JaMake: A Java Compiler Environment*

Zoran Budimlić and Ken Kennedy

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

Abstract. A language of choice for general-purpose programming, Java is quickly becoming popular in more specialized areas, such as scientific computing. However, even though the compilation technologies have significantly improved Java execution, performance is still the main obstacle to the use of Java for scientific applications. Although good Java Virtual Machine implementations are approaching the performance of Fortran on similarly-coded applications, significant performance problems remain because of the power of the object-oriented programming paradigm. Our experiments show that full use of polymorphic, object-oriented programming can result in performance penalties of up to two orders of magnitude. To address this performance difficulty, the authors have developed the JaMake Java transformation system, which uses advanced program analysis and transformation techniques to allow programmers to create extensible and maintainable programs using object-oriented design, while generating Java programs whose performance approaches that of hand-optimized, Fortran-style code. Experiments on our collection of object-oriented scientific programs have shown that transformation by JaMake can yield speed-ups of a factor of ten or more, bringing the performance of these object-oriented programs to within 75% of hand-optimized, Fortran-style code.

1 Introduction

Over the past several years, Java has become a language of choice for general-purpose programming because of its advanced features, including security, portability, availability, ease of use, familiar syntax and object-oriented approach. The popularity of Java has spread into areas traditionally dominated by other languages, such as scientific computing, where researchers are beginning to use Java to implement scientific algorithms on high performance computers.

The latest just-in-time compilation technologies have brought the performance of hand-optimized, Fortran-style Java programs to within a factor of four of equivalent hand-optimized native Fortran programs [6]. The performance gap between Java and Fortran is slowly closing, making Java even more attractive for high-performance applications.

In spite of these gains, the largest source of performance degradation of Java programs lies in the style of programming it encourages [7]. Programmers prefer using the full power of object-oriented programming to writing hand-optimized,

* This work is sponsored by Compaq and LACSI.

Fortran-like code in Java syntax because the object-oriented style results in programs that are elegant, extensible, easy to maintain and debug. Unfortunately, programs written using the full power of object-oriented programming in Java can be up to two orders of magnitude slower than their Fortran-style counterparts [7]. This problem threatens the long-term prospects for acceptance of Java by the scientific community—scientific programmers will not convert to a new language unless it offers significant benefits in programming power or application performance. If using the full power of the language leads to huge performance degradations, no one will do it.

To address this problem the authors have developed the JaMake program transformation system, a Java compilation environment that uses advanced program analysis and transformation techniques, such as exact type inference to determine the targets of method calls, *class specialization* to create class clones based on the exact type of data the original class contains, and *object inlining* to transform objects and arrays of objects into primitive data variables and arrays of primitive data variables. It also addresses the problem of reduced flexibility and usability of programs that are restricted to whole-program compilation, which is the common case for such advanced techniques, through *almost-whole-program compilation*, a technique that allows the programmer to determine the tradeoff between program flexibility and performance. JaMake is also an excellent infrastructure for research of more traditional compilation techniques, such as code motion and SSA manipulation.

We designed have JaMake (except for the back-end compiler) as a source-to-source transformation tool. This allowed us to concentrate on developing high-level compiler strategies while relying on the existing technologies (javac, Java VM, JIT or native compilation) to enable efficient execution of the transformed code. Thus, JaMake can be viewed as a programming tool aimed at assisting programmers to write a better *and* more efficient code. Our experiments on an object-oriented, polymorphic version of the Linpack library show that such programs demonstrate a speed-up of a factor of ten or more when transformed with the JaMake compiler, and have a performance of within a factor of two or better when compared to their hand-optimized Fortran-style equivalent. These results are very encouraging for further research in development of similar techniques based on the whole-program and almost-whole-program compilation approach.

2 Compiler Techniques Used by JaMake

In order to allow easier reading of the remainder of this paper, we will in this section briefly describe some of the advanced compiler techniques used in the JaMake compilers. The complete description of these techniques can be found elsewhere [4, 3].

Class specialization clones a class containing polymorphic fields based on the possible subtypes of those fields. It generates several specialized versions of the original class, exposing the subtype distinctions in the revised class hierarchy [5, 9]. The specialized classes are monomorphic with respect to the selected fields,

enabling subsequent optimizations such as object inlining. This transformation extends the classical notion of cloning [10, 11], where a function is cloned based on the values or types of its parameters, by allowing a class to be cloned based on the exact type of the polymorphic data it contains [5, 9]. Class specialization results in only modest performance benefits; its main result is that it enables subsequent object inlining.

Object inlining is a novel program transformation that converts Java objects into *inlined objects*, a new data representation for the objects, and transforms one program into another that operates on this new representation. This transformation eliminates the indirection in object accesses by "flattening" the data structures in the program. It enables method inlining on the inlined objects by eliminating the privacy restrictions of the original code. Moreover, it improves the memory hierarchy performance by increasing the locality of the data. It reduces the overhead of dynamic memory management by allocating some objects on the stack and reducing the total number of objects in the program. In addition, it reduces the memory footprint of the program by reducing the number of objects among which the program data is distributed. Finally, it creates more space for local compiler optimizations by performing these transformations. Object inlining can improve the performance of polymorphic, object-oriented, scientific Java programs by up to two orders of magnitude.

Almost-whole-program compilation is a novel compilation strategy in which the compiler assumes a static class hierarchy at compile time [5] and the programmer specifies the classes that would be publicly visible. This strategy uses Java visibility rules, novel implementation techniques, and novel class packaging techniques to allow for extensive program optimization in Java. These techniques allow both good design *and* good performance by dividing the compilation process into two fundamentally different processes: development and distribution.

If Java is to be used for high performance computing, some of the well-understood code-moving optimizations must also be implemented. As our other work [3] demonstrates, implementation of these optimizations is not straightforward in Java. In particular, the Java exception mechanism prevents or seriously limits most code motion optimizations. *Exception hiding* is a program transformation that enables more efficient code motion. This technique uses loop peeling [1] and guard insertion to create "exception-free zones" for code motion. Code motion transformations can then freely move the code within these zones, without concern for exceptions.

Static Single Assignment (SSA) form, the standard intermediate representation for programs in a compiler, enables efficient implementation of many classical code optimization techniques [12, 2]. An efficient SSA representation is needed to improve the implementation of classical optimizations. We have developed a novel algorithm [3] for converting SSA form into a control flow graph (CFG). This is a very fast algorithm — taking time that is nearly linear linear in the size of the CFG, while inserting many fewer copies than the standard SSA-to-CFG conversion algorithm. This algorithm uses proven properties of the SSA form to check for variable interference only when necessary.

3 JaMake Structure and Implementation

The JaMake infrastructure is written completely in Java. It consists of more than 90,000 lines of Java source code distributed over approximately 400 files with more than 500 classes. Most of the code was designed at Rice University, with some components written by one of the authors while at Sun Microsystems, and some components modified from Sun Microsystems and Compaq.

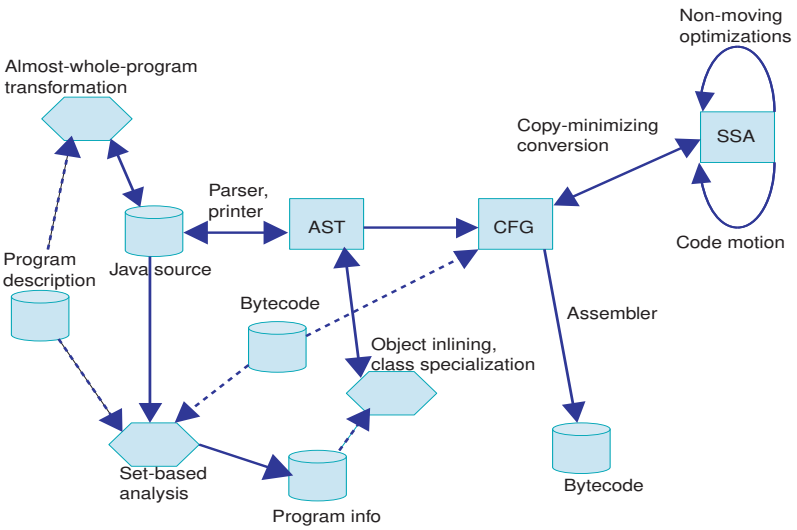

Fig. 1. The structure of the JaMake project

Figure 1 shows the structural diagram of the JaMake compiler infrastructure, with our main data structures as the nodes in the graph. Disk icons represent the data structures that are stored on the disk, while rectangle icons symbolize the internal data structures. Labeled edges in the diagram symbolize the transformations between data structures. The unlabeled solid edges symbolize writes, and the dashed edges represent reads. Hexagons represent analysis and transformation components that require the knowledge of multiple data structures.

For implementation simplicity, most of the transformations in the JaMake compiler operate on data structures that are stored on the disk in the intermediate form. For example, object inlining and class specialization [4, 5] read the source code from the disk, create the abstract syntax trees (AST's), transform the AST's and then generate the source code again. This technique eliminates many dependencies and incompatibilities (due to different origins) between the different components of the infrastructure and greatly simplifies the debugging of the components. Of course, this is not an optimal arrangement since there are some unnecessary I/O operations during the execution the JaMake compiler,

but that is a small price to pay for the flexibility of the design. Naturally, a production compiler would eliminate the unnecessary source-to-AST-to-source and CFG-to-SSA-to-CFG conversions and avoid storing and reading the intermediate data structures to and from the disk.

The main disk data structures are the source code for all the classes in the program, the assembler-generated bytecode, the bytecode for the classes that are not available in the source form at the compile time (or are intentionally left only in bytecode form for almost-whole-program transformation purposes), the almost-whole-program description, and the concrete type information generated by the set-based analysis. The main internal data structures are the abstract syntax trees, the control flow graph (CFG) and the static single assignment (SSA) form. The main transformations are almost-whole-program encapsulation [3], object inlining and class specialization [5, 3], copy minimizing conversion, code motion and exception hiding [3], parsing, printing, AST-CFG conversions, local optimizations on the SSA and CFG, and assembling.

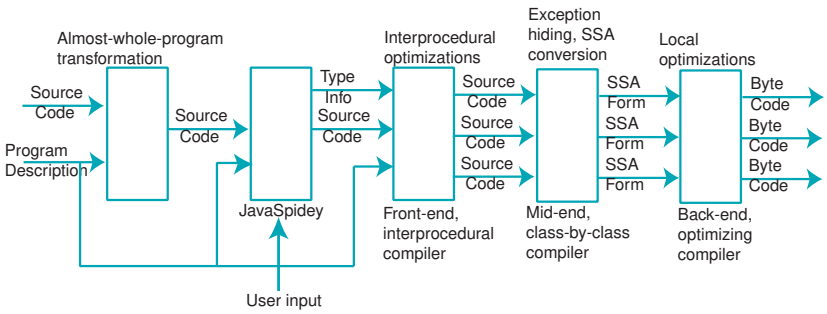

Fig. 2. The flow of the source code through the JaMake compiler

Figure 2 shows the flow of the source code through the JaMake compiler. Almost-whole-program infrastructure takes the source code for the almost-whole program and the program description. The program description contains the information that tells the almost-whole-program transformation component which classes are private, non-extensible or public. This component parses the source code, generates the abstract syntax trees and does its program modifications on AST's. After it handles all the classes in the development package, it generates the source code for the distribution package [3].

Almost-whole-program framework then passes the source code to JavaSpidey, which does the whole-program set-based analysis of the program. JavaSpidey also needs the program description to detect the classes and method that the developer has designated as public and to augment the set-based analysis accordingly. JavaSpidey was written by Cormac Flanagan, a former Rice student, at Compaq SRC lab. It was originally meant to be used for static debugging purposes; JaMake project uses it at the front end of the compiler infrastructure.

JavaSpidey has a graphic user interface that allows the user to augment the type information it has generated. This is an excellent tool for correcting the imprecisions in the analysis, as we have done in Section 4. JavaSpidey passes the type information directly to our front-end interprocedural compiler.

Since it is written completely in Java, JaMake executes identically on all Java platforms, with the exception of JavaSpidey. Even though JavaSpidey is written in Java as well, it depends on some native system information for getting the types of the binary classes (such as *Object.java*) and as such it generates the type information only on the Solaris platforms. This limitation will be corrected in the future versions of JavaSpidey.

The interprocedural compiler performs class specialization and object inlining [4, 3]. It converts the source code of all the methods of all the classes into abstract syntax trees and performs these optimizations directly on the AST's. It uses a modified *Visitor* pattern [14] to traverse and modify the AST's. On the front end of this compiler is a parser from the *javac* compiler from Sun Microsystems, that generates the AST's. After doing class specialization and object inlining, our interprocedural compiler generates the modified source for classes in the program and passes them to our mid-end, class-by-class compiler.

This mid-end compiler is again based on the *javac* compiler from Sun Microsystems. It compiles one class at a time. As before, it parses the source and creates an AST for all the methods in the class. It then converts the AST into a control flow graph. It also has the ability to create a CFG directly from the bytecode of the class by using symbolic code interpretation of the stack machine code. This compiler then performs exception hiding [3]. It then uses the classical SSA construction algorithm [2, 12] to generate the SSA form for all the methods in the class. It then passes the SSA form to our back-end optimizing compiler.

This back-end compiler operates on the SSA representation of the code. It performs numerous classical optimizations (dead code elimination, value numbering, constant propagation, value-driven code motion, local common subexpression elimination e.t.c.). As a last step, this compiler executes copy folding and then performs our SSA-to-CFG conversion algorithm [3] to generate the final CFG. The CFG is then passed to our assembler for bytecode generation.

4 Experimental Results

This section presents the performance results for experiments that show that advanced compilation techniques can virtually eliminate the overhead introduced by object-oriented programming in high performance scientific applications.

Table 1 shows the execution times for our standard collection of OwlPack benchmarks [7, 6]. OwlPack is an object-oriented implementation of the Linpack library with matrices, vectors, pairs and, most importantly, individual numbers implemented as objects.

The "Fortran style" columns show the execution times for a version of Linpack written in Java style that very closely resembles Fortran [13] (all methods

Table 1. Sun Ultra 5 execution times, showing the 5-10 times speedup of the optimized code over the original code

	jdk 1.1.5 interpreter				**jdk 1.2 JIT**			
	Fortran style	*"Lite" OO*	*OO style*	*Optimized OO style*	*Fortran style*	*"Lite" OO*	*OO style*	*Optimized OO style*
dpofa	4.669	5.548	115.155	16.057	1.353	1.584	21.862	2.789
dposl	8.82	10.362	127.392	20.382	2.582	2.831	12.036	3.822
dpodi	12.736	14.315	297.346	46.410	2.614	3.462	70.965	5.743
dgefa	3.226	3.511	81.981	13.101	0.698	0.838	18.496	1.569
dgesl	4.079	4.640	35.868	7.714	0.865	0.967	4.488	1.195
dgedi	6.234	6.971	161.868	26.091	1.428	1.677	33.288	3.256
dqrdc	21.197	25.139	538.53	88.225	6.921	8.118	123.438	7.95
dqrsl	14.504	15.757	162.044	59.144	3.953	3.903	15.459	3.991
dsvdc	9.008	15.439	226.495	39.074	1.456	3.043	64.054	1.393
average	1	1.202	20.483	3.568	1	1.259	18.745	1.75

are static, arrays are passed directly as arguments, the data is accessed directly and there is a version of the code for each primitive number type).

"Lite OO" columns show the execution times for a version of OwlPack we suspect the performance-conscious programmers would be most likely inclined to write today — all the data is 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.

"OO style" columns show the execution times for object-oriented OwlPack. 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 fourth and eighth columns show the execution times for the full object-oriented style version of OwlPack, optimized with the JaMake compiler. As of this writing, class specialization has not been fully implemented in our infrastructure, so we have performed the class specialization of the OO code by hand, while the object inlining is performed automatically by JaMake.

The last row in the table shows the average execution times of the "Lite" OO, full OO and optimized OO versions of the code, relative to the Fortran style version on the same platform. The table shows that on the average, the "Lite" OO version is 20-25% slower than the Fortran style version.

The optimized object-oriented version shows tremendous improvement over the object-oriented version. Optimized code is 5 to 10 times faster than the original, around 6 times faster on average when interpreted and around 9 times faster when executed on a VM with a JIT. Even though the optimized object-oriented version approaches the Fortran style and "Lite" OO style in performance, there is still a significant performance difference. Optimized code is still about 75% slower than the Fortran style code on a JIT VM.

There is a very simple reason for this disparity. To ensure the correctness of the transformation without complicating the implementation of the object-inlining transformation, the JaMake compiler inserts many extraneous instructions in the code. It handles method inlining by inserting copies of the actual arguments into formal arguments of the method, followed by the method body. It handles returns by assigning the value to the return variable and breaking out of the method body. This results in many unnecessary instructions that will be easy to eliminate in the future in the back end of our optimizing compiler.

5 Conclusions and Future Work

Our experiments with the JaMake infrastructure have provided practical evidence that the full power of Java can be used in high-performance, scientific computing without suffering unacceptable performance penalties. The powerful features supported by Java make it possible for programmers to write elegant, extensible and maintainable programs. Advanced compilation techniques such as those that are implemented in JaMake can optimize those programs to achieve performance comparable to that of hand-optimized, Fortran-style code.

The JaMake project and the compilation technologies that we have developed for it form an excellent foundation for further advancements in compilation technology. There are several directions we are exploring for further research:

- *Better array inlining.* Using the regular-section array analysis techniques [16], it may be possible to inline heterogeneous arrays of object, or only inline parts of the arrays of objects.
- *Fast register allocation* can be developed using our SSA reconstruction algorithm which enables fast variable interference checking.
- *Semantic-changing transformations* can be easily implemented within JaMake. In that case JaMake would serve as a programming tool that assists the programmer in writing better and faster source code.
- *Exception recovery* would be the next step in reducing the effect of exceptions on code motion. Symbolic debugging techniques [17] can allow state-changing optimizations on the code, and "undo" them if an exception should occur.
- *More precise and faster type analysis* using iterated Hindley-Milner [18] type inference or the Kennedy-Hall call graph construction strategy [15] can find more opportunities for object inlining, further enhancing its effectiveness.
- *Coordinated compilation* [8] is an attractive solution to the bytecode bottleneck in Java execution. Native instructions can be encoded in the bytecode, which can then safely execute on *any* virtual machine, and much faster on a VM that understands the encoding.

With further development of static whole-program and almost-whole-program optimizations, as well as additional progress in the back-end of the Java execution model (interpretation, JIT and native compilation) we expect Java to achieve performance competitive with native Fortran and C++, threatening the preeminence of those languages for high-performance, scientific computing.

References

1. F. Allen and J. Cocke, *A catalogue of optimizing transformations*, in Design and Optimization of Compilers, Prentice-Hall, 1972, pp. 1–30.
2. P. Briggs, K. Cooper, T. Harvey, and T. Simpson, *Practical improvements to the construction and destruction of static single assignment form*, Software: Practice and Experience, 28 (1998), pp. 859–881.
3. Z. Budimlić, *Compiling Java for High Performance and the Internet*, PhD thesis, Rice University, 2001.
4. Z. Budimlić and K. Kennedy, *Optimizing Java: Theory and practice*, Concurrency: Practice and Experience, 9 (1997), pp. 445–463.
5. ———, *Static interprocedural optimizations in Java*, Tech. Rep. CRPC-TR98746, Rice University, Dec. 1998.
6. ———, *Prospects for scientific computing in polymorphic, object-oriented style*, in Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, Texas, Mar. 1999.
7. Z. Budimlić, K. Kennedy, and J. Piper, *The cost of being object-oriented: A preliminary study*, Scientific Programming, 7 (1999), pp. 87–95.
8. R. Cartwright, *Encoding machine-specific optimization in generic byte code by using local variables as pseudo-registers*, United States Patent, 6075942 (2000).
9. C. Chambers and D. Ungar, *Customization: Optimizing compiler technology for* SELF*, a dynamically-typed object-oriented programming language*, Proceedings of the ACM SIGPLAN 89 Conference on Program Language Design and Implementation, 24 (1989), pp. 146–160.
10. K. Cooper, M. Hall, and K. Kennedy, *Procedure cloning*, in Proceedings of the 1992 International Conference on Computer Languages, Oakland, California, Apr. 1992, pp. 96–105.
11. ———, *A methodology for procedure cloning*, Computer Languages, 19 (1993), pp. 105–117.
12. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *An efficient method of computing static single assignment form*, in Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages, Austin, Texas, Jan. 1989, pp. 25–35.
13. FPL Statistics Group, *Linear algebra for statistics Java package*. http://www1.fpl.fs.fed.us/linear_algebra.html.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Mass.: Addison-Wesley, June 1999.
15. M. W. Hall and K. Kennedy, *Efficient call graph analysis*, ACM Letters on Programming Languages and Systems, 1 (1992), pp. 227–242.
16. P. Havlak and K. Kennedy, *An implementation of interprocedural bounded regular section analysis*, IEEE Transactions on Parallel and Distributed Systems, 3 (1991).
17. J. L. Hennessy, *Symbolic debugging of optimized code*, ACM Transactions on Programming Languages and Systems, 4 (1982), pp. 323–344.
18. R. Milner, *A theory of type polymorphism in programming*, Journal of Computer and System Sciences, 17 (1978), pp. 348–375.