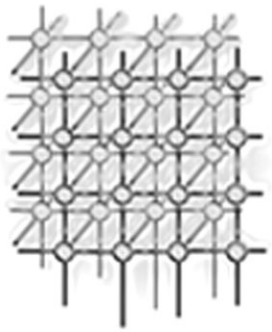
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **17**:573–587
Published online 22 February 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.850

Compiling almost-whole Java programs



Zoran Budimlić*,† and Ken Kennedy

Center for High Performance Software Research, Rice University, Houston, TX 77005, U.S.A.

SUMMARY

This paper presents a strategy, called *almost-whole-program* compilation, for extending the benefits of whole-program optimization to large collections of Java components that are packaged as a group *after* the development phase. This strategy has been implemented in a framework that uses Java visibility and scoping rules to transform a collection of classes into a package that is amenable to whole-program optimizations, without precluding extensions to the optimized and compiled code. Thus, it enables the Java developer to balance performance against flexibility of the program *after* the development phase, without compromising the design process. The transformation is shown to incur only modest performance penalties, which are more than compensated for by the interprocedural optimizations it enables. The paper concludes with experimental results showing the benefits that can be achieved using this approach. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: whole-program compilation; almost-whole-program compilation; incremental compilation; Java; object-oriented optimization

1. MOTIVATION

In a project to support the use of Java for scientific programming, our investigations have established that interprocedural optimizations, such as object inlining and class specialization, can yield integer factor speedups on code that uses the full power of object-oriented programming [1,2]. These results were obtained in a source-to-source optimization tool called JaMake, which transforms the program before it is presented for execution to a Java virtual machine (JVM). While the current implementation of JaMake uses Java source as the input and output file type, there is nothing inherent in the techniques implemented in JaMake and described here that limits their applicability to source code. These techniques can just as easily be implemented in work on binary class files.

*Correspondence to: Zoran Budimlić, Center for High Performance Software Research, Rice University, Houston, TX 77005, U.S.A.
†E-mail: zoran@cs.rice.edu

Contract/grant sponsor: NSF; contract/grant number: ACI-0234345

Received 15 January 2003
Revised 29 August 2003
Accepted 14 October 2003

To achieve performance improvements, the JaMake framework needs to see the entire Java program so that it can perform cross-procedural analysis and optimizations. However, for many contexts, such as the development of component libraries, it is not convenient to deliver software in whole-program form. What is needed is a strategy that combines the flexibility of the standard Java compilation model with the performance improvements of whole-program optimization.

Current Java implementations adopt a simple compilation model: classes are compiled one at a time and, in the process of compiling one class, the compiler cannot assume anything about the internals of any other class [3,4]. This gives users great flexibility in the ways they can use the program—they can add, remove and replace classes in the package at will, without the need for recompilation. Unfortunately, it also prohibits many interprocedural optimizations that could lead to significant performance benefits.

A whole-program compilation model, on the other hand, assumes that the whole program is available at compile time and that the programmer will not add, remove or modify any classes without recompiling the complete program [5]. This allows the compiler to perform a wide range of aggressive interprocedural optimizations that can lead to dramatic performance improvements. Unfortunately, the whole-program compilation model provides little flexibility to the user.

In the real world, the need for flexibility falls somewhere in between these two models. Some parts of a program are fixed and the user is not expected to modify or augment them, while other parts should be available to the user for extension. Currently, there are no mechanisms by which programmers can control this: they can only write the program, compile it and distribute it to the user. Consequently, they are forced to trade performance for flexibility or *vice versa*.

A principal reason for this problem is that a single process (compiling the program) is used for two very distinct purposes: *development* of the program and its *distribution*. These two phases often have conflicting goals.

While developing the program, programmers want maximal code flexibility, so that they can change, extend and debug it. Also, they want their code to be easily upgraded in later versions of the program. Consequently, programmers use object-oriented techniques such as encapsulation and polymorphism when designing their systems. Most of the classes in such a system will be public (enabling their later extension), most of the methods will be virtual (enabling their overriding) and public.

On the other hand, a main goal during the software *distribution* process is achieving the highest possible performance. Many powerful optimizations require cross-procedural analysis and optimization to achieve the best performance. In most frameworks, carrying out such optimizations requires that the entire program be fixed at compile time. In the Java framework, this requirement means that all of the classes have to be made private and final, and that all of the methods are final and/or static. Only the main class and main method of the program should be made visible to the user to allow the program to run. Putting the program in this form would permit the static analysis and optimizations to be effective, but it would overly restrict the potential uses of the program.

Of course, programmers can choose to transform the code by hand to achieve 'optimal' performance. Alternatively, when the development phase is finished and the program is debugged and ready for shipping, the developer can manually create another version of the program in which many of the classes and methods are final and private. This transformation enables the compiler to perform a better job of optimizing the whole program. However, such a process is tedious and error-prone, even though it is not uncommon in the software industry. In our personal correspondence with Java JDK developers, we have learned that many of the classes and methods that were public and extensible in

the earlier versions were converted into private and final in the later versions for performance reasons (for example, String manipulation methods). A significant amount of programmer time has been spent to manually make this transformation in order to enable `javac` to inline these modified methods, but resulting in only modest performance improvements. Furthermore, such a transformation is not even possible in the general case, since non-leaf classes in Java class hierarchy cannot be final.

Figure 1(a) depicts an abstract relation between the flexibility of the generated code and its performance. As parts of the code are fixed to enable Java source compiler to perform more optimizations, the flexibility of the code that is shipped to the end-user is reduced. This reduction in flexibility is a logical consequence and generally accepted by the end-user. Unfortunately, the programmer is forced to tolerate the same reduction in flexibility, since there is direct mapping from the source code to the code that is deployed to the user.

If we observe the current *automatic* Java compilation systems, the performance versus flexibility diagram looks more like that in Figure 1(b). There is a precipitous drop in user flexibility when the systems abandon the class-by-class compilation model and assume a whole-program compilation. Depending on the number and effectiveness of the optimizations applied, performance of the generated code improves, but the user flexibility stays the same. The user can run only the generated program and nothing more.

The almost-whole-program compilation model addresses this problem by bridging the gap between the whole-program and class-by-class compilation models. It is less flexible than the class-by-class model, in that the user's choices are more restricted, but it is nevertheless more flexible than the whole-program model. Similarly, the range and effectiveness of the interprocedural optimizations that can be applied are smaller than in the whole-program model, but far greater than in the class-by-class model. Figure 1(c) shows this relation.

In this paper, we present an *almost-whole-program* transformation tool that specializes a Java program based on directives from the programmer. The framework, which can be invoked after the development phase, makes most of the classes and methods final and inaccessible to the end-user. It then passes this converted program to the JaMake whole-program optimization system described in our previous work [2]. The whole-program optimizer can now derive more precise type information from the program and perform more effective object inlining. These techniques are implemented in our JaMake compiler infrastructure.

This framework allows the programmer to specify which classes in a partial program are open (extensible) and thus restricts all supplemental classes to extending those classes. To enforce constraints on supplemental classes, the system places the compiled partial program in a single Java package and marks non-extensible classes as package private, relying on Java class visibility rules and run-time checks to prevent extension of forbidden classes.

This framework is usable in situations when the program (or most of the program) is being developed, when the program can be transformed with little additional effort into an almost-whole-program library. This framework is not that useful, however, when applied to existing Java programs, where a part of the program is subjected to almost-whole-program compilation, since the parts of the program that are *not* the part of the almost-whole library would have to conform to certain conventions described below. Depending on the size and complexity of the program code not in the almost-whole part, this approach would require some additional programmer effort.

There are other advantages that almost-whole-program transformation provides. Systems that perform code obfuscation [6] can take advantage of the class repackaging and renaming mechanism

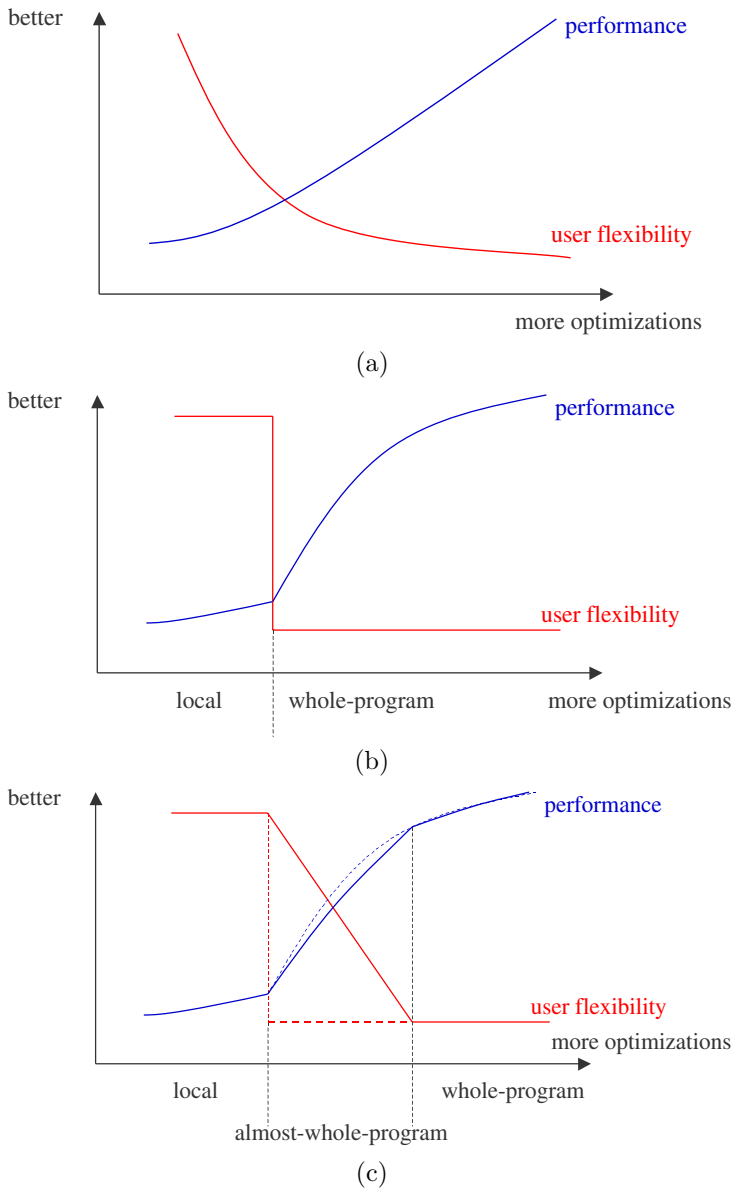


Figure 1. Performance versus flexibility diagrams, showing the trade-off between performance and flexibility in different systems: (a) manual program transformation; (b) current automatic techniques; (c) almost-whole-program compilation.

that almost-whole-program transformation implements to further protect the original code from reverse engineering. Although it is not its primary goal, almost-whole-program transformation is effective in compacting the distribution code, as only the reachable methods and classes are included in the distribution package. Other systems such as Jax [7] take this idea even further and perform code compaction using techniques that are very similar to those applied in almost-whole-program transformation.

Almost-whole-program transformation is *not* a replacement for class-by-class compilation and is *not* targeted towards compilation of highly dynamic systems that require extensive flexibility. It is best applied in situations where programmers are forced to use whole-program compilation to get high performance, but would still like to provide *some* flexibility to the end-user without sacrificing much of the performance. By using almost-whole-program transformation they can achieve that goal. Almost-whole-program transformation is not targeted to provide flexibility that approaches that of class-by-class compilation by transforming *most* of the development code into public and visible. The optimizations (if any) that would remain possible in such a system would likely be negated by the overhead of the almost-whole-program transformations. In terms of Figure 1(c), almost-whole-program compilation is most effective in the area close to the whole-program area: very high performance, but still *some* flexibility. Thus the term *almost-whole*, as opposed to *partial* or *modular* compilation.

2. RELATED WORK

Tip *et al*. [7] use an approach similar to our treatment of *private classes* in their Jax application extractor to extract a Java application from a collection of classes. They use class repackaging and renaming to minimize the size of the extracted application. However, they do not have anything similar to our transformation of non-extensible or public classes, and as such their approach is not suitable for partial program compilation. Jax is an excellent whole-program application extractor, but it does not address the almost-whole-program compilation issues that we concentrate on in this paper.

Zaks *et al*. [8] used interprocedural analysis to find virtual calls within a package that are guaranteed to target methods within that package. If that package is *sealed* [9] those calls can be safely devirtualized. Their analysis can be viewed as an attempt to *exploit* the almost-whole properties of a sealed Java package to gain performance, and as such it is an '*application* of almost whole program compilation' [8, Section 1, Paragraph 2].

Dean, Shultz and others [5,10,11] have developed a whole program optimization framework for object-oriented languages. They have proposed several optimization and analysis techniques for compilation of such languages when the whole program is known at the compile time. Our work concentrated on further developing some of their techniques and some new ones in the context of Java programming language and the unique problems created by Java's execution model, especially in the context of high-performance scientific programs written in Java. We also exploit the impact the almost-whole-program optimization framework has on applicability of their techniques.

3. AN ALMOST-WHOLE-PROGRAM TRANSFORMATION FRAMEWORK

Our approach makes almost-whole-program compilation possible without changing any of the rules that are currently in effect for Java systems. There are no changes to the Java language.

The JVM specification is respected, so the generated bytecodes may be executed on any JVM. The rules for Java name visibility are unchanged. In fact, it is the rules of Java package name visibility together with the run-time checks that make it possible to perform almost-whole-program optimization while still generating pure Java programs.

To provide the desired flexibility for the programmer, our almost-whole-program transformation system must exhibit three main properties:

- it must permit the programmer to specify which classes are inaccessible to the end-user, regardless of the original class layout in the development project;
- it must allow the programmer to specify classes the end-user is permitted to see (instantiate and run), but not to extend; and
- it must permit the developer to specify which classes from the project are to be made completely available to the end-users.

In the following sections we show how our framework achieves all three of these goals.

3.1. Private classes

It is easy to imagine a scenario in which some of the classes in the programmer's project are specified as public, but the programmer does not want to allow an end-user to access them. These classes are specified as public for development reasons, perhaps because some classes in other packages may need to access them. If the original project is translated directly into code for the JVM, the end-user will be able to access, instantiate and even extend any classes that are specified as public.

The first goal of the almost-whole-program framework is to allow the programmer to specify which classes are inaccessible to the end-user, regardless of the original class layout in the development project. Figure 2 shows an example of what happens to the classes the programmer has specified as inaccessible during the transformation of the development project into the distribution package. The emphasized classes are the ones that are publicly accessible in both packages—only the *Main* class is accessible in the distribution package.

All classes that are written as *private* in the original class layout remain private in the distribution package (class *top.sub1.class2* in the example in Figure 2). The tool converts all the classes that were *public* in the development package and marked as inaccessible by the programmer into *package private* (classes *top.class1*, *top.sub1.class1*, *top.sub2.class1*, *top.sub2.class2* in Figure 2). The framework renames and repackages all the classes from the development project into a single distribution package. This transformation is necessary to allow the mutual access of the classes that were originally *public* and in different packages but are now *package private*.

Only the *Main* class of the whole development project remains public (but final, and thus non-extensible) in the distribution package. The end-user will only be able to run the program by calling *Main.main()* and nothing else.

3.2. Non-extensible classes

The second goal of the almost-whole-program framework is to allow the programmer to specify classes the end-user is allowed to see (instantiate and run), but not to extend. This case arises in situations in which a class should be made available to the user first to instantiate the class and then to call its public

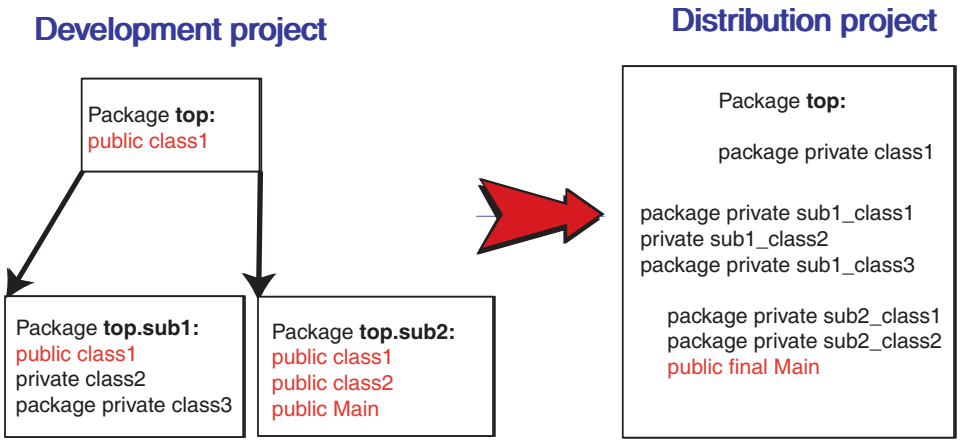


Figure 2. The development-to-distribution transformation of private classes.

method, but the programmer wants to preserve the class hierarchy for performance reasons. A logical approach would be to convert these classes into *final*; unfortunately this is not always possible. Some of these non-extensible classes may have sub-classes already defined in the development project, and the back-end compiler will not allow the compilation of the sub-classes if their super-classes are declared as final.

We propose an alternative approach. JaMake converts all the classes that the developer specifies as visible but non-extensible into *private*. It then creates auxiliary public interfaces (implemented as abstract Java classes) that summarize the member information of the non-extensible classes. Figure 3 shows the structure of this transformation.

Figures 4 and 5 display an example of the development-to-distribution project conversion. *Foo* on Figure 4 is a class that the developer has marked as public but non-extensible. The almost-whole-program framework converts this class into the class *Foo* on the right-hand side of Figure 4. All of the public methods and fields are converted to be package private, the same as for private classes described in Section 3.1.

The end-users need a mechanism for instantiation of objects of type *AFoo*. Since the class *Foo* is package private, they cannot directly create objects of type *Foo*. Instead, the abstract class *AFoo* provides static *Create* methods, which serve as an abstract factory [12] for obtaining new instances of *Foo*. For each constructor of *Foo*, class *AFoo* contains a corresponding method *Create*. The only way the end-users can create new instances of *Foo* is by calling the *AFoo.Create()* method. Since the generated factory methods *Create* are static, most of the current JVM implementations are able to perform run-time inlining of these calls, making them as fast as if the original constructors were called directly.

The almost-whole-program framework ensures the non-extensibility of the classes the developer marked as non-extensible through a run-time mechanism. When converting the project from the

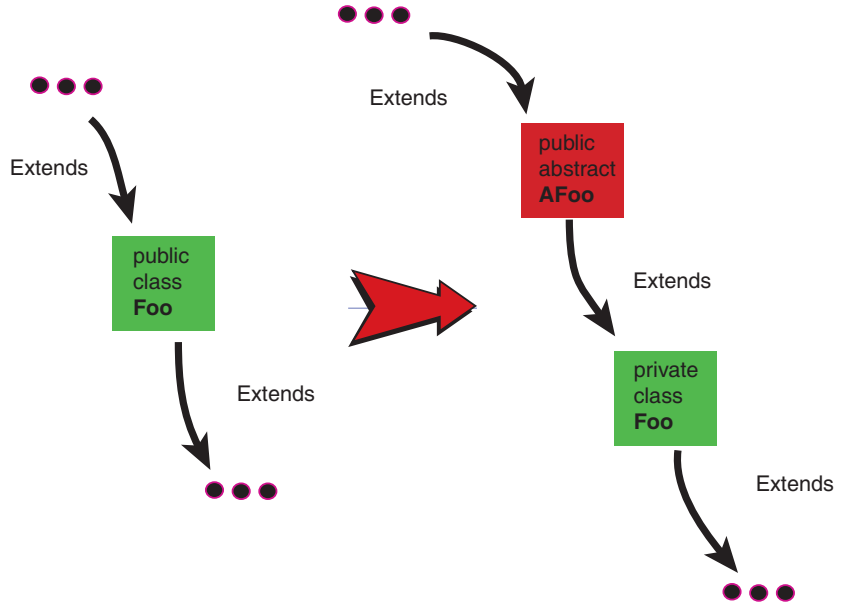


Figure 3. The development-to-distribution transformation of a part of the class hierarchy.

```
public class Foo extends Goo{
       public T field1;
       public static T field2;
       private T field 3;
       public Foo(T arg){
           /*implementation*/
       }
       public Foo(T1 arg1, T2 arg2){
           /*implementation*/
       }
       public T Meth1(T2 arg){
           /*implementation*/
       }
       static public T Meth2(){
           /*implementation*/
       }
}
```

```
class Foo extends AFoo{
       private T field 3;
       Foo(T arg){
           /*implementation*
       }
       Foo(T1 arg1, T2 arg2){
           /*implementation*/
       }
       T Meth1(T2 arg){
           /*implementation*/
       }
}
```

Figure 4. Original class *Foo*, marked as non-extensible, and the generated class *Foo*.

```
public abstract class AFoo extends Goo{
      public T field1;
      public static T field2;
      public static final AFoo Create(T arg){
           return new Foo(arg);
      }
      AFoo(T arg){
           super();
           if(!(this instanceof Foo)) throw new Error(``Not Foo!'');
      }
      public static final AFoo Create(T1 arg1, T2 arg2){
           return new Foo(arg1, arg2);
      }
      AFoo(T1 arg1, T2, arg2){
           super();
           if(!(this instanceof Foo)) throw new Error(``Not Foo!'');
      }
      abstract public T Meth1(T2 arg);
      static public T Meth2(){ /*implementation*/ }
}
```

Figure 5. Generated auxiliary abstract class *AFoo*.

development to the distribution form, this tool inserts run-time tests in all of the constructors of the auxiliary abstract class. In the example from Figure 5, all of the constructors have a test of whether the object being currently instantiated is of type *Foo*. If not, the constructor will throw a run-time exception. This technique prevents the end-user from instantiating his own version of sub-classes of *AFoo*. The end-user can still write and compile sub-classes of *AFoo*, but cannot do anything else with those classes. This restriction ensures that there cannot be any instances of *AFoo* that are not also instances of *Foo* at run-time. The whole-program optimization techniques [1,2,5] can take advantage of this fact and optimize the code that uses *Foo*.

The development-to-distribution implementation as described above (marking whole classes as either private, public or non-extensible) would extract all of the public methods from the class that is being converted into the auxiliary abstract class. It would also generate wrapper methods for all of the public static methods. Such an implementation would require less effort from developers, since they would only have to specify the classes which are non-extensible, but it would be less flexible. In contrast, our implementation allows developers to specify precisely which methods and fields should be exported in each class, in addition to the per-class specification. This approach allows more flexibility to developers, although requiring more input from them. This approach also generates shorter code, since it only converts the requested methods and fields.

This development-to-distribution conversion completely hides the class *Foo* from the end-users, but still enables them to create instances of it, call its public methods and access its public fields. The end-users cannot extend the class *Foo*, since it is not visible outside the generated package. They can create new classes that extend the class *AFoo*, but cannot create instances of those new classes.

It is important to note that this framework does not provide complete protection of the inaccessible and non-extensible classes from the misuse by the end-users. The users can breach these measures by decompressing and unpacking the generated JAR file, and replacing some of these unpacked classes with their own. Alternatively, they can mimic the directory structure of the package and create their own classes with the same name as some of the classes from the package, and put their own classes before the package on the classpath. Even if their classes conform to the interfaces of those they have replaced and have the same functionality, the program created in such a way would still be incorrect. If the end-users wish to make such changes, they need the whole-program or almost-whole-program recompilation framework.

3.3. Public classes

Finally, the developers can specify which classes from the development project are to be made completely available to the end-users. These classes are declared as public and can be used by end-users in any way that regular public Java classes can be used. They can instantiate them, call their methods and (for the non-final classes) write their own classes that extend them. The conversion process from the development project into the distribution project is trivial: these classes are simply declared as *public*.

However, the implementation details in the context of the concrete type analysis [13,14] and whole-program optimization are more complex. The JaMake system has to capture the information that these classes could be extended by the end-users and pass it to the set-based type analysis tool as well as to the optimizing back-end of the compiler.

The almost-whole-program framework achieves this behavior by inserting 'unknown' classes, which represent the portion of the program that the end-user can add to the final package. Let us assume that the developer marked the class *Foo* as available. The almost-whole-framework will create the type *UnknownFoo*, which is a subtype of *Foo*. The set-based analysis which computes the concrete type information for the program inserts this information into the analysis. For every public method (public methods of public classes and public methods of the non-extensible classes described in the previous section) that takes an argument of the type *Foo*, it will add *UnknownFoo* to the set of types of that argument. The analysis then proceeds as usual.

Interprocedural optimization techniques [2,5,15] are aware of this special type and limit the transformations that they perform on the program accordingly. Class specialization can specialize the classes based only on the polymorphic data that *do not* contain the unknown type. It will keep a default implementation (basically the original code of the class) for the classes that *do* contain the unknown type. After class specialization, object inlining will inline only objects of a precise (and known) type. These restrictions ensure that the program that the end-user creates by using the distribution package and extending the available classes from it is observationally equivalent to the program in which the distribution package is replaced with the development package.

Naturally, marking too many classes as public in the distribution package severely limits the range of interprocedural optimizations and reduces their effectiveness. In the extreme case, marking *all* of the classes available will completely prevent global object inlining (although some limited local object inlining will still be possible). The developer has to have this fact in mind when creating the distribution package, and balance its flexibility and performance.

Table I. Execution times (in ms) for OwlPack using JDK 1.4.2, javac -O.

	Development package			Distribution package		
	Fortran style	'Lite' OO	OO style	OO style	Dev. − Dist. OO % increase	Optimized OO *
dpofa	180	220	6860	7421	8.2	550
dposl	261	360	3585	4456	24.3	461
dpodi	350	511	27 029	27 610	2.1	1012
dgefa	120	150	8512	8643	1.5	320
dgesl	81	130	1322	1452	9.8	231
dgedi	170	281	13 549	13 960	3.0	661
dqrdc	1142	1221	48 319	51 504	6.6	2674
dqrsl	531	561	5949	7020	18.0	1192
dsvdc	290	400	18 877	21 411	13.4	581
Average	1.000	1.327	36.504	41.173	9.7	2.647

4. PERFORMANCE RESULTS

The almost-whole-program transformation may incur some additional overhead over the original program. This transformation adds a level of indirection to the classes declared as non-extensible. All the method calls on the non-extensible class become virtual. The most significant effect in the performance is that every instantiation has an added constructor call (to the constructor of *AFoo* in our example) in the chain of *super()* calls in the constructors. In addition, that constructor is performing a run-time check to determine whether the object being instantiated is of a proper type (*Foo* in our case).

All performance figures in this section are obtained on Sun's HotSpot VM 1.4.2, on an 800 MHz Athlon PC with Windows XP SP1 and 256 MB of memory, with the -O option for javac 1.4.2 and using the fastest of three runs for all the tests.

Table I shows the execution times for our standard set of OwlPack benchmarks, described in great detail elsewhere [16], both for the original (development) and the almost-whole-program (distribution) package.

The *Fortran style* column shows the execution times for a version of Linpack written in Java style that very closely resembles Fortran [17] (all methods are static, arrays are passed directly as arguments, the data are accessed directly and there is a version of the code for each primitive number type). We like to think of this code as 'Fortran programming using Java syntax'.

'*Lite*' *OO* is a version of the Linpack library that makes an attempt of introducing objects to the Linpack library without compromising the performance of the *Fortran style* version. We suspect that this is the version performance-conscious programmers without access to advanced compilation frameworks such as JaMake would most likely be inclined to write today—all of the data are still stored in two-dimensional arrays, there are four versions of the code for four data types, but the arrays are wrapped in objects that represent matrices, vectors and similar data structures.

The *OO style* version of the Linpack library uses the full power of object-oriented programming. This code uses polymorphic classes to represent numbers, and there is only one version of the code that operates on generic numbers. Numbers are rarely mutated; most of the number operations instantiate a new object for the result. The *OO style* version of the code is about four times shorter than the '*Lite*' *OO* version, with the same functionality.

We choose to mark the classes *LNumber* and *LDouble* as non-extensible for the distribution package, simulating a situation where the programmer anticipates that the end-user might directly use these classes in their program extensions, but not to extend their hierarchy. Another reason is that these classes are extensively used by the benchmarks from Table I, and we wanted to show the performance penalties that would occur from revealing *critical* classes in this way. All of the elements of matrices in these benchmarks are of type *LNumber* and are instantiated to the type *LDouble*. Table I shows the execution times for Fortran style, the 'Lite' OO version and the object-oriented version of the benchmarks in its first three columns. The fifth column shows the execution time of the OO version of the code after it has been transformed by our almost-whole-program framework. The sixth column shows the percentage increase in the running time of the OO version of the code in the distribution package. Since they do not utilize *LNumber* and *LDouble* classes, the numbers for 'Lite' OO style and for Fortran style computation exhibit only the measurement noise and are not shown in Table I.

The *Average* row shows the average slowdown factor of different versions of the code, relative to the Fortran style version. The exception is the sixth column where the average row shows the average of that column for each of the given platforms.

Table I shows some performance degradations for the object-oriented version of the benchmarks in the distribution package. On average, these benchmarks took about 9.7% more time to execute than their equivalents in the development package. This is because *every* element of *every* matrix involved in the computation now has a level of indirection for all method calls and *two* added levels of constructor calls (both for *ALNumber* and for *ALDouble*) for every instantiation of a new number. Since the object-oriented version of OwlPack utilizes copy-in, copy-out semantics for number computation (i.e. every operation on two numbers creates a new instance), these added constructors are called very frequently. Modern JVMs such as the HotSpot JVM used in obtaining results in Table I do a much better job of optimizing away this overhead than the older JVMs as previously reported, where the relative performance degradation was up to 40%. We expect that further improvements in JVM optimization technology will bring the performance of development and distribution packages even closer.

Table II shows the performance of a different application, Parsek from the Los Alamos National Laboratory [18]. In this case, we have marked the class *Particle* from the package as visible but non-extensible. *Particle* is a particle implementation, and represents the finest grained object-oriented element of the Parsek package, and as such should encounter the largest performance penalty from almost-whole-program transformation. However, Table II shows that the performance overhead from transforming the development version of Parsek into a distribution version is minimal. Since Parsek does not create any new objects during the computation once the initial object creation is done, the virtual machine in HotSpot 1.4.2 is able to almost completely eliminate any overhead caused by almost-whole-program transformation.

The results from Tables I and II suggest that the developer should utilize the almost-whole-program infrastructure carefully. If a key component of the program (a class that is involved in most of the computation and is extensively instantiated during the computation) is made non-extensible, the added overhead of extra constructor calls may increase the execution time. This is exactly what has happened

Table II. Execution times (in ms) for Parsek using JDK 1.4.2, javac -O.

Number of particles	Development package OO style	Distribution package	
		OO style	Dev. − Dist. OO % increase
1000	190	190	0
2000	340	340	0
5000	721	731	1.3
10 000	1622	1633	0.7
20 000	5418	5417	0
50 000	14 060	14 080	0.2
10 000	27 931	27 770	−0.6
Average			0.2

with our benchmark package. Unless the end-user *really* needs to see the classes *LNumber* and *LDouble*, these classes should be left as private. On the other hand, if the program does not extensively instantiate it, even the key class in the computation such as *Particle* in Parsek can be made visible to the end-user with virtually no performance penalty.

The almost-whole-program transformation and whole-program optimizations are fully implemented in our JaMake framework. The last column in Table I shows the execution times of the OO version of the *development* code, optimized with our whole-program optimization framework. JaMake has not yet implemented the details of the interaction between the almost-whole-program transformation and the whole-program optimizations, so we are unable to report the performance results for the optimized *distribution* package. Nevertheless, we believe that the last column closely reflects its expected performance, as the added constructor calls will not prevent *LDouble* and *LNumber* classes from being object-inlined, which is *the* major optimization in JaMake. Additionally, we do not have the Fortran style and 'Lite' OO style versions of Parsek available to compare the performance improvements of our whole-program optimizations, but it is reported elsewhere [18] that JaMake optimized object-oriented Parsek is right on par with hand-optimized Fortran style and Lite object-oriented versions of Parsek.

As Table I shows, our optimizations are extremely effective. They were able to reduce the execution times of object-oriented programs by an order of magnitude and bring the performance of the object-oriented version of the code to within a factor of two to three of the Fortran-style, hand-optimized code. Moreover, as is reported elsewhere [18], our optimizations can bring the performance of the object-oriented version of Parsek to within a few per cent of the performance of the Fortran-style, hand-optimized Parsek.

5. SUMMARY AND CONCLUSIONS

We have presented a transformation framework which is designed to make it possible to apply whole-program optimizations to collections of components that do not form a whole program.

The framework makes it possible for a programmer to finalize significant portions of a Java program so that, after the development phase, the finalized program may be optimized using powerful interprocedural techniques. Unlike previous whole-program optimization frameworks, our approach permits selected classes to be used outside of the compiled code and after the program has been compiled. It also permits the programmer to select the classes that will still be extensible after being compiled.

Our almost-whole-program transformation framework enables the developers to take advantage of the fact that most of the code they are writing will be finalized before the distribution by allowing the whole-program optimizations on the finalized part of the code. Our preliminary experiments show that almost-whole-program transformation results in modest performance penalties for programs that are significantly extended after the compilation, while allowing optimizations of the finalized parts of the code that can yield an order of magnitude performance improvement.

These methods have been implemented in the JaMake Java optimization framework, which performs source to source transformations before a program is presented to javac for class-by-class compilation and the JVM for execution. Along with other technologies in the JaMake compiler, the almost-whole-program transformation framework creates a programming environment that enables high-level, object-oriented programming without sacrificing performance, flexibility, extensibility, safety or reliability.

ACKNOWLEDGEMENT

This work was supported by NSF grant ACI-0234345.

REFERENCES

1. Budimlić Z, Kennedy K. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience* 1997; **9**(6):445–463.
2. Budimlić Z. Compiling Java for high performance and the Internet. *PhD Thesis*, Rice University, 2001.
3. Gosling J, Joy B, Steele G. *The Java™ Language Specification*. Addison-Wesley: Reading, MA, 1996.
4. Lindholm T, Yellin F. *The Java™ Virtual Machine Specification*. Addison-Wesley: Reading, MA, 1996.
5. Dean JA. Whole program optimization of object-oriented languages. *PhD Thesis*, University of Washington, 1996.
6. Low D. Protecting Java code via code obfuscation. *ACM Crossroads* 1998; **4**(3).
7. Tip F, Laffra C, Sweeney PF. Practical experience with an application extractor for Java. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM Press: New York, 1999.
8. Zaks A, Feldman V, Aizikowitz N. Sealed calls in Java packages. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM Press: New York, 2000.
9. Sun Microsystems. Java 2 software development kit version 1.2.2, July 1999. http://java.sun.com/products/jdk/1.2. Documentation: docs/guide/extensions/specs.html#sealing.
10. Dean J, Chambers C, Grove D. Selective specialization for object-oriented languages. *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation. SIGPLAN Notices* 1995; **30**(6):93–102.
11. Shultz U, Lawall J, Consel C. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*. ACM Press: New York, 2003.
12. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1999.
13. Flanagan C, Felleisen M. Modular and polymorphic set-based analysis: Theory and practice. *Technical Report TR96-266*, Rice University, 1996.
14. Agesen O. Concrete type inference: Delivering object-oriented applications. *PhD Thesis*, Stanford University, 1995.

15. Dolby J. Automatic inline allocation of objects. *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation. SIGPLAN Notices* 1997; **32**(6):7–17.
16. Budimlić Z, Kennedy K, Piper J. The cost of being object-oriented: A preliminary study. *Scientific Programming* 1999; **7**(2):87–95.
17. FPL Statistics Group. Linear algebra for statistics Java package. http://www1.fpl.fs.fed.us/linear_algebra.html.
18. Markidis S, Lapenta G, VanderHeyden WB, Budimlić Z. Implementation and performance of a particle-in-cell code written in Java. *Concurrency and Computation: Practice and Experience* 2005; **17**:821–837.

