

# EXTRACTING LIBRARY-BASED JAVA APPLICATIONS

Reducing the size of Java applications by  
creating an application extractor.

**T**he popularity of the Java programming language [2] has resulted in significant growth of the number of third-party class libraries that perform common tasks such as creating and manipulating collections of objects, parsing XML files, and constructing graphical user interfaces. The use of these libraries can greatly increase developer productivity and code reliability. Developers can focus on the aspects that are unique to the application they are developing without being burdened with the unexciting task of building (and debugging!) standard infrastructure.

Unfortunately, the use of class libraries also has a disadvantage: the prerequisite libraries that a given application depends on may not be available in the deployment environment. Therefore, if an application uses, say, the Xerces (see [xml.apache.org](http://xml.apache.org)) library for parsing XML files, one must either require users to download and install Xerces, or one must ship Xerces along with the application. The for-

mer approach is often undesirable because it makes the installation process more cumbersome and error-prone (for example, customers may inadvertently install an incompatible version of Xerces, which may cause unexpected failures). The latter approach—shipping prerequisite libraries along with applications that use them—often significantly increases the size of the distribution. Increased distribution size

has the disadvantage of increased download time for applications distributed via the

Internet, and increased cost and power consumption for embedded applications stored in Read-Only or Flash Memory. Some embedded environments have imposed physical restrictions on the size of applications. For example, certain kinds of

J2ME-enabled phones have a maximum size of 64K for an application JAR and 512K total storage space for all applications. Clearly, the smaller the application, the more opportunity exists for increasing either the quality or the quantity of downloadable applications. In fact, it is precisely the existence of this type of constraint that led to the development of a standardized small

Java environment such as the Java 2 Platform, Micro Edition (J2ME).

Although the use of prepackaged libraries containing reduced functionality such as J2ME is an important step in the right direction, more advanced techniques are often needed to meet stringent space requirements. Application extraction techniques [1] reduce code size by extracting exactly those parts from referenced class libraries used by a specific application, and then optimizing the code that remains.

Third-party class libraries are often distributed in the form of Java bytecode [5]. To use extraction tools with distributions that include bytecode libraries, such tools must be able to operate on this presentation. Fortunately, Java bytecode is a high-level representation amenable to precise analysis because of the existence of class hierarchies, the availability of type information, and the absence of type-unsafe casting and pointer arithmetic.<sup>1</sup> Consequently, extraction tools can accurately determine the library functionality that is needed. This environment differs considerably from those in which native-code post-pass compaction tools must operate.

Application extraction tools rely on a static analysis of the program. These tools read the class files [5] associated with a Java application and its prerequisite libraries, and perform a static analysis to determine which classes, methods, and fields are actually used. After removing unused classes, methods, and fields, various size-reducing program transformations (including name compression and class hierarchy transformations) are applied to what remains and a JAR file is produced containing the extracted application. This extracted application has the same behavior as the original application, but is typically much smaller (in the benchmarks we examined, on average, applications are reduced to less than 40% of their original size).

Extraction techniques are particularly effective where applications rely on third-party class libraries. This is the case because applications tend to use only a small fraction of the functionality of the libraries they rely on. Application extractors are especially effective in “pruning away” unused library functionality.

We have experimented with a number of extraction techniques in the context of Jax, an application extractor developed at IBM Research. Here, we present a high-level overview of the various extraction

techniques implemented in Jax and assess the impact of these techniques on a set of benchmark applications.

**T**he first step of the extraction process consists of loading the seed classes from the original application that contain the application’s entry points. For each class loaded, an in-memory representation is constructed. Other classes in the application and its prerequisite libraries that are directly or indirectly referenced by the seed classes are also loaded. Some size reductions can be obtained by discarding class file attributes such as line number tables and local variable name tables only essential for symbolic debugging.

The first step identifies the set of classes the application might use, but in general, only a subset of the methods in these classes will be needed. There are various reasons why unreachable methods may arise. For example, an application may contain code associated with features that have become obsolete, or that is associated with features that have not yet been completed. Of particular interest to us is the situation where an application uses a class library that was developed elsewhere. In this case, unreachable methods may arise because the application uses only a small fraction of the library’s functionality.

To identify unreachable methods, a call graph is constructed. A call graph is a conservative approximation of the calls between methods that might arise during program execution. The key step in call graph construction for OO languages is to approximate conservatively the “target” methods that can be invoked by dynamically dispatched method calls. Starting from the application’s entry points, each method body is examined to identify call sites and to approximate their potential targets. This is an iterative process: as new reached methods are encountered, additional call sites may need to be examined, which may lead to the identification of still more reached methods. The process terminates when no additional methods are found. At that point, all methods that are not reached can be removed.

Many call graph construction algorithms for OO languages have been proposed [4]. These algorithms differ in the tradeoffs they make between cost and precision, which both increase as more accurate type information is computed by an algorithm. We originally used the RTA algorithm [3], which is very efficient because only a single set of types is computed, and which computes call graphs that are reasonably precise. Later, we adopted the more precise XTA

<sup>1</sup>In principle, the extraction techniques presented here can be applied to other languages (for example, C++), provided suitable high-level representations (for example, source code) are available for analysis.

algorithm [9], which computes significantly more precise call graphs in some cases, and still has acceptable performance characteristics. XTA computes one set of types for each method in the program. For an in-depth comparison of the performance of these algorithms, see [11].

The elimination of methods can give rise to redundant fields: any field that is only accessed from unreachable methods can be removed from the application. Moreover, fields that are only written to (but not read) can also be removed because their value cannot affect the program's behavior [8]. In the latter case, instructions that store values into the redundant field must also be removed. The removal of such "write-only" fields is particularly worthwhile in situations where fields are initialized in the constructor of a class, but never subsequently used because the field is only accessed by unreachable methods. This frequently happens in cases where unused library functionality is removed.

Although our primary goal has been the reduction of application size, we also mention a few simple performance optimizations that can be performed during extraction. These include call devirtualization, the inlining of method calls in situations where this does not increase application size, and intra-method bytecode optimizations.

### **Class Hierarchy Transformations**

When an application uses only part of a library's functionality, there may be opportunities to reduce application size through compaction of the class hierarchy. For example, if a given class and its members are not referenced from any reachable method, it is possible to remove that class under certain conditions such as when a class has no subclasses. Furthermore, merging classes that are adjacent in the hierarchy can be done without affecting program behavior and without increasing runtime object size. Determining whether or not program behavior is affected and runtime object size is increased involve structure analysis of the class hierarchy, and information about which classes are instantiated [10]. Class merging reduces application size because in Java, each class file is self-contained, and has its own pool of constants and literals. Merging classes reduces the duplication of literals across constant pools.

### **Name Compression**

A Java class file is a self-contained unit of executable code [5]. 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

---

**EXTRACTION**  
**techniques are**  
**particularly effective for**  
**applications that rely on**  
**class libraries because**  
**they are very effective at**  
**removing unused library**  
**functionality.**

---

`"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 will result in smaller archives. Our approach is to rename classes, methods, and fields to `a, b, c ...` More ambitious naming schemes, in which methods with different signatures get the same name, are possible.

### **Extraction Hazards**

Although we have offered readers a sense of the number of important extraction techniques in the preceding sections, some issues that complicate extraction are described here.

In general, a static analysis cannot determine the classes, methods, and fields that are accessed using

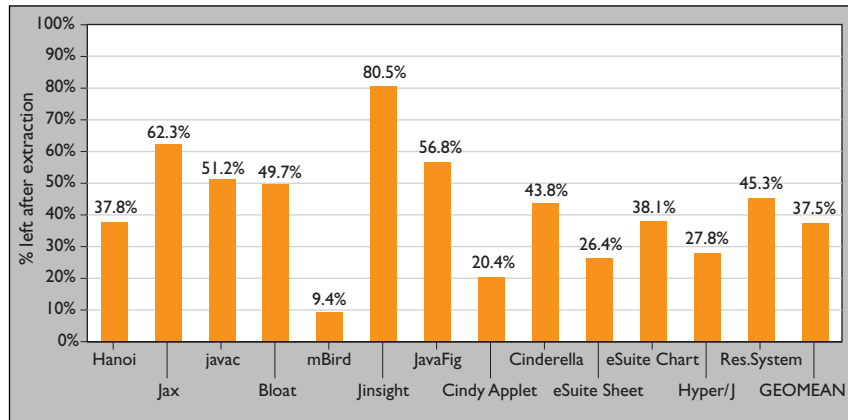
dynamic loading or reflection.<sup>2</sup> Certain special cases can be detected (for example, through the propagation of string constants), but this will not always be possible. Since removing or transforming these arti-

names of constructors and static initializers.

When a program is modified by the extractor, the extractor must ensure that the order in which Java classes are loaded by the runtime system remains the same because some applications rely on that order.

## Jax

We implemented the techniques discussed here in Jax, an application extraction tool. Jax was originally developed at IBM Research as a vehicle for research on application extraction [7, 10, 11] and call graph construction [9], and has been available from IBM's alphaWorks Web site since June 1998. A number of software products (both inside and outside



**Figure 1. Size of the extracted benchmark applications as a percentage of the size of the original archive. Resource files are excluded from both the initial and processed archives.**

facts may affect program behavior, the user must identify all uses of reflection and dynamic loading and list them in a configuration file.

There are cases where unreachable methods cannot be removed for syntactic reasons. For example, if a (non-abstract) class implements an interface, all of the methods declared in the interface must be retained, even though some of these methods may be unreachable. In such cases, one can still replace the body of such unreachable methods with a single return statement.

Merging classes across package boundaries may require that classes, methods and fields need to be made public. If security concerns are an issue, class merging needs to be disallowed in such cases.

Java code may contain native methods that are implemented in a different language (for example, C). Such methods may instantiate classes, invoke methods, and access fields. As native code is difficult to analyze, a common approach is to rely on the user or library designer to specify the behavior of native methods [7].

Some names cannot be compressed. These include any name of a class, method, or field in an external library; a method that overrides a method in an external library; the name of any program component accessed using reflection; the name of the class containing the main routine; and the

of IBM) have been shipped after extracting them with Jax. Moreover, a number of major components of Jax have recently been incorporated in the link-time optimizer of IBM's WebSphere Studio Developer (see [www.ibm.com/embedded](http://www.ibm.com/embedded)), an integrated environment for developing embedded systems applications in Java, in order to reduce the size of embedded Java applications. An evaluation copy of Jax can be downloaded for free from [www.alphaworks.ibm.com/tech/jax](http://www.alphaworks.ibm.com/tech/jax).

Table 1 lists the size, essential characteristics, and a short description of each of a collection of Java applications used to evaluate the effectiveness of Jax. Figure 1 shows, for each of the benchmark applications, the size of the ZIP file containing the extracted application as a percentage of the size of the ZIP file containing the original application.

Figure 2 shows, for each benchmark application, the number of classes, methods, and fields in the extracted application as a percentage of the number of classes, methods, and fields that were originally present. As can be seen in Figure 1, the applications are reduced to 37.5% of their original size on average,<sup>3</sup> but in some cases even greater reductions can be achieved. The size of the mBird benchmark is reduced dramatically because we extracted only the batch client from a distribution containing both a batch client and a GUI-based client, and shows that Jax is very effective at removing the unused (GUI-related) functionality. The other benchmarks for which we obtained significant size reductions (Hanoi, eSuite Sheet, eSuite Chart, and Hyper/J) all use one or more class libraries, and a large part of the

<sup>2</sup>Dynamic loading and reflection are mechanisms that allow programmers to load and instantiate classes, invoke methods, and access fields, using runtime (string) values to identify the accessed program constructs.

<sup>3</sup>All average percentages reported here are computed using the geometric mean.

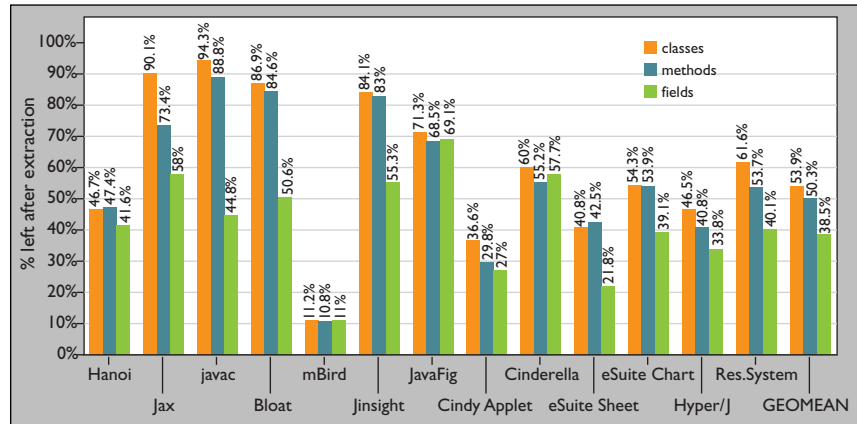
reported reductions is attributable to removing unused library functionality. It is clear from the similarity of the shapes of the bar charts of Figures 1 and 2 that the reduction in distribution size is strongly correlated to the number of classes, methods, and fields that are removed.

One benchmark on which Jax is particularly effective is Hyper/J (see Figure 1). Table 2 shows how the reduction of Hyper/J to 27.8% of its original size can be attributed to the various steps performed by Jax. Removing unreferenced classes reduces the ZIP file to 86.0% of its original size. Next, removing redundant attributes further reduces the ZIP file to 67.1% of its original size. A similar amount of space is taken up by unreachable methods, and removing these results in a ZIP file that is 46.7% of the original size. The combined impact of dead field removal, inlining, and devirtualization is a relatively minor 4.1%, shrinking the ZIP file to 42.6% of its original size. The contribution of class hierarchy transformations is quite noticeable. Hyper/J is written in a highly OO style, with heavy use of interfaces. Large sections of the class hierarchy are compacted by Jax, which reduces the number of classes from 789 to 428, and the ZIP file is reduced to 35.2% of its original size. Finally, name compression reduces Hyper/J to a mere 27.8% of its original size. In our experience, the techniques described here are highly scalable. For example, processing Reservation System, the largest of our benchmarks, with Jax, takes about 4 minutes on an 800MHz Intel-based machine with 512MB of memory.

## Conclusion

We have demonstrated that application extraction techniques can significantly reduce the size of Java applications by constructing an application extractor called Jax. Extraction techniques incorporated in Jax include the removal of unreachable methods and redundant fields, and compaction of the class hierar-

Figure 2. Number of classes, methods, and fields in the extracted benchmark applications shown as a percentage of the original number of classes, methods, and fields, respectively.



Benchmark	Description	Classes	Methods	Fields	Archive Size
Hanoi	Tower of Hanoi program	45	378	233	55,765
Jax	Java extraction tool	302	2,900	1,274	534,658
javac	SPECjvm98 version of Sun's bytecode compiler	210	1,512	1,107	452,125
bloat	Bytecode optimizer from Purdue University	282	2,677	1,255	506,736
mBird	Proprietary IBM tool for multi-language operability	2,050	17,946	6,739	2,950,543
jinsight	IBM research developed performance analysis tool	264	2,974	1,875	393,857
JavaFig	Java version of the xfig program	160	2,108	1,526	394,432
CindyApplet	Educational self-study, interactive geometry tool applet	467	4,449	3,075	881,555
Cinderella	Educational self-study, interactive geometry tool applet	467	4,449	3,075	881,555
Lotus eSuite Sheet	Productivity application, part of the now obsolete Lotus eSuite	588	5,590	4,305	1,251,765
Lotus eSuite Chart	Productivity application, part of the now obsolete Lotus eSuite	733	8,302	5,448	1,570,569
Hyper/J	Advanced separation of concerns system by IBM research [6]	921	8,776	2,733	1,523,670
Reservation System	IBM customer airline, hotel, and car reservation system front-end	2,326	21,495	12,487	3,810,120

Table 1. Characteristics of the benchmark applications used to evaluate Jax. The size of the initial archive shown here is in bytes and excludes any resource files contained in the shipped archives. The CindyApplet and Cinderella benchmarks are a closely related applet and application that are contained in the same distribution.

Hyper/J	Archive Size	%
original size	1,523,670	100.0%
unreferenced classes	1,309,962	86.0%
Redundant attributes	1,021,698	67.1%
Unreachable methods	711,678	46.7%
Redundant fields	650,461	42.7%
Inlining/devirtualizing	648,646	42.6%
Class transformations	535,606	35.2%
Name compression	424,074	27.8%

Table 2. Archive size for Hyper/J as a percentage of the original archive size after (1) removal of unreferenced classes, (2) removal of redundant attributes, (3) removal of unreachable methods, (4) removal of redundant fields, (5) method inlining/devirtualizing, (6) class hierarchy transformations, and (7) name compression.

chy. We evaluated Jax on a collection of large benchmark applications, and measured that, on average, these benchmarks are reduced to 37.5% of their original size. Extraction techniques are particularly effective for applications that rely on class libraries because they are very effective at removing unused library functionality. This largely eliminates the main drawback of using third-party class libraries. **C**

## REFERENCES

1. 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, Oct. 1994), 355–370.
2. Arnold, K., Gosling, J., and Holmes, D. *The Java Programming Language, Third Edition*. Addison-Wesley, 2000.
3. Bacon, D.F. and Sweeney, P.F. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, (San Jose, CA, Oct. 1996), 324–341.
4. Grove, D. and Chambers, C. A framework for call graph construction algorithms. *ACM TOPLAS* 23, 6 (Nov. 2001), 685–746.
5. Lindholm, T. and Yellin, F. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.
6. Ossher, H. and Tarr, P.L. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM* 44, 10 (Oct. 2001), 43–50.
7. Sweeney, P.F. and Tip, F. Extracting library-based object-oriented applications. In *Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE-8)*, (San Diego, CA, Nov. 6–10, 2000), 98–107.
8. 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 (PLDI'98)*, (Montreal, CA, June 17–19), 1998.
9. Tip, F. and Palsberg, J. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, (Minneapolis, MN, Oct. 15–19, 2000), 281–293.
10. Tip, F. and Sweeney, P.F. Class hierarchy specialization. *Acta Informatica* 36 (2000), 927–982.
11. Tip, F., Sweeney, P.F., Laffra, C., Eisma, A. and Streeter, D. Practical extraction techniques for Java. *ACM TOPLAS* 24, 6 (Nov. 2002), 625–666.

---

**FRANK TIP** (ftip@us.ibm.com) is a research staff member at the IBM T.J. Watson Research Center in Hawthorne, NY.

**PETER F. SWEENEY** (pfs@us.ibm.com) is a research staff member at the IBM T.J. Watson Research Center in Hawthorne, NY.

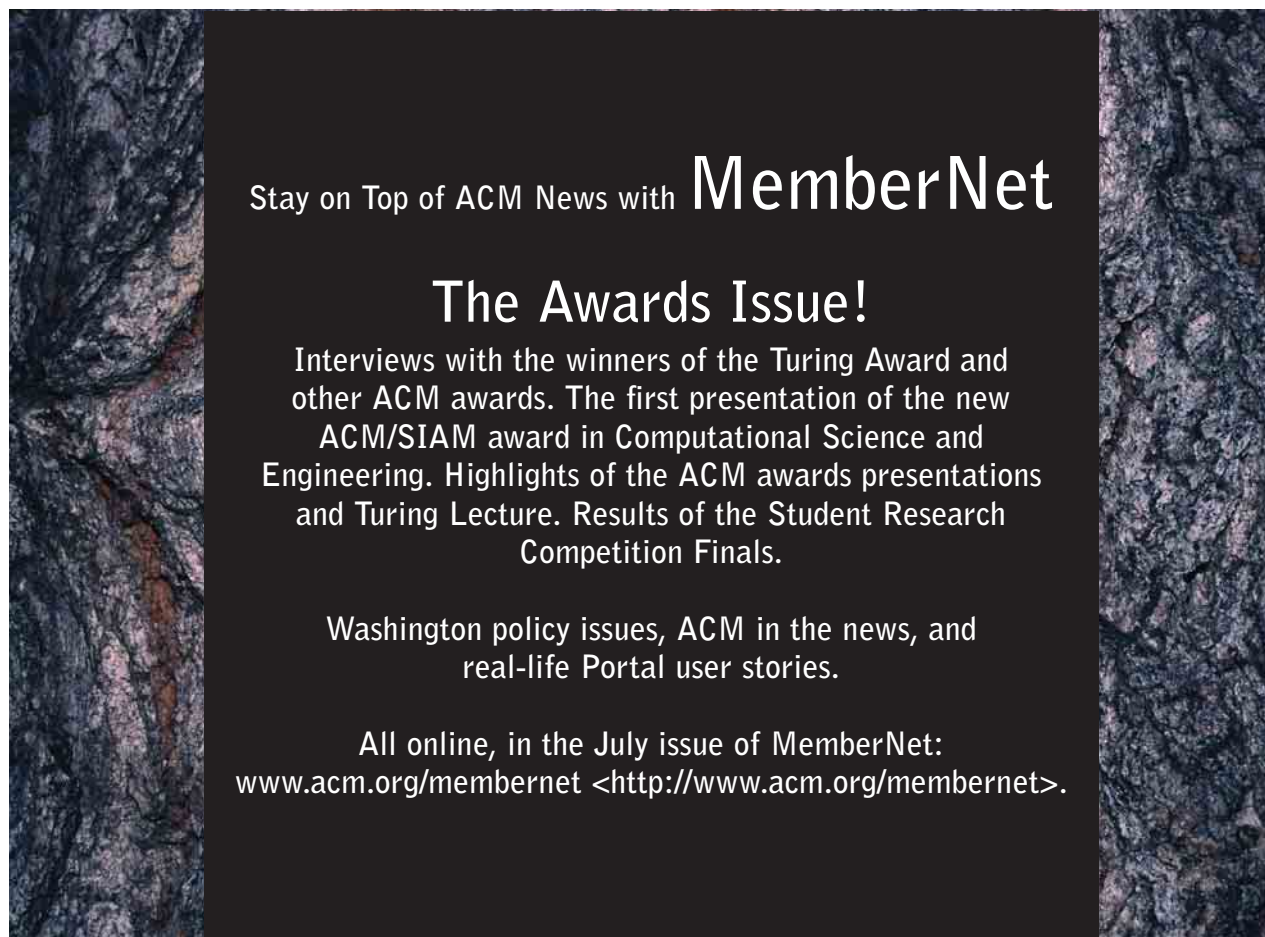
**CHRIS LAFFRA** (Chris\_Laffra@ca.ibm.com) is a senior software engineer at the IBM Ottawa Lab, in Ottawa, Canada.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

---

© 2003 ACM 0002-0782/03/0800 \$5.00



Stay on Top of ACM News with **MemberNet**

**The Awards Issue!**

Interviews with the winners of the Turing Award and other ACM awards. The first presentation of the new ACM/SIAM award in Computational Science and Engineering. Highlights of the ACM awards presentations and Turing Lecture. Results of the Student Research Competition Finals.

Washington policy issues, ACM in the news, and real-life Portal user stories.

All online, in the July issue of MemberNet:  
[www.acm.org/membernet](http://www.acm.org/membernet) <<http://www.acm.org/membernet>>.