program after merging `B` into `A`. Note that the '`new B`' statement in method `Example.main()` is changed into '`new A`'. Note that this class merging operation could not have been applied to the original class hierarchy, because both classes `A` and `B` contained a non-abstract method `foo()` originally. Hence, this class merging operation was *enabled* by the removal of unreachable method `A.foo()`. Figure 3(d) shows the program after merging `I` with `A`. Note that the types of variables `i1` and `i2` have changed from `I` to `A`.

2.6 Performing optimizations

Thus far, we have primarily focused on archive size reduction, and optimization was only a secondary goal. However, a few simple and easy-to-implement optimizations have been implemented in *Jax*. Non-overridden methods are inlined in cases where this does not increase application size. Moreover, in cases where a virtual dispatch has only one potential target, we "devirtualize" the call by replacing an `invokevirtual` with an `invokespecial` bytecode. Unfortunately, the Java Virtual Machine Specification [27] only permits this in a very limited number of situations: the callee has to occur in a superclass of the caller class, or has to be a private method. *Jax* marks non-overridden virtual methods `final` so that a just-in-time compiler can inline these calls where appropriate.

The example program of Figure 2 illustrates how transformations enable each other. We have already seen how elimination of (the body of) unreachable methods enables class merging. Note that, in the resulting program of Figure 3(d) only a single method `bar()` remains in the entire class hierarchy. This implies that the call to `bar()` can be devirtualized and subsequently inlined.

2.7 Name compression

A Java class file is a self-contained unit of executable code. References to other classes, methods, and fields are made through string literals. For example, if a class contains a method call `Thread.sleep(500)`, the constant pool for that class contains the strings "`java/lang/Thread`", "`sleep`", and "`(J)V`", representing the fully qualified class name, the method name, and a string representation of the method's signature (one argument of type `long`; returning `void`). Because all linking information is represented in string form, and is replicated in each class file, it is obvious that shortening class, method, and field names by shorter ones will result in smaller archives. *Jax* currently renames classes, methods, and fields `a`, `b`, `c`, ... but more ambitious naming schemes, in which methods with different signatures get the same name are possible. Certain names cannot be compressed. This includes any reference to a class, method, or field in an external library, methods that override methods in external libraries, any program component accessed using reflection, the name of class containing the `main` routine, and and the names of constructors and static initializers.

2.8 Constant pool compression

Removing redundant methods and fields may render constant pool entries unnecessary. In the in-memory representation of class files constructed by *Jax*, references to constant pool entries are replaced by explicit references to objects representing the classes, methods, fields, and constants normally contained in the constant pool. After the transformations described above have been performed by *Jax* the class is written out again, and a new constant pool is created from scratch. Only the classes, methods, fields, and constants that are actually referenced will be added to this constant pool. The resulting constant pool has minimal size and is typically much smaller than the constant pool originally found in the class file.

Class merging has interesting repercussions for the size of constant pools. Adjacent classes in the hierarchy are likely to share many literal values, which are *duplicated* in their constant pools. Merging these classes allows us to eliminate this duplication. We will see in Section 3.5 that the contribution of class hierarchy transformation to archive size reduction can be significant.

benchmark	# classes	# methods	#fields	archive
Hanoi	45	378	233	55,765
Jax	302	2,900	1,274	534,658
javac	210	1,512	1,107	452,125
bloat	282	2,677	1,255	506,736
mBird	2,050	17,946	6,739	2,950,543
Jinsight	264	2,974	1,875	393,857
JavaFig	160	2,108	1,526	394,432
CindyApplet	467	4,449	3,075	881,555
Cinderella	467	4,449	3,075	881,555
eSuite Sheet	588	5,590	4,305	1,251,765
eSuite Chart	733	8,302	5,448	1,570,569
Hyper/J	921	8,776	2,733	1,523,670
Res. System	2,326	21,495	12,487	3,810,120

Table 1: Characteristics of the benchmark applications used to evaluate *Jax*. For each benchmark, the initial number of classes, methods, and fields is shown. The size of the initial archive shown here is in bytes and excludes any resource files contained in the shipped archives.

3 Results

3.1 Overview of the benchmarks

Table 1 lists the Java applications used to evaluate *Jax*. The benchmarks cover a wide spectrum of programming styles and are publicly available (except for *mBird* and *Reservation System*). For each benchmark, the initial number of classes, methods, and fields are shown, as well as the initial size of the archive.

Hanoi is an interactive applet version of the well-known "Towers of Hanoi" problem, and is shipped with *Jax*. An earlier version of *Jax* itself (version 7.1) was used as a benchmark. *javac*[5] is the SPEC JVM 98 version Sun's source to byte-code compiler. *bloat*[6] is a byte-code optimizer developed at Purdue University. *mBird* is a proprietary IBM tool for multi-language operability. It relies on, but uses only limited parts of, several large class libraries (including Swing, now part of the standard libraries, and IBM's XML parser). *Jinsight*[7] is a performance analysis tool developed at IBM Research. *JavaFig*[8] (version 1.43 (22.02.99)) is a Java version of the `xfig` drawing program. *Cinderella*[9] is an interactive geometry tool used for education and self-study in schools and universities. *CindyApplet* is an applet that allows users to solve geometry exercises interactively. It is contained in the same class file archive as *Cinderella*. *Lotus eSuite Sheet*[10] is an interactive spreadsheet applet, which is part of the examples shipped with Lotus' *eSuite*, a productivity suite. *Lotus eSuite Chart* is an interactive charting applet, another example shipped with Lotus *eSuite*. *Hyper/J*[11] is a system for advanced separation of concerns developed at IBM research. *Reservation System* is an interactive front-end for an airline, hotel, and car rental reservation system developed by an IBM customer.

[5]See `www.specbench.org`.

[6]See `www.cs.purdue.edu/homes/hosking/pjama.html`.

[7]See `www.research.ibm.com/jinsight`.

[8]See `tech-www.informatik.uni-hamburg.de/applets/javafig`.

[9]See `www.cinderella.de`.

[10]See `www.esuite.lotus.com`.

[11]See `www.research.ibm.com/hyperspace`.

benchmark	# classes	# methods	#fields	archive	time
Hanoi	21	179	97	21,055	7.0
Jax	272	2,128	739	333,268	22.8
javac	198	1,342	496	231,605	21.1
bloat	245	2,266	635	251,822	23.9
mBird	230	1,934	743	276,884	19.6
Jinsight	222	2,469	1,036	317,166	22.0
JavaFig	114	1,443	1,055	223,913	15.1
CindyApplet	171	1,325	831	179,702	21.5
Cinderella	280	2,456	1,773	385,900	28.7
eSuite Sheet	240	2,378	939	330,677	28.0
eSuite Chart	398	4,472	2,129	598,985	38.9
Hyper/J	428	3,584	924	424,074	80.0
Res. System	1,432	11,540	5,004	1,725,421	256.4

Table 2: The number of classes, methods, and fields and the archive size for the benchmark applications of Table 1 after processing by *Jax*. The rightmost column shows the time (in seconds) required by *Jax* to process the application.

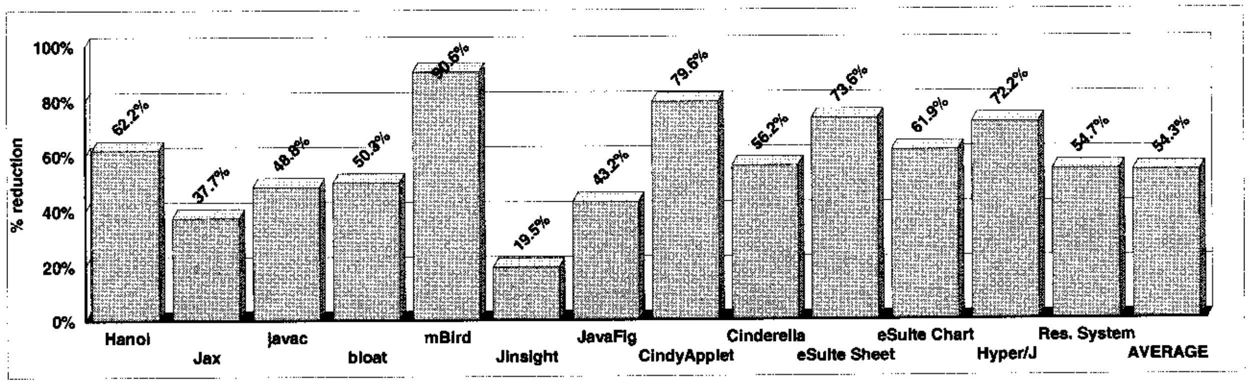


Figure 4: Percentage reduction in archive size for the benchmark applications of Table 1. Resource files are excluded from both the initial and processed archives.

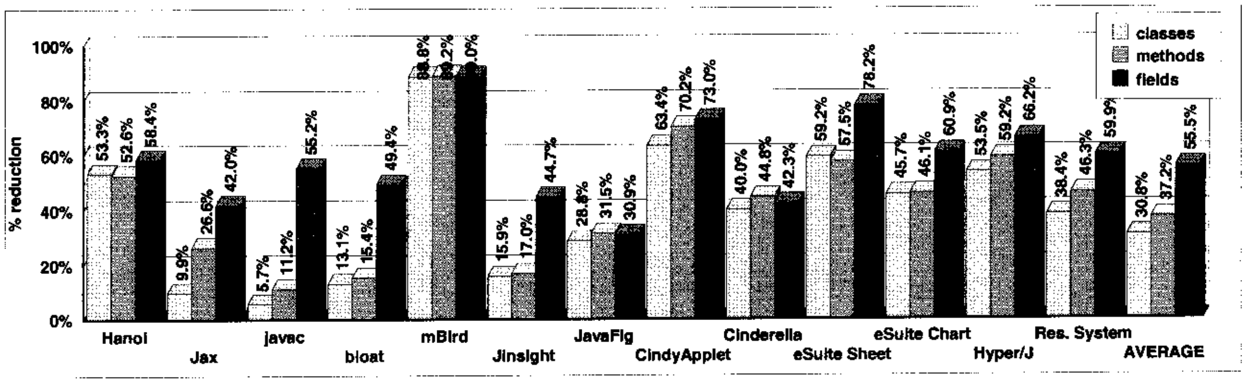


Figure 5: Percentage reduction in numbers of classes, methods, and fields for the benchmark applications of Table 1.

3.2 Measurement issues

For a number of the benchmarks, the shipped version of the initial archive contains resource files such as properties files and image files. Since our techniques only address the transformation of class files, we moved all resource files to a separate archive. This "resources archive" is unaffected by *Jax*, and its contents should be added to the archive produced by *Jax* in order to run the compressed application. Currently, our techniques do not address the issue of determining which resources are actually used by an application, and we have observed cases where archives contained many unneeded resources.

Another issue is that different implementations of *zip* and *jar* tend to produce slightly different results. In order to give a consistent evaluation, all archives mentioned in this paper have been unzipped, and subsequently re-zipped (into a single archive) using WinZip 7.0[12].

3.3 Reductions in archive size, classes, methods, and fields

Table 2 shows the overall size reductions obtained by applying *Jax* (version 7.4) to the benchmarks of Table 1, as well as the time required by *Jax* to process the benchmarks[13]. *Reservation System*, the largest benchmark, was processed in about 4.5 minutes. We consider these processing times to be quite acceptable, especially since application extraction is typically an infrequent activity that is only performed when applications are shipped.

Figure 4 depicts the percentage by which the archive size is reduced for each benchmark. As can be seen from the figure, the reduction ranges from 19.5% to 90.6% (54.3% on average). Figure 5 depicts the percentage by which the number of classes, methods, and fields are reduced for each benchmark. As is shown in the figure, the number of classes is reduced by 5.7% to 88.8% (30.8% on average), the number of methods by 11.2% to 89.2% (37.2% on average), and the number of fields by 30.9% to 89.0% (55.5% on average).

3.4 Evaluation

A number of observations can be made about the results reported above.

The benchmark for which we measured the smallest reduction in archive size is *Jinsight*. A discussion with the developers revealed that this bench-

[12]See `www.winzip.com`.

[13]All measurements were taken on a Pentium III/800Mhz PC with 1 processor, a 64K L2-cache and 512MB memory running Windows 2000. All measurements were conducted using Sun JDK 1.2.2, in combination with a Just-In-Time compiler developed by IBM [23]

mark does not rely on any class libraries other than the standard libraries, and that most of the removed methods correspond to future extensions that were never fully implemented, and to functionality that had become obsolete.

The benchmark for which we measured the highest reduction in archive size, *mBird*, is a special case. *mBird* consists of two distinct components: a tool with an interactive GUI and a command-line "batch" tool. These tools are usually shipped together as a single class file archive. In our evaluation, we extracted *only* the batch component from this archive. The main reason for the very large size reductions is that *Jax* is very effective in removing the unused GUI-related library classes. Other benchmarks for which we measure large reductions such as *Hanoi* with 62.2%, *Lotus eSuite Sheet* with 73.6%, *Lotus eSuite Chart* with 61.9%, and *Hyper/J* with 72.2% either rely on class libraries, or are structured as a class library with a client application. The large size reductions we measure for these benchmarks are in agreement with the general perception that applications typically use only a small fraction of the functionality in class libraries that they rely on, and it shows that our techniques are quite successful in eliminating redundant library functionality.

The *Cinderella* and *Cinderella Applet* benchmarks are another interesting case because they are derived from the same original archive. In this case, *Cinderella Applet* contains (roughly) a subset of the *Cinderella*'s functionality. For development purposes, it is desirable to have the two applications share the same archive, but for distribution purposes it is undesirable to ship the entire archive for *Cinderella Applet*. A common solution to such problems consists of splitting the classes into a package with "core functionality", and separate packages with additional functionality that is used by different components. This approach has some obvious organizational drawbacks. In such cases, an application extractor can extract the desired functionality for each application. The fact that *Jax* is capable of eliminating unused functionality is evident from the fact that we see a significantly larger size reduction for *Cinderella Applet* (79.6% archive size) than for *Cinderella* (56.2%). Section 4.5 explores different distribution scenarios for *Cinderella* in more detail. Encouraged by these results, the creators of *Cinderella* decided to use *Jax* to create their various distributions.

3.5 Breakdown of the results

Measuring the individual contributions of each step in *Jax* is complicated by the fact that each step's effectiveness strongly depends on the preceding one. For example, the removal of useless fields is performed after the removal of unreachable methods (otherwise, we would not be able to remove fields that are only referenced bwithiny unreached methods). Hence, correlating the contributions of dead fields or methods *in isolation* to the reduction in archive size would be meaningless. Another example along these lines is that the number of classes that can be merged is strongly dependent on removal of unused fields and methods in a previous step. Consequently, what we will study in the remainder of this section is the *cumulative* effect of each step. By selectively disabling steps performed by *Jax*, we measure the additional impact of each step.

Table 3 shows detailed statistics gathered for the *Hyper/J* benchmark, indicating the size of the archive, and the numbers of classes, methods, and fields after each step. Figures 6 and 7 show the contributions of the successive steps in graphical form. These figures reveal several interesting facts. A substantial number of classes in the initial archive (132 out of 921) is not loaded. Removal of these unreferenced classes reduces archive size by 14.0%. Most of these unused classes are library classes that are shipped with, but not used by the application. Removal of redundant attributes such as line number tables and local variable name tables contributes 18.9% to the reduction in archive size, bringing the total size reduction to 32.9%. Removal of unreachable methods results in an additional reduction in archive size of 20.4%. The contribution of useless field removal is relatively small: 4.0%. Many of the removed fields are static final fields. Java compilers apply constant propagation and replace each occurrence of such a field by its value. The original field remains, even though it is now redundant. In the case, of *Hyper/J*, almost 2,800 fields are removed from the application.

Hyper/J	classes	methods	fields	archive
original size	921	8,776	2,733	1,523,670
unreferenced classes	789	7,674	2,602	1,309,962
redundant attributes	789	7,674	2,602	1,021,698
dead methods	789	3,870	2,602	711,678
redundant fields	789	3,870	924	650,461
inlining/devirtualizing	789	3,844	924	648,646
class transformations	428	3,584	924	535,606
name compression	428	3,584	924	424,074

Table 3: Detailed measurements for the *Hyper/J* benchmark.

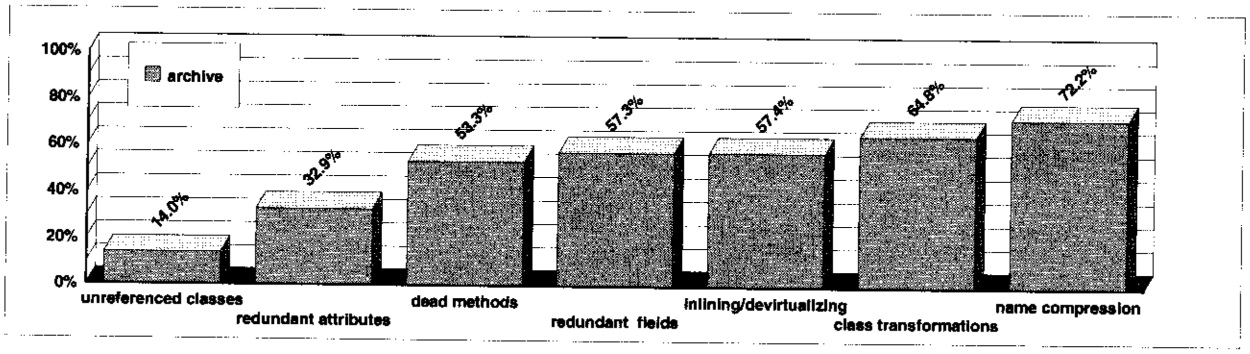


Figure 6: Percentage reduction in archive size for *Lotus eSuite Sheet* w.r.t. the original archive after (1) loading, (2) removal of unreachable methods, (3) removal of useless fields, (4) method inlining/devirtualizing, (5) class hierarchy transformations, and (6) name compression.

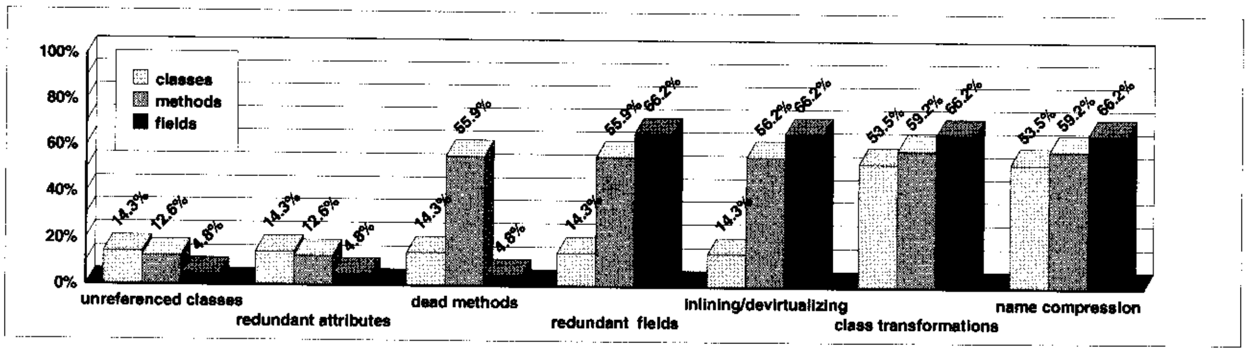


Figure 7: Percentage reduction in the number of classes, methods, and fields w.r.t. the original archive for *Hyper/J* after (1) loading, (2) removal of unreachable methods, (3) removal of useless fields, (4) method inlining/devirtualizing, (5) class hierarchy transformations, and (6) name compression.

Method inlining and devirtualizing have a small positive effect on the result (0.1%). The contribution of class hierarchy transformations is a significant 7.8%. The *Hyper/J* benchmark is written in a highly object-oriented style, with heavy use of interfaces. Large sections of the class hierarchy are flattened by *Jax*, as is evidenced by the fact that the number of classes is reduced from 789 to 428. From Table 3 and Figure 7 it can be seen that the class hierarchy transformations remove an additional 3.0% of the methods (260 methods). These methods are `abstract` methods that disappear as a result of class merging. Name compression reduces the resulting archive by another 7.4%, bringing the cumulative size reduction to 72.2%.

3.6 Other call graph construction algorithms

Although the "direct" contribution of detecting and removing unreachable methods is only 20.4% for the *Hyper/J* benchmark, the removal of unreachable methods may:

- lead to the identification of useless fields,
- lead to the removal of constant pool entries, and
- enable merging of classes and interfaces in the hierarchy.

All previously discussed results are based on the use of the XTA algorithm [46] for identifying unreachable methods. In order to investigate the transitive effects of method removal, we conducted an experiment in which we used Class Hierarchy Analysis (CHA) [13] and Rapid Type Analysis (RTA) [6] instead of XTA to determine a set of reachable methods. Both of these algorithms are less precise than XTA. CHA uses only class hierarchy information to resolve virtual method calls, and RTA uses a single set of instantiated classes in resolving virtual method calls.

Table 4 shows a comparison of the reductions in archive size, and number of classes, methods, and fields we obtained for *Hyper/J* using CHA, RTA, and XTA. These results are depicted in Figure 8. As can be seen from the chart, CHA removes only 47.1% of the methods, in contrast to 59.0% and 59.2% for RTA and XTA, respectively. This results in an archive size reduction of 63.6%, as opposed to 71.8% for RTA, and 72.2% for XTA. Observe that the removal of additional methods has a significant impact on the number classes that can be merged or removed. Extracting *Hyper/J* using CHA results in an archive with 551 classes, compared with 434 classes using RTA, and 428 classes using XTA.

It is clear from Table 4 and Figure 8 that XTA is only marginally more precise than RTA as far as detecting unreached methods is concerned. However, XTA can be significantly more effective than RTA when it comes to detecting redundant call graph edges. A closer look revealed that XTA constructs a call graph with 19,826 edges compared an RTA call graph with 20,754 edges. Upon further examination, we found that the RTA call graph contained 4,592 monomorphic call sites and 2,385 polymorphic call sites. XTA found a unique target for 380 of these RTA-polymorphic call sites, i.e., in 16.1% of all cases. Although *Jax* could not devirtualize many of these call sites due to the constraints imposed on Java byte codes, XTA may enable more devirtualization and inlining when a target representation other than class files is used or when class file annotations are used in conjunction with a JIT. Section 7.2 discusses annotations as future work.

3.7 Download Time Results

Table 5 shows the time required to download the class file archives for each of the benchmarks of Table 1, before and after applying *Jax*. We measured the download time using `ftp`, over a 56K modem connection, as well as over a fast LAN connection.

As could be expected, the reduction between archive size reduction and download time reduction is proportional. For the measurements over modem connections, the reductions in archive size and download time are generally the same within a few percentage points. This is also the case for the larger benchmarks over the LAN connections. The results for the smaller benchmarks over LAN connections are somewhat erratic, which is probably due to the fact

Hyper/J	classes	methods	fields	call graph edges	archive
original size	921	8,776	2,733	N/A	1,523,670
processed with CHA	551	4,640	1,141	30,791	554,091
processed with RTA	434	3,597	924	20,754	429,153
processed with XTA	428	3,584	924	19,826	424,074

Table 4: Comparative measurements for *Hyper/J* using Class Hierarchy Analysis and Rapid Type Analysis.

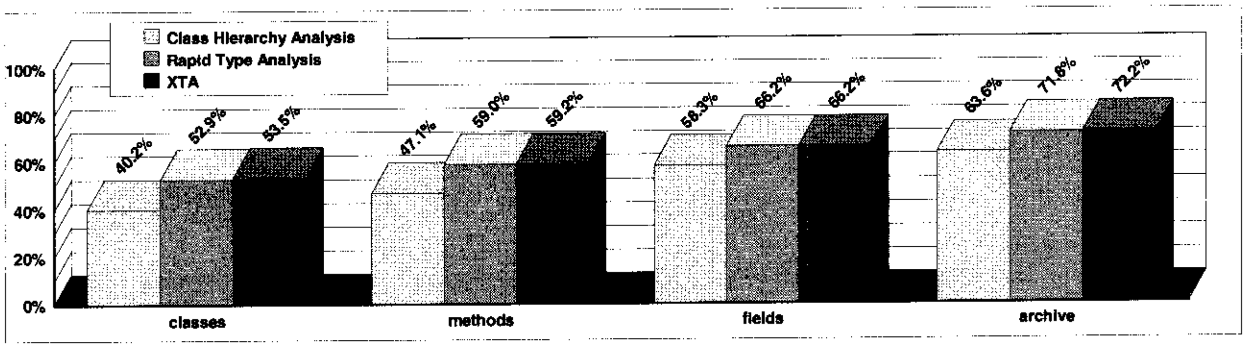


Figure 8: A comparison of the reduction in the number of classes, number of methods, number of fields, and archive size obtained for *Hyper/J* using Class Hierarchy Analysis and Rapid Type Analysis to determine reachable methods.

benchmark	original (modem)	processed (modem)	reduction	original (LAN)	processed (LAN)	reduction
Hanoi	12.0	4.6	61.7%	0.06	0.02	66.7%
Jax	125.2	65.2	47.9%	0.71	0.39	45.1%
javac	88.2	45.8	48.0%	0.57	0.29	49.1%
bloat	103.9	55.8	46.3%	0.56	0.34	39.3%
mBird	675.3	54.2	92.0%	3.48	0.36	89.7%
Jinsight	94.6	63.0	33.4%	0.47	0.42	10.6%
JavaFig	91.2	45.9	49.7%	0.49	0.26	46.9%
CindyApplet	187.8	34.7	81.5%	1.03	0.24	76.7%
Cinderella	187.8	93.5	50.2%	1.03	0.50	51.5%
eSuite Sheet	333.5	90.7	72.8%	1.52	0.48	68.4%
eSuite Chart	368.5	148.6	59.7%	1.67	0.80	52.1%
Hyper/J	386.4	107.9	72.1%	1.99	0.49	75.4%
Res. System	980.5	465.7	52.5%	4.12	1.95	52.7%

Table 5: Time required to download the class file archives before and after running *Jax*, for each of the benchmarks of Table 1. Measurements are shown for downloading over a 56K modem connection (throughput 3.9–5.1KB/sec), and for a fast LAN connection (throughput 0.8–1.0MB/sec). All times shown are in seconds.

benchmark	execution time (original)	execution time (processed)	speedup
Jax	7.43	7.20	3.1%
javac	20.37	19.58	3.9%
mbird	2.02	1.95	3.5%
Hyper/J	5.58	4.78	14.3%

Table 6: Speed-up measurements for the non-interactive benchmarks. Times shown are in seconds.

that the download times are so small that they are difficult to measure.

In addition to reducing download time, the reduction in archive size also has benefits on the server side. Since each data transfer to a client requires less time, more clients can be served in principle.

An interesting statistic is that for most benchmarks, the time required to process the application with *Jax* is *less* than the reduction in download time over 56K modem connections.

3.8 Execution time speed-up

Table 6 shows the running time for the four non-interactive benchmarks (*Jax*, *javac*, *mBird*, and *Hyper/J*) before and after applying *Jax*. The other benchmarks are all interactive GUI-based applications, so that direct speedup measurements are difficult to conduct. For *Jax*, *javac* and *mBird* the speedups are small: 3.1%, 3.9%, and 3.5%, respectively. For *Hyper/J*, the speedup is a more significant 14.3%.

4 Extracting Other kinds of Software Distributions

Thus far, we have only studied the extraction of complete applications. In Section 4.1, we conduct an in-depth analysis of the requirements and design issues associated with the extraction of software distributions other than complete applications. Section 4.2 presents a high-level, uniform solution in the form of a small, modular specification language MEL (Modular Extraction Language) that a programmer uses to provide the information required to extract various kinds of software distributions. Section 4.3 presents a low-level specification language that is used by an extraction tool, and presents a translation from the high-level to the low-level specification language. Section 4.4 discusses how we implemented MEL in the context of *Jax*, and, in particular, how several of the program transformations and optimizations of Section 2 can be adapted to take into account MEL scripts. Section 4.5 presents a small case study in which different extraction scenarios are applied to *Cinderella*, a commercially available library-based Java application.

4.1 Requirements and design issues

In a number of situations, the extraction of software distributions other than complete applications requires information that cannot be obtained using static analysis alone, and that has to be provided to the extraction tool by the user:

- Different kinds of software distributions (e.g., complete applications, web-based applications that execute in the context of a browser, and extensible frameworks) have different sets of entry points, and require the application extractor to make different assumptions about the deployment environment. In fact, the same unit of software may even play different roles, depending on the deployment scenario.

- Modern object-oriented applications typically rely on one or more independently developed class libraries. With the advent of virtual machine technology, library code is amenable to the same analyses as application code, because the same representation is used in each case.

When an application is distributed separately from the libraries it depends upon, an extraction tool needs to be aware of the *boundary* between the two.

- Dynamic features such as *reflection* poses additional problems for extraction tools, because a static analysis alone is incapable of determining the program constructs that are used, and hence the program constructs that can be removed.
- There are also some interesting interactions between the above issues. For example, consider a situation where an application A is to be distributed together with an independently developed class library L in which reflection is used. In general, the use of reflection in L may depend on the *features* in L that are used by A. We will discuss how this observation affects extraction.

We will now investigate each of these issues in more detail.

4.1.1 Roles of software units

We will adopt the term *software unit* to denote any collection of classes that constitutes a logical entity. Recall that there is no difference between code in a class library and code in an executable application, and it is only the way in which software units are *used* and *composed* that determines how extraction should be performed. In the remainder of this paper, the term *role* will be used to refer to the way in which a software unit is used. We will consider four roles that frequently occur in the context of Java:

- An *application* is an executable software unit with an external interface consisting of a single `main()` method. It is assumed that classes in applications are not further extended by derivation after extraction.
- An *applet* is an executable software unit that is executed in the context of a browser. An applet extends class `java.applet.Applet` and its external interface consists of the methods in `java.applet.Applet` that it overrides. It is assumed that classes in applets are not further extended by derivation after extraction.
- A *library* is not assumed to be executable by itself, but is used as a building block by other units. Classes in libraries may be extended by derivation. The external interface of a library consists of any method and any field that has `public` or `protected` access rights.
- A *component* is similar to a library in the sense that it is an incomplete program used as a building block by other units. But, unlike a library, it is assumed that classes in a component cannot be extended by derivation. The external interface of a component contains every method and field with `public` access rights.

Other roles such as JavaBeans [40] and servlets [10] can be modeled similarly.

4.1.2 Specifying the extraction domain

There is no distinction between classes in different software units at the language level. Consequently, it is necessary to specify the "boundaries" between software units when performing extraction. We propose an approach in which the user selects the set of classes that should be extracted, and worst-case assumptions are made about the behavior of classes that are not selected. It is important to realize that the boundary between software units is not merely an issue of avoiding redundant work and shipping redundant code, but potentially also one of correctness. If an application class contains a call to a method in a class that is not extracted, inlining that call on one platform may result in code that does not work on another platform, because the implementations of the class on the two platforms may be different.

In the case of embedded systems and network PC's that run a fixed set of applications it may be desirable to extract all classes that are needed. The extraction of embedded systems applications will be discussed in Section 5.

4.1.3 Dealing with dynamic features

Java's reflection mechanism allows programs to do various forms of self-inspection. Figure 9(a) shows an example program that uses structural

```
import java.io.*;
import java.lang.Class;
import java.lang.reflect.Method;

public class Example1 {
  public static void main(String args[]){
    T t = new T();
    Class c = T.getClass();
    Method[] methods = c.getDeclaredMethods();
    for (int i=0; i < methods.length; i++){
      Method m = methods[i];
      String methodName = m.getName();
      System.out.println(methodName);
    }
  }
};

class T {
  void foo(){ ··· };
  void bar(){ ··· };
};
```

(a)

```
import java.io.*;
import java.lang.Class;

public class Example2 {
  public static void baz(String name){
    try {
      Class c = Class.forName(name);
      Object o = c.newInstance();
      I i = (I)o;
      i.zap();
    }
    catch (ClassNotFoundException e){
      System.out.println("Error:  " +
        "Could not find " + name); }
    catch (IllegalAccessException e){
      System.out.println("Error:  " +
        "Illegal access to " + name); }
    catch (InstantiationException e){
      System.out.println("Error:  " +
        "Abstract " + name); }
  }
};

interface I {
  public void zap();
};
```

(b)

Figure 9: **(a)** Example Java program that uses structural reflection. **(b)** Example Java program that uses dynamic loading.

reflection (sometimes referred to as introspection). In this program, the class that represents the type T of object t is retrieved using a call to method `java.lang.Object.getClass()`, and stored in variable c. The program then calls method `java.lang.Class.getDeclaredMethods()` to obtain a vector of objects representing the methods in T. For each method in this vector, the name is retrieved (by way of a call to `java.lang.reflect.Method.getName()`), and printed to standard output. Hence, the program generates the following output:

```
foo
bar
```

Clearly, program behavior depends on the *presence* and the *name* of the methods in T, even though these methods are not invoked anywhere. It is obvious that this use of reflection precludes program transformations such as the removal or renaming of methods in T because such actions would affect program behavior.

Dynamic loading, another form of reflection, is a heavily-used[14] mechanism for instructing a Java Virtual Machine to load a class X with a specified name s, and return an object c representing that class. Reflection can be applied to c to create X-objects on which methods can be invoked. The crucial issue is that s is computed at *run-time*. This implies that, in general, a static analysis cannot determine which classes are dynamically loaded.[15]

Figure 9(b) shows a program fragment that exhibits a fairly typical use of dynamic loading. Class `Example2` contains a method `baz` which takes a single argument of type `String`, and dynamically loads a class with that name by calling method `java.lang.Class.forName()`. A reference to this class is stored in variable c. The program then calls

[14]Nine of the thirteen benchmarks studied in Section 3 use dynamic loading.

[15]In some cases, the type of a dynamically loaded class can be inferred by constant propagation of the string literals that represent the class name. However, we have observed that these names are often read from files or manipulated in non-trivial ways.

method `java.lang.Class.newInstance()` to create a new object of the dynamically loaded type, casts it down to an interface type `I`, and calls method `zap` on the object. Observe that class instantiation (of the dynamically loaded class) and method invocation (of the default constructor of that class) occur implicitly. This poses problems for optimizations such as dead method removal because the analyses upon which these optimizations are based typically require knowledge about instantiated classes and invoked methods.

Java provides a mechanism for implementing methods in a platform-dependent way, typically using C. The mechanism works roughly as follows: The `native` keyword is used to designate a method as being implemented in a different language, and the corresponding method definition is provided in an object file (e.g., a dynamically linked library) associated with the Java application. The native code in the object file may instantiate classes, invoke methods, and access fields in the application. This obviously poses problems for any program transformation that relies on accurate information about class instantiation and method invocation, because object code is notoriously hard to analyze.

It should be evident from the above examples that, without additional information, the use of reflection, dynamic loading, and native methods requires that *extremely* conservative assumptions be made during extraction: It would essentially be impossible to remove, rename, or transform any program construct. The approach taken in this paper relies on the user to specify a list of program constructs (i.e., classes, methods, and fields) that are accessed using these mechanisms, and to make the appropriate worst-case assumptions about these constructs.

It can be difficult to determine where reflection is used in unfamiliar code, especially if source is unavailable. In order to help users with determining where reflection is used in unfamiliar applications, the Jax distribution includes a simple tool that instruments calls to the reflection API. Running the instrumented application produces a file that lists the classes, methods, and fields that are accessed via reflection in that specific execution. We found this tool to be very effective for finding uses of reflection in unfamiliar applications. Although there is no way to guarantee that all uses of reflection are exposed by this tool[16], in practice it usually suffices to exercise all menus, buttons and other GUI components.

4.1.4 Modeling different usage contexts

Section 1.1 already alluded to issues related to the use of third-party libraries in which reflection is used. In order to create MEL scripts that are *reusable* in different contexts, it is often desirable to specify that a given program construct is only accessed using reflection under certain conditions. To illustrate this issue, Figure 10(a) shows a small class library consisting of three classes `L`, `M` and `N`. Class `L` has two methods: `f` and `g`. A call to `f` results in the dynamic loading of class `M`, and a call to `g` results in the dynamic loading of class `N`. Note that a client that calls `f` but not `g` will only access `M`, and a client that calls `g` but not `f` will only access `N`. A specification of the library's behavior that states that *any* client of `L` accesses both `M` and `N` would clearly be overly conservative. Section 4.2 introduces a mechanism that allows conditional specifications of the form "program construct X should be preserved when method m is executed". This allows one to express how dynamic loading or reflection is dependent on the part of a software unit's functionality that is *used*. Consequently, it enables the creation of a single, reusable configuration file for a software unit that can be used to extract that unit accurately in the context of different clients.

We conclude this discussion with an observation. In Section 1.1, we sketched two very different scenarios involving library L. In one example (the distribution of L_{ext} by l), all externally accessible L-methods should be treated as entry points in determining which methods are reachable. In the other scenario, (the distribution of AL_{ext} by a), only L-methods invoked from A and methods transitively reachable from those methods should be preserved. Hence, the decision on which methods to preserve re-

[16]It is easy to see that even exercising all control-flow paths would not necessarily expose all program components accessed via reflection: the computation of different *run-time* (string) values at a call to `Class.forName()` leads to different classes being dynamically loaded.

```
import java.lang.Class;

public class L {
  public static void f(){
    ...
    Class c = Class.forName("M");
    ...
  }
  public static void g(){
    ...
    Class c = Class.forName("N");
    ...
  }
};
class M { ··· };
class N { ··· };
```

(a)

```
import L;

public class A {
  public static void main(String args[]){
    ...
    L l = new L();
    l.g();
    ...
  }
};
```

(b)

Figure 10: **(a)** Example class library that uses dynamic loading. **(b)** Example application that uses the library of **(a)**.

quires information *not present in the code* of L. This precludes an approach based on annotating the code of L with additional information, unless different annotations are used to support different scenarios.

4.2 A Specification Language

Figure 11 presents a BNF grammar for a simple specification language, MEL (Modular Extraction Language), that allows users to specify at a high level how to extract a library-based application. A MEL script comprises:

1. A *domain specification*, consisting of a class `path` where classes can be found, and a set of `include` statements that specify the extraction domain. Any class not listed in an `include` statement is considered external to our analyses in the sense that it will not be extracted, and that worst-case assumptions will be made about its behavior.

2. A set of *statements*. There are two kinds of statements. *Role* statements designate the role of some or all of the classes included in the extraction domain as `application`, `applet`, `component`, or `library`. The semantics of these roles were discussed earlier in Section 4.1.1. *Preserve* statements specify that program constructs (i.e., classes, methods, and fields) should be preserved because they are accessed either outside of the extraction domain or through reflection, and that worst-case assumptions should be made about these constructs. Following the discussion of Section 4.1.4, program constructs can be conditionally preserved depending on the reachability of a specified method.

3. A list of imported configuration files. The semantics of the `import` feature consist of textual expansion of the imported file into the importing file.

Figure 10(b) shows an example application `A` that uses the library of Figure 10(a). Observe that `A`'s `main()` routine creates an L-object and invokes L's method `g()`. Figure 12 presents MEL scripts *L.mel* and *A.mel* for the library of Figure 10(a) and the application of Figure 10(b), respectively. The conditional preserve statements in *L.mel* ensure that class `M` is preserved if method `L.g()` is reached, and that class `N` is preserved if method `L.f()` is reached. Since `A` only calls method `L.g()`, class `N` will not be extracted.

4.3 Implementation Strategy

The specification language presented in Figure 11 was designed to make it easy for programmers to specify how a collection of software units should be extracted.

MELScript	::=	Item*
Item	::=	DomainSpecifier | Statement | Import
DomainSpecifier	::=	ClassPath | Include
ClassPath	::=	**path** <Directory> | **path** <ZipFile>
Include	::=	**include** <Class> | **include** <PackageName>
Statement	::=	Role | Preserve
Role	::=	**application** <Class> | **applet** <Class> | **library** <Class> | **component** <Class>
Preserve	::=	SimplePreserve | CondPreserve
SimplePreserve	::=	**preserve** <Class> | **preserve** <Method> | **preserve** <Field>
CondPreserve	::=	SimplePreserve **when reached** <Method>
Import	::=	**import** <FileName>

Figure 11: BNF Grammar for the user-level information in MEL

```
path ···
include L
library L
preserve M when reached L.g()
preserve N when reached L.f()
```
(a)

```
path ···
include A
application A
import L.mel
```
(b)

Figure 12: **(a)** Specification *L.mel* for the class library of Figure 10(a) **(b)** Specification *A.mel* for the application of Figure 10(b).

However, the algorithms used by extraction tools typically require low-level information such as methods that are potentially executed, and classes that are potentially instantiated. To bridge the gap between user-level and extractor-level information, we add a number of assertion constructs to MEL, and provide a translation from user-level statements to these assertions. An important benefit of this approach is that all roles and usage scenarios can be treated uniformly by the extractor.

Figure 13 shows a BNF grammar for MEL assertions. The `instantiated`, `reached`, and `accessed` assertions are provided for expressing that a class is instantiated, a method is reached, or a field is accessed, respectively. The `preserveIdentity` assertions express that a program construct may be accessed from outside the extraction domain or accessed through reflection, which implies that the construct's name or signature should not be changed. The `extendible` and `overridable` assertions serve to express that a class may be extended, and that a method may be overridden after extraction, respectively. In Section 4.4, we discuss the impact of the latter two types of assertions on the closed-world assumptions made by optimizations such as call devirtualization.

The `instantiated`, `reached`, `accessed`, and `preserveIdentity` assertions also have a conditional form, which is used to model conditional preserve statements that specify situations where reflection is used in a specific method.

Table 7 shows how statements are translated to assertions. The table shows for each type of MEL statement the assertions generated for that statement. The translation process for roles can be summarized as follows:

- Worst-case assumptions are made to determine a set of methods that can be invoked from outside the extraction domain. Each such method is assumed to be `reached`, and its identity is preserved to indicate that external references may rely on its name and signature. Different roles require different treatment. For example, for `applications`, only the `main()` method is referenced externally and needs to be added

Assertion	::=	SimpleAssertion | ConditionalAssertion
SimpleAssertion	::=	LHS_Assertion
SimpleAssertion	::=	`extendible` <Class>
SimpleAssertion	::=	`overridable` <Method>
LHS_Assertion	::=	`instantiated` <Class>
LHS_Assertion	::=	`reached` <Method>
LHS_Assertion	::=	`accessed` <Field>
LHS_Assertion	::=	`preserveIdentity` <Class>
LHS_Assertion	::=	`preserveIdentity` <Method>
LHS_Assertion	::=	`preserveIdentity` <Field>
CondAssertion	::=	LHS_Assertion `when reached` <Method>

Figure 13: BNF grammar for the extractor-level information in MEL.

statement	derived assertions

`application` C	`preserveIdentity` C
	`reached` C`.main(java.lang.String[])`
	`preserveIdentity` C`.main(java.lang.String[])`

`applet` C	`instantiated` C `preserveIdentity` C `preserveIdentity` $C.m$ for every $C.m$ that overrides `java.applet.Applet.`m `reached` $C.m$ for every $C.m$ that overrides `java.applet.Applet.`m
`component` C	`preserveIdentity` C `preserveIdentity` $C.m$ for every `public` method $C.m$ `reached` $C.m$ for every `public` method $C.m$ `preserveIdentity` $C.f$ for every `public` field $C.f$ `accessed` $C.f$ for every `public` field $C.f$
`library` C	`preserveIdentity` C `extendible` C `reached` $C.m$ for every `public` or `protected` method $C.m$ `preserveIdentity` $C.m$ for every `public` or `protected` method $C.m$ `overridable` $C.m$ for every `public` or `protected` method $C.m$ `accessed` $C.f$ for every `public` or `protected` field $C.f$ `preserveIdentity` $C.f$ for every `public` or `protected` field $C.f$
`preserve` C	`instantiated` C when C is not an interface or an abstract class `preserveIdentity` C
`preserve` $C.m$	`reached` $C.m$ `preserveIdentity` $C.m$
`preserve` $C.f$	`accessed` $C.f$ `preserveIdentity` $C.f$
`preserve` C `when reached` $D.n$	`instantiated` C `when reached` $D.n$ `preserveIdentity` C `when reached` $D.n$
`preserve` $C.m$ `when reached` $D.n$	`reached` $C.m$ `when reached` $D.n$ `preserveIdentity` $C.m$ `when reached` $D.n$
`preserve` $C.f$ `when reached` $D.n$	`accessed` $C.f$ `when reached` $D.n$ `preserveIdentity` $C.f$ `when reached` $D.n$

Table 7: Translation of statements into assertions.

to the set. However, for classes that play a `library` role all `public` and `protected` methods are added.

- For each `role` of a class, the appropriate assumptions are made to determine the fields that may be accessed from outside the extraction domain, and all such fields are asserted to be `accessed`. For example, all `public` fields of `components` are assumed to be accessed.
- Any class that plays an `applet` role is instantiated by the JVM when the applet is started. We model this by asserting that each applet class is `instantiated`.
- For classes that play a `library` role, we have to assume that subclassing and method overriding may take place after extraction. To this end, we assert that the class is `extendible` and all of its `overridable`.

Any program construct referenced in a `preserve` statement receives the `preserveIdentity` assertion, because the identity of program constructs accessed from outside the extraction domain or through reflection should be preserved. Moreover, we make the conservative assumption that a `preserved` class is instantiated if it is not `abstract` or an `interface`. Each preserved method is assumed to be invoked, and is therefore asserted to be `reached`. Similarly, each preserved field is assumed to be `accessed`. The translation of conditional `preserve` statements involves carrying over the condition from the statement to the assertion, but is otherwise completely analogous.

It is hard to make completeness arguments about MEL. In our design of the high-level MEL statements, we have attempted to make it easy for the user to specify commonly occurring extraction scenarios. The low-level MEL assertions are sufficient to ensure that a program construct will not be affected by an extractor. In our implementation, we have given the user direct access to the lower-level MEL assertions as a fall-back option for extraction scenarios that are not currently supported. One instance where this has already been useful is a situation where the main class of an application contained an unaccessed field called "copyright" containing a copyright message. Since these fields are not accessed, an explicit `preserveField` assertion had to be supplied to preserve the field.

4.4 Implementation

We have implemented MEL in the context of *Jax*. In addition to the previously discussed functionality, our implementation has mechanisms for specifying the name of the generated zip file, and for selectively disabling optimizations. *Jax* provides two mechanisms to support MEL. In "batch mode", a MEL script is read from a file, and the application is processed accordingly. A graphical user interface that allows users to create MEL scripts interactively is also provided. We will describe how a number of the program transformations and optimizations of Section 2 can be adapted to operate on various kinds of library-based applications by taking into account MEL assertions. While we do not claim to be the first to adapt these optimizations to library-based programs, we are not aware of any previous systematic treatment of the subject.

4.4.1 Call graph construction

We will now discuss how the XTA algorithm [46] that was described in Section 2 can be adapted to take into account MEL assertions. Other call-graph construction algorithms can be modified similarly. In order to explain the role of MEL assertions in the call graph construction process, it will be necessary to explain the XTA algorithm in some detail. What follows is a high-level overview of XTA; for complete details the reader is referred to [46].

XTA keeps track of a set of reached methods R, and associates a set of types S_m with each method m and a set of types S_f with each field f. R is initialized to contain the methods called from outside the application (e.g., an application's `main()` method), and the sets of types associated with all methods and fields are initially empty. Then, the following steps are performed repeatedly:

- When a method m is found to be reachable, its body is scanned and all call sites and read/write accesses of fields are recorded. Moreover, any type C that is instantiated in m's body is added to set S_m.

- Each call to a method $C.f$ that occurs in the body of method m is resolved with respect to each type D in S_m, where D is a subclass of C. This involves performing a method lookup for f in class D. If the lookup resolves to a method that was not previously reached, it is added to R, and the call graph is updated with edges that reflect the flow of control between caller and callee.
- If a method m writes to a field f with type T, then any subtype of T in S_m is propagated to S_f. Conversely, if a method m reads field f, then any element that occurs in S_f is added to S_m.
- If a method m calls a method m', then any type C in S_m that is a subtype of a type that is used as a parameter of m' is propagated to $S_{m'}$. Conversely, any type D in $S_{m'}$ that is a subtype of the return type of m' is propagated to S_m.

This iterative process continues as long as additional methods, call sites, and instantiated classes are found. In cases where a class C in the extraction domain overrides a method f in a class outside the extraction domain, we conservatively assume that f is called on an object of any type passed into the libraries.

In order to adapt XTA to take into account MEL assertions, R is initialized to contain any method m for which an assertion `reached` m was generated. Moreover, each class C for which an assertion `instantiated` C was generated is added to the sets for all methods and fields. Then, in the iterative part of the algorithm, we add the following additional steps, which are executed when a method m is added to the worklist of reached methods:

- whenever a method m is added to R such that an assertion `instantiated` C `when reached` m exists, C is added to S_m if it does not occur in S_m, and
- whenever a method m is added to R such that an assertion `reached` m' `when reached` m exists, m' is added to R if it does not occur in R.

4.4.2 Removal of dead methods and fields

Dead method removal relies solely on call graph information. No information is needed beyond what was discussed in Section 4.4.1. Dead field removal can be adapted to handle MEL assertions by considering a field $C.f$ to be read-accessed if there exists a `accessed` $C.f$ assertion. Conditional `accessed` assertions can be treated in the same way as conditional `reached` assertions.

4.4.3 Call devirtualization

Call devirtualization [9, 3], the transformation of a dynamically dispatched call site x that calls a method $C.m$ can be transformed into a direct call if only one method can be reached from x, and if method $C.m$ cannot be overridden after extraction of the application. The first condition can be verified by inspection of the call graph, and the second condition is met if there is no assertions `overridable` $C.m$ or `extendible` C, where $C.m$ is the method invoked at call site x.

Other optimizations that rely on closed-world assumptions such as inlining [36] can be adapted similarly. In the presence of MEL assertions, a call to a virtual method $C.m$ for which an assertion `overridable` $C.m$ exists cannot be inlined, because the method may be overridden after extraction, and we conservatively assume that the call site may resolve to these overriding method definitions.

4.4.4 Name compression

The presence of MEL assertions imposes additional constraints on the renaming of program constructs. Any program construct x for which there exists an assertion `preserveIdentity` x cannot be renamed, any method m for which there exists an assertion `overridable` m cannot be renamed, and any class c for which there exists an assertion `extendible` c cannot be renamed.

4.4.5 Method finalization and class finalization

MEL assertions are accommodated by not finalizing any class C for which an assertion `extendible` C exists, and by not finalizing any method m for an assertion `overridable` m exists.

4.4.6 Class hierarchy transformations

In order to accommodate MEL assertions, any class C for which there exists an assertion `preserveIdentity` C should not be removed, or merged into its base class.

4.5 A case study

We have conducted a small case study in which different extraction scenarios are applied to the *Cinderella* benchmark, an interactive geometry tool used for education and self-study in schools and universities. *Cinderella* consists of an application, which can be used for constructing interactive geometry exercises, and an applet in which students can attempt to solve these exercises. Two interesting observations can be made about Cinderella. First, the application and the applet are derived from the same code base, which is contained in a single zip file. Second, `Cinderella` relies on a class library called *Antlr* for parsing.

Table 8 shows different distribution scenarios for *Cinderella.* The first two rows, labeled `Application + Applet (unprocessed)` and `Antlr (unprocessed)` are concerned with the original distributions of *Cinderella* and *Antlr*, respectively. The columns of the table show the size of the zip file, and the numbers of classes, methods and fields, respectively. The next row, labeled `Antlr (processed)` shows the result of extracting *Antlr* as a stand-alone library. The reduction in size was obtained by removing several methods and fields that are only accessible inside the library, and by removing the redundant class file attributes. The next three rows, labeled `Applet (processed)`, `Application (processed)` and `Applet + Application (processed)` shows the size of extracting the application, the applet, and their combination without *Antlr*. Finally, the last three rows, labeled `Applet + Antlr (processed)`, `Application + Antlr (processed)`, and `Applet + Application + Antlr (processed)` show the results of extracting the application, the applet, and their combination with the parts of *Antlr* that they use.

The following observations can be made:

- The applet contains (roughly) a subset of the application's functionality, since the `Application + Applet (processed)` distribution (400,593 bytes) is only marginally bigger than the `Application (processed)` distribution (390,833 bytes).

- On the other hand, the size of the applet (176,397 bytes) is significantly smaller than the combined distribution (400,593 bytes). Hence, users that only require the applet will prefer a distribution containing only that.

- From the fact that the distributions that include *Antlr* are not much bigger than the distributions without *Antlr*, we can infer that *Cinderella* uses only a small subset of *Antlr*'s functionality.

- Interestingly, the distribution of *Cinderella* without *Antlr* (`Application (processed)`, 390,833 bytes) is *larger* than the distribution of *Cinderella* with *Antlr* (`Application + Antlr (processed)`, 385,900 bytes). This is due to the fact that only a small part of *Antlr*'s functionality is used by *Cinderella*, and, more importantly, that *not including Antlr* forces *Jax* to make conservative assumptions about the *Antlr* library that increase the size of the distribution.

- Extracting *Antlr* by itself results in a nontrivial (about 20%) reduction of distribution size. This confirms that extracting stand-alone class libraries is worthwhile.

Cinderella	# classes	# methods	# fields	archive
Application + Applet (unprocessed)	337	3,057	2,391	658,397
Antlr (unprocessed)	130	1,392	684	225,324
Antlr (processed)	130	1,369	677	180,171
Applet (processed)	154	1,225	788	176,397
Application (processed)	265	2,363	1,732	390,833
Applet + Application (processed)	270	2,405	1,742	400,593
Applet + Antlr (processed)	171	1,325	831	179,702
Application + Antlr (processed)	280	2,456	1,773	385,900
Applet + Application + Antlr (processed)	292	2,508	1,786	411,328

Table 8: Results of multiple distribution scenarios for "Cinderella".

5 Application to Embedded Systems

Traditionally, embedded systems have been programmed in languages such as C and assembler because the applications running on these devices were simple and static and had relatively long life cycles. The advent of pervasive computing has given rise to all sorts of new dynamic, networked and complex embedded devices, and a strong interest in programming these devices in Java. Embedded devices have a number special characteristics that make the applications running on these devices excellent candidates for packaging[17]:

- Embedded devices tend to have little RAM, and less powerful CPUs than desktop computers, because of cost, size, and power consumption considerations.
- Embedded devices tend to be diskless, and operating system and applications are stored in ROM or flash memory.
- Battery-powered devices often need to be turned off or hibernated when not in use, but long start-up times are generally not acceptable. Both extraction and the use of an execute-in-place target representation may reduce start-up time.

[17]In the remainder of this section, the term "extraction" will refer to size-reducing program transformations. The term "packaging" will be used to refer to extraction followed by translation to a target execution format.

Several of the present authors have been involved in the development of a packaging tool called SmartLinker, which is part of IBM VisualAge® Micro Edition[18] (VAME), an environment for developing Java applications for embedded systems. SmartLinker incorporates several of the previously presented extraction techniques, and shares a significant amount of infrastructure with *Jax*. SmartLinker collects Java class files and resources and extracts a run-time image that can be stored in ROM. The goals of SmartLinker are both to reduce application size and to improve execution speed. This section outlines the extraction algorithms implemented in SmartLinker, and compares them with the techniques implemented in *Jax*.

5.1 Configuring the packaging process

SmartLinker uses a specification language that is logically equivalent to (but syntactically different from) MEL (see Section 4.2) to configure many aspects of the packaging process. Convenience features such as the use of regular expressions for concisely referring to sets of similar class and method names have been added. We refer the reader to the SmartLinker VisualAge® Micro Edition product documentation [22] for syntactic details on the specification language.

By default, SmartLinker assumes that it is packag-

[18]IBM VisualAge Micro Edition can be downloaded from `www.ibm.com/software/ad/embedded/`.

ing a "closed world" application, and it marks all leaf classes and leaf methods as final. The specification language allows the user to configure which classes should be considered extensible, thereby preventing a class and its visible methods to be marked as final. The run-time verifier enforces these closed-world assumptions at run-time by preventing final classes from being subclassed, and final methods from being overridden.

SmartLinker supports both application libraries that are analyzed and extracted along with the application, and independently packaged prerequisite libraries. In the latter case, a map file is generated that contains for every packaged method:

- references to other methods, fields and classes packaged in the library, and
- references to methods, fields and classes not packaged in the library that need to be resolved if the library method turns out to be reachable.

The created map files serve to support a feature currently not offered by *Jax*. When packaging a dependent application, the map files of the prerequisite libraries can be loaded and the associated information can be used in the call graph construction process (see Section 5.4). The constructed call graphs are often more precise than those constructed by making worst-case assumptions about library behavior.

An additional advantage of having complete reference information available for prerequisite libraries is that it is possible to verify that all required classes, methods and fields are indeed available (except of course for reflective references that the user inadvertently omitted).

5.2 Loading

Two modes of operation are provided to determine the set of classes that are loaded. Similar to *Jax*, SmartLinker can start loading the classes explicitly specified by the user, and load any class transitively referred to by these classes (see Section 2.1). Like *Jax*, SmartLinker relies on the user to specify classes, methods, and fields referenced using reflection, the Java Native Interface (JNI), and dynamic loading.

SmartLinker also provides the option of loading an explicitly specified set of classes. This option is useful in cases where one does not know which program artifacts in a component are accessed using reflection, and in cases where one is not allowed to remove or transform classes due to licensing restrictions.

5.3 Class file attributes

Unlike *Jax*, SmartLinker does not discard bytecode attributes that represent debugging information. The debugging information is (optionally) merged back into the class files generated during the output phase, or written in symbol information files that can be used with the VAME Debug Proxy, a utility that translates addresses in the embedded target execution format back to the standard Java Debug Wire Protocol. This enables remote source-level debugging of extracted applications.

5.4 Call graph construction

SmartLinker uses RTA [6, 5] to construct a call graph. As was mentioned in Section 5.2, the generated map files for prerequisite libraries allow SmartLinker to construct more precise call graphs than *Jax*, since this approach avoids making conservative assumptions about library behavior.

An important practical consideration is that extracting an embedded application using the map file for the complete run-time libraries often results in the inclusion of many methods, fields and classes that are not used by that particular application. This is due to required static initializers and many (required) internal dependencies between classes and packages. The approach chosen to remedy this problem is to create a number of custom run-time libraries that provide different subsets of the functionality provided by the complete run-time libraries. These custom run-time libraries range from a 100 KB "Extreme" custom library to a 3 MB "Max" custom library. It is currently the responsibility of the user to select the smallest custom library that will support a particular application

5.5 Removing redundant methods and fields

Using the call graph constructed in the previous step, unreachable methods are removed, as was described in Section 2.3, as well as unaccessed fields (see Section 2.4). The removal of write-only fields has not been implemented yet in SmartLinker.

5.6 Class hierarchy transformations

Uninstantiated classes without derived classes, and that do not contain any live methods or fields, are removed, as described in Section 2.5. However, the more general class hierarchy transformations implemented in *Jax*, have not yet been implemented in SmartLinker. A complicating factor is that source-level debugging requires a mapping from the transformed hierarchy back to the original source files.

5.7 Name compression

Unlike *Jax*, SmartLinker does not rename packages, classes, methods and fields. The main reason name compression has not been implemented is the fact that strings are already shared in the JXE format, and the few percent extra savings given by name compression did not outweigh the implementation effort.

5.8 Inlining

SmartLinker inlines methods in order to decrease application size and to improve execution speed. SmartLinker is capable of inlining static and instance methods, with their exception ranges, line number tables and local variable tables at selected call sites, as long as visibility constraints are not violated for the inlined code. Similar to *Jax* (see Section 2.6), methods are inlined automatically in cases where it can be determined that the resulting application becomes smaller after inlining a method and removing its call sites removed.

An experimental feedback-directed optimization feature in SmartLinker allows the user to profile one or more application runs and feed the profile data back into SmartLinker [15]. Based on the profile data SmartLinker will inline a user-specified percentage of the "hottest" call sites (the most frequently executed calls). Performance improvement due to the inlining of hot call sites varies greatly, and more investigation of the results is needed.

5.9 Peephole optimizations

Following inlining, various simple peephole optimizations and flow-sensitive optimizations are performed, such as:

- Replacing consecutive identical load instructions with load, dup,
- Removing load, pop combinations,
- Removing jumps to the immediately following instruction,
- Replacing load, const, add, store instruction sequences with iinc,
- Removing no-ops,
- Removing store (load) when there are no more stores (loads) of the same local,
- Removing store instructions that are followed by a return instruction.

Although simple, peephole optimization often has significant benefits in terms of application size and execution speed.

5.10 Ahead-of-time compilation

In general, bytecodes are more compact than equivalent machine code, but interpreted execution is significantly slower. Cost and power consumption constraints on embedded systems usually require that at least part of the application is translated into machine code. Machine code requires fewer CPU cycles, which allows for a smaller processor (cost) and a lower clock rate (power). VAME supports three execution modes that can be mixed in different combinations, depending on the level of support for a particular target platform:

1. Interpreted bytecode execution.
2. Execution of machine code generated from the bytecode at run-time by a just-in-time (JIT) compiler.
3. Execution of machine code generated from the bytecode by the ahead-of-time (AOT) compiler integrated in SmartLinker.

For many embedded systems, AOT compilation is preferred over JIT compilation, because the footprint of a JIT compiler is too high for most embedded systems, and because of the negative effect of JIT compilation on the latency and predictability of the application, which is especially unwanted in "instant-on" type applications. Another advantage of AOT compilation over JIT compilation is that more processing power and time are available for analysis, and because of the closed world assumptions that can be made and enforced at link time. A disadvantage of AOT compilation is that the application loses the platform independence of Java bytecode. An alternative could be to generate platform-independent optimization hints for an embedded JIT compiler to support the generation of high-quality machine code at run-time without the expense of a full-scale optimizing JIT compiler.

Using the feedback-directed optimization feature described in Section 5.8, SmartLinker precompiles the set of methods in which a user-specified percentage of the total measured CPU time is spent. For most applications only a small subset of the methods need to be precompiled in order to obtain nearly all of the performance benefits of AOT compilation [15].

5.11 Conversion to the target execution format

SmartLinker supports two target execution formats. The primary output format is the Java Executable (JXE) format, a format specifically designed for packaging embedded applications in the context of VAME. It is also possible to output standard Java class files. The JXE target execution format has the property that it can be executed in-place by the JVM, which implies that the application can be stored in flash memory or in ROM. The JXE format uses a single, shared representation for strings and other constants, as opposed to using a copy per class in the standard Java class file format. The JXE format also uses significantly less indirection than the standard class file format.

One of the targets platforms supported by VAME is development of applications for Personal Digital Assistants running PalmOS on the Motorola 68K CPU. Packaging an application for an embedded device usually involves an additional conversion to a device-specific format. In the case of PalmOS applications, the PalmOS-specific "PRC" format is commonly used. The `Jxe2Prc` utility that is provided with VAME converts the JXE representation of an application together with Palm resources compiled by the `PilRC` Palm resource compiler into the PRC format. After transferring the generated PRCs to the device (or emulator), the PRC can be executed on the device (possibly using the debug proxy with the SmartLinker-generated debug symbol file for remote source-level debugging).

5.12 An example: packaging a PalmOS application

Table 9 shows some results that were obtained for the "Sticks" PalmOS application, one of the examples shipped with VAME. The Sticks application is a simple multi-threaded application that displays a bouncing line for each stroke made by the user on the display. The three rows at the top of Table 9 pertain to the inputs to the packaging process: the Sticks application itself (5 classes contained in a 10,956 byte JAR file), a small library containing several high-level PalmOS utility classes (13 classes in a 28,727 byte JAR file), and the VisualAge® Micro Edition run-time PalmOS libraries (344 classes in a 1,068,560 byte JAR file), which provides a (small) subset of the functionality of the standard Java class library and a (large) subset of the PalmOS API.

Table 9 also shows the size of the JXE representations of these three archives: 9,544 bytes, 18,117 bytes, and 617,481 bytes, respectively. This brings the total size of the original distribution to 362 classes

benchmark	# classes	size (ZIP/JAR)	size (JXE)	size (PRC)
Sticks	5	10,956	9,544	15,388
PalmOS utility classes	13	28,727	18,117	18,206
VAME PalmOS runtime libraries (complete)	344	1,068,560	617,481	634,196
VAME PalmOS runtime libraries (subset)	78	134,081	83,002	99,626
Sticks (packaged, no reduction/optimization)	63	1,108,243	240,748	246,624
Sticks (packaged, reduction, no optimization)	12	20,583	12,751	18,596
Sticks (packaged, reduction + optimization)	10	16,724	10,999	16,844

Table 9: Results of running SmartLinker on the PalmOS "Sticks" application. The JAR files in this figure contain uncompressed entries.

and 645,142 bytes[19]. Since PalmOS devices typically only have a few megabytes of storage, it is clear that some reduction in footprint is needed.

The next row of the table (labeled "VAME PalmOS runtime libraries (subset)") refers to a custom runtime library that contains a subset of the functionality of the complete runtime libraries. This custom library contains 78 classes (JXE size: 83,002 bytes) and packages the classes that are required by nearly every PalmOS application. In order to avoid unnecessary duplication of this functionality across applications, the developers of VAME have chosen to share this custom library between every packaged PalmOS application. Hence, an extracted PalmOS application A generally requires: (i) the classes associated with A itself, (ii) the 78 run-time classes in the custom run-time library, (iii) additional classes of the run-time libraries used by A, and (iv) any resources required by A. This approach for packaging library functionality is illustrated by Figure 14.

The three rows at the bottom of Table 9 illustrate the impact of some of the steps in the packaging process. Packaging the application without any removal of unused classes, methods, and fields, and without any optimization results in 63 classes, and a PRC size of 246,624 bytes. Specifically, these 63 classes consist of the 5 classes of the Sticks application itself, 1 utility class, and 57 classes from the run-time libraries that do not occur in the custom library. The significant increase in size is due to the fact that some of the classes that Sticks refers to are very large (and includes, among other things, a 128KB class containing all 1200+ PalmOS function wrappers). However, it is clear that not much functionality in these classes is actually used, since removal of unused classes (i.e., uninstantiated classes that do not contain live methods or fields), methods, and fields results in an application with 12 classes, and a PRC size of 18,596 bytes. Inlining and peephole optimizations further reduce the application to 10 classes (the 5 Sticks application classes, 1 utility class, and 4 run-time classes not in the custom library), and a PRC size of 16,844 bytes. The removal of two classes in this last step is due to the fact that all calls to the methods in these classes are inlined away. In particular, most PalmOS function wrapper methods are eliminated completely and replaced at the call site by a call to the native `trap()` method that is the wormhole provided in the Micro Edition run-time to all PalmOS functions.

The bottom row of Table 9 shows the final result: a PRC file of 16,844 bytes. Adding the custom runtime libraries of 99,626 bytes and the VAME JVM run-time system itself of 86,018 bytes brings the total footprint to just under 200KB.

6 Related Work

6.1 Historical perspective

Several of the techniques incorporated into *Jax* borrow from previous work that some of the present authors were involved in. *Jax* implements the RTA [6, 5] and XTA [46] call graph construction algorithms. Accurate treatment of methods that override methods

[19]The PRC distribution also contains native start-up and library code, and resources (application icons) that are needed.

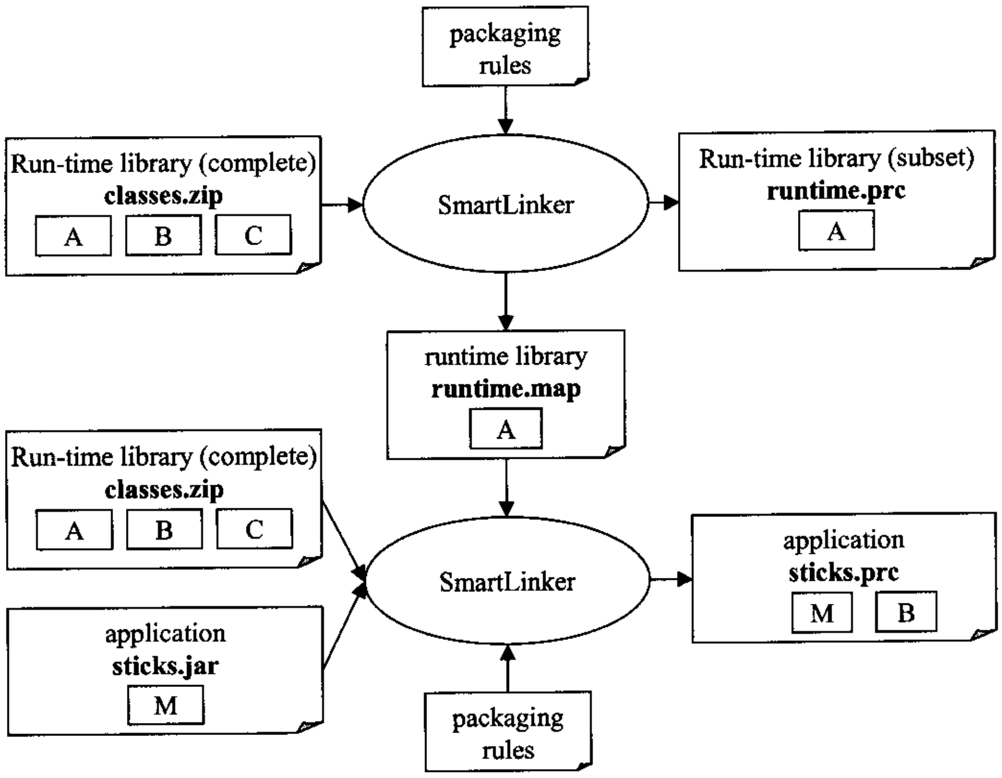


Figure 14: Schematic view of the SmartLinker approach for packaging the VAME PalmOS run-time libraries. The commonly used library classes are pre-packaged in a shared PRC distribution, and other library classes are packaged in another PRC distribution along with the application.

in external class libraries is very important for reducing archive size due to the importance of class libraries in Java. The detection of useless fields, including write-only fields, was previously studied by Sweeney and Tip [42], and Tip et al. [45] for C++. In this work, RTA was used to construct a call graph, and an average of 12.5% useless fields was measured for a set of C++ applications. In the context of *Jax*, we found an average 55.5% of all fields to be useless. We conjecture that this difference is partly due to the pervasive use of class libraries in Java, and that these libraries tend to contain a lot of unused functionality. Furthermore, the larger percentage of unaccessed fields in Java applications could be due to the fact that Java lacks a macro facility and that Java programmers use static final fields to define constants. Java compilers propagate these constants so that no accesses to these fields remain, but the fields themselves are not removed.

The class hierarchy transformations used by *Jax* were originally proposed in the context of specializing class hierarchies [47]. The goal of this work is to remove members from objects. Ignoring a number of details, the specialization algorithm constructs a new class hierarchy in which a new class is constructed for each variable and each member in the program. Inheritance relations between these classes reflect member access relationships between variables and class members, and subtype relationships between variables that must be retained to preserve program behavior. The class hierarchy transformations used in *Jax*were introduced in order to reduce the complexity of the resulting class hierarchy.

6.2 Application extraction for other languages

The extraction of applications was pioneered in the Smalltalk community, where it is usually referred to as "packaging" [21, 14, 29]. Smalltalk packaging tools typically have mechanisms for excluding certain standard classes and objects from consideration, and for forcing the inclusion of objects and methods. While the latter mechanism is sufficient to handle programs that use reflection, we are not aware of any Smalltalk extractor that models different types of applications, or that provides a feature to preserve certain program constructs conditionally.

Agesen and Ungar [2, 1] describe an application extractor for the Self language that eliminates unused slots from objects (a slot corresponds to a method or field). In his PhD thesis [1, page 146], Agesen writes that there is no easy solution to dealing with reflection other than "rewriting existing code on a case by case basis as is deemed necessary" and suggests "encouraging programmers writing new code to keep the limitations of extraction technology in mind". In contrast, we allow the user to specify where reflection occurs, so that applications that use reflection can be extracted.

Chen et. al [11] describe Acacia, an extraction tool for C/C++ based on a repository that records several relationships between program entities. Several types of reachability analyses can be performed, including a forward reachability analysis for determining entities that are unused. Chen et al. identify several issues that make extraction difficult such as the use of libraries for which code is unavailable, and situations where functionality should be preserved because source modules are shared with other applications. Unlike our work, Acacia is an analysis tool aimed at providing information to the user, and does not actually perform any program transformations such as dead code elimination. A number of issues that we study such as the use of reflection are not discussed, and no mechanism appears to be available for supplying additional information to the extractor.

6.3 Extraction and obfuscation tools for Java

DashO-Pro[20] is the only tool we are aware of that aims at archive size reduction and that goes beyond simple name compression. We are unfamiliar with the algorithms used by DashO-Pro, and it is unclear to us how DashO-Pro compares to *Jax* in terms of size reduction and processing speed. An interesting feature of DashO-Pro is its name compression algorithm which apparently attempts to reuse the same name as many times as it can. This includes using

[20]See `www.preemptive.com`.

the same name for classes, methods, and fields where possible.

Several other Java tools aim at obfuscation (i.e., to make decompilation of class files into understandable source code more difficult). In addition to DashO-Pro, hashjava[21], SourceGuard[22], ObfuscatePro[23], Jshrink[24], Jmangle[25], and JZipper[26] perform some form of compression of class names, method names, field names, and package names. Of these tools, only DashO-Pro and SourceGuard go beyond simple name compression, and perform other transformations such as modifying an application's control flow. We are not aware of any published work on the algorithms used in any of these tools, nor on their internal architecture.

Rayside et al. [32] present a technique that uses an entity-relationship dependency graph for extracting embedded systems applications. A crucial difference with our work is that Rayside et al. do not use call graph information as the basis for detecting unused program constructs, and only remove program constructs that are not referred to. The techniques by Rayside et al. are much less accurate than ours, since they are incapable of removing anything that is referred to from dead code. Unlike *Jax*, this work does not explore the extraction of software distributions other than complete applications.

Thies [43, 44] presents a static analysis of class libraries where the goal is to compute "summary" information that concisely represents the side-effects of method invocations on actual parameters. This information can be used by a JIT at run-time to detect additional opportunities for optimizations such as common subexpression elimination, and loop invariant code motion. The analysis information that Thies computes is similar in spirit to the information captured in our MEL configuration files, although it is different in the sense that it is computed automatically. Thies does not discuss how he handles the use of reflection and dynamic loading in class libraries. It appears that Thies uses Class Hierarchy Analysis as the basis for computing the side-effects of methods that call other methods. It would be interesting to investigate how much precision can be gained by using a more sophisticated call graph construction algorithm.

6.4 Alternative representations for Java class files

Recently, Pugh [31] and Horspool et al. [20, 7] proposed alternative, more space-efficient representations for Java class files. These representations rely on techniques such as the use of a global constant pool, efficiently representing names that share a common prefix, and separating different streams of information (e.g., opcodes and operands) and compressing the resulting streams separately. Similar "wire-format" representations that rely on creating streams of opcodes and operands that can be compressed separately were explored previously by Ernst et al. [16] for compressing x86 machine code. Pugh reports archives that range between 17% and 49% of the size of the original representation for a representative set of benchmarks. Pugh also evaluates the Jazz representation by Horspool et al. on his benchmarks, and measures archives ranging in size from 31% to 181% of the original representation. An important advantage of these representations is the enabling of sharing of information between different class files. *Jax* can only introduce a limited amount of sharing by merging classes. On the other hand, application extractors can achieve significant size reductions by eliminating unused methods, classes, and fields, which is not addressed by compression techniques. Therefore, one would expect application extraction and more efficient class file representations to be largely "orthogonal" techniques for reducing application size.

In order to verify this conjecture, we converted the original class file archives of Table 1 and the archives produced by *Jax* as shown in Table 2 to the Jazz representation [7], and to Pugh's Packed representation [31]. We also converted these archives to the JXE format used in SmartLinker that was discussed

[21] See `www.meurrens.org/ip-Links/Java/CodeEngineering/blackDown/hashjava.htlm`.

[22] See `www.4thpass.com`.

[23] See `www.jammconsulting.com`.

[24] See `www.e-t.com`.

[25] See `www.elegant-software.com/software/jmangle`.

[26] See `www.www.vegatech.net/jzipper`.

benchmark	JXE (original)	JXE (processed)	Pugh (original)	Pugh (processed)	Jazz (original)	Jazz (processed)
Hanoi	69,229	33,706	14,051	6,989	26,384	11,225
Jax	621,039	487,832	113,644	92,351	280,830	173,150
javac	411,631	337,944	77,233	64,813	164,441	115,896
bloat	497,995	375,564	88,354	70,642	252,187	129,617
mBird	2,921,538	410,381	505,585	85,184	467,219	160,055
Jinsight	526,474	437,191	99,804	83,875	207,027	184,537
JavaFig	481,361	343,658	92,819	67,853	220,183	144,543
CindyApplet	1,083,673	282,666	197,243	51,839	518,779	105,367
Cinderella	1,083,673	585,048	197,243	106,699	518,779	246,654
eSuite Sheet	1,950,291	519,192	570,522	114,513	644,734	214,344
eSuite Chart	2,349,267	869,856	654,357	187,728	811,585	410,801
Hyper/J	1,538,894	584,878	287,028	106,016	714,396	207,894
Res. System	4,101,554	2,338,994	735,801	430,188	2,581,288	1,443,430

Table 10: Results of converting the archives of Tables 1 and 2 to the JXE format (see Section 5), Pugh's packed representation, and to the Jazz representation. For each of the benchmarks, we show the size of the original archive, and the size the archive produced by Jax (both after conversion).

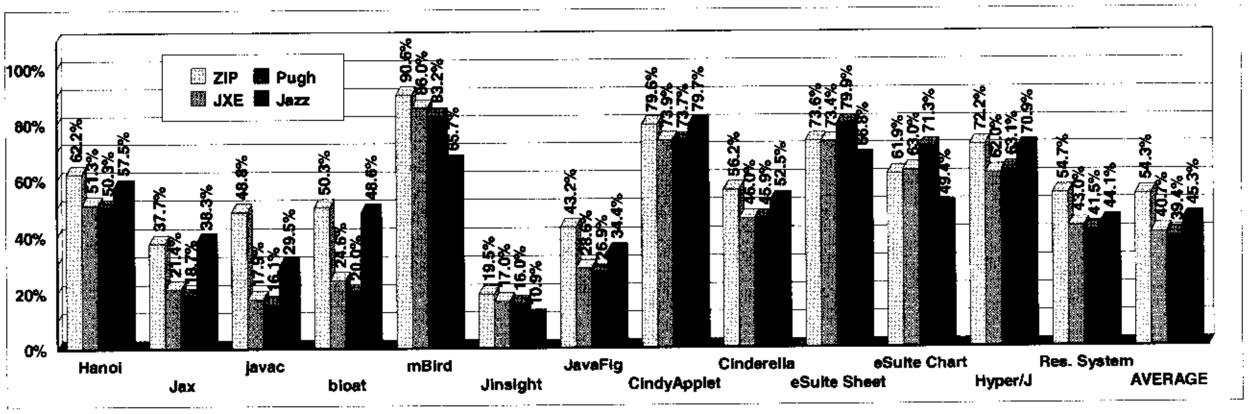


Figure 15: Comparison of the archive size reductions obtained with Jax in each of the four class file representations under consideration. For each benchmark, the four bars indicate, from left to right, the percentage reduction using the standard class file representation, the percentage reduction using the JXE format, the percentage reduction using Pugh's representation, and the percentage reduction using the Jazz representation.

in Section 5. Table 10 shows the archive sizes for the original application and the corresponding extracted application in ZIP files, JXE files, Pugh's packed representation, and the Jazz representation.

Figure 15 depicts the percentage reduction in each of the four representations. In summary, the figure shows that we measured size reductions:

- ranging from 19.5% to 54.3% (average: 54.3%) using ZIP file,
- ranging from 17.0% to 86.0% (average: 40.7%) using the JXE representation,
- ranging from 16.0% to 83.2% (average: 39.4%) using Pugh's representation, and
- ranging from 10.9% to 79.7% (average: 45.3%) using the Jazz representation.

These results demonstrate that application extraction clearly remains a highly useful size reduction technique when alternative representations are used, and that the benefits of application extraction are largely independent from the class file representation that is used.

A different approach is taken by Franz and Kistler [24, 25] who, instead of representing Java byte-codes more efficiently, propose a new intermediate representation called slim binaries as an alternative to Java byte-codes. This representation is based on high-level, abstract syntax tree and encoded using an adaptive compression scheme.

Rayside et al. [33] proposed a new and smaller interpretable format for Java binaries instead of compressing the existing structure by focusing on the constant pool and code array. Unlike the work of Pugh and Horspool et al. but like *Jax*, this representation does not require decompression before execution. Nevertheless, the new interpretable format requires either a slightly modified virtual machine or a customized class loader.

Clausen et al. [12] explored an alternative approach to compression of Java byte codes that aims at reducing the memory footprint of low-end embedded systems applications. The approach by Clausen et al. replaces frequently recurring sequences of byte code instructions by new "macro" instructions, and requires minor modifications to a VM to recognize and expand these macros at run-time. Similar techniques were previously applied to x86 machine code by Ernst et al. [16]. Clausen et al. report a memory footprint reduction of 15%, and a speed penalty that varies between 2% and 30%. It is important to realize that the work by Clausen et al. only addresses compression of instruction sequences, and not, for example, compression of constant pool entries.

Krintz et al. [26] study an approach for reducing application start-up time using an alternative, less strict execution model in which an application starts executing even as parts of it (classes and methods) are still being downloaded. The benefits of this work are likely to be orthogonal to those of application extraction.

7 Conclusions and Future Work

7.1 Summary of contributions

We implemented a number of application extraction techniques such as the removal of redundant methods and fields, inlining of method calls, class hierarchy transformations, and renaming of program entities in a practical extraction tool named *Jax*. We evaluated the effectiveness of *Jax* on a set of large benchmark applications, and measured an average size reduction of 54.3%. More dramatic size reductions of well over 70% are common in situations where class libraries are distributed along with an application, and where the library code can be treated as if it were part of the application.

Extracting software distributions other than complete applications requires additional user input to define the role(s) played by a set of classes, to delineate the boundary between the application and any external classes it depends upon, and to define the (conditional) use of reflection. We have presented a uniform solution in the form of MEL, a small, modular specification language for providing the information required to extract various kinds of software distributions. MEL was implemented in the context of *Jax*, and we have presented a case study to illustrate the practicality of extraction of various kinds of software distributions.

Another domain where extraction techniques are highly useful is the packaging of applications deployed on embedded devices. Several of the extraction techniques that were prototyped and validated in *Jax* have been applied successfully in VisualAge® Micro Edition SmartLinker and make a crucial difference for scaling Java to embedded systems. We have discussed several important issues that arise in the embedded systems domain, such as the support for partial (pre-)linking of libraries, profile-driven ahead-of-time compilation of bytecode into machine code for the most frequently executing methods, and carefully designed custom class libraries that obtain the best results for a particular type of embedded application.

7.2 Future Work: Optimizations of Embedded Applications

We are currently investigating the use of the XTA algorithm [46] in packaging embedded systems applications, where it may enable more inlining and interprocedural optimizations. In the embedded systems setting, the standard class libraries are available for analysis, and using XTA may result in a bigger improvement over other call graph construction algorithms.

Other plans for future work in the area of optimization for embedded systems applications include various interprocedural optimizations of the generated code such as object inlining, interprocedural liveness analysis, and demand-driven context-sensitive analysis driven by profiling feedback. We have already conducted some initial experiments with an approach in which we continue to generate optimized bytecodes, using a translation to a register-based intermediate representation (IR) [8], followed by regeneration of bytecodes from this IR (similar to [30]). However, bytecode-level optimizations have their limitations. For example, there are many cases where a given `invokevirtual` call site always resolves to a specific method, but where the call cannot easily be devirtualized because the use of the `invokespecial` bytecode is restricted to certain situations [27]. An alternative approach would be to store analysis information as attributes in class files. A JIT could be adapted to recognize this information and exploit it by performing more on-line optimizations, or by making existing optimizations more efficient.

Other optimization opportunities exist in the area of flow-sensitive bytecode-level and IR-level optimizations. These include more sophisticated peephole optimizations, dead code elimination, and constant folding.

7.3 Future work: interactive program development tools

We plan to incorporate the analysis performed by *Jax* to visualize unnecessary program components in an interactive program development environment. One can easily imagine the benefits of a tool that could inform the user about unreachable methods and unaccessed fields, enabling the user to determine if these components are simply redundant, or unreachable as the result of a bug. Such a tool could also provide hints to the user about classes and methods that can be declared final. Other software engineering applications that can benefit from analysis information such as call graphs include change impact analysis [35], refactoring [17], and regression test selection [34].

Acknowledgements

Lisa Martin and Sarvamangala Jagadeesh are gratefully acknowledged for their help with the development of *Jax*. The bug reports and comments from the many people who participated in the *Jax* discussion group on alphaWorks have been invaluable. We are grateful to Robert Berry, John Field, Harold Ossher, Ramalingam, Vivek Sarkar, Gregor Snelting, Alan Stevens, and Robert Weir for their constructive comments and feedback. We are also grateful to Bill Pugh, Quetzalcoatl Bradley, Nigel Horspool, and Jan Vitek for making their class file compression tools available.

More information and downloading *Jax*

More information about the *Jax* project can be found at: `www.research.ibm.com/JAX`. A free evaluation copy of *Jax* (along with documentation and examples) can be downloaded from: `www.alphaWorks.ibm.com/tech/JAX`.

References

[1] AGESEN, O. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, Dec. 1995. Appeared as Sun Microsystems Laboratories Tech. Rep. SMLI TR-96-52.

[2] AGESEN, O., AND UNGAR, D. Sifting out the gold: Delivering compact applications from an exploratory object-oriented programming environment. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)* (Portland, OR, 1994), pp. 355–370. *ACM SIGPLAN Notices* 29(10).

[3] AIGNER, G., AND HÖLZLE, U. Eliminating virtual function calls in C++ programs. In *Proc. of the Tenth European Conf. on Object-Oriented Programming (ECOOP'96)* (Linz, Austria, July 1996), vol. 1098 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 142–166.

[4] ARNOLD, K., AND GOSLING, J. *The Java Programming Language*, second edition ed. Addison-Wesley, 1997.

[5] BACON, D. F. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, Computer Science Division, U. of California, Berkeley, Dec. 1997. Report No. UCB/CSD-98-1017.

[6] BACON, D. F., AND SWEENEY, P. F. Fast static analysis of C++ virtual function calls. In *Proc. of the Eleventh Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, Oct. 1996), pp. 324–341. *ACM SIGPLAN Notices* 31(10).

[7] BRADLEY, Q., HORSPOOL, R. N., AND VITEK, J. Jazz: An efficient compressed format for java archive files. In *CASCON'98* (November/December 1998), pp. 294–302.

[8] BURKE, M. G., CHOI, J.-D., FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M. J., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings of the ACM Java Grande Conference* (San Francisco, CA, June 1999).

[9] CALDER, B., AND GRUNWALD, D. Reducing indirect function call overhead in C++ programs. pp. 397–408.

[10] CALLAWAY, D. R. *Inside Servlets: Server-Side Programming for the Java Platform.* Addison-Wesley, 1999.

[11] CHEN, Y.-F., GANSNER, E. R., AND KOUTSOFIOS, E. A C++ data model supporting reachability analysis and dead code detection. *IEEE Trans. on Software Engineering 24*, 9 (Sept. 1998), 682–694.

[12] CLAUSEN, L. R., SCHULTZ, U. P., CONSEL, C., AND MULLER, G. Java bytecode compression for low-end embedded systems. *ACM Trans. on Programming Languages and Systems 22*, 3 (May 2000), 471–489.

[13] DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. of the Ninth European Conf. on Object-Oriented Programming (ECOOP'95)* (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77–101.

[14] DIGITALK INC. *Smalltalk/V for win32 Programming,* 1993. Chapter 17: "Object Libraries and Library Builder.

[15] EISMA, A. Feedback directed ahead-of-time compilation for embedded Java applications. In *Java Optimization Strategies for Embedded Systems* (Genova, Italy, April 2001), U. Assmann, Ed., pp. 105–112. Workshop held at ETAPS'01.

[16] ERNST, J., EVANS, W., FRASER, C., LUCCO, S., AND PROEBSTING, T. Code compression. In *Proc of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '07)* (Las Vegas, NE, June 1997), pp. 358–365.

[17] FOWLER, M. *Refactoring.* Addison-Wesley, 1999.

[18] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification.* Addison-Wesley, 1996.

[19] GROVE, D., DEFOUW, G., DEAN, J., AND CHAMBERS, C. Call graph construction in object-oriented languages. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)* (1997), pp. 108–124. *SIGPLAN Notices* 32(10).

[20] HORSPOOL, R. N., AND CORLESS, J. Tailored compression of java class files. *Software—Practice and Experience 28*, 12 (1998), 1253–1268.

[21] IBM CORPORATION. *IBM Smalltalk User's Guide*, version 3, release 0 ed., 1995. Chapter 36: Introduction to Packaging, Chapter 37: "Simple Packaging, Chapter 38: "Advanced Packaging.

[22] IBM CORPORATION. *VisualAge® Micro Edition 1.4 Reference Manual*, 2001.

[23] ISHIZAKI, K., KAWAHITO, M., YASUE, T., TAKEUCHI, M., OGASAWARA, T., SUGANUMA, T., ONODERA, T., KOMATSU, H., AND NAKATANI, T. Design, implementation, and evaluation of optimizations in a Just-In-Time compiler. In *Proc. of the ACM Java Grande Conf.* (San Francisco, CA, June 1999), pp. 119–128.

[24] KISTLER, T., AND FRANZ, M. Slim binaries. *Communications of ACM 40*, 12 (Dec. 1997), 87–94.

[25] Kistler, T., and Franz, M. A tree-based alternative to java byte-codes. *International Journal of Parallel Programming 27*, 1 (Feb. 1999), 21–34.

[26] Krintz, C., Calder, B., Lee, H. B., and Zorn, B. G. Overlapping execution with transfer using non-strict execution for mobile programs. In *Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, October 1998), pp. 159–169.

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

[28] Pande, H. D., and Ryder, B. G. Data-flow-based virtual function resolution. In *Proc. of the Third Int. Symposium on Static Analysis (SAS'96)* (September 1996), pp. 238–254. Springer-Verlag LNCS 1145.

[29] ParcPlace Systems. *ParcPlace Smalltalk*, objectworks release 4.1 ed., 1992. Section 16: Deploying an Application, Section 28: Binary Object Streaming Service.

[30] Pominville, P., Qian, F., Vallée-Rai, R., Hendren, L., and Verbrugge, C. A framework for optimizing java using attributes. In *Proc. of the International Conf. on Compiler Construction (CC'2001)* (Eisenstadt, Austria, 2001), R. Wilhelm, Ed., pp. 334–354. Springer-Verlag LNCS 2027.

[31] Pugh, W. Compressing java class files. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (Atlanta, GA, May 1999), pp. 247–258.

[32] Rayside, D., and Kontogiannis, K. Extracting Java library subsets for deployment on embedded systems. In *Proc. of the 3rd IEEE Conf. on Software Maintencance and Re-engineering (CSMR '99)* (Amsterdam, Mar. 1999), pp. 102–110.

[33] Rayside, D., Manas, E., and Hons, E. Compact Java binaries for embedded systems. In *Proc. of the 9th NRC/IBM Centre for Advanced Studies Confernece (CASCON '99)* (Toronto, CA, Nov. 1999), pp. 1–14.

[34] Rothermel, G., and Harrold, M. J. A safe, efficient regression test selection technique. *ACM Trans. on Software Engineering and Methodology 6*, 2 (April 1997), 173–210.

[35] Ryder, B. G., and Tip, F. Change impact analysis for object-oriented programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)* (Snowbird, UT, June 2001).

[36] Scheifler, R. W. An analysis of inline substitution for a structured programming language. *Commun. ACM 20*, 9 (Sept. 1977), 647–654.

[37] Shapiro, M., and Horwitz, S. Fast and accurate flow-insensitive points-to analysis. In *Conf. Record of the Twenty-Fourth ACM Symp. on Principles of Programming Languages* (Paris, France, 1997), pp. 1–14.

[38] Shivers, O. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, CMU, May 1991. CMU–CS–91–145.

[39] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, January 1996), pp. 32–41.

[40] Sun Microsystems. *JavaBeans*, version 1.01 ed. 2550 Garcia Avenue, Mountain View, CA 94043, July 1997.

[41] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., and Godin, C. Practical virtual method call resolution for Java. In *Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)* (Minneapolis, Minnesota), 2000), pp. 264–280. *ACM SIGPLAN Notices* 35(10).

[42] Sweeney, P. F., and Tip, F. A study of dead data members in C++ applications. In *Proc. of the ACM SIGPLAN'98 Conf. on Programming Language Design and Implementation* (Montreal, Canada, June 1998), pp. 324–332. *ACM SIGPLAN Notices* 33(6).

[43] Thies, M. A closer look at inter-library dependencies in Java-software. In *JIT'99 Java Informations-Tage 1999* (1999), Informatik Aktuell, Springer Verlag.

[44] Thies, M. Static compositional analysis of libraries in support of dynamic optimization. Technischer Bericht tr-ri-99-210, University of Paderborn, Aug. 1999.

[45] Tip, F., Choi, J.-D., Field, J., and Ramalingam, G. Slicing class hierarchies in C++. In *Proc. of the Eleventh Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 179–197. *ACM SIGPLAN Notices* 31(10).

[46] Tip, F., and Palsberg, J. Scalable propagation-based call graph construction algorithms. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)* (Minneapolis, MN, 2000), pp. 281–293. *SIGPLAN Notices* 35(10).

[47] Tip, F., and Sweeney, P. Class hierarchy specialization. In *Proc. of the Twelfth Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)* (Atlanta, GA, 1997), pp. 271–285. *ACM SIGPLAN Notices* 32(10).

[48] Tip, F., and Sweeney, P. Class hierarchy specialization. *Acta Informatica 36* (2000), 927–982.