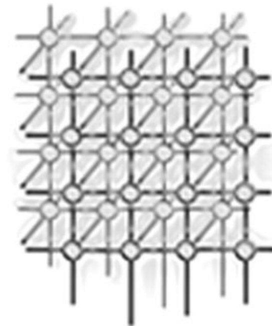
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **15**:207–222 (DOI: 10.1002/cpe.657)



Automatic translation of Fortran to JVM bytecode

Keith Seymour*,† and Jack Dongarra

Department of Computer Science, University of Tennessee, Knoxville, Knoxville, TN 37996, U.S.A.

SUMMARY

This paper reports on the design of a Fortran-to-Java translator whose target language is the instruction set of the Java Virtual Machine. The goal of the translator is to generate Java implementations of legacy Fortran numerical codes in a consistent and reliable fashion. The benefits of directly generating bytecode are twofold. First, compared with generating Java source code, it provides a much more straightforward and efficient mechanism for translating Fortran GOTO statements. Second, it provides a framework for pursuing various compiler optimizations, which could be beneficial not only to our project, but to the Java community as a whole. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: Fortran; Java; JVM; bytecode; numerical libraries

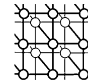
1. INTRODUCTION

The Java programming language [1] has grown drastically in popularity in recent years, in industry as well as in academia. The properties of Java, such as portability, memory management, and security make it an attractive programming environment for a wide range of applications. Despite some properties that would make Java seem less attractive to programmers in the high-performance and scientific computing community (such as the lack of a complex primitive data type and the lack of operator overloading), interest in using Java for scientific and engineering applications has also increased, as evidenced by the number of mathematical libraries [2–6] and scientific and engineering applications [7,8] developed in Java over the past few years.

The primary by-product of our earlier work on the Fortran-to-Java translator has been one such mathematical library—JLAPACK [9]. The JLAPACK library provides application programming interfaces (APIs) to numerical libraries from Java programs. The numerical libraries will be distributed as class files produced by a Fortran-to-Java translator, f2j. The first version of f2j was used to translate

*Correspondence to: Keith Seymour, Department of Computer Science, University of Tennessee, Knoxville, Knoxville, TN 37996, U.S.A.
†E-mail: seymour@cs.utk.edu

Copyright © 2003 John Wiley & Sons, Ltd.
Received 31 August 2001
Revised 21 January 2002



the BLAS [10–12] and LAPACK [13] numerical libraries from their Fortran 77 reference source code to Java source code and was subsequently distributed as a library of class files. These libraries are established, reliable and widely used linear algebra packages and are, therefore a reasonable first testbed for f2j. This report describes an extension to the f2j compiler that allows the generation of class files directly from Fortran source code.

2. MOTIVATION

First we describe the motivation behind writing a Fortran-to-Java translator and then describe why we have chosen to extend the code generator to directly emit bytecode.

The original goal of f2j was to facilitate the translation of legacy Fortran numerical libraries to Java, with LAPACK and BLAS being the primary libraries of interest. Given the goal of producing a Java implementation of LAPACK, there are three options:

1. wrap the native routines in Java interfaces;
2. rewrite the routines in Java from scratch;
3. develop a tool to automate the translation.

We avoided the first method because we wanted the Java version of LAPACK to be used by applets as well as applications, thus requiring a pure Java implementation. The second option would have required hand-translating, testing and debugging hundreds of routines. Given the large amount of code in LAPACK, the second option could be very time-consuming and error-prone. We chose the third option because it allows us to generate pure Java code in a consistent and reliable way from the original Fortran source. In addition, after pursuing the third option, we have a tool which could be applied successfully to other numerical libraries and eventually to a wide range of Fortran code.

There are two primary factors motivating the development of a bytecode generator for f2j—handling GOTO statements and exploring code optimization techniques.

The handling of Fortran GOTO statements has been a difficult problem due to Java's lack of a goto statement. As described in [9], we can use Java's labeled `break` and `continue` statements to translate certain types of Fortran GOTOs, but there are still some branches that do not correspond to a `break` or `continue` statement. For these GOTOs, the technique we have been using is to generate 'placeholders' in the Java source code. The placeholders are method calls which specify the target of the GOTO statement. For example, the Fortran statement

```
GO TO 100
```

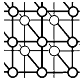
would be translated to the following Java method call:

```
Dummy.go_to("Progunit",100);
```

whereas the corresponding label becomes:

```
Dummy.label("Progunit",100);
```

The first argument is the name of the current program unit and the second argument is the branch target or label number. Once the resulting Java source code is compiled with `javac`, we use a GOTO



translation tool to parse the bytecode and identify the placeholders, which are then emitted as JVM branch instructions. This is discussed in greater detail in [9].

The post-processing has worked acceptably except that the multi-stage process is cumbersome and something that many users find confusing. In fact, it is so easy to forget to run the GOTO translation tool that we implemented the Dummy methods such that they warn the user when GOTO translation has not been performed. The first argument to the Dummy method is used to inform the user which program unit has not been transformed. The GOTO translation process has remained separate from f2j for two reasons. First, we wanted to allow the users to modify the resulting Java source before GOTO translation. Second, since the GOTO translation requires Java compilation before the patching, we kept the process separate from f2j to allow for users to choose their own Java compiler and flags.

Since the JVM instruction set includes an unconditional branch instruction, generating the bytecode directly does not require any tricky manipulation or post-processing. This greatly simplifies and speeds the translation process.

Another benefit to generating bytecode is that it allows us to explore various optimization techniques since we can directly control how the stack is used, which instructions are generated, and in which order. We plan to employ traditional compiler optimizations such as loop unrolling and code motion, for which well-understood optimization techniques exist [14], as well as exploring alternate techniques which may be specific to Java. The application of such techniques to Java source code compilers is complicated by Java's precise exception and multithreading semantics [15].

While there are several Java assembler formats available which would make debugging a bit easier (one popular example being Jasmin [16]), we chose to generate the bytecode directly in order to minimize the dependence on external packages. The benefit to the user is that there is one fewer package to install and the benefit to us (the developers) is that we do not depend on ongoing support for any external packages.

The development of the bytecode generator has not rendered obsolete the prior Java source code generator. In some cases, it may be desirable to modify the translated source code, which is much easier when the target language is Java source code. Therefore, we have designed the bytecode generator to coexist with the previous code generator without interference, providing the user with the choice of target language.

3. IMPLEMENTATION

The f2j compiler operates in four stages, as described below and illustrated in Figure 1.

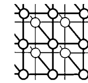
3.1. Lexical analysis and parsing

In this stage the lexer separates the Fortran source code into tokens and the parser builds a complete AST (abstract syntax tree) and symbol tables for each program unit. Subsequent compilation stages obtain all information about the program structure from the AST built during parsing.

3.2. Optimizing the use of scalar wrappers

In Fortran, values are passed to functions and subroutines by reference. This implies that if a Fortran subroutine modifies one of its parameters, then that modification also takes effect in the calling routine.

Copyright © 2003 John Wiley & Sons, Ltd.



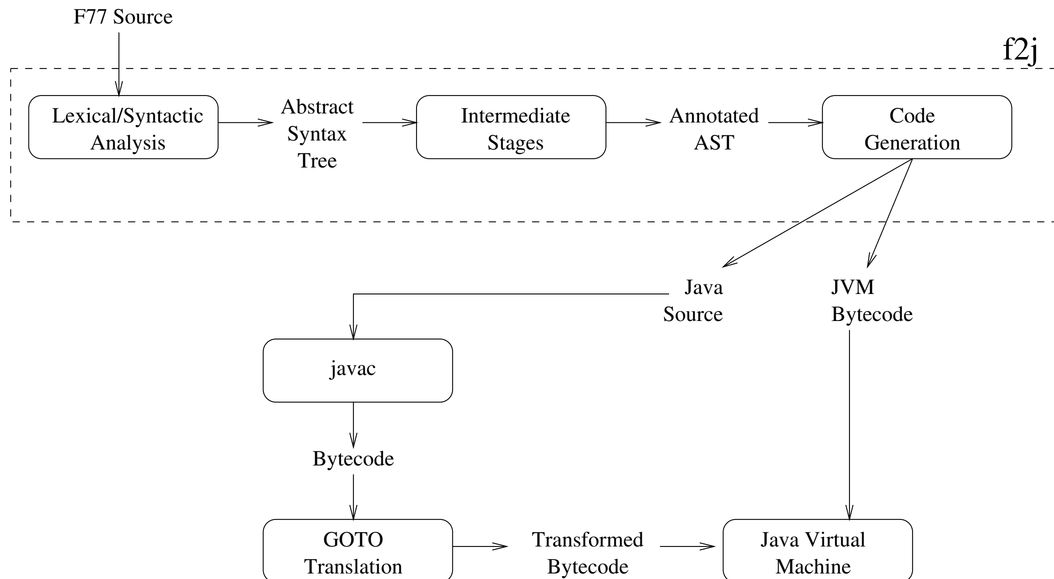


Figure 1. Stages of translation.

However, Java uses pass-by-value, which implies that any modifications would *not* take effect in the caller. In order to simulate pass-by-reference in Java, we must wrap the scalar in an object. Then instead of passing the integer value, we would pass the object wrapper whose scalar field may be modified in the subroutine.

During the scalar 'optimization' phase, f2j determines which parameters of each subroutine absolutely need to be wrapped. The rest are passed as Java primitive data types (`int`, `double`, etc) in order to improve access times and save memory. The determination is made as follows.

A variable must be wrapped if:

1. the variable is an argument to this function and it is on the left side of an assignment statement in this program unit;
2. the variable is an argument to this function and it is an argument to a READ statement;
3. the variable is passed to a function or subroutine that modifies it.

The last rule implies that every function or subroutine that the current program unit depends on must be checked before this unit can be completely verified. f2j resolves the dependencies before continuing to check the current unit. Of course, the assumption is that there are no cycles in the dependency graph.

3.3. Type assignment

This stage is not 'typechecking' in the semantic analysis sense. In this stage, f2j performs a traversal of the AST and assigns type information to each node, propagating information up the tree. For example,

Copyright © 2003 John Wiley & Sons, Ltd.

f2j looks at both sides of an addition operation and assigns the widest type to the addition node and so on up the tree. This information helps the code generator emit the appropriate type-specific opcodes and type casts when necessary. Such type casts are not as common when generating Java source, but explicit casts are necessary when making a narrowing conversion.

3.4. Code generation

Code generation is by far the largest and most complicated stage in the translator. In this stage, f2j traverses the AST, generating code as it traverses through the tree. The code generator depends on the information determined in all the prior steps to generate correct code. Currently, Java source code and JVM bytecode are generated during the same pass (over the AST) because using separate passes would have resulted in a lot of duplicated code and made maintenance more difficult.

4. GENERATING BYTECODE

This section elaborates on the design and implementation of the final stage of the translator, the code generator.

4.1. Design

In this section, we briefly mention some of the design considerations made during the development of f2j.

4.1.1. General

Each Fortran program unit is generated as a separate Java class containing a single static method. For example, the Fortran subroutine `DGEMM` would be translated to a Java class named `Dgemm` containing only a single method named `dgemm`.

All arrays are laid out in memory in column-major fashion, with multi-dimensional Fortran arrays being translated as linearized one-dimensional Java arrays.

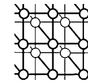
4.1.2. GOTO statements

Java source code does not provide a GOTO statement. Thus, we must perform some post-processing on the class files that were generated from Java source (using `javac` or an equivalent Java compiler) in order to correctly generate the GOTO statements. However, Fortran GOTO statements are easily translated to JVM bytecode since there exists a `goto` opcode.

4.1.3. Variables

All variables are emitted as static class variables and initialized in a special class initialization method named `<clinit>`, which is only required when directly generating bytecode. When writing Java source code, `javac` generates `<clinit>`. Since the variables are declared as static class variables,

Copyright © 2003 John Wiley & Sons, Ltd.



any SAVE statements in the Fortran source may be ignored since essentially *all* variables are already saved. This was originally done for simplicity so that the code generator could emit all variables at the same time and in the same manner. However, such convenience usually comes at a price and in this case, it is performance that suffers. We find that the JVM instruction to load a static class variable takes longer than the instruction to load a local variable, thus decreasing the overall performance of the generated code.

As mentioned in Section 3, all variables are emitted as primitives unless f2j determines that they must be wrapped in objects.

4.1.4. DATA statements

The initialization performed by DATA statements is emitted as part of the special method named `<clinit>`. This works well under the current assumption that all variables are all emitted as static class variables.

4.1.5. Intrinsic functions

Some Fortran intrinsic functions may be translated directly to a corresponding method in the Java core API. For example, many mathematical functions such as `ABS`, `SQRT` and `LOG` have direct analogues in the `java.lang.Math` package. However, many intrinsics do not directly correspond to any existing Java method and, in these cases, we have implemented the intrinsics in Java. The intrinsics we implemented in Java include `DIM`, `LOG10` and hyperbolic trigonometric functions. It is worth noting that the code used to implement these unsupported intrinsics could have been inlined in the bytecode. However, there are several reasons for not inlining. First, as there is an upper bound on the code size of a Java method, it is prudent to avoid code expansion where possible. Also, modern JIT compilers are likely to inline these small methods where appropriate. Finally, implementing these unsupported intrinsics in Java simplifies the implementation and maintenance of the code generator.

4.1.6. Common blocks

Each common block in the Fortran source is emitted in a separate class file, containing all the variables of the original common block as static variables. If multiple declarations of the same common block exist in the Fortran source, f2j merges the variable names from each declaration into one name.

Tables I and II show examples of the code generated for common blocks. All variables in a common block are wrapped in objects because it may be difficult to determine whether one of the variables is used in a pass-by-reference context when program units are translated separately.

4.1.7. EQUIVALENCE statements

Generally, EQUIVALENCE statements are difficult to translate since Java does not allow overlapping memory regions. However, f2j can handle a limited form of EQUIVALENCE as long as the variables being equivalenced do not differ in type and are not offset from each other. This restriction implies that any two arrays being equivalenced must specify indices of 1. However, it is allowable to equivalence arrays of different dimensions (e.g. a one-dimensional integer array to a two-dimensional integer array),

Copyright © 2003 John Wiley & Sons, Ltd.

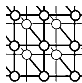


Table I. Common block example.

Fortran source	Bytecode (disassembled with D-Java [17])

`PROGRAM CTEST`
`INTEGER A,B,C`
`COMMON /CBLK/A,B,C`
`...`
	`public final class ctest_cblk extends Object`
	`>> ACC_SUPER bit set <<`
	`{`
	`    public static org.netlib.util.intW a;`
	`    public static org.netlib.util.intW b;`
	`    public static org.netlib.util.intW c;`
	`...`

Table II. Common block example with merged variable names.

Fortran source	Bytecode (disassembled with D-Java [17])

`PROGRAM CTEST2`
`INTEGER A,B,C`
`COMMON /CBLK/A,B,C`
`...`
`SUBROUTINE SUB()`
`INTEGER X,Y,Z`
`COMMON /CBLK/X,Y,Z`
`...`
	`public final class ctest2_cblk extends Object`
	`>> ACC_SUPER bit set <<`
	`{`
	`    public static org.netlib.util.intW x_a;`
	`    public static org.netlib.util.intW y_b;`
	`    public static org.netlib.util.intW z_c;`
	`...`

as all arrays are linearized to one dimension and the access is basically the same regardless of the number of dimensions.

To handle the limited EQUIVALENCE, we simply merge the equivalenced variable names into a single variable, as shown in Table III. Then this single variable is loaded in place of any of the variables that were previously equivalenced.

4.1.8. I/O statements

While I/O is not the most critical aspect of translating numerical libraries such as LAPACK, we have found it useful to partially implement Fortran I/O in order to translate the test routines, which read in the parameters and write out the results. Unformatted WRITE statements are easily implemented with Java's `println()` method, but formatted WRITE statements first require analyzing the corresponding FORMAT statement and creating a `StringBuffer` to hold the output before calling `println()`. READ is implemented using an external library, EasyIn [18]. File I/O is not yet implemented.

Copyright © 2003 John Wiley & Sons, Ltd.

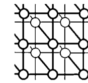


Table III. EQUIVALENCE example.

Fortran source	Bytecode (disassembled with D-Java [17])

`PROGRAM ETEST`
`INTEGER A(100), B(10,10)`
`EQUIVALENCE (A,B)`
`...`
	`public final class Etest extends Object`
	`>> ACC_SUPER bit set <<`
	`{`
	`    public static int[] a_b;`
	`...`

4.1.9. Passing functions as arguments

Strictly speaking, passing a function name as an argument to another function, as done in Fortran, is not possible in Java. A Java programmer may pass a class as an argument, but not an individual method. The closest Java analogue is passing an *interface* which implements a method with a pre-determined name. That technique is not really suitable for use with f2j because every generated class file would be forced to implement the interface, whether it was actually intended to be passed to another function or not, thus adding extra overhead even if the implementation is simply a call to the translated Fortran routine. Another option is using *inner classes* to implement the callback, but this also mandates generating extra code.

Rather than using interfaces, f2j uses Java's 'reflection' mechanism to determine the appropriate method to invoke, based on the assumption that f2j always places the generated Fortran routine in the first method of the class. The caller only needs to pass a new instance of the class corresponding to the translated Fortran routine.

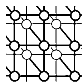
4.1.10. Other limitations

Aside from the limitations already mentioned (I/O, EQUIVALENCE, etc.), f2j does not currently support multiple entry points (ENTRY statement), alternate returns, statement label assignment or complex arithmetic.

Although it is not part of the Fortran 77 specification, some compilers support the RECURSIVE keyword to signify that recursion is allowed. The Java platform places no artificial restrictions on recursion, but support for recursive functions is still not completely implemented in f2j. The obstacle to fully implementing recursion is allowing recursion interferes with the scalar wrapper optimization phase (see Section 3.2), but only when the recursion is indirect and one of the arguments requires an object wrapper.

Some of these restrictions are more serious than others. For example, code that relies heavily on EQUIVALENCE will not likely translate well because such a language construct does not map well to the Java language. The LAPACK source code only uses EQUIVALENCE in a few cases. In those cases, the EQUIVALENCE statements are in a form that f2j can handle correctly, as described in Section 4.1.7.

Copyright © 2003 John Wiley & Sons, Ltd.



These limitations did not prevent us from completely translating the double-precision routines in BLAS and LAPACK. The current release of JLAPACK consists of all 346 double-precision routines translated from LAPACK and all 33 double-precision routines from BLAS, totaling 137 406 lines of Java source. Additionally, all of the double-precision BLAS and LAPACK testers have been translated, totaling over 100 000 lines of Java source. Complex arithmetic is not supported by f2j yet, but given the existence of several Java libraries [19–21], we expect that the f2j code generator could be straightforwardly modified to support complex arithmetic using one of these implementations.

4.2. Implementation

The code generator is implemented in two passes. The first pass generates the appropriate instructions and the second pass calculates the maximum stack depth and fills in any branch targets that were not known during the first pass. There are backpatching techniques that would allow filling in the branch targets in one pass [14], but since we are using two passes in any case, we can more easily perform this task during the second pass.

4.2.1. Instruction generation

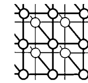
First, the code generator traverses the abstract syntax tree, generating Java source code as well as the JVM opcodes. However, at this point the opcodes have empty operand fields for branch target addresses because for forward branches, we do not yet know the address of the target instruction as it has not yet been generated. In such cases, we simply save a pointer to the current node and update its branch target pointer after generating the nodes in between the current node and the target. This is useful when the GOTO is 'implicit' in the sense that it never appeared as a GOTO in the original source (i.e. it is used to implement some Fortran control structure such as a DO loop). GOTO statements which appear explicitly in the Fortran source require slightly different handling since the branch target is an arbitrary label whose corresponding node we do not have access to at this point. Also there is no inherent structure as with a `DO` loop, where the `goto` always branches to a specific instruction. Thus, during this phase, we create a table which maps the labels in the original source to the corresponding instruction addresses. Then, in the next phase, we can easily fill in the branch target address for any GOTO statement by looking up its branch label.

4.2.2. Calculating stack depth and branch targets

At this point, the AST has been fully traversed, we have generated the Java source code and we have built a control flow graph representing the bytecode. Each node may have pointers to other nodes, which represent the branch targets. Each instruction may or may not have a single branch target depending on the instruction type[‡]. For example, an `iload` instruction has no branch target whereas `icmpeq`, being a conditional branch, does have a branch target. Now, as we traverse this graph, we can fill in the empty branch target offsets by following the pointer to the target node and examining its address.

[‡]We are conveniently ignoring `tableswitch` which has many branch targets because f2j never actually generates this opcode.

Copyright © 2003 John Wiley & Sons, Ltd.



Also during this phase, we maintain information about the current stack depth at each node because the class file format requires specifying the maximum stack depth that will be encountered during execution of the method. The stack depth at any given node is a function of the stack depth at the prior node and the characteristics of the current instruction (e.g. `iadd` would pop two integers off the stack and push one integer on, for a net difference of one). This also provides a nice opportunity for sanity checking—for example, if the current instruction branches back to another instruction for which the stack depth has already been calculated, then we can check whether the expected stack depth matches the current stack depth. In other words, a given instruction could be the target of multiple other instructions and the stack depth at all of those instructions must be consistent (otherwise, this indicates an error in the code generator).

Finally, at this point we have built a complete data structure representing the class file—this includes constant pool, fields and methods—which we emit in the format dictated by the Java Virtual Machine Specification [22].

4.3. Differences in code generation

Since we did not want to eliminate the existing code generator (which emits Java source), we designed the new code generator to emit both Java source and JVM bytecode during the same pass. For the most part, the bytecode can be generated simultaneously with the Java source code, but there are some exceptions as follows.

- Obviously, GOTO statements are handled differently in bytecode since we can easily emit a `goto` JVM instruction, but we must generate 'dummy' method calls in Java source.
- `DO` loops are emitted as `for` loops in Java source, with the initial, terminal and incrementation parameters straightforwardly translated to Java-style loop control expressions. However, when translating a `DO` loop directly to bytecode, we follow the sequence outlined in the Fortran 77 specification [23] and calculate the iteration count[§] before entering the loop. Then at each iteration, the iteration count is decremented until it reaches 0, at which point the loop is terminated.
- Variable declarations are handled a bit differently in bytecode. Each variable must be stored in the *fields table* of the current class, but explicit initialization code is only generated for array and reference data types. When generating bytecode, we must create a special method named `<clinit>` into which we place the initialization code. However, with Java source, this is handled by `javac`.
- Type casts are much more important when generating bytecode than Java source since each instruction is type-specific. Thus, in the many instances that we could 'get away' with generating an expression in Java without any explicit casts, we must generate type conversion instructions in bytecode.

When the code generator needs to toggle between modes—such as to suspend Java source code generation and begin generating code in bytecode only—we simply set the appropriate global file

[§]The iteration count is defined as $\max(\text{int}((m_2 - m_1 + m_3)/m_3), 0)$, where m_1 is the initial value m_2 is the terminal value and m_3 is the incrementation value [23].

Copyright © 2003 John Wiley & Sons, Ltd.

pointer to `/dev/null` and call the routine as usual. There is no need for any modification to the code generation routines.

4.4. Resolving calls to external functions

This section describes a technique for resolving calls to functions or subroutines which do not appear in the original source file. By 'resolving', we mean determining the correct calling sequence for the function call, which depends on its method signature. For example, consider the following Fortran program segment:

```
      INTEGER X(10)

      CALL FUNC1( X(5) )
      CALL FUNC2( X(5) )
[...]
      SUBROUTINE FUNC1(A)
      INTEGER A
[...]
      SUBROUTINE FUNC2(A)
      INTEGER A(*)
```

The first subroutine, `FUNC1`, expects a scalar argument, while `FUNC2` expects an array argument. These two calls would be generated identically in a standard Fortran compiler, regardless of how `FUNC1` and `FUNC2` were defined—the address of the fifth element of `X` would be passed to the subroutine in both cases. However, things are not as simple in Java due to the lack of pointers. To simulate passing array subsections, as necessary for the second call, we actually pass two arguments—the array reference and an additional integer offset parameter, as shown in the right column of Table IV.

However the first subroutine expects a scalar, so we should pass only the value[¶] of the fifth element, without any offset parameter, as shown in the left column of Table IV.

Notice that the primary difference between the two calling sequences is that when calling `FUNC1`, the array is first dereferenced using the `iaload` instruction. Also note that the purpose of the arithmetic expression is to decrement the index by 1 to compensate for the fact that Java has 0-based indexing whereas Fortran has 1-based indexing.

The only way to determine the correct calling sequence for any given call is to examine the parameters of the corresponding subroutine or function declaration. This is only possible if the declaration had been parsed at the same time as the current program unit, meaning that for code generation to work properly all the source files had to be joined into a big monolithic input file.

This was a serious limitation, especially for large libraries, because a modification to any part of the code requires re-compiling *all* of the source. There are at least a couple of ways to solve this problem. One way would be to obtain the parameter information directly from class files that have already been

[¶]In this case, assume that `FUNC1` does not modify the argument, otherwise things get even more complex. See [9] for a description of handling that case.

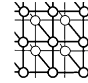


Table IV. Differences in argument passing.

Calling FUNC1	Calling FUNC2

`getstatic #15 <Field Hello.x:int[]>`
`iconst_5`
`iconst_1`
`isub`
`iaload`
`invokestatic #22`
`    <Method Func1.func1(int):void>`
	`getstatic #15 <Field Hello.x:int[]>`
	`iconst_5`
	`iconst_1`
	`isub`
	`invokestatic #28`
	`    <Method Func2.func2(int[],int):void>`

generated. While this would work well, f2j is written in C and does not have access to nice Java features like reflection, so it would require a lot of extra code to parse the class files. Instead, we use a more lightweight procedure in f2j. At compile-time, f2j creates a *descriptor file*, which is a text file containing a list of every method generated. Each line of the descriptor file contains the following information.

- Class name—the fully qualified class name which contains the given method.
- Method name—the name of the method itself.
- Method descriptor—this method's descriptor, which is a string representing the types of all the arguments as well as the return type.

To resolve a subroutine or function call, we search all the descriptor files for the matching method name and examine the method descriptor. Based on the method descriptor, we can then correctly generate the calling sequence. The code generator locates the descriptor files based on colon-separated paths specified on the command line or in the environment variable `F2J_SEARCH_PATH`.

5. EXPERIMENTAL RESULTS

When evaluating the results of our code generator, the two aspects we are most concerned with are correctness and efficiency of the generated code. Correctness means that the generated code produces the same numerical results as the native-compiled Fortran code, within a certain degree of tolerance inherent in performing floating-point calculations on different systems. We also measure efficiency in terms of the original Fortran code, using the performance of optimized Fortran as the standard by which we evaluate the efficiency of our code. Optimized Fortran is almost certainly going to represent an upper-bound on the performance potential of the code that f2j generates.

5.1. Correctness

To date, the BLAS and LAPACK libraries have been the main testbed for f2j. Thus, when evaluating correctness we are primarily concerned with the results generated by the Java implementation of the BLAS and LAPACK libraries. Fortunately the original Fortran distributions of these libraries include

Copyright © 2003 John Wiley & Sons, Ltd.

Table V. Number of test cases.

Test category	Number of test cases
BLAS Level 1	Unreported
BLAS Level 2	30 409
BLAS Level 3	27 864
LAPACK Linear Solver	316 206
LAPACK Eigenvalue	719 291

Table VI. Performance on the double-precision Linpack benchmark ($n = 500$).

Compilation method	Command line	Raw performance (Mflop/s)	Performance relative to optimized Fortran
Optimized Fortran	`f77 -O3`	34.7	1.00
Unoptimized Fortran	`f77`	14.1	0.41
Bytecode	`f2java`	10.9	0.31
Bytecode (-server)	`f2java`	25.9	0.75
Java Source	`f2java ; javac`	10.3	0.30
Java Source (-server)	`f2java ; javac`	27.9	0.80

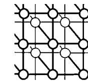
comprehensive testing routines to verify the numerical results of the computations. To determine the correctness of the code generated by f2j, we translated all of the double-precision BLAS and LAPACK test routines to Java and ran them against the Java implementations of the BLAS and LAPACK libraries. The total number of test cases executed is quite large, as shown in Table V. All the numerical tests passed within the default thresholds given in the original LAPACK test input files.

5.2. Efficiency

To measure the performance of the code generated by f2j, we translated the Fortran 77 source code for the Linpack benchmark [24] in several different ways, as shown in Table VI. The first two entries represent native-compiled Fortran code, both optimized and unoptimized. The third entry represents the performance of the JVM bytecode generated by f2j. The fourth entry is the same as the third except that the `-server` flag was specified when running the benchmark. The last two entries represent the Java source code generated by f2j (which includes subsequent compilation with `javac` and GOTO translation).

The test machine is a Sun Ultra-5 running Solaris 2.7 with Sun's J2SE 1.3.0 (with HotSpot enabled). We used Sun's f77 5.0 compiler to obtain the results for native-compiled Fortran code.

Copyright © 2003 John Wiley & Sons, Ltd.



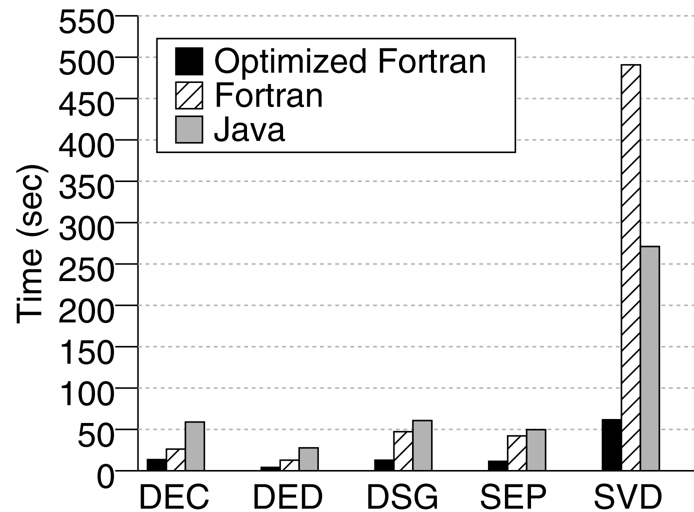


Figure 2. Elapsed time executing various LAPACK test routines.

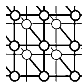
As Table VI shows, the bytecode generated by f2j achieves roughly one-third the performance of optimized Fortran code in 'client' mode and over three-quarters of the performance of optimized Fortran code in 'server' mode. Since f2j generates the bytecode and Java source simultaneously, it is convenient to compare the performance of the directly-generated bytecode to the bytecode resulting from generating Java source and subsequently compiling using `javac`. Depending on the JVM used (and in the case of Hotspot, which JVM flag is specified), in some cases the Java source is faster and in other cases the directly-generated bytecode is faster.

As another informal benchmark, we compared the time required to run several of the LAPACK test routines. As Figure 2 shows, the results are similar to the Linpack benchmark results. The code generated by f2j is close to the performance of unoptimized Fortran, but still lags behind optimized Fortran speeds.

5.3. The object-oriented approach

At the outset of the f2j project, JVM technology had not matured to the point that it was feasible from a performance standpoint to make heavy use of object-oriented techniques in linear algebra code. However, with recent developments in Virtual Machines and JIT compilers, much of the overhead associated with such techniques has been eliminated. As an example, we compare the performance of the code automatically generated by f2j with a representative of the object-oriented approach—HARPOON/JLAPACK [4]. Figure 3 shows the performance of double-precision matrix multiply, in which the object-oriented approach can exceed the performance of procedural-style code.

Copyright © 2003 John Wiley & Sons, Ltd.



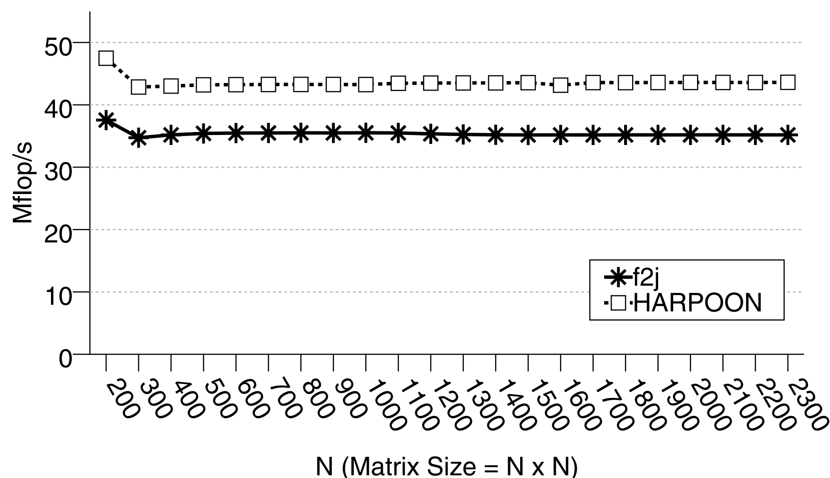


Figure 3. Comparison of f2j-generated DGEMM with the object-oriented version.

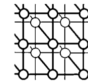
The performance difference is due to the cumbersome array indexing expressions in our automatically-generated code.

The clear conclusion is that we need not fear the performance consequences of using object-oriented techniques. To a certain extent, it would be feasible to rewrite our code generator in terms of object-oriented techniques. For example, it would be straightforward to modify our code generator such that instead of using standard Java arrays, it generates the code in terms of a multidimensional array package, such as those proposed in [25]. Other techniques, such as recognizing vector operations and generating the appropriate method call, would require more significant modifications.

6. CONCLUSION

We have demonstrated that it is feasible to automatically convert very large Fortran libraries to JVM bytecode with reasonable performance. Certainly at this point the performance of the translated numerical code does not match hand-tuned Java algorithms, but that is not the problem f2j is designed to address. The f2j project intends to bootstrap the use of Java for numerical and scientific computing by providing the widest possible range of useful and reliable numerical routines in a pure Java format. However, having said that, we think there is still a lot of opportunity for improving the performance of the generated code. Currently the bytecode is generated in a very straightforward manner, without any optimization. The next stage in the development of the bytecode generator will be the implementation of a code optimization stage to increase the performance of the generated code. In particular, we would like to investigate the impact of various compiler optimizations on the performance of the JLAPACK

Copyright © 2003 John Wiley & Sons, Ltd.



library routines. These techniques could be beneficial not only to our project, but to the Java community as a whole.

We also plan to remedy some of f2j's limitations—complex arithmetic support, better I/O handling and some syntactic restrictions inherent in our parser. The eventual goal is to have a tool with a usefulness more general than just translating numerical libraries.

REFERENCES

1. Sun Microsystems Inc. *The Java Language Environment*. Sun Microsystems, Mountain View, CA, 1995.
2. Hicklin J, Moler C, Webb P, Boisvert RF, Miller B, Pozo R, Remington K. JAMA: A Java Matrix Package. http://math.nist.gov/javanumerics/jama [June 1999].
3. Stewart GW. JAMPACK: A Java Package for Matrix Computations. ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html [February 1999].
4. Blount B, Chatterjee S. An evaluation of Java for numerical computing. *Scientific Programming* 1999; **7**(2):97–119.
5. DeriVision Inc. LinJa. http://www.derivision.com/products/index.html [April 2000].
6. DRA Systems. OR-Objects. http://opsresearch.com/OR-Objects/index.html [December 1999].
7. Riley C, Chatterjee S, Biswas R. High-performance Java codes for computational fluid dynamics. *Proceedings of the ACM 2001 Java Grande/ISCOPE Conference*. ACM, 2001.
8. VanderHeyden W, Dendy E, Padial-Collins N. CartaBlanca—A pure-Java, component-based systems simulation tool for coupled non-linear physics on unstructured grids. *Proceedings of the ACM 2001 Java Grande/ISCOPE Conference*. ACM, 2001.
9. Doolin D, Dongarra J, Seymour K. JLAPACK—Compiling LAPACK Fortran to Java. *Scientific Programming* 1999; **7**(2):111–138.
10. Lawson C, Hanson R, Kincaid D, Krogh F. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software* 1979; **5**:308–325.
11. Dongarra J, Du Croz J, Hammarling S, Hanson R. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 1988; **14**(1):1–32.
12. Dongarra J, Du Croz J, Duff I, Hammarling S. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 1990; **16**(1):1–17.
13. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D. *LAPACK Users' Guide, Third Edition*. SIAM: Philadelphia, PA, 1999.
14. Aho AV, Sethi R, Ullman JD. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley: Reading, MA, 1988.
15. Artigas P, Gupta M, Midkiff S, Moreira J. High performance computing in Java: Language and compiler issues. *Proceedings of the 12th Workshop on Language and Compilers for Parallel Computers*. Springer, 1999.
16. Meyer J, Downing T. *Java Virtual Machine*. O'Reilly & Associates: Sebastopol, CA, 1997.
17. Meyer J. D-Java. http://www.cat.nyu.edu/meyer/jvm/djava [December 1996].
18. van der Linden P. EasyIn. http://www.afu.com/EasyIn.txt [May 1997].
19. Wu P, Midkiff S, Moreira J, Gupta M. Efficient support for complex numbers in Java. *Proceedings of the ACM 1999 Java Grande Conference*. ACM Press: New York, NY, 1999.
20. Anderson A. Complex Arithmetic for Java. http://www.almide.demon.co.uk/html/Complex/Complex_for_Java.html [March 2001].
21. Brophy J. Design of Class Complex. http://www.vni.com/corner/garage/grande/complex.htm [2001].
22. Lindholm T, Yellin F. *The Java Virtual Machine Specification*. Addison-Wesley: Berkeley, CA, 1997.
23. American National Standards Institute. American National Standards Institute Programming Language FORTRAN. *X3.9-1978*, ANSI, New York, 1978.
24. Dongarra J. Linpack Benchmark. http://www.netlib.org/benchmark/linpackd [October 1992].
25. Moreira J, Midkiff S, Gupta M. A comparison of three approaches to language, compiler, and library support for multidimensional arrays in Java. *Proceedings of the ACM 2001 Java Grande/ISCOPE Conference*. ACM Press: New York, NY, 2001.

Copyright © 2003 John Wiley & Sons, Ltd.