Almost-Whole-Program Compilation *

Zoran Budimlić and Ken Kennedy

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

Abstract. This paper presents the motivation and an implementation strategy for a new *almost-whole-program* compilation framework for Java programs. This framework improves the flexibility of the generated code, while still permitting whole-program optimizations and incurring only modest performance penalties. It enables the programmer to balance performance and flexibility of the program *after* the development phase, instead of compromising the design process for the sake of performance. The almost-whole-program compilation framework uses Java visibility and scoping rules to transform a collection of classes into a package that is amenable to whole-program optimizations, and still allows for extensions to the optimized and compiled code. The paper concludes with experimental results showing the benefits that can be achieved using this approach.

1 Motivation

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 [9, 10]. 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 the 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, including object inlining and class specialization [3, 2], which the authors have shown can yield integer factor speedups on code that uses the full power of object-oriented program. Unfortunately, the whole-program compilation model provides no 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

* This work was supported by Compaq Computer Corporation and the Los Alamos Computer Science Institute

control this: they can only write the program, compile it and distribute it to the user. Thus, 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 virtual (enabling their overriding) and public.

On the other hand, the main goal during the distribution process is improving performance. To carry out whole-program optimizations, the entire program must 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 running the program. Such formulation of the program allows the static analysis to be effective, but it overly restricts the potential uses of the program.

Of course, programmers can choose to perform manual "optimization" of the code. When the development phase is finished and the program is debugged and ready for shipping, the programmer 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. This is evidenced by the Java JDK, in which many of the classes and methods that were public and extensible in the earlier versions are private and final in the later versions. A significant amount of programmer time has been spent to make this transformation manually in order to enable `javac` to inline these modified methods, but resulting in only modest performance improvements.

The diagram in Figure 1a. shows 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 the one on Figure 1b. 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

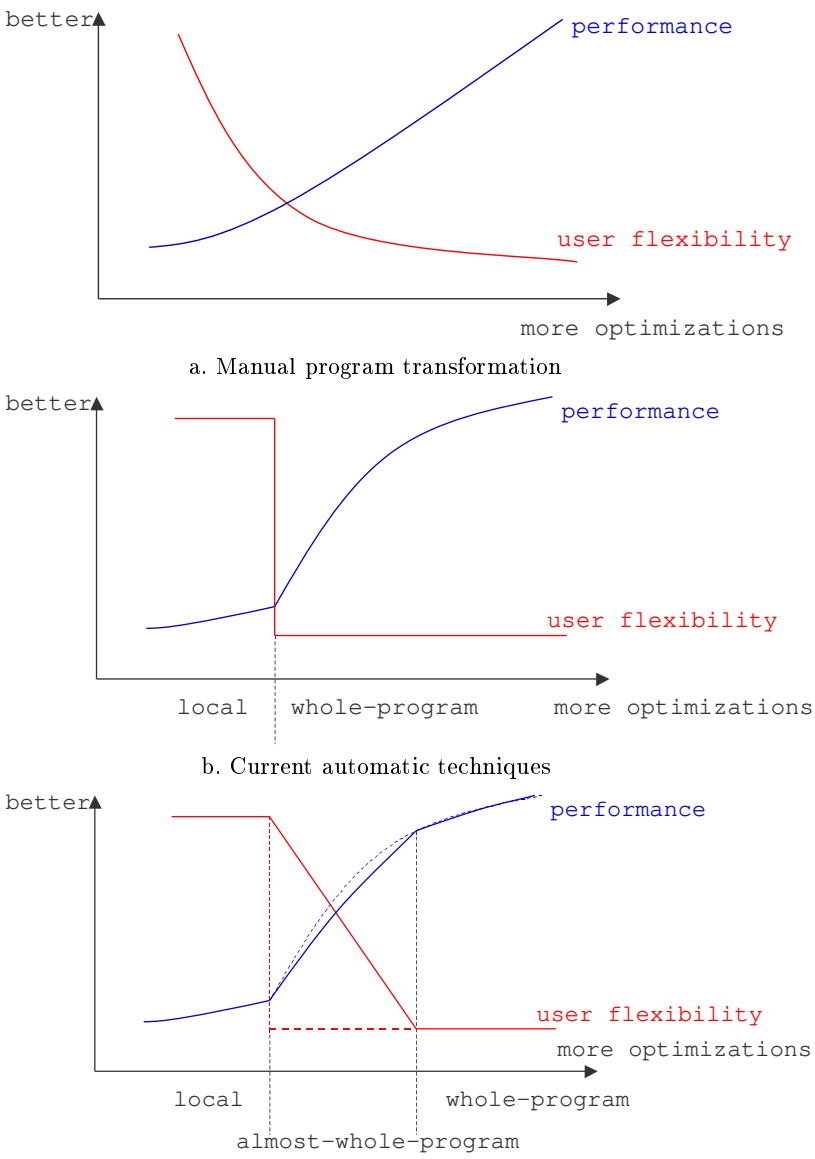

Fig. 1. Performance versus flexibility diagrams, showing the trade-off between performance and flexibility in different systems

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 1c. 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 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 incorporates machinery for specifying 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 nonextensible classes as package private, relying on Java class visibility rules and run-time checks to prevent extension of forbidden classes.

2 An Almost-Whole-Program Transformation Framework

Our 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 Java virtual machine specification is respected, so the generated bytecodes may be executed on any Java VM. 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.

2.1 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.

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 *packageprivate* (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 *packageprivate*.

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 be able only to run the program by calling $Main.main()$ and nothing else.

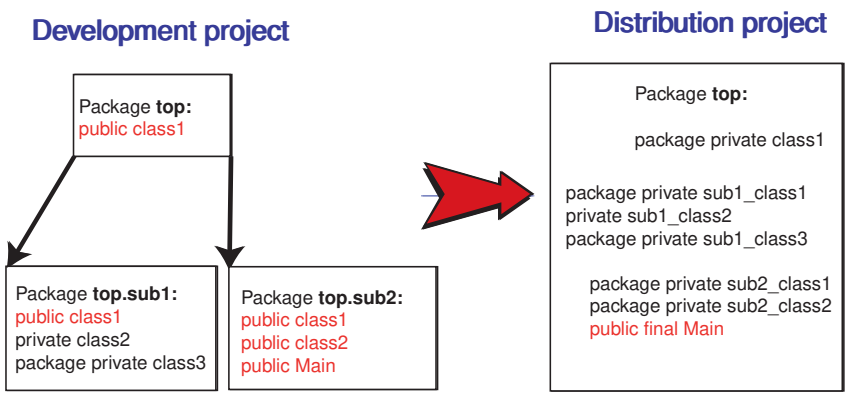

Fig. 2. The development-to-distribution transformation of private classes

2.2 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 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 subclasses already defined in the development project, and the back-end compiler will not allow the compilation of the subclasses 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 that summarize the member information of the non-extensible classes. Figure 3 shows the structure of this transformation.

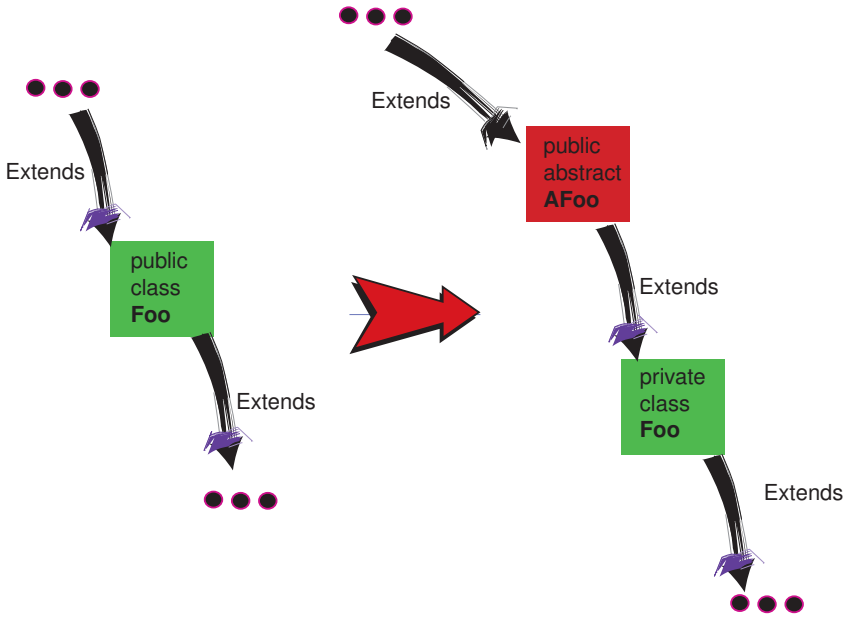

Fig. 3. The development-to-distribution transformation of a part of the class hierarchy

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 the public methods and fields are converted to be package private, the same as for private classes described in Section 2.1.

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 [8] 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 Java VM implementations are able to perform run-time

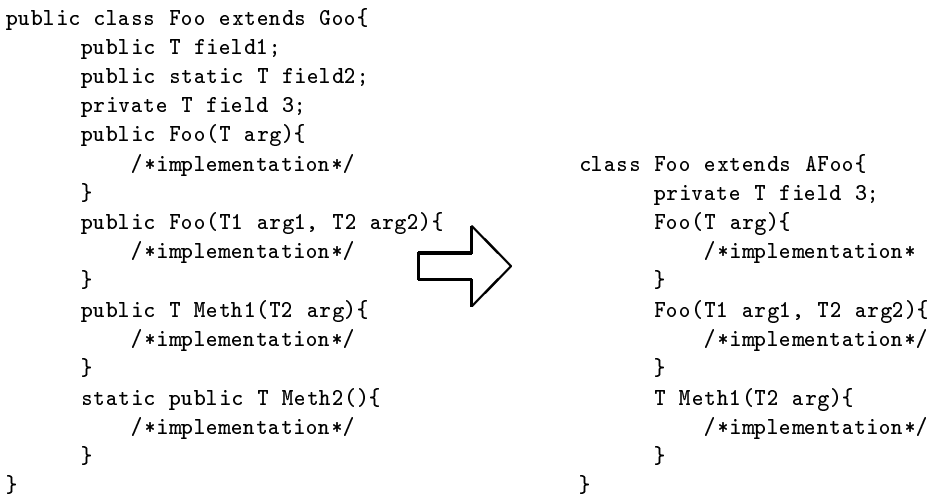

Fig. 4. Original class Foo, marked as non-extensible, and the generated class Foo

```
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*/ }
}
```

Fig. 5. Generated auxiliary abstract class AFoo

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 development to the distribution form, this tool inserts run-time tests in all the constructors of the auxiliary abstract class. On the example from Figure 5, all the constructors have a test of whether the object being currently instantiated is of the type *Foo*. If not, the constructor will throw a run-time exception. This technique prevents the end-user from instantiating his own version of subclasses of *AFoo*. The end-user can still write and compile subclasses 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 [2, 3, 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 the public methods from the class that is being converted into the auxiliary abstract class. It would also generate wrapper methods for all 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 the ones 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.

2.3 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 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 [7, 1] 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 possible types of that argument. The analysis then proceeds as usual.

Interprocedural optimization techniques [2, 5, 6] 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.

3 Performance Results

The almost-whole program transformation incurs 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).

Table 1 shows the execution times for our standard set of benchmarks, described in a great detail elsewhere [4], both for the original (development) and the almost-whole-program (distribution) package. We choose to mark the classes *LNumber* and *LDouble* as non-extensible for the distribution package. These classes are extensively used by the benchmarks from Table 1. All the elements of matrices in these benchmarks are of type *LNumber* and are instantiated to the type *LDouble*. Table 1 shows the execution times for Fortran style, "Lite" OO version and object-oriented version of the benchmarks in its first three columns, respectively. The fourth column shows the execution time of the OO version of the code after it has been transformed by our almost-whole-program framework. The fifth column shows the percentage the running time of the OO version of the code increased 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 1.

The *average* rows show the average slowdown factor of different versions of the code, relative to the Fortran style version. The exception is the fifth column where the average rows show the average of that column for each of the given platforms.

Table 1 shows significant performance degradations for the object-oriented version of the benchmarks in the distribution package. On average, these benchmarks took over 23% more time to execute than their equivalents in the development package in JDK 1.1, and almost 49% more in JDK 1.2. 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. The absolute value of this overhead is lower in JDK 1.2, but the relative impact is higher, since JDK 1.2 JIT does a much better job of optimizing the rest of the code than the JDK 1.1 JIT does.

The results from Table 1 suggest that the developer should utilize the almost-whole-program infrastructure very carefully. If a key component of the program (a class that is involved in most of the computation and could be potentially object-inlined) is made non-extensible, the added overhead of extra constructor calls can be quite substantial. This is exactly what has happened with the our benchmark package. Unless the end-user *really* needs to see the classes *LNumber* and *LDouble*, these classes should be left as private.

Almost-whole-program transformation and whole-program optimizations are fully implemented in our JaMake framework. The last column in Table 1 shows the execution times of the OO version of the *development* code, optimized with our whole-program optimization framework. Even though the details of the interaction between the almost-whole-program transformation and the whole-

Table 1. Sun Ultra 5 execution times, showing the overhead for almost-whole-program transformation and the effectiveness of almost-whole-program optimizations

	Development package			**Distribution package**		
	Fortran style	*"Lite" OO*	*OO style*	*OO style*	*Dev. - Dist. OO % inc.*	*Optimized* OO *
JDK 1.1.5						
dpofa	4.66	5.534	116.93	158.363	26.16	16.057
dposl	9.097	10.234	159.951	213.035	24.92	20.382
dpodi	10.992	12.26	254.46	474.266	46.34	46.410
dgefa	3.182	3.599	97.558	119.227	18.17	13.101
dgesl	4.329	4.854	60.992	70.495	13.48	7.714
dgedi	6.322	7.073	195.231	257.112	24.06	26.091
dqrdc	21.848	25.531	781.383	1035.304	24.52	88.225
dqrsl	14.012	16.058	222.061	234.468	5.29	59.144
dsvdc	9.079	15.584	279.86	375.884	25.55	39.074
average	1	1.202	20.483	25.229	23.17	3.568
JDK 1.2.2.05a						
dpofa	1.353	1.584	21.862	35.806	64.85	2.789
dposl	2.582	2.831	12.036	19.451	61.6	3.822
dpodi	2.614	3.462	70.965	113.824	60.39	5.743
dgefa	0.698	0.838	18.496	28.042	51.61	1.569
dgesl	0.865	0.967	4.488	6.108	36.09	1.195
dgedi	1.428	1.677	33.288	51.226	53.88	3.256
dqrdc	6.921	8.118	123.438	175.993	42.58	7.95
dqrsl	3.953	3.903	15.459	26.182	69.36	3.991
dsvdc	1.456	3.043	64.054	68.593	7.08	1.393
average	1	1.259	18.745	27.919	48.94	1.75

program optimizations are not fully implemented yet (thus preventing us from reporting the performance results for optimized *distribution* package) we believe that the last column reflects closely what its performance will be after the implementation is completed. With the knowledge from almost-whole-program phase, object inlining will be able to eliminate *all* of the overhead of the additional constructor calls and class checking for non-extensible classes, effectively rendering the performance of optimized code with and without the non-extensible classes identical.

As the Table 1 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 four on JDK 1.1, and even to within 75% on JDK 1.2 of the Fortran-style, hand-optimized code.

4 Summary and Conclusions

We have presented a transformation tool designed to make 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 strategies, 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.

The 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 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 Java Virtual Machine 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.

References

1. Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, 1995.
2. Zoran Budimlić. *Compiling Java for High Performance and the Internet.* PhD thesis, Rice University, 2001.
3. Zoran Budimlić and Ken Kennedy. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, June 1997.
4. Zoran Budimlić, Ken Kennedy, and Jeff Piper. The cost of being object-oriented: A preliminary study. *Scientific Programming*, 7(2):87–95, 1999.
5. Jeffrey A. Dean. *Whole Program Optimization of Object-Oriented Languages.* PhD thesis, University of Washington, 1996.
6. Julian Dolby. Automatic inline allocation of objects. *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, 32(6):7–17, June 1997.
7. Cormac Flanagan and Matthias Felleisen. Modular and polymorphic set-based analysis: Theory and practice. Technical Report TR96-266, Rice University, 1996.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Mass.: Addison-Wesley, June 1999.
9. James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification.* Mass.: Addison-Wesley, 1996.
10. Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification.* Mass., Addison-Wesley, 1996.