Benchmarking FALCON's MATLAB-to-Fortran 90 Compiler on an SGI Power Challenge

Luiz De Rose and David Padua *
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801, U.S.A.
({derose,padua}@cs.uiuc.edu)

Abstract

This paper presents an overview of the FALCON MATLAB-to-Fortran 90 compiler. FALCON is a programming environment for the development of high-performance scientific programs. It combines static and dynamic inference methods to translate MATLAB programs into Fortran 90. The static inference is supported with advanced value propagation techniques and symbolic algorithms for subscript analysis. The experiments presented in this paper show that FALCON's MATLAB compiler can generate code that performs more than 1000 times faster than the interpreted version of MATLAB and substantially faster than a commercially-available MATLAB compiler on one processor of an SGI Power Challenge. Furthermore, for most of the programs we have tested, the compiler-generated codes are as fast as the corresponding hand-written programs.

Resumo

Esse artigo apresenta uma visão geral do compilador de MATLAB para Fortran 90 desenvolvido para o ambiente de programação para processamento de alto desempenho FALCON. Esse compilador utiliza técnicas estáticas e dinâmicas para inferência. Essas técnicas sao otimizadas com análise simbólica e com métodos desenvolvidos para determinação do domínio das variáveis. Os testes de desempenho apresentados nesse artigo mostram que os programas gerados pelo compilador podem ser executados até 1000 vezes mais rápidos que a versão interpretada pelo MATLAB. Esses programas tambem tem um desempenho muito superior aos programas gerados pelo compilador MATLAB disponível comercialmente. Finalmente, para a mairia dos programas testados, o compilador gerou programas que rodam tão rápido quanto versões escritas a mão para os mesmos algoritmos.

*This work was supported in part by Army contract DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

1 Introduction

The development of software for scientific computation on high-performance computers is a very difficult and time-consuming task, requiring not only an understanding of the algorithms to be implemented, but also a detailed knowledge of the target machine and the software environment. We believe the development of scientific programs should start with a language as close as possible to the mathematical description of the problem, albeit in the form of a simple and easy-to-use procedural language. The use of a very high-level language facilitates the development process by enhancing the ease of programming and portability of applications.

Interactive array languages such as APL [12] and MATLAB [13] are powerful programming tools for the development of programs for numerical computation. A convenient feature in these languages that facilitates prototyping of applications is the lack of specification of dimensions and intrinsic type of variables. Moreover, interactive array languages are usually contained within problem-solving environments which include easy-to-use facilities for displaying results both graphically and in tabular form [10]. Furthermore, the interactive nature of these languages provide an environment that tends to increase productivity in software development. The trade-off is that in order to provide this nicer programming environment, array languages are usually interpreted, with the resulting negative effect on performance.

The option of prototyping the program in an interactive language like MATLAB and then, after program development is complete, rewriting in a compiled language, like C or Fortran, is often mentioned as a possibility, but seldom done. The situation in practice is that the overhead of reimplementing programs in a different language is sufficiently large that most people never get around to doing it. Clearly the best solution is for programmers to use a compiler that generates efficient code from MATLAB programs.

FALCON [8, 9] is a programming environment for the development of scientific libraries and applications. It attempts to facilitate the development process by taking advantage of both the power of interactive array languages and the performance of compiled languages. One of FALCON's main components is a MATLAB to Fortran 90 compiler. Some of the issues that need to be addressed when compiling MATLAB programs are the lack of intrinsic type definitions and specification of dimensions of variables, the possibility that any of these variable properties could change during run-time, and the overload of operators that have different semantics depending on the rank of the variables being operated. Our ultimate goal is to generate parallel code by integrating FALCON with Polaris [3, 4], a parallelizing compiler developed at Illinois.

This paper describes the main ideas of FALCON's MATLAB to Fortran 90 compiler and presents performance comparisons of Fortran 90 programs generated by FALCON against the interpreted MATLAB programs, C programs generated by a commercial MATLAB to C compiler (MCC) [14], and Fortran 90 hand coded programs for the same algorithms. The rest of this paper is organized as follows: Section 2 presents an overview of FALCON's MATLAB to Fortran 90 compiler. Section 3 discuss experimental results. Finally, our conclusions are presented in Section 4.

2 FALCON's Compiler Overview

MATLAB is a procedural language that operates on only one kind of data structure: a rectangular numerical matrix [13]. A MATLAB program consists of one or more Fortran-like statements which may involve function calls. There are two types of functions in MATLAB: *built-ins* and *M-files*. Built-in functions are intrinsic functions, such as SQRT, INV (for matrix inverse), and EIG (for eigenvalues and eigenvectors). M-files consist of a sequence of MATLAB statements, which possibly include references to other M-files.

The main challenge of the MATLAB compiler is to perform inference on the input program to determine the following variable properties, necessary to generate the Fortran 90 declarations and to optimize the output code:

intrinsic type: which could be COMPLEX, REAL, INTEGER, or LOGICAL;

rank: which could be SCALAR, VECTOR, or MATRIX; and

shape: which indicates the size of each dimension.

FALCON's MATLAB compiler uses conventional data-flow analysis [1], and inference techniques that are either *static* inference mechanisms used to generate declarations at compile time, or *dynamic* strategies that are applied at execution time when a lack of statically available information prevents the automatic generation of a particular variable's declaration. The static inference mechanism was built upon techniques developed for SETL [16] and APL [5], extended to deal with peculiarities of the MATLAB language. Also, additional techniques were used to improve accuracy and performance. Examples of these additional techniques are advanced value propagation techniques and symbolic algorithms for subscript analysis [7].

FALCON's static inference algorithms are applied to a *Static Single Assignment* (SSA) [6] representation of the MATLAB program in the form of an *abstract syntax tree* (AST) [1]. Before the AST is generated, all function calls to M-files are inlined in the program. Inlining makes it easier to perform the program analysis on a function for the correct combination of input variables, as subsequent calls to the same function may have input variables with different types or ranks.

The static inference mechanism extracts information from four main sources: input files; program constants; operators; and built-in functions. From input files the compiler extracts the initial intrinsic type and rank of the variables being loaded. Variable shapes are not extracted from the input files because they are much more likely than intrinsic type and rank to differ between runs. Program constants are used for the inference of intrinsic type, rank, and shape. From MATLAB operators we extract intrinsic type information, in the case of operators that produce logical values, and rank and shape information, by taking into consideration the conformability requirements imposed by the operators. Finally, the fourth source of information comes from MATLAB built-in functions that can provide inference information for their output parameters based on type of the input parameters.

Although MATLAB operates on REAL and COMPLEX values only, FALCON's static inference mechanism also considers INTEGER and LOGICAL values. The intrinsic types are inferred according to the following type hierarchy: LOGICAL $\prec$ INTEGER $\prec$ REAL $\prec$ COMPLEX.

This hierarchy means that it is correct for variables with INTEGER values to be declared as REAL or COMPLEX and for variables with REAL values to be declared as COMPLEX. In fact, intrinsic type inference could be avoided if all variables were considered COMPLEX. This, however, would affect the performance of the code due to the number of additional arithmetic operations required by COMPLEX variables. Hence, the determination of the correct variable intrinsic type during compile time is very important for the efficiency of the generated code.

The static mechanism for intrinsic type inference propagates intrinsic types through expressions using a *type algebra* similar to that described in [16] for SETL. For the case of logical operators, the result is always considered to be of intrinsic type LOGICAL. For the other operators and built-in functions, this algebra operates on the intrinsic type of MATLAB objects and is implemented using tables for all operations. For each operation these tables contain the intrinsic type of the result as a function of the intrinsic type of the operands. When the intrinsic types of the operands are different, the static type inference promotes the output of the expression to be of an intrinsic type that subsumes the intrinsic types of both operands, according to the intrinsic type hierarchy described above. In some cases, such as division, the output is promoted to a higher intrinsic type from the type hierarchy, even if the intrinsic type of the operators are the same. for example, the division of two INTEGER variables result in an output of intrinsic type REAL.

In some cases, the outcome of an expression or a built-in function can be of different intrinsic types, depending on the values of the operands (such as the power operator, square root, and inverse trigonometric functions). For example, the outcome of $\sqrt{1-A}$ would be REAL if $A \leq 1$, and COMPLEX otherwise. We refer to these expressions as *ambiguous-typed expressions*. In these cases, to improve the accuracy of the intrinsic type inference, we perform *value-propagation* analysis [7]. This value-propagation analysis keeps track statically of the range of possible values for each variable in the program. If this range of values is not sufficient to determine the intrinsic type of an ambiguous-typed expression, the output intrinsic type of the expression is promoted to COMPLEX.

Rank and shape information are obtained with the use of *conformability analysis*. We define the operators that require conformability for only one of the dimensions of the operands (e.g., "*", "/", and "\") as a *single-dimension conformable operator*, and operators that require both operands to have the same shape (e.g., +, -, and logical operators) as a *shape conformable operator*. The conformability analysis uses tables such as Table 1 for the rank and shape inference for the multiplication operator (a single-dimension conformable operator) and Table 2 for the shape inference of a shape conformable operator. In these tables the shape information is indicated with the letters (m, n, p, and q) that represent the exact values for the number of rows or the number of columns. Column VECTORS and row VECTORS are represented as VECTOR(p,1) and VECTOR(1,q) respectively, and UNKNOWN values are represented with "?".

If any of the variable attributes necessary for the generation of the Fortran 90 declarations

A * B	B			
A	SCALAR	VECTOR(p,1)	VECTOR(1 ,q)	MATRIX(p,n)
SCALAR	SCALAR	VECTOR(p,1)	VECTOR(1 ,q)	MATRIX(p,n)
VECTOR(p,1)	VECTOR(p,1)	error	MATRIX(p ,q)	error
VECTOR(1,q)	VECTOR(1,q)	SCALAR[1]	error	VECTOR(1,m)[1]
MATRIX(n,q)	MATRIX(n,q)	VECTOR(n,1)[1]	error	MATRIX(n,m)[1]
[1]Only if p = q; otherwise error.				

Table 1: Rank and shape inference for the multiplication operator.

A+B	B								
	SCALAR	VECTOR		NOTMATRIX		NOTSCALAR		MATRIX	UNKNOWN
A	(1,1)	(p,1)	(1,q)	(1,?)	(?,1)	(p,?)	(?,q)	(p,q)	(?,?)
(1,1)	(1,1)	(p,1)	(1,q)	(1,?)	(?,1)	(p,?)	(?,q)	(p,q)	(?,?)
(p,1)	(1,1)	(p,1)	error	(p,1)	(p,1)	(p,1)	error	error	(p,1)
(1,q)	(1,q)	error	(1,q)	(1,q)	(1,q)	error	(1,q)	error	(1,q)
(1,?)	(1,?)	(p,1)	(1,q)	(1,?)	(?,?)	(p,?)	(?,q)	(p,q)	(?,?)
(?,1)	(?,1)	(p,1)	(1,q)	(?,?)	(?,1)	(p,?)	(?,q)	(p,q)	(?,?)
(p,?)	(p,?)	(p,1)	error	(p,?)	(p,?)	(p,?)	error	(p,q)	(p,?)
(?,q)	(?,q)	error	(1,q)	(?,q)	(?,q)	error	(?,q)	(p,q)	(?,q)
(p,q)	(p,q)	error	error	(p,q)	(p,q)	(p,q)	(p,q)	(p,q)	(p,q)
(?,?)	(?,?)	(p,1)	(1,q)	(?,?)	(?,?)	(p,?)	(?,q)	(p,q)	(?,?)

Table 2: Shape inference for a conformable operator.

(i.e., intrinsic type, number of rows, and number of columns) have an UNKNOWN value at the end of the static inference phase, dynamic code to determine the necessary attribute at run-time is generated. FALCON associates tags for each variable that have any UNKNOWN attribute value. These tags are updated at execution time. Based on these tags, which are stored in *shadow variables*, conditional statements are used to allocate the necessary space and to select the operation on variables of the appropriate intrinsic type.

To avoid the excessive number of conditional tests necessary to detect the intrinsic type of the outcome of an expression, the dynamic inference mechanism considers only two intrinsic types: REAL and COMPLEX. If the intrinsic type of a variable can be determined statically, a Fortran declaration for the inferred intrinsic type is generated. Otherwise, a conditional statement is generated to test during run-time the shadow value for the intrinsic type. Each branch of the conditional statement receives a clone of the operation that uses the variable requiring the shadow test. In one branch the variable is assumed to be of type COMPLEX, while in the other it is assumed to be of type REAL. In addition, the variable is renamed and declared twice, once for each dynamic intrinsic type.

Dynamic code is also generated to compute during run-time the necessary space for dynamic allocation. To this end, two shadow variables are used to keep track of variable

```
S1: if (A_D1 .ne.  B_D1 .or.  A_D2 .ne.  B_D2) then
S2:   if (ALLOCATED(A)) DEALLOCATE(A)
S3:   A_D1 = B_D1
S4:   A_D2 = B_D2
S5:   ALLOCATE(A(A_D1,A_D2))
S6: end if
S7: A = B + 0.5
```

Figure 1: Example of shadow variables for shape.

dimensions during execution-time. So, for example, if the shape of B in an assignment of the form A=B+0.5 were UNKNOWN after the static phase, the compiler would generate the code presented in Figure 1. This code uses shadow variables A_D1, A_D2, B_D1, and B_D2, to store the run-time information about the size of each dimension of A and B. These shadow variables are initialized to zero at the beginning of the program. This figure shows the general form for the dynamic allocation. However, only the necessary statements are generated. If, for example, this were the first definition of variable A, statements S1 and S2, that are needed only to reallocate the variable, would not be generated.

Some optimizations in the shape inference mechanism are necessary to avoid the excessive number of tests and allocations. To this end, two static techniques were developed to support dynamic shape inference: *coverage analysis* and *efficient placement of dynamic allocation* [7]. The objective of the first technique is to determine whether an indexed array assignment may increase the size of the array. If this information is known at compile time, it is not necessary to generate an allocation test for the indexed assignment. To determine whether there is definition coverage, we use a simplified version of a demand-driven symbolic analysis algorithm developed by Tu and Padua [17]. When an allocation test is required, the second technique is used to place the test where it will minimize the overhead.

3 Experimental Results

To evaluate the overall effectiveness of the compiler, twelve MATLAB programs were tested. These programs implement algorithms including the iterative solution of linear system, the solution of differential equations, and matrix factorizations. Where possible, the program parameters were set so that the time required to execute the program as a MATLAB M-file would be around 60 seconds. A brief description of each program is presented next.

AQ – This code uses the Simpson's rule in conjunction with adaptive quadrature [15] to numerically approximate the integral:

$$\int_{-1}^{6} 13 * (x - x^2) e^{\frac{-3x}{2}} dx.$$

CG – An iterative method for the solution of linear systems that uses the preconditioned Conjugate Gradient method with a diagonal preconditioner [2]. The input data is a 420×420 stiffness matrix from the Harwell-Boeing Test Set (BCSSTK06).

CN – A numerical approximation method for the solution of parabolic differential equations (the heat equation) [15]. The problem size is a 321×321 grid.

Di – An iterative method for the solution of Laplace's equations [15]. The problem size is a 41×41 grid.

FD – A numerical approximation method for the solution of hyperbolic differential equations (the wave equation) [15]. The problem size is a 451×451 grid.

Ga – A numerical approximation method to solve the Poisson equations in two dimensions using the Galerkin method [11]. The problem size is a 40×40 grid.

IC – Calculates the incomplete Cholesky factorization of a matrix.[1] The input data is a 400×400 matrix.

EC – A numerical approximation method to solve ordinary differential equations. This program solves the orbit of a comet around the sun using the Euler-Cromer method [11]. The program executes 6240 steps.

RK – A numerical approximation method to solve ordinary differential equations. This program solves the orbit of a comet around the sun using a 4th order Runge-Kutta method [11]. The program executes 3200 steps.

QMR – An iterative method for the solution of linear systems that uses the Quasi-Minimal Residual method [2]. The input data is the 420×420 stiffness matrix BCSSTK06.

SOR – A third iterative method for the solution of linear systems that uses the Successive Overrelaxation method [2]. The input data is the 420×420 stiffness matrix BCSSTK06.

3D – Generates a three-dimension surface based on calculating Eigenvalues within a triply-nested loop.[1] The dimensions of the surface are $51 \times 31 \times 21$.

Table 3 presents the best execution times in seconds for all programs in our test set. The Fortran 90 programs were compiled using the SGI native Fortran 90 compiler with the optimization flag "O3". The C programs generated by the MathWorks MATLAB to C Compiler (MCC) were compiled both with the SGI native C compiler using the highest possible optimization flag ("O2") and with the GNU C compiler using the optimization flag "O3". For each program, the best execution time out of the two compilers was chosen. MCC does not support the `load` statement; hence, the input data was loaded using interpreted MATLAB commands (not timed) and provided to the C programs as function parameters.

[1]Code was provided by a colleague.

Program	MATLAB	MCC	FALCON	Hand coded
AQ	19.95	2.30	1.477	0.877
CG	5.34	5.51	0.588	0.543
CN	44.10	0.70	0.098	0.097
Di	44.17	1.50	0.052	0.050
FD	34.80	0.37	0.031	0.031
Ga	31.44	0.56	0.156	0.154
IC	32.28	1.35	0.245	0.052
3D	34.95	11.14	3.163	3.158
EC	8.34	3.38	0.012	0.007
RK	20.60	5.77	0.038	0.025
QMR	7.58	6.24	0.611	0.562
SOR	18.12	18.14	2.733	0.641

Table 3: Execution times (in seconds) running on an SGI Power Challenge.

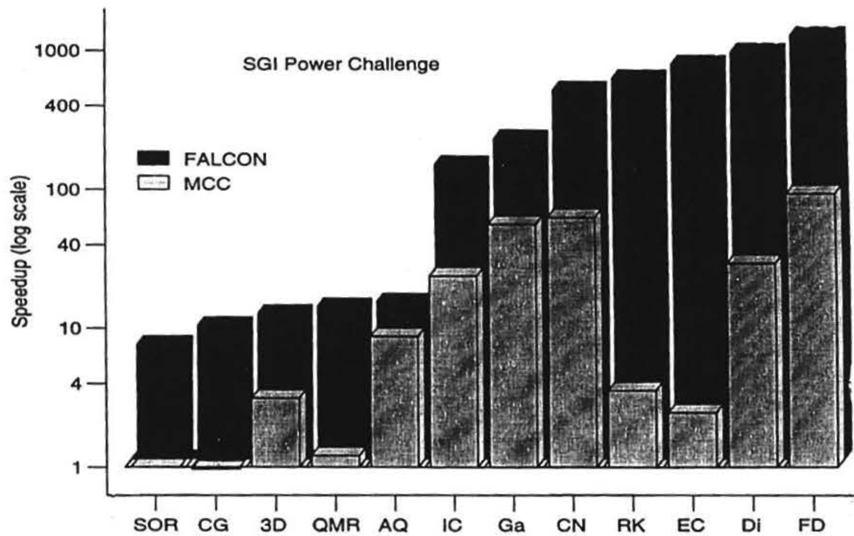

Figure 2: Speedup of compiled programs over MATLAB, running on the SGI Power Challenge.

Assertions indicating the intrinsic type and rank of the loaded variables were added to the M-files to provide MCC with the same information that was extracted by our compiler from the loaded variables.

Figure 2 presents the speedups of the compiled codes over the interpreted MATLAB execution. The darker bars represent the speedup of FALCON's compiler over MATLAB, while the lighter bars represent the speedup of MCC over MATLAB. Due to the large difference in performance for some of the programs, the speedups are shown in logarithmic scale.

The following sections discuss the performance of FALCON's compiler with respect to the interpreted MATLAB programs; the hand-written Fortran 90 programs; and the performance of the C programs generated by MathWorks MCC Compiler.

3.1 Comparison of Compiled Fortran 90 Programs to MATLAB

Our experimental results show that for all programs in the test set, the performance of the compiled Fortran 90 code is better than the respective interpreted execution, and the range of speedups is heavily dependent on the characteristics of the MATLAB program.

Programs that have execution time dominated by built-in functions (CG, SOR, QMR, and 3D) have a small speedup compared to MATLAB. In these programs, the speedup obtained is small because in general, the built-ins use the same optimized library functions that are called by FALCON's compiler and by MCC. Programs that perform mostly elementary scalar operations using SCALARS or element-wise access of VECTORS or MATRICES (FD, Di, EC, RK, and CN) are the ones that benefit the most from compilation. This improvement is due to the more efficient loop control structure of the compiled code and the larger overhead of the indexed assignments within the interpreted code.

Finally, the speedup obtained by AQ resulted from the better handling of indexed assignments and the reallocation of matrices by the compiled program. However, according to a scalability study to determine how problem size affects the relative speed of the programs [7], this improvement varies considerably, depending upon the number of reallocations required by the program, which is in turn dependent upon the input data set and the function being used for the numerical integration.

3.2 Comparison of Compiler Generated Programs with the Hand-written Fortran 90 Programs

Figure 3 presents the speedups of the hand-written Fortran 90 programs over the compiler generated versions. We observe that, for most programs, the performance of the compiled versions is very close to the performance of the hand-written programs.

The largest performance differences occur with IC and SOR. In both cases, the hand-written code performed more than four times faster than the compiler generated code. The

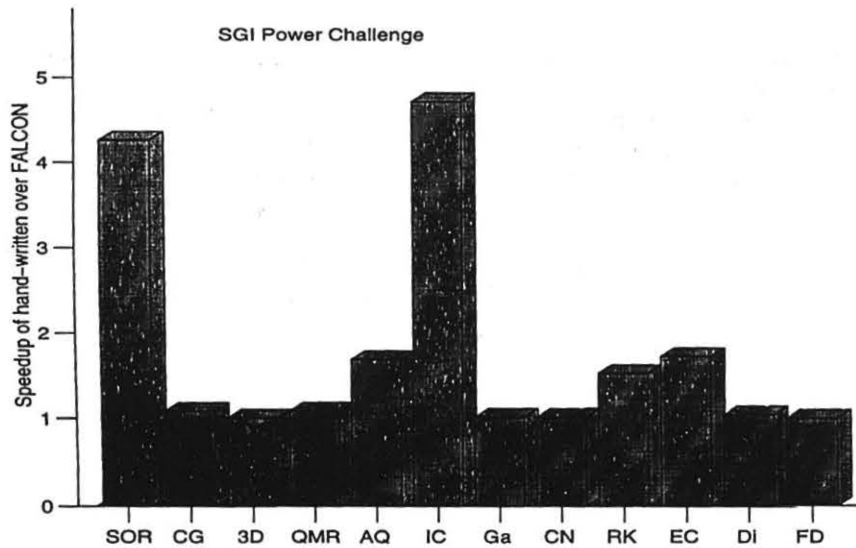

Figure 3: Speedup of hand-coded Fortran 90 programs over the compiler generated versions, running on the SGI Power Challenge.

```
S0:  load %(L) (L is initially REAL)
S1:  for j = 1:n
     ...
S2:    r = sqrt(L(j,j) - S);
S3:    if (r <= 0)
S4:      Error = j;
S5:      L(j,j) = 1;
S6:    else
S7:      L(j,j) = r;
S8:    end
       ...
S9:  end
```

Figure 4: MATLAB code segment for the Incomplete Cholesky Factorization (IC).

reason for the performance difference with the IC program was the inability of the inference mechanism to detect that the conditional statement S3, shown in Figure 4, would prevent the array L to became COMPLEX. Due to the ambiguous-typed expression (sqrt) in S2, the variable r is inferred to be COMPLEX. Thus, L in S7 is also inferred to be COMPLEX. However, since both L and S are REAL, the result of the square root function in S2 can be a REAL non-negative value or a COMPLEX value with its real component equal to zero. Thus, due to the conditional statement[2] S2, S7 will only be executed if r is REAL. Since the inference mechanism is unable to infer the real intrinsic type for the array L, the compiled code performs COMPLEX arithmetic for most of the program, while the optimized program uses REAL variables for the same operations.

The main reason for the performance degradation in the SOR case is attributed to the computation of the following MATLAB expression inside a loop: `x = M \ (N * x + b);` where x and b are vectors, M is a lower triangular matrix, and N is an upper triangular matrix. The hand-coded version considers the shape of M and N and calls specialized routines from the BLAS library to compute the solve operation (\) and the matrix multiplication (N * x) for triangular matrices. The compiled version (as well as MATLAB) uses a run-time test to detect that M is a triangular matrix, and computes the solve using specialized functions. However, $\mathcal{O}(n^2)$ operations are needed to detect the triangular structure of the matrix. Moreover, in both cases, the matrix multiplication is performed using a generalized function for full matrices that performs $2n^2$ operations, while the BLAS function for triangular matrix multiplication performs roughly half the number of operations. Furthermore, it would not be worthwhile to test if the matrix is triangular during run-time because, as mentioned above, the test itself has an $\mathcal{O}(n^2)$ cost.

Other performance differences that are worth mentioning are from EC, RK, and AQ. In the first two programs, the difference in performance is attributed to the computation of the built-in "norm". Both EC and RK compute several norms of vectors of two elements. The compiled programs call a library function for the computation of the norm, while the hand-written programs perform the computation using a single expression which takes into consideration that the vector has only two elements.

Finally, the performance difference observed with AQ is primarily the result of the code generated by the compiler for the reallocation of matrices. In the hand-written code, due to a better knowledge of the algorithm, it was possible to optimize the reallocation process.

3.3 Comparison with the MathWorks MATLAB Compiler

Figure 5 presents the speedups of the codes generated by FALCON over MCC's codes. We observe that all codes generated by FALCON ran faster than their MCC counterparts. In similar experiments, running on a SPARCstation 10 [7], we observed that in three cases (CG,

[2] MATLAB accepts COMPLEX variables in logical operations, but only take into consideration the real component of the variable.

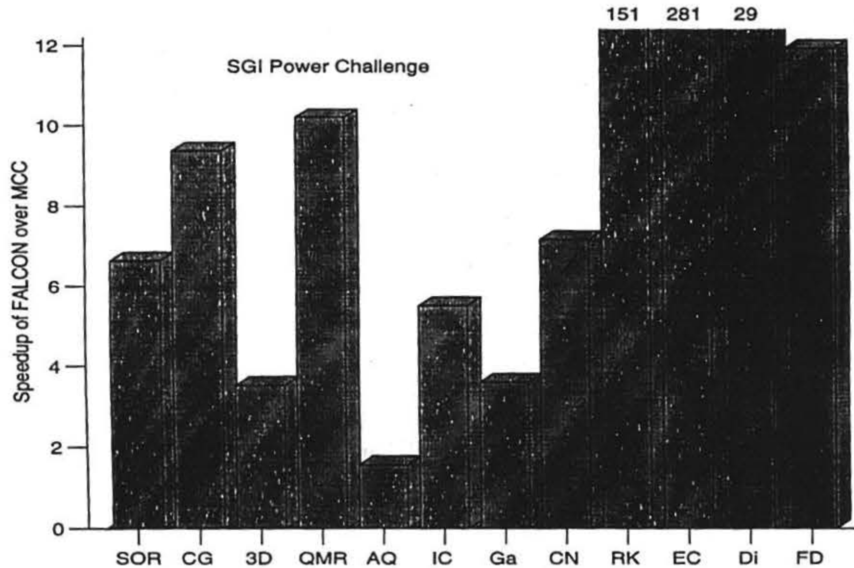

Figure 5: Speedup of FALCON's compiler over MCC on an SGI Power Challenge.

SOR, and QMR) MCC generated programs that ran slower than MATLAB, while all codes generated by FALCON ran faster than MATLAB.

Three programs generated by FALCON (RK, EC, and Di) had significantly better performance than the corresponding MCC versions. The primary reason for these differences, is the lack of more in depth inference analyses by the MathWorks compiler. The MathWorks compiler does not perform use-coverage analysis and simplifies the handling of ambiguous-typed expressions by considering that they always return COMPLEX output. Moreover, as described in [14], the code generated by MCC cannot handle COMPLEX values nor perform subscript checking. To solve these problems, the code generated by MCC calls MATLAB using "*callback functions*" provided in their library.

RK and EC perform several elementary vector operations using vectors of size 2. the code generated by MCC is very inefficient for these kinds of operations because it calls the MATLAB functions to perform VECTOR and MATRICES operations. These callback functions generate an overhead that, in this case, is not amortized due to the size of the vectors. Furthermore, the lack of pre-allocation of variables[3] in the MATLAB code is also responsible for the degradation of the performance of these MCC codes. FALCON's compiler, by contrast, is able to allocate all matrices in these programs outside the main loop, due to

[3] A common practice of MATLAB programmers is to pre-allocate VECTORS and MATRICES using built-ins, such as zeros, to avoid allocation overhead inside of loops.

its symbolic propagation analysis.

Finally, the better performance from Di results from the value-propagation analysis. In this case, the intrinsic type inference mechanism can determine that the Expression:

$$\omega = \frac{4}{2 + \sqrt{4 - \left[cos\left(\frac{\pi}{n-1}\right) + cos\left(\frac{\pi}{m-1}\right)\right]^2}} \tag{1}$$

will always return a REAL value between 1 and 2 for ω, whereas MCC assumes the output of this ambiguous-typed expression to be of type COMPLEX. Although ω is only a scalar variable, for performance reasons it is important to be able to infer its intrinsic type; this variable is used to update an $n \times m$ rectangular grid which is in turn used to solve Laplace's equation in a iterative process. Therefore, if ω is assumed to be COMPLEX, the $n \times m$ matrix that contains the grid will have to be declared and operated as COMPLEX. Thus, the code generated by MCC for Di uses COMPLEX variables for most of its computations, whereas FALCON's generated code uses only REAL variables.

4 Conclusions

We are building a programming environment for the development of scientific libraries and applications. By using MATLAB as the source language and producing Fortran 90 as output, this environment takes advantage of both the power of interactive array languages and the performance of compiled languages. In order to generate code from the interactive array language, we developed a compiler that combines static and dynamic inference methods for intrinsic type, shape, and rank inference, and is optimized with value and symbolic dimension propagation.

As shown by our experimental results, the compiled programs performed better than their respective interpreted executions, with performance improvement factors varying according to the characteristics of each program. For certain classes of programs, FALCON generates code that executes as fast as hand-written Fortran 90 programs, and more than 1000 times faster than the corresponding MATLAB execution on an SGI Power Challenge. Loop-base programs with intensive use of scalars and array elements are the ones that benefit the most from compilation.

Finally, comparisons against a commercial MATLAB compiler (MCC) show that programs generated by FALCON performed up to 280 times faster than the programs generated by the MathWorks compiler, on the SGI Power Challenge. This difference in performance is attributed to the more enhanced inference mechanism utilized by FALCON's compiler.

References

[1] Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1985.

[2] Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1993.

[3] Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., and Tu, P. Parallel Programming with Polaris. *IEEE Computer 29*, 12 (December 1996), 78–82.

[4] Blume, W., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, B., Rauchwerger, L., Tu, P., and Weatherford, S. Polaris: Improving the Effectiveness of Parallelizing Compilers. In *Languages and Compilers for Parallel Computing* (August 1994), K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., Lecture Notes in Computer Science, vol. 892, Springer-Verlag, pp. 141–154. 7th International Workshop, Ithaca, NY, USA.

[5] Budd, T. *An APL Compiler.* Springer-Verlag, 1988.

[6] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language and Systems 13*, 4 (October 1991), 451–490.

[7] De Rose, L. A. *Compiler Techniques for MATLAB Programs.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1996.

[8] DeRose, L., Gallivan, K., Gallopoulos, E., Marsolf, B., and Padua, D. FALCON: A MATLAB Interactive Restructuring Compiler. In *Languages and Compilers for Parallel Computing* (August 1995), C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., Lecture Notes in Computer Science, vol. 1033, Springer-Verlag, pp. 269–288. 8th International Workshop, Columbus, Ohio.

[9] DeRose, L., Gallivan, K., Gallopoulos, E., Marsolf, B., and Padua, D. FALCON: An Environment for the Development of Scientific Libraries and Applications. In *Proc. of the KBUP95: First international workshop on Knowledge-Based systems for the (re)Use of Program libraries* (Sophia Antipolis, France, November 1995).

[10] Gallopoulos, E., Houstis, E., and Rice, J. R. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Computational Science & Engineering 1*, 2 (Summer 1994), 11–23.

[11] Garcia, A. L. *Numerical Methods for Physics.* Prentice Hall, 1994.

[12] Gilman, L., and Rose, A. *APL : An Interactive Approach.* Wiley, 1984.

[13] The Math Works, Inc. *MATLAB, High-Performance Numeric Computation and Visualization Software. User's Guide*, 1992.

[14] THE MATH WORKS, INC. *MATLAB Compiler*, 1995.

[15] MATHEWS, J. H. *Numerical Methods for Mathematics, Science and Engineering*, 2nd ed. Prentice Hall, 1992.

[16] SCHWARTZ, J. T. Automatic Data Structure Choice in a Language of a Very High Level. *Communications of the ACM 18* (1975), 722–728.

[17] TU, P., AND PADUA, D. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *Proceedings of the 9th ACM International Conference on Supercomputing* (Barcelona, Spain, July 1995), pp. 414–423.