Practical Experience with an Application Extractor for Java

Frank Tip Chris Laffra Peter F. Sweeney
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
{tip,laffra,pfs}@watson.ibm.com

David Streeter
IBM Toronto Laboratory
1150 Eglinton Ave. East
Toronto, Ontario, Canada
daves@ca.ibm.com

Abstract

Java programs are routinely transmitted over low-bandwidth network connections as compressed class file archives (i.e., zip files and jar files). Since archive size is directly proportional to download time, it is desirable for applications to be as small as possible. This paper is concerned with the use of program transformations such as removal of dead methods and fields, inlining of method calls, and simplification of the class hierarchy for reducing application size. Such "extraction" techniques are generally believed to be especially useful for applications that use class libraries, since typically only a small fraction of a library's functionality is used. By "pruning away" unused library functionality, application size can be reduced dramatically. We implemented a number of application extraction techniques in *Jax*, an application extractor for Java, and evaluate their effectiveness on a set of realistic benchmarks ranging from 27 to 2,332 classes (with archives ranging from 56,796 to 3,810,120 bytes). We report archive size reductions ranging from 13.4% to 90.2% (48.7% on average).

1 Introduction

Java[1] [10] programs are routinely transmitted over the internet as compressed class file archives (i.e., zip files and jar files). A typical example of this situation consists of downloading a web page that contains one or more embedded Java applets. The downloading of class file archives is increasingly often the distribution mechanism of choice for stand-alone Java applications as well (especially for "network computers"). Since the time required to download an application is proportional to the size of the archive, it is desirable for the archive to be as small as possible.

In this paper we evaluate the effectiveness of a number of compiler-optimization and program transformation techniques for extracting Java applications[2]. These transformations include:

- removal of redundant methods and fields,
- devirtualization and inlining of method calls,
- transformation of the class hierarchy, and
- renaming of packages, classes, methods and fields,

and have the effect of reducing application size. Application extraction is generally believed to be especially useful when an application is shipped with a (proprietary) class library, because typically only a small fraction of the library's functionality is used. In such cases, "pruning away" unused library functionality can dramatically reduce application size.

We implemented a number of application extraction techniques in the context of *Jax* (short for Jikes Application eXtractor). *Jax* reads in the class files [15] that constitute a Java application, and performs a whole-program 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* removes the unnecessary components, performs several size-reducing transformations to the application, and writes out a class file archive containing the extracted application. *Jax* relies on user input to specify the components of the application that are accessed using Java's reflection mechanism [3], but the extraction process is fully automatic otherwise. *Jax* has been available on IBM's alphaWorks web site[3] since June 1998 and has been downloaded over 10,000 times since then. We are planning to ship *Jax* as a Technology Preview with an IBM product (IBM VisualAge Java 3.0, Enterprise Edition) later this year.

We evaluate the performance of *Jax* on a set of real-life benchmarks ranging from 27 to 2,332 classes (the corresponding archives range from 56,796 to 3,810,120 bytes), and measure a reduction in archive size ranging from 13.4% to 90.2% (48.7% on average[4]). Measurements over modem and LAN connections confirm that download times are reduced proportionally.

[1] Java is a trademark of Sun Microsystems.

To appear in the Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), Denver, Colorado, USA, November 1–5, 1999.

[2] In what follows, the word 'application' will be used to refer to applications as well as applets, unless otherwise stated.

[3] www.alphaWorks.ibm.com/tech/JAX

[4] All average percentages reported in this paper are computed using the geometric mean.

Recently, Pugh [18] and Bradley et al. [6, 11] designed alternative, more space efficient representations for Java class files. We repeated our experiments using Pugh's and Bradley et al.'s class file representations, and measured average size reductions for our benchmarks of 33.1% and 42.2%, respectively (detailed information about these measurements is presented in Section 4). This shows that the benefits of application extraction are largely independent from the class file representation that is used.

Although the primary motivation for developing *Jax* has been to reduce archive size, and therewith download time, there are a number of additional benefits:

- Execution speed may be improved due to several of the optimizations we perform (in particular, inlining, devirtualizing, and removal of redundant fields). We measured a speed improvement ranging from 0.4% to 5.4% percent for the non-interactive benchmarks.
- By eliminating unused components (in particular, classes) from applications, the amount of initialization time spent by a Java virtual machine may be reduced.
- The removal of redundant components may reduce an application's memory requirements, by decreasing the amount of space required for storing the application itself, and the amount of memory allocated by the application at run-time due to the removal of redundant fields from objects. In other words, it may be possible to run the application on a "smaller" machine after extraction.
- It is a well-known fact that Java class files can be decompiled into Java source code fairly easily. Although extracting an application does not prevent decompilation, several of the transformations and optimizations we perform make the source code resulting from such decompilation harder to understand, and less useful.

The remainder of this paper is organized as follows. Section 2 discusses the optimizations and transformations used in *Jax*. Section 3 presents the results of running *Jax* on a realistic set of benchmarks. Section 4 discusses related work. Section 5 outlines ongoing activities and plans for future work.

2 Overview of approach

We will now present a high-level overview of the most significant transformations and optimizations that are incorporated into *Jax*.

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. Superfluous bytecode attributes such as local variable name tables and line number tables (debugging information) are discarded at this point.

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. Since these strings are computed at run-time, it is in general undecidable to determine using static analysis which classes are dynamically loaded by an application. Therefore, *Jax* must rely on the user to specify all classes that are dynamically loaded, and list these in a configuration file. Such classes are treated as additional entry points for the class loading process. In Section 2.7, we describe tool support for determining where dynamic loading and reflection are used.

```
interface I {
  public void f();
}
class A implements I {
  int x, y, z;
  public void f() {int j=x; g();}
  public void g() {y=0; }
}
class B extends A {
  public void f() {y=0; z=0; int j=z;}
}
public class program {
  public static void main(String argv[]){
    I i = new B();
    i.f();
  }
}
```

Figure 1: Example program.

2.2 Finding reachable methods and fields

In general, not all of the methods and fields in the loaded classes are required to run an application. In order to determine which methods are reachable, *Jax* constructs a call graph by conservatively approximating the "target" methods that can be invoked by a dynamic dispatch. Additional entry points such as initializer methods are also taken into account during call graph construction.

Various algorithms have been proposed to determine the potential targets of a dynamic dispatch, including Class Hierarchy Analysis [8], and algorithms for alias or points-to analysis [16, 19, 20]. *Jax* uses a variation of Rapid Type Analysis [5, 4] (RTA) to resolve virtual calls. RTA uses global class instantiation information in conjunction with class hierarchy information.

We will use the example program of Figure 1 to illustrate RTA. The program contains classes `A` and `B` and an interface `I` such that `A` implements `I`, `B` inherits from `A`, and `A` declares three fields `x`, `y`, and `z`. The program contains one call site (a dynamically dispatched call to `I.f`) and creates one object, of type `B`. What are the reachable methods in this program? Class Hierarchy Analysis [8] would determine both `A.f` and `B.f` as potential targets since classes `A` and `B` both provide overriding definitions for `I.f`. RTA uses the fact that class `A` is not instantiated *anywhere in the program* to rule out `A.f` as a potential target. Therefore, method `A.f` is not a reachable method. Method `A.g` is unreachable because the call site in the body of `A.f` is not reachable.

Unfortunately, not all unreachable methods can simply be removed. Method `A.g` can be removed without any problem, but removing method `A.f` would lead to an invalid class file, because class `A` promises to implement interface `I`, but it would not contain a definition of method `f`, which is specified in the interface. In such cases, we remove the *body* of the method and replace it with a return statement.

The elimination of methods can lead to the elimination of fields. In our example, `x` is only accessed from `A.f`, and since `A.f` 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 since their value cannot affect the program's behavior [21]. In

addition to removing the field itself, this involves removal of the instructions that store the value in the field. In our example, field `A.y` is write-only, and can be eliminated.

Figure 2 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 2(a) shows the original program. Figure 2(b) shows the program after removing unreachable method `A.g`, removing the body of unreachable method `A.f`, removing unaccessed field `A.x`, and removing write-only field `A.y`.

The use of class libraries 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 f in L. Then, a virtual dispatch inside the library 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 and implemented library method may be called. However, in the RTA style, we use global class instantiation information to rule out certain targets.

Java's reflection mechanism [3] further complicates the task of determining reachable methods, because it essentially allows one to invoke a method by specifying its name as a string (computed at run-time) and its signature. Since 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. Constructors of dynamically loaded classes are treated as entry points as well.

2.3 Class hierarchy transformations

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

- Merging classes may enable the transformation of virtual method calls into direct method calls.
- Merging classes may reduce of the duplication of literals across the constant pools of different classes (this will be discussed in more detail in Section 2.6).

The class hierarchy transformations used in *Jax* are an adaptation of the ones in [23, 24], where they serve as a simplification phase after the generation of specialized class hierarchies (the relationship with this work will be discussed in Section 4).

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. A more interesting transformation is the *merging* of classes that are adjacent in hierarchy under certain conditions. A base class B and a derived class C are merged if there is no live non-`abstract` method f that occurs in both B and C, and one of the following conditions holds:

1. B is uninstantiated, *or*
2. C 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., contains more fields) as a result of the merge.

Merging a base class B with a derived class C involves a number of steps. All live methods and fields of C are moved to B. Cases where B and C have fields or static methods with identical names pose no problem, since we can simply rename any field or static method in cases where name conflicts occur. Constructors required special treatment because B and C 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 B and C both have static initializer methods, we merge the initializer for C into the initializer for B[5]. If B and C both contain an instance method f, then at least one these methods must be `abstract` due to conditions (1) and (2) above. If both methods f are abstract, $C.f$ is simply removed. Otherwise, the non-abstract method is preferred over the abstract method. Finally, all references to class C, as well as methods and fields in C are updated to reflect their new "location" in class B.

Other class hierarchy transformations include the merging of classes with interfaces, and are very similar to the transformation described above. Due to space limitations, we will not present these transformations in detail here. 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 public. Finally, classes that are explicitly referenced using Java's reflection mechanism cannot be merged with other classes.

Figure 2(c) shows the example program after merging class `B` into class `A`. Note that the '`new B`' statement in method `program.main` is changed into '`new A`'. Another issue of interest is the fact that this class merging operation could not have been applied to the original class hierarchy, since both classes `A` and `B` contained a non-abstract methods `f` originally. Hence, this class merging operation was *enabled* by the removal of unreachable method `A.f`. Figure 2(d) shows the program after merging interface `I` into class `A`. Note that the type of variable `i` in method `program.main` is changed from `I` to `A`.

2.4 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*. *Jax* inlines non-overridden methods whose only function is to set or retrieve a field's value. 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 [15] only permits this in a very limited number of situations: the invoked method has to occur in a superclass of the current method's class, or has to be a private method. Finally, *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 1 illustrates how transformations can enable each other. We have already seen

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

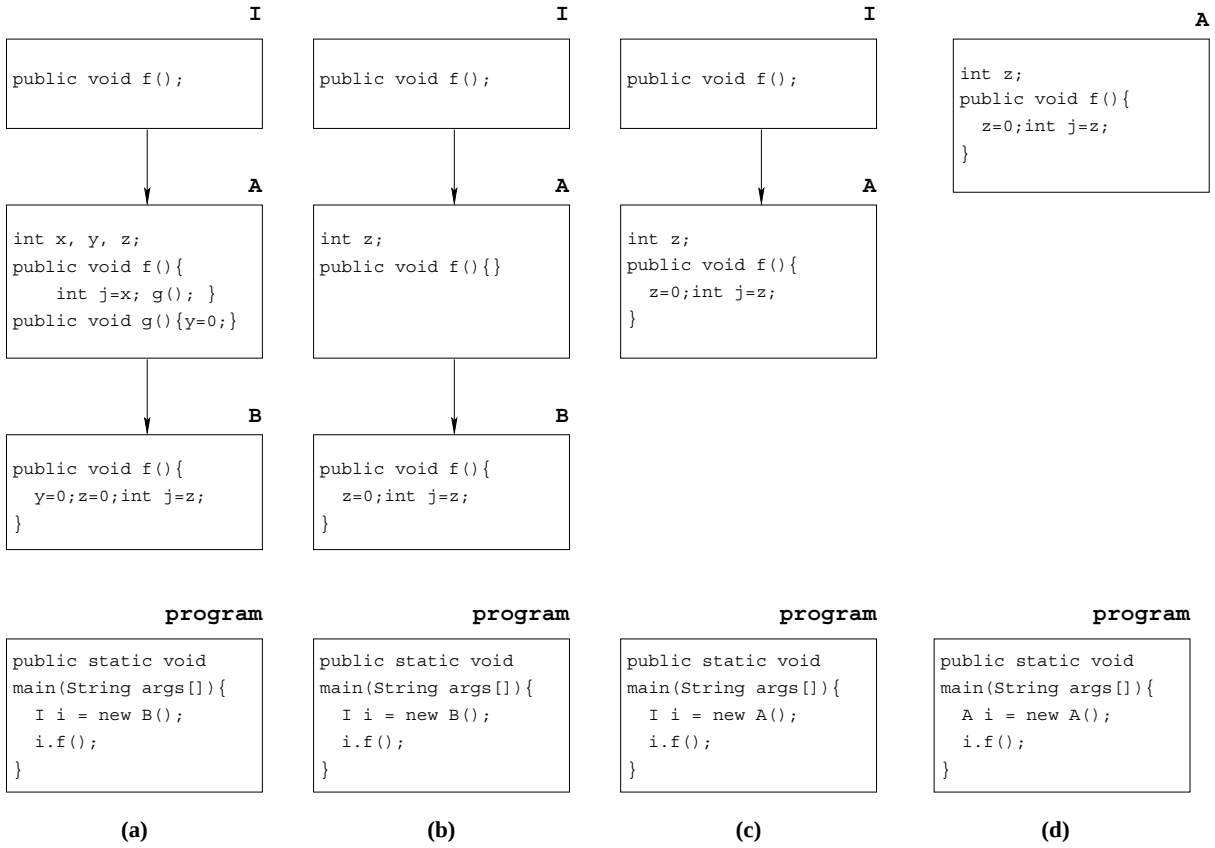

Figure 2: Successive steps in transformation of the example program of Figure 1 by *Jax*. **(a)** the original program, **(b)** the program after removing unreachable method `A.g`, removing the body of unreachable method `A.f`, removing unaccessed field `A.x`, and removing write-only field `A.y`. **(c)** the program after merging class `B` into class `A`. **(d)** the program after merging interface `I` into class `A`.

how elimination of (the body of) unreachable methods can enable class merging. Note that, in the resulting program of Figure 2(d) only a single method `f` remains in the entire class hierarchy. This implies that the call to `f` can be devirtualized, and that `f` can subsequently be inlined.

2.5 Name compression

A Java class file is a self-contained unit of executable code. References to other classes, methods, and fields are made through literal strings. For example, suppose that a class makes the method call `Thread.sleep(500)`. Then, the constant pool would contain the strings "`java/lang/Thread`", "`sleep`", and "`(J)V`". These strings represent the fully qualified class name, the method name, and a string representation of the method's signature (one argument of type `long`; returning `void`). Since 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 envisioned eventually.

Certain names cannot be compressed. In particular, any reference to a class, method, or field in an external library cannot be changed. Methods that override methods in classes or interfaces in external libraries cannot be renamed. Any class, method, or field accessed using Java's reflection mechanism cannot be changed. The name of the main class (the applet class, or the class containing the `main` routine) and the `main` method cannot be changed, and neither can the names of constructors and static initializers be changed.

2.6 Constant pool compression

When methods are removed by *Jax* entries in the constant pools of class files may be rendered unnecessary. In the in-memory representation of class files constructed by *Jax*, references to constant pool entries are replaced by explicit references to actual 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.

Early Java compilers were a little careless with the constant pool and added entries to it that were not referenced from the class file at all. Such entries are not recreated by *Jax*.

2.7 Dealing with reflection

We already discussed the repercussions of the use of Java's reflection mechanism in several places. Certain uses of reflection are relatively harmless. For example, we encountered a program that used reflection to retrieve the name of an object's class, and subsequently load an image (`.gif`) file with the same name. This case is harmless because *Jax* makes the conservative assumption that the name of any class that is accessed by reflection should not be changed.

Other scenarios include various forms of self-inspection. For example, the reflection mechanism could be used to determine the number of methods that occur in a given class. If the program's behavior depends on this number in any way, none of the methods in that class can be removed, even if they are known to be unreachable. Since static analysis cannot determine which classes, methods, and fields are accessed via reflection, *Jax* relies on the user to specify these components by listing them in a configuration file. *Jax* conservatively assumes that these components cannot be changed in any way (i.e., renamed, removed, or merged).

In order to assist the user with the process of determining where dynamic loading and other uses of reflection are used, the *Jax* distribution provides a simple tool that instruments calls to the reflection API. Running the instrumented application will produce a log file that lists the classes, methods, and fields that are accessed via reflection in that specific execution of the program. We found this tool to be a very effective for finding uses of reflection in unfamiliar applications.

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 *Mockingbird* 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.

IBM Host-on-Demand is a terminal emulator that is shipped with Netscape Communicator 4.x. *Hanoi* is an interactive applet version of the well-known "Towers of Hanoi" problem, and is shipped with *Jax*. *ICE Browser*[6] is a simple internet browser. *Jax* itself (version 5.3) was used as a benchmark. *JavaFig*[7] (version 1.43 (22.02.99)) is a Java version of the `xfig` drawing program. *jMark20*[8] is well-known Java performance benchmark. *javac*[9] is the SPEC JVM 98 version Sun's `javac` compiler. *Cinderella*[10] is an interactive geometry tool used for education and self-study in schools and universities. *Cinderella Applet* 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*[11] is an interactive spreadsheet applet, which is part of the examples shipped with Lotus' *eSuite* (DevPack 1.5 version). *Lotus eSuite Chart* is an interactive charting applet, which is another example shipped with Lotus *eSuite*. *Mockingbird* 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 JDK 1.2, and IBM's XML parser). *Reservation System* is an interactive front-end for an airline, hotel, and car rental reservation system developed by an IBM customer.

[6] See `www.icesoft.no`.
[7] See `tech-www.informatik.uni-hamburg.de/applets/javafig`.
[8] jMark is a trademark of Ziff-Davis. See `www.zdnet.com/zdbop`.
[9] See `www.specbench.org`.
[10] See `www.cinderella.de`.
[11] See `www.esuite.lotus.com`.

benchmark	# classes	# methods	# fields	archive size
IBM Host-on-Demand	27	470	1,052	131,190
Hanoi	44	379	232	56,796
ICE Browser	76	761	500	106,081
JAX	101	1,342	790	214,362
JavaFig	161	2,111	1,526	394,432
JMark 2.0	196	723	1,221	186,808
javac	210	1,512	1,107	452,125
Cinderella	468	4,449	3,075	891,552
Cinderella Applet	468	4,449	3,075	891,552
Lotus eSuite Sheet	580	7,321	4,351	1,264,419
Lotus eSuite Chart	730	8,276	5,433	1,570,569
Mockingbird	2,050	17,944	6,738	2,950,543
Reservation System	2,332	21,508	12,493	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.

benchmark	# classes	# methods	# fields	archive size	processing time
IBM Host-on-Demand	26	422	426	82,100	8
Hanoi	21	183	100	21,528	4
ICE Browser	64	651	430	91,891	9
JAX	94	898	267	128,257	12
JavaFig	115	1,456	1,112	231,783	20
JMark 2.0	76	387	492	87,432	8
javac	198	1,342	499	230,981	21
Cinderella	327	2,745	1,968	468,748	62
Cinderella Applet	249	1,984	1,542	312,623	45
Lotus eSuite Sheet	302	3,125	1,169	383,233	59
Lotus eSuite Chart	391	4,497	2,162	594,999	80
Mockingbird	237	1,980	757	289,308	30
Reservation System	1,409	11,641	5,337	1,732,576	412

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.

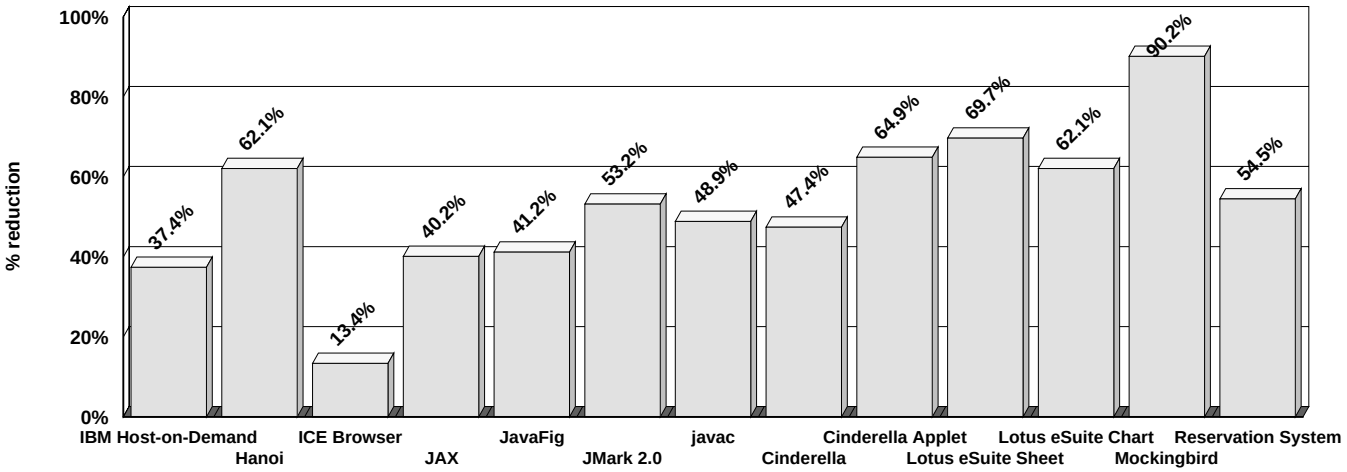

Figure 3: Percentage reduction in archive size for the benchmark applications of Table 1. Resource files are excluded from both the initial and processed archives.

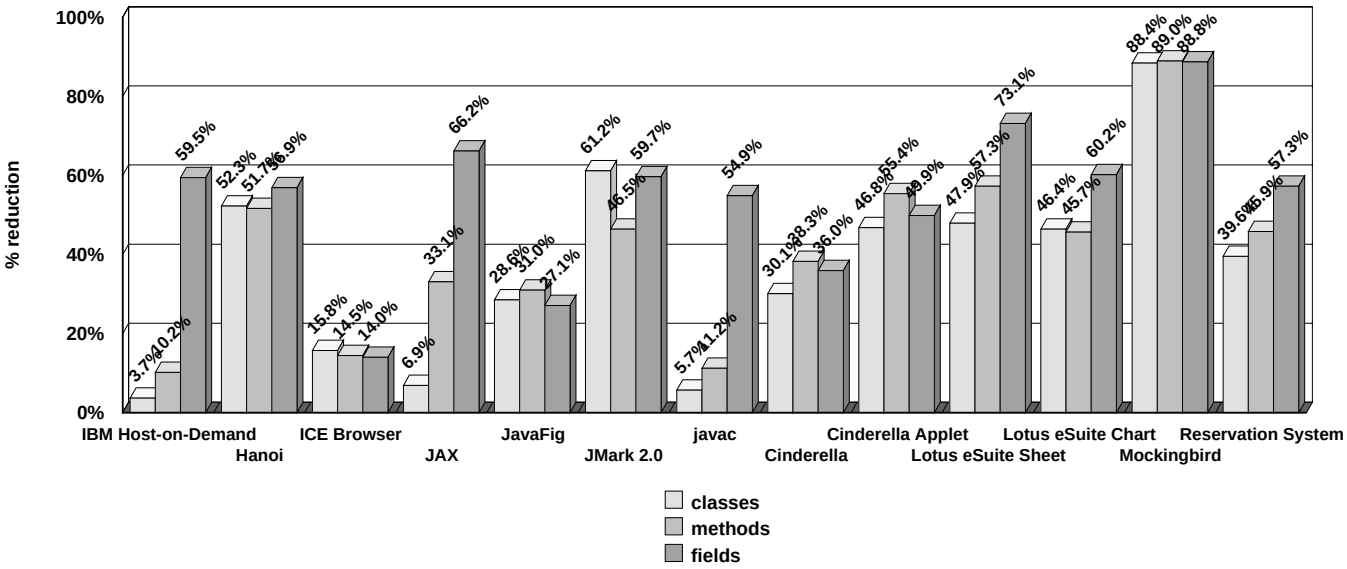

Figure 4: Percentage reduction in numbers of classes, methods, and fields for the benchmark applications of Table 1.

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 (both the original archives and the archives produced by *Jax*) 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 5.3) 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 7 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 3 depicts the percentage by which the archive size is reduced for each benchmark. As can be seen from the figure, the reduction ranges from 13.4% to 90.2% (48.7% on average). Figure 4 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 3.7% to 88.4%, the number of methods by 10.2% to 89.0%, and the number of fields by 14.0% to 88.8%. The average percentage reductions for classes, methods, and fields are 25.9%, 34.4%, and 49.7%, respectively.

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 *ICE Browser*. Analysis of this application revealed that the class files of this program had already been processed by some other obfuscation/extraction tool. In particular, we observed that many of the names the class files were obviously generated—consecutive fields and methods had names consisting of consecutive unprintable characters. The fact that *Jax* manages to further reduce the size of this application implies that it may perform more sophisticated analyses and transformations than the tool used to process *ICE Browser*.

The benchmark for which we measured the highest reduction in archive size, *Mockingbird*, is a special case as well. *Mockingbird* 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 reduction we measured is that *Jax* is very effective in removing the unused GUI-related library classes from batch component. Other benchmarks for which we measure large reductions such as *Hanoi* with 62.1%, *Lotus eSuite Sheet* with 69.7%, and *Lotus eSuite Chart* with 62.1% 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* con-

[12] See www.winzip.com.

[13] Measurements taken on a Pentium II/300Mhz PC with 128MB memory, using the just-in-time compiler developed at the IBM Tokyo Research Laboratory [13].

Lotus eSuite Sheet	classes	methods	fields	archive
original size	580	7,321	4,351	1,264,419
loaded	487	5,658	2,017	775,676
dead methods removed	487	3,335	2,017	559,897
dead fields removed	487	3,335	1,169	544,746
inlining/devirtualizing	487	3,202	1,169	542,449
class transformations	302	3,125	1,169	464,777
obfuscation	302	3,125	1,169	383,233

Table 3: Detailed measurements for the *Lotus eSuite Sheet* benchmark.

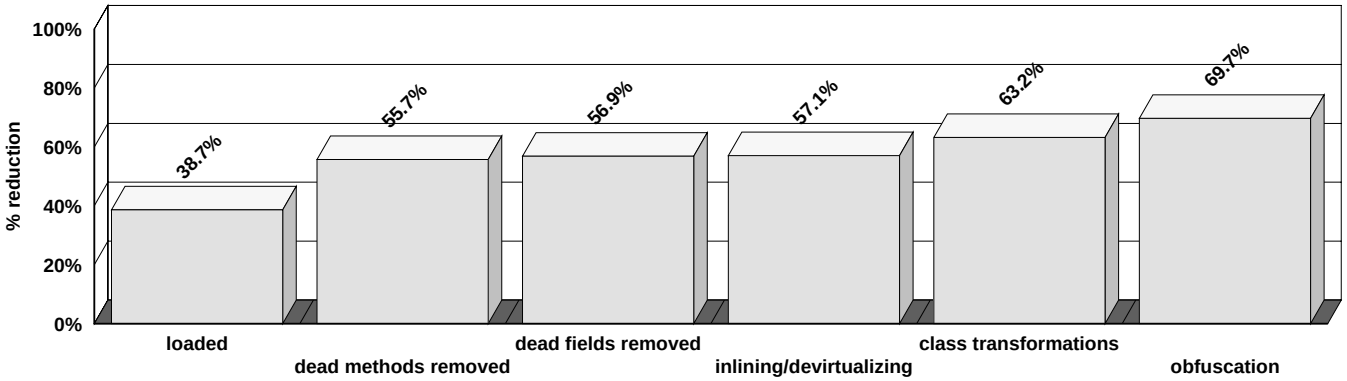

Figure 5: Percentage reduction in archive size for *Lotus eSuite Sheet* w.r.t. the original archive after (1) loading, (2) removal of unreachable methods, (3) removal of useless fields, (4) method inlining/devirtualizing, (5) class hierarchy transformations, and (6) name compression.

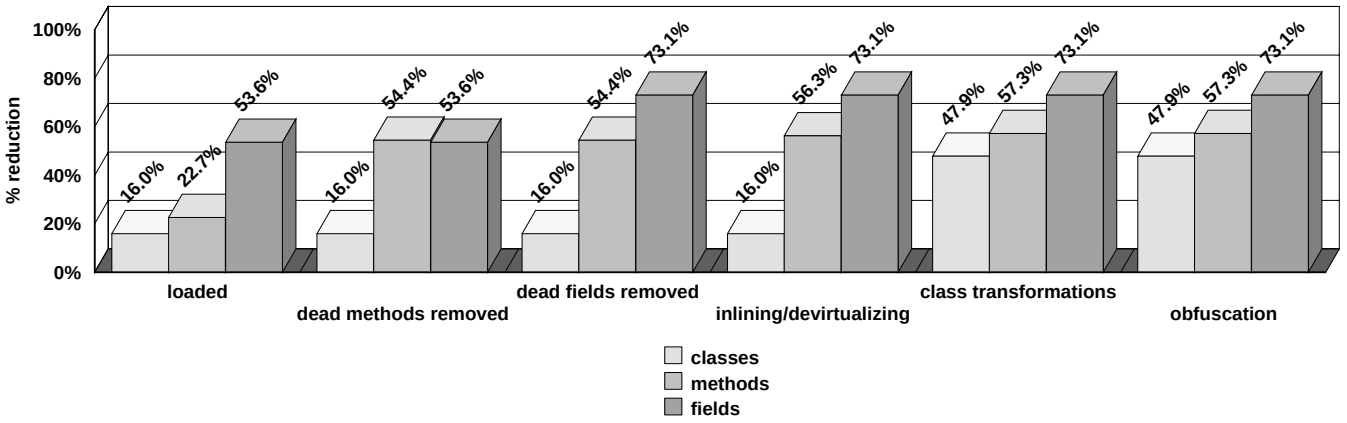

Figure 6: Percentage reduction in the number of classes, methods, and fields w.r.t. the original archive for *Lotus eSuite Sheet* after (1) loading, (2) removal of unreachable methods, (3) removal of useless fields, (4) method inlining/devirtualizing, (5) class hierarchy transformations, and (6) name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

Lotus eSuite Sheet	classes	methods	fields	archive
original size	580	7,321	4,351	1,264,419
processed by JAX using CHA	417	4,408	1,765	563,029
processed by JAX using RTA	302	3,125	1,169	383,233

Table 4: Comparative measurements for *Lotus eSuite Sheet* using Class Hierarchy Analysis and Rapid Type Analysis.

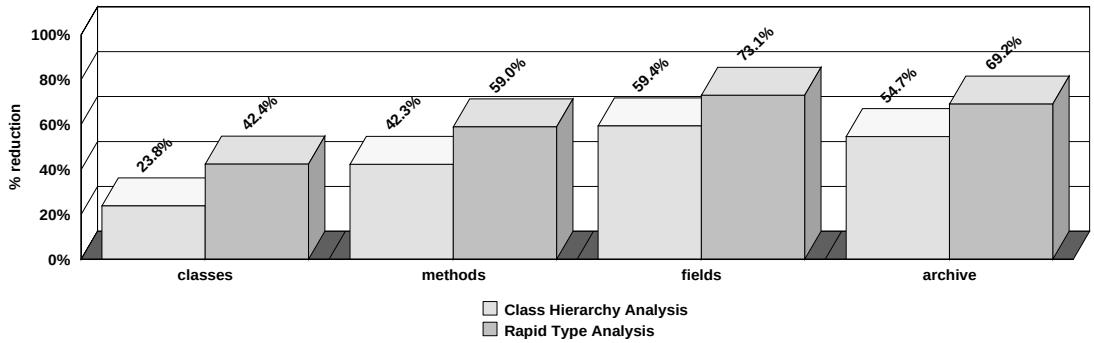

Figure 7: A comparison of the reduction in the number of classes, number of methods, number of fields, and archive size obtained for *Lotus eSuite Sheet* using Class Hierarchy Analysis and Rapid Type Analysis to determine reachable methods.

tains (roughly) a subset of *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* (64.9% archive size) than for *Cinderella* (47.4%).

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 from within 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 *Lotus eSuite Sheet* benchmark, while Figures 5 and 6 show the contributions of each successive steps in a graphical form.

Figure 5 reveals a number of interesting facts. The initial archive contains a substantial number of classes that are not loaded. Removal of these unreferenced classes has the largest effect on the overall result: 38.7%. Most of these unused classes are library classes that are shipped with, but not used by the application. Removal of unreachable methods is the next biggest contributor to the overall result: 17.0%. The contribution of useless field removal is relatively small: 1.2%. Method inlining and devirtualizing have a positive effect on the result (0.2%) because we currently only inline very small methods. Inlining such a method does not increase code size at call sites for the method, and in cases where all call sites for the method can be inlined, the method can be removed.

The contribution of class hierarchy transformations is 6.1%. From Table 3 and Figure 6 it can be seen that the class hierarchy transformations remove an additional 1.0% of the methods (77 methods). All of these methods are `abstract` methods which disappear as a result of class merging. Name compression reduces the resulting archive by another 6.5%.

3.6 An experiment

Although the "direct" contribution of detecting and removing unreachable methods is only 17.0% for the *Lotus eSuite Sheet* benchmark, the removal of unreachable methods has a number of indirect effects:

- The identification of more unreachable methods may lead to the identification of more useless fields.
- The identification of more unreachable methods may lead to the removal of more entries from the constant pool.
- The removal of more methods and fields may enable more merging of classes and interfaces in the hierarchy.

All previously discussed results are based on the use of Rapid Type Analysis 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) [8] instead of RTA to determine a set of reachable methods. CHA is less powerful than RTA because it only uses class hierarchy information to resolve virtual method calls, whereas RTA uses instantiated class information as well. Table 4 shows a comparison of the reductions in archive size, and number of classes, methods, and fields we obtained for *Lotus eSuite Sheet* using Class Hierarchy Analysis and Rapid Type Analysis. These results are depicted in Figure 7. The difference between the resulting archive sizes is pronounced: The archive produced using CHA is 563,029 bytes, whereas the archive produced using RTA is only 383,233 bytes. This is due to the fact that RTA is capable of identifying 1283 more unreachable methods and 596 more useless fields than CHA. This, in turn, led to the elimination of an additional 115 classes.

The conclusion we draw from this experiment is that the detection of unreachable methods is key to obtaining smaller archives. In Section 5, we discuss our current work on more sophisticated algorithms for detecting unused methods.

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

benchmark	original (modem)	processed (modem)	reduction	original (LAN)	processed (LAN)	reduction
IBM Host-on-Demand	27.3	16.9	38.1%	0.170	0.110	35.3%
Hanoi	12.0	4.9	59.1%	0.060	0.020	66.7%
ICE Browser	21.8	18.5	14.9%	0.130	0.110	15.4%
JAX	42.1	25.4	39.7%	0.260	0.160	38.5%
JavaFig	83.7	45.5	45.6%	0.480	0.280	41.7%
JMark 2.0	37.2	17.5	53.0%	0.220	0.100	54.5%
javac	93.9	44.7	52.4%	0.550	0.270	50.9%
Cinderella	189.3	90.3	52.3%	1.210	0.630	47.9%
Cinderella Applet	189.3	60.3	68.1%	1.130	0.380	66.4%
Lotus eSuite Sheet	254.0	74.2	70.8%	1.530	0.460	69.9%
Lotus eSuite Chart	314.4	118.4	62.3%	2.070	0.740	64.3%
Mockingbird	584.2	55.8	90.4%	3.750	0.360	90.4%
Reservation System	753.3	330.1	56.2%	4.780	2.140	55.2%

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 up to 5.0 KB/sec), and for a fast LAN connection (average throughput around 800 KB/sec). All times shown are in seconds.

benchmark	execution time (original)	execution time (processed)	savings
JAX	82.9	81.2	2.1%
javac	68.2	67.9	0.4%
Mockingbird	49.1	46.4	5.4%

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

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 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 three non-interactive benchmarks (*Jax*, *Mockingbird*, and *javac*) before and after processing them with *Jax*. For javac, we used the standard driver that is shipped with the SPEC JVM98 benchmarks, and the *Cinderella* benchmark was used as an input to *Jax*. The other benchmarks are all interactive GUI-based applications, so that direct speedup measurements are difficult to conduct.

The speedups we measured turned out to be small and somewhat erratic, and highly dependent on the (version of the) used VM and JIT. Using the JIT developed in IBM's Tokyo Research Laboratory [13], we measured speedups of 5.4%, 2.1%,and 0.4%. We believe that the fact that we measured the relatively large percentage of 5.4% for *Jax* because *Jax* spends a relatively small amount of time doing I/O, compared to *Mockingbird* and *javac*.

4 Related Work

Related work falls into a number of different categories.

4.1 Related Java tools

We are aware of a number of Java tools that have similar objectives as *Jax*.

DashO-Pro[14] and Condensity[15] are commercially available tools that aim at archive size reduction, obfuscation, and optimization. We are unfamiliar with the algorithms used in these tools, and it is unclear to us how *Jax* performs compared to these them.

Several Java tools aim at obfuscation (i.e., to make decompilation of class files into understandable source code more difficult). To mention just a few of these, SourceGuard[16], ObfuscatePro[17], and Jmangle[18] all perform some form of compression of class names, method names, field names, and package names. To our knowledge, of these tools only SourceGuard goes beyond simple name compression, and performs other transformations such as modifying an application's control flow.

4.2 Alternative representations for Java class files

Pugh [18] and Horspool et al. [11, 6] have proposed alternative, more space-efficient representations for Java class files. These representations rely on techniques such as the use of a global constant pool to enable sharing of constants across

[14] DashO-Pro is a trademark of preEmptive Solutions, Inc. See `www.preemptive.com`.

[15] Condensity is a trademark of Plumb Design, Inc. See `www.condensity.com`.

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

[17] See `www.anet-dfw.com/~neil/index.html`.

[18] Jmangle is a trademark of Taylor Computing. See `www.access.digex.net/~rrl`.

benchmark	original application	processed with JAX	reduction
IBM Host-on-Demand	87,452	61,891	29.2%
Hanoi	25,984	11,228	56.8%
ICE Browser	67,462	58,435	13.4%
JAX	106,212	66,509	37.4%
JavaFig	220,186	146,962	33.3%
JMark 2.0	85,118	51,858	39.1%
javac	164,441	110,589	32.7%
Cinderella	550,653	309,982	43.7%
Cinderella Applet	550,653	168,288	69.4%
Lotus eSuite Sheet	644,737	242,496	62.4%
Lotus eSuite Chart	811,556	392,953	51.6%
Mockingbird	457,025	151,816	66.8%
Reservation System	2,910,029	1,221,174	58.0%

Table 7: Results of converting the archives of Tables 1 and 2 to the Jazz representation. For each of the benchmarks, we show the size of the original archive converted to Jazz, the archive produced by *Jax* converted to Jazz, and the corresponding percentage reduction.

benchmark	original application	processed with JAX	reduction
IBM Host-on-Demand	42,848	36,614	14.5%
Hanoi	13,786	7,020	49.1%
ICE Browser	36,452	32,923	9.7%
JAX	50,518	38,807	23.2%
JavaFig	92,819	69,416	25.2%
JMark 2.0	35,934	27,784	22.7%
javac	77,233	64,930	15.9%
Cinderella	197,246	127,420	35.4%
Cinderella Applet	197,246	81,665	58.6%
Lotus eSuite Sheet	548,645	121,158	77.9%
Lotus eSuite Chart	633,228	187,009	70.5%
Mockingbird	505,553	86,429	82.9%
Reservation System	735,629	433,505	41.1%

Table 8: Results of converting the archives of Tables 1 and 2 to Pugh's class file representation. For each of the benchmarks, we show the size of the original archive converted to Pugh's representation, the archive produced by *Jax* converted to Pugh's representation, and the corresponding percentage reduction.

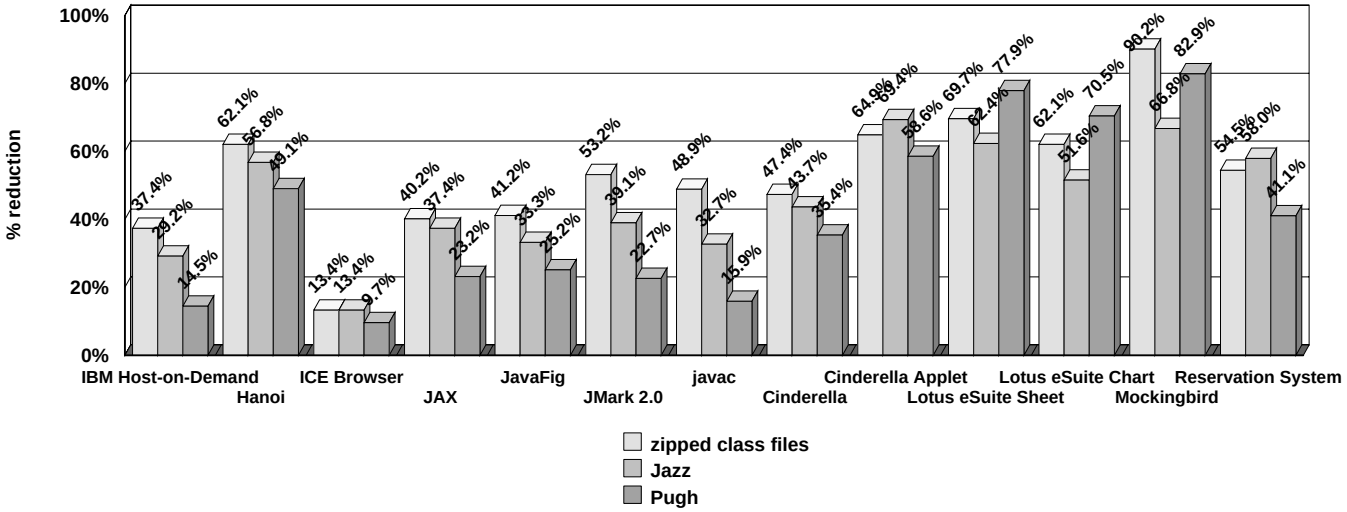

Figure 8: Comparison of the archive size reductions obtained with *Jax* in each of the three class file representations under consideration. For each benchmark, the leftmost bar indicates the percentage reduction using the standard class file representation, the middle bar shows the percentage reduction using the Jazz representation, and the rightmost bar shows the percentage reduction in Pugh's representation.

class files, 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. 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 the representations of [18, 11, 6] is the enabling of sharing of various information between different class files. *Jax* can only eliminate a limited amount of duplication by merging classes. On the other hand, application extractors can achieve significant size reductions by eliminating unused methods, classes, and fields, something that is not addressed by the compression techniques of [18, 11, 6]. 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 took the original class file archives of Table 1 and the archives produced by *Jax* as shown in Table 2, and converted them to the Jazz representation [6], and to Pugh's Packed representation [18]. The results of these conversions are shown in Tables 7 and 8. Table 7 shows for each of the benchmarks the size of the Jazz archives before and after applying JAX. As can be seen from Table 7, the percentage reduction ranges from 13.4% to 69.4% (42.2% on average). Table 8 shows, for each of our benchmarks, the size of the Packed archives (Pugh's representation) before and after applying JAX, as well as the percentage reduction. As can be seen from Table 8, the percentage reduction ranges from 9.7% to 82.9% (33.1% on average). Figure 8 depicts the percentage reduction in each of the three representations (zipped class files, Jazz, Pugh's representation) for each of the benchmarks. Application extraction clearly remains a highly useful size reduction technique, even when these more efficient class file representations are used.

Another line of work for reducing the amount of time needed before a Java application can start executing is the work by Krintz et al. [14, 7] on eager class loading. Krintz et al. study 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.

4.3 Application extraction for other languages

The extraction of applications was pioneered in the Smalltalk community, where it is usually referred to as "packaging" [12, 9, 17]. In order to use a packaging tool for Smalltalk, the user typically specifies a set of initially live "root" objects. The tool then analyzes the application, and determines all objects that are referenced from these objects, and continues this process recursively. Finally, an image is produced that only includes the objects. Smalltalk packaging tools typically have mechanisms for excluding certain standard classes and objects from consideration, and for forcing the inclusion of objects and methods that are accessed using reflection.

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). For each message send that may be executed, a set of slots that is needed to preserve that send's behavior is computed. Finally, a source file is produced in which redundant slots have been eliminated.

4.4 Previous work on C++

Many of the techniques incorporated into *Jax* borrow from earlier work on C++.

The call graph construction algorithm used by *Jax*, RTA was originally proposed by Bacon [4], who proposed RTA primarily as a means for identifying dynamically dispatched method calls that can be devirtualized. Bacon and Sweeney [5] evaluated RTA on a set of C++ benchmarks, and compared its effectiveness to that of Class Hierarchy Analysis [8]. Due to the importance of class libraries for Java, we found that the accurate treatment of methods that override methods in external class libraries is important for reducing archive size.

The detection of useless fields, including write-only fields, was previously studied by Sweeney and Tip [21], and Tip et al. [22] for C++. Sweeney and Tip used RTA to construct a call graph, and measured an average percentage 12.5% useless fields in a set of C++ applications. In the context of *Jax*, we found that, on average, 54.1% of all fields are useless. We conjecture that this difference is partly due to the fact that class libraries are pervasive 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 actually removed.

The class hierarchy transformations used by *Jax* were originally proposed by Tip and Sweeney [23] in the context of specializing class hierarchies. The purpose of specialization 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, method, and field in the program. Inheritance relations between these classes reflect access relationships between variables and methods and fields, and subtype relationships between variables that must be retained to preserve program behavior. In order to reduce the complexity of the resulting class hierarchy, a number of semantics-preserving transformation rules was presented.

5 Future Work

We are interested in the following areas of future research.

5.1 More experiments

We conjecture that a reduction in archive size should lead to smaller representation of the application inside a VM. Furthermore, the removal of fields from classes leads to smaller objects being created at run-time. For these reasons, one would expect an application's memory requirements to be reduced. We intend to conduct more experiments in order to verify this assertion. Also planned are experiments to measure the effect of application extraction on the initialization (load) time required by a VM.

5.2 Optimization

The version of *Jax* discussed in the present paper performs only a limited number of optimizations. Current experiments include improved inlining, and a transformation in

which we redeclare final virtual methods as static methods, and replace virtual dispatches to those methods with a static (direct) dispatch. Preliminary results of these experiments are very encouraging.

As a more ambitious effort, we are developing a bytecode-to-bytecode optimizer in order to further reduce class file size and to improve an application's run-time performance. In this approach, a register-based IR is built from the bytecodes, since many optimizations are more easily performed on a register-based than on a than stack-based representation. Simple optimizations such as constant folding, constant propagation, expression strength reduction, and dead code elimination are performed while the IR is being built from bytecodes, and inlining is also done at this time. After performing the optimizations, a new bytecode stream is assembled from the IR. The IR is designed with this in mind, and its opcodes reflect those of the bytecode. The bytecode assembler performs peephole optimizations, local variable elimination and local variable renumbering.

Future optimizations include traditional intraprocedural optimizations such as copy propagation and common subexpression elimination, as well as some Java-specific optimizations such as removing redundant type checks and inlining virtual calls. Knowledge of the whole application will allow us to do interprocedural optimizations, such as constant propagation and live variable analysis. Type propagation through method arguments may result in more dead methods and opportunities for inlining at virtual call sites. We also plan to look at additional optimizations such as loop optimizations.

5.3 Supplying information to a JVM

In certain cases, *Jax* is capable of determining useful information that it cannot easily exploit. For example, there are cases where it can be determined that 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 [15]. In such cases, *Jax* can pass on the information to a JVM in the form of a bytecode attribute.

5.4 Program understanding tools

One can imagine incorporating the analysis performed by *Jax* to detect unnecessary classes, methods, and fields into a program development environment. For example, a tool could inform the user of unreachable methods and fields, allowing the user to determine if these components are simply redundant, or unreachable as the result of a bug. The tool could also inform the user which methods can be declared as final.

Acknowledgements

We are grateful to John Field, Ramalingam and Vivek Sarkar for comments on drafts of this paper. The feedback from Robert Weir, and the many people who participated in the *Jax* discussion group on alphaWorks is also much appreciated.

We also would like to thank Bill Pugh, Quetzalcoatl Bradley, Nigel Horspool, and Jan Vitek for making their class file compression tools available to us.

Downloading *Jax*

A free evaluation copy of *Jax* can be downloaded from:

`www.alphaWorks.ibm.com/tech/JAX`

Detailed instructions on how to use *Jax* are also supplied there.

References

[1] Agesen, O. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, December 1995. Appeared as Sun Microsystems Laboratories Technical Report 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] Arnold, K., and Gosling, J. *The Java Programming Language*, second edition ed. Addison-Wesley, 1997.

[4] Bacon, D. F. *Fast and Effective Optimziation of Statically Typed Object-Oriented Languages.* PhD thesis, University of California at Berkeley, Dec 1997. Forthcoming.

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

[6] 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.

[7] Calder, B., Krintz, C., and Hölzle, U. Reducing transfer delay using java class file splitting and prefetching. In these proceedings.

[8] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)* (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77–101.

[9] Digitalk Inc. *Smalltalk/V for win32 Programming*, 1993. Chapter 17: "Object Libraries and Library Builder.

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

[11] Horspool, R. N., and Corless, J. Tailored compression of java class files. *Software—Practice and Experience 28*, 12 (1998), 1253–1268.

[12] 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.

[13] 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 *Proceedings of the ACM SIGPLAN JavaGrande Conference* (San Francisco, CA, June 1999).

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

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

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

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

[18] PUGH, W. Compressing java class files. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation* (Atlanta, GA, May 1999), pp. 247–258. *ACM SIGPLAN Notices* 34(5).

[19] SHAPIRO, M., AND HORWITZ, S. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages* (Paris, France, 1997), pp. 1–14.

[20] 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.

[21] SWEENEY, P. F., AND TIP, F. A study of dead data members in C++ applications. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation* (Montreal, Canada, June 1998), pp. 324–332. *ACM SIGPLAN Notices* 33(6).

[22] TIP, F., CHOI, J.-D., FIELD, J., AND RAMALINGAM, G. Slicing class hierarchies in C++. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 179–197. *ACM SIGPLAN Notices* 31(10).

[23] TIP, F., AND SWEENEY, P. Class hierarchy specialization. In *Proceedings of the Twelfth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)* (Atlanta, GA, 1997), pp. 271–285. *ACM SIGPLAN Notices* 32(10).

[24] TIP, F., AND SWEENEY, P. F. Class hierarchy specialization. Tech. Rep. RC21111, IBM T.J. Watson Research Center, February 1998.