In C.-H. Huang, P.Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (Editors), **Languages and Compilers for Parallel Computing**, pages 269-288. Springer-Verlag, August 1995. (8th International Workshop, LCPC'95, Columbus OH.)

FALCON: A MATLAB Interactive Restructuring Compiler

L. DeRose, K. Gallivan, E. Gallopoulos, B. Marsolf and D. Padua

September 1995

CSRD Report No. 1448

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, Illinois 61801

FALCON: A MATLAB Interactive Restructuring Compiler

L. De Rose* K. Gallivan** E. Gallopoulos*** B. Marsolf** D. Padua†

Center for Supercomputing Research and Development
and Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
{derose, gallivan, stratis, marsolf, padua}@csrd.uiuc.edu

Abstract. The development of efficient numerical programs and library routines for high-performance parallel computers is a complex task requiring not only an understanding of the algorithms to be implemented, but also detailed knowledge of the target machine and the software environment. In this paper, we describe a programming environment that can utilize such knowledge for the development of high-performance numerical programs and libraries. This environment uses an existing high-level array language (MATLAB) as source language and performs static, dynamic, and interactive analysis to generate Fortran 90 programs with directives for parallelism. It includes capabilities for interactive and automatic transformations at both the operation-level and the functional- or algorithm-level. Preliminary experiments, comparing interpreted MATLAB programs with their compiled versions, show that compiled programs can perform up to 48 times faster on a serial machine, and up to 140 times faster on a vector machine.

1 Introduction

The development of efficient numerical programs and library routines for high-performance parallel computers is a complex task requiring not only an understanding of the algorithms to be implemented, but also detailed knowledge of the target machine and the software environment. Today, few effective tools are available to help the programmer of high-performance computers perform the transformations necessary to port parallel programs and library routines across

* Supported by the CSRD Affiliates under grant from the U.S. National Security Agency.

** Supported by the National Science Foundation under Grant No. US NSF CCR-9120105 and by ARPA under a subcontract from the University of Minnesota of Grant No. ARPA/NIST 60NANB2D1272.

*** Supported by the National Science Foundation under Grant No. US NSF CCR-9120105.

† 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.

machines. This task is made even more difficult by the great diversity in overall organization between the different classes of parallel computers. This difficulty holds true even when the objective is to develop code for a single target machine.

Several approaches to facilitate the development and maintenance of parallel code are currently under study. One approach is the automatic translation of conventional programming languages, notably Fortran, into parallel form. SUIF [2], a compiler developed at Stanford, and Parafrase-2 [27] and Polaris [25], developed at Illinois, are examples of this first approach. Another approach is to extend conventional languages with simple annotations and parallel constructs. This second approach also involves the development of translation techniques for the extensions. Examples include High Performance Fortran [19] and pC++ [7]. Finally, a third approach is to accept a very high-level description of the mathematical problem to be solved and automatically compute the solution in parallel. Examples of this approach are //ELLPACK [20], developed at Purdue, ALPAL [12], developed at Lawrence Livermore Laboratories, and EXTENT [15], developed at Ohio State University.

FALCON is a programming environment that includes capabilities for interactive and automatic transformations at both the operation-level and the function- or algorithmic-level. This environment supports the development of high-performance numerical programs and libraries, combining the transformation and analysis techniques used in restructuring compilers with the algebraic techniques used by developers to express and manipulate their algorithms in an intuitively useful manner [16]. As we envision it, the development process should start with a simple prototype of the algorithm and then continue with a sequence of automatic and interactive transformations until an effective program or routine is obtained. The prototype and the intermediate versions of the code should be represented in an interactive array language. The interactive nature of the language is important to facilitate development and debugging. Array operations are the natural components of most numerical code; therefore, including array extensions in the base language should make the programs easier to develop and modify.

The ideal interactive array language should be easy to learn and capable of accessing powerful graphics and other I/O facilities. We are using an existing language in order to shorten the learning curve for our system and give us immediate access to existing support routines. APL and MATLAB [22] are the most widely used interactive array languages today. We have chosen MATLAB because it is the more popular of the two and has a conventional syntax that facilitates learning it. In MATLAB, as in any interactive array language, the type, rank and shape of variables are dynamically determined. However, for the generation of efficient code, static declarations are necessary. In our system, declarations are inserted automatically by the compiler. If the compiler is unable to determine type, rank, or shape statically, code is generated to make the determination at runtime. To facilitate code development, the system will also provide capabilities for the user to insert optional declarations. By compiling the array language, we expect to improve the performance of the programs by

eliminating inefficiencies caused by the interpretation. As discussed in Section 3, our preliminary experiments show, that for programs that require certain types of operations, the compiled version performed up to 140 times faster than the MATLAB version.

Another effort in this research is devoted to the development of the interactive restructuring system which, through correctness-preserving transformations, will lead to efficient code from the original prototype. Our system includes capabilities for interactive restructuring of both expressions and program statements via standard restructuring techniques, algebraic restructuring techniques, and the combination of the two. We designed a uniform transformation system based on an extensible collection of rewriting rules. Some transformations will operate on a single expression, while others, such as data structure selection, have a global effect on the code. Also, transformations for library routine selection are part of the interactive transformation. Library selection will help the programmer take advantage of a library of routines highly tuned for a particular target machine. Many other transformations, including those to enhance locality and exploit implicit parallelism, are also being developed.

As discussed in more detail below, we take advantage of the collection of functions and operations in MATLAB to facilitate the translation process. Our programming environment incorporates knowledge of the algebraic and computational properties of many MATLAB functions. This enables us to perform complex algebraic transformations on the source program. The availability of information at this semantic level will substantially increase the chances of success where traditional high-level language restructurers fail, due to their almost exclusive focus on low-level operations. The chances of successfully transforming MATLAB programs are further increased by the simplicity of the language. In our experience, Fortran restructurers often fail because of aliasing or the undisciplined use of GO TOs.

The rest of this paper is organized as follows. In Section 2, we present a more detailed description of the system. The current status and some experimental results are presented in Section 3. Finally, our conclusions are presented in Section 4.

2 System Description

In this section, we describe the overall organization of the system as it was designed, and then present additional information about the techniques already implemented. As mentioned in the Introduction, our objective is to develop a programming environment for implementing, maintaining, and extending scientific programs and numerical library routines. Our long-term goal is to generate code for a variety of machines, including conventional workstations, vector supercomputers, shared-memory multiprocessors, and multicomputers. However, to be able to solve several difficult problems that arise in the translation of MATLAB codes, we concentrated our initial efforts on a conventional workstation and vector supercomputers.

The main target language of the system is Fortran 90, which is very convenient for our purposes because of its similarities with MATLAB. Later, we plan to also generate pC++ as a target language in order to explore object-oriented issues that arise in numerical programming.

The environment consists of four main modules, namely the Program Analysis module, the Interactive Restructuring and Routine Selection module, the Code Generator module, and the Parallelizer module. Figure 1 presents the flow of information between the main system components. The process starts with a prototype in the form of a MATLAB program. This prototype is analyzed to infer information such as type, shape, and dependences. Whenever this information cannot be determined statically, the program analysis phase generates code to make the determination at runtime. Hence, after this transformation, the type and shape of each variable in the program are known. This is necessary before proceeding to the next phase: interactive restructuring. The reason is that, as discussed below, runtime type checking usually involves code replication and each copy should be available for transformation.

Making use of a collection of transformation rules in the database and the information gathered by the program analyses, the user can interactively transform the original program by changing the structure of the code, selecting data structures for variables, and selecting library routine implementations for each invocation. In Figure 1, control flows back and forth between restructuring and analysis because, after the structure of the code has changed, it might be necessary to repeat the analysis. The user could extend the database of transformation rules and library routine information as new transformations are conceived or new library routines are incorporated into the system.

The code generator makes use of the information gathered by the program analysis phase to generate a Fortran 90 program or routine. Furthermore, the code generator will also produce a MATLAB program. The actual generation of parallel code will be done by a restructuring compiler, such as Polaris [25] or KAP [21]. Our system facilitates the translation into parallel form by annotating the target Fortran program and by using array operations whenever possible. Furthermore, the Fortran restructurer should be capable of applying transformations automatically whenever possible, thereby saving the user of the interactive restructurer additional work.

In the rest of this section we discuss the program analysis and the restructuring modules.

2.1 Program Analysis Module

The Program Analysis Module reads in the MATLAB program and converts it into an abstract syntax tree (AST). During this process, the system performs program analysis to identify parallelism and to determine *variable properties*, such as type, rank (e.g vector, square matrix, scalar, etc.), shape (i.e. size of each dimension), and structure (e.g. upper triangular, diagonal, Toeplitz). Previous work on type inference has been done for several high-level languages. In particular, type inference techniques developed for SETL [28] and APL [8, 11]

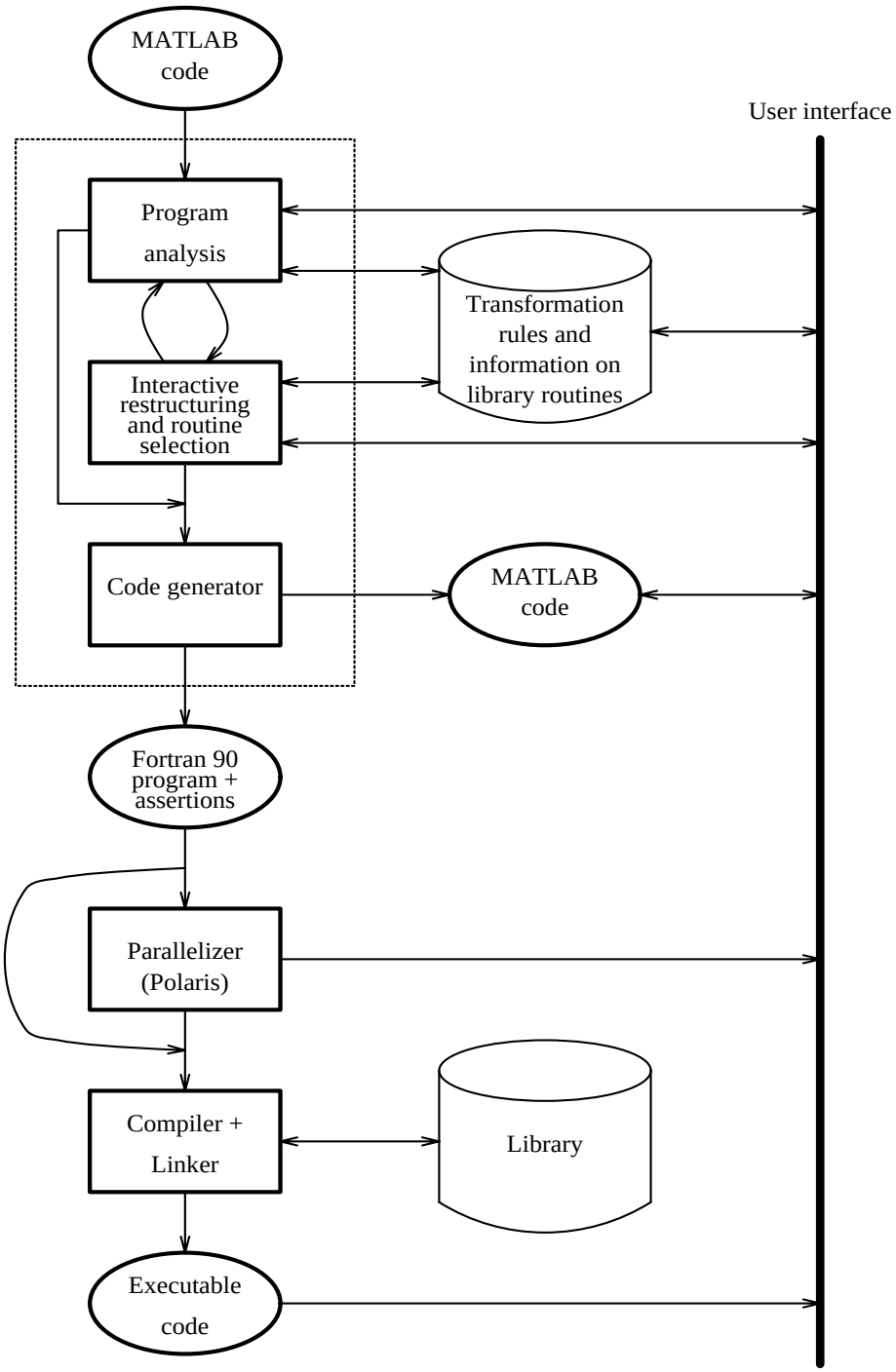

Fig. 1. Program development environment.

are most relevant to our work. These two languages are similar to MATLAB in that they can be executed interactively, are usually interpreted, and operate on aggregate data structures. We make use of some of the techniques developed for these two languages and extend them, with techniques originally developed for Fortran, to analyze array accesses within loops and represent the information gathered in a compact form [29]. These techniques are necessary for MATLAB, since arrays are often built using Fortran-like loops and assignments that may be distributed across several sections of the code. In contrast, the techniques developed for APL assume that arrays are usually built by a single high-level array operation. Determining variable properties automatically should facilitate the program development process. Even though the system also accepts user assertions, the users may prefer to leave the determination of the properties to the system, especially when the program under development makes use of routines whose local variables have different shapes and types, depending on the context in which they are invoked.

To determine variable properties, we make use of *high-level summary information* on the behavior of the MATLAB built-in functions. For example, the system will take into account the fact that the function `lu` returns triangular matrices; using this information, it is able to utilize optimized functions, instead of generalized methods, on these returned matrices in the compiled code. The use of this high-level information should greatly increase the accuracy of the analysis over the approach used in conventional high-level language compilers, where only the information from elementary operations is used.

Use-Definition Coverage The MATLAB program is internally represented in Static Single Assignment (SSA) form [14]. This is a convenient representation for many of the analysis algorithms that are implemented. In the SSA representation, it is evident which definitions affect (or cover) a particular use of a scalar variable. Using this information, it is easy to perform variable renaming and privatization. The usefulness of renaming to improve the accuracy of the analysis is illustrated in the following pseudocode, which uses `i` as the imaginary unit.

```
a = 2 * i
b = exp(a)
a = pi / 4.
z = sin(a)
```

In this example, the variable `a` is used for two different purposes. Giving a different name to the second use of `a` makes it possible to assign the type `complex` to the first use of `a`, and type `real` to the second. In the absence of renaming, it would be necessary either to perform dynamic analysis, as discussed below, or to assume `a` to be `complex` all the time. While either alternative would produce a correct program, the code would clearly be more efficient if the correct type of every use were determined statically.

While it is easy to determine which definitions cover a given use in the case of scalars using the SSA representation, a more complex approach is necessary

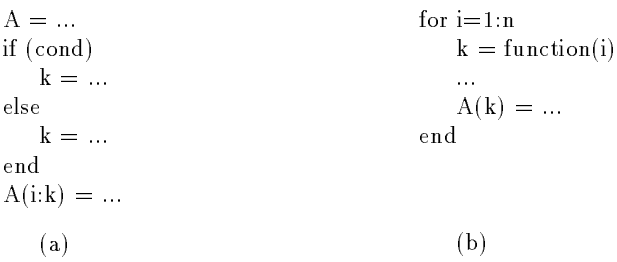

Fig. 2. Examples requiring use-definition coverage analysis

in the case of arrays. For arrays, this information is useful not only to determine whether different types of values are assigned to array elements in different sections of the code, but also to determine where to allocate and reallocate arrays and how much storage is required in each case.

This last problem, which seems to be ignored by current APL compilers [8, 11], is illustrated by the pseudocode segment in Figure 2. In this case, the compiler needs to determine if the partial assignment to `A` is a subrange of the previous definitions of the variable. If not, reallocating `A` may be necessary.

As a second example, consider the pseudocode segment in Figure 2(b). In this case, the compiler needs to estimate the maximum value of `k` to avoid the overhead of dynamic allocation for every iteration of the loop.

We plan to attack the problem of array definition coverage using the analysis techniques described in [29]. These have proven quite effective in detecting privatizable arrays in Fortran programs and should be at least equally effective in this context. The strategy uses data flow analysis to determine which definitions cover the uses of array elements. It also makes use of a number of conventional analysis algorithms, including induction variable recognition [18] to compute upper and lower bounds of these variables, and propagation of the range of values of scalars and of those arrays used as subscripts [6, 13]. This analysis is performed interprocedurally whenever the modules referenced in the program are available to the system. In the case of intrinsic functions, information on the elements accessed are available in a database that is consulted during the computation of ranges. The analysis of ranges and induction variables is facilitated by the use of the SSA representation.

Type Inference To generate Fortran declarations and to support structure selection, the system infers variable properties. Although described separately here, shape and structure inference work in coordination with the coverage analysis discussed in the previous subsection. Variable properties are estimated using a forward/backward scheme [1].

For type inference, we use a *type algebra* similar to the one described in [28] for SETL. This algebra operates on the type of the MATLAB objects and is

implemented using tables for all operations. Each node, of the graph representing the program, contains attribute fields to store inference information. These fields are filled during the static inference phase and propagated through the graph whenever a new attribute is synthesized. If these attributes are inferred to have different (or unknown) types, then the node is marked to be resolved during the dynamic phase, or a type that subsumes all possible types is assigned to the variable.

Array shapes are estimated using the induction variable and range propagation algorithms mentioned in the previous subsection. Structural information is computed in similar ways. Again, in the case of rank and shape, code to apply dynamic analysis is generated whenever the static analysis fails. For structural inference, the system uses semantic information to identify special matrix structures, such as diagonal and triangular. This structural information is propagated using an "algebra of structures" which defines how the different structures interact for the various operations. This information will be used by the compiler, or by the interactive restructurer, to replace general methods that operate on regular dense matrices with specialized functions for structured sparse matrices. For example, consider the following MATLAB code for the solution of a linear system $Ax = b$, using a LU decomposition:

```
[L, U, P] = lu(A);
y = L \ (P * b);
x = U \ y;
```

The first statement calls a built-in function (`lu`) that returns a lower triangular matrix `L`, an upper triangular matrix `U`, and a permutation matrix `P`. For a regular compiler, the second statement should perform a matrix-vector multiplication (Pb) and solve the linear system $Ly = Pb$. Finally, the last statement should solve the linear system $Ux = y$. However, by taking into consideration the semantics of the array language and knowing the properties of `L`, `U`, and `P`, the system will infer that the matrix-vector multiplication (`P * b`) is only a permutation on the vector `b`, and that the two linear systems to be solved are triangular systems. Using this information, the multiplication operation and the general linear system solve can be replaced by specialized algorithms during the restructuring phase.

As mentioned above, a facility for user interaction is an important component of the analysis phase. To simplify the interaction, the system will internally record how the properties of a variable affect other variables. The objective is to create a hierarchy of information interdependence and use it to query the user only about the topmost level of the hierarchy.

Dynamic Analysis In some cases, when the static information is insufficient, the user may not be able (or willing) to provide enough information to determine variable properties. In those cases, the system generates code to determine these properties at runtime and, based on this determination, allocates the necessary space and selects the appropriate code sequence. We follow an approach similar

to that used in many systems: associating tags with each array of unknown type or shape. Based on these tags that are stored in *shadow variables*, conditional statements are used to select the appropriate operations and to allocate the arrays. Similar shadow variables are generated for dynamic type inference. For example, if the type of `A` in an assignment of the form `B = A + 0.5` were unknown at compile-time, the system would generate two variables for `A` ($\texttt{A}^{\texttt{complex}}$ and $\texttt{A}^{\texttt{real}}$) along with a shadow variable for the type of A (`A_type`) and similar variables for `B`. Then the assignment is transformed into:

```
B_type = A_type
if(A_type == t_complex)
    B^complex = A^complex + 0.5
else
    B^real = A^real + 0.5
end if
```

Clearly, such transformations cannot be indiscriminately applied. For example, if `A` is a scalar, it may be faster to assume throughout the program that it is a complex variable and generate code accordingly. However, if `A` is a large array, or if it is a scalar that is operated with large arrays, the overhead of the `if` statement will be minimal compared to the extra cost of the assignment, assuming that `A` is often a real.

Also, in the case of right-hand sides with many operands, or in the case of loops containing many statements, the number of possibilities would grow exponentially. This problem is reduced by transforming the long expressions into shorter ones, or by distributing the loop; in some cases, however, the best strategy may be to assume that the variable is `complex`. In any case, in our final version, a cost model that incorporates information from typical input data sets will be defined to decide the best strategy in each case. In the current version, static analysis techniques are used to minimize the number of tests applied at runtime.

2.2 Interactive Restructuring and Routine Selection Module

Code restructuring will be done within this module by applying transformations to expressions and blocks of statements. The system will have the necessary interface mechanisms to accommodate the use of automatic transformation modules. Initially, however, we are focusing on interactive transformations based on an easy-to-extend database of rewriting rules. The user will be able to mark any section of the program, including compound statements, subexpressions, and function invocations, for restructuring. The system will then determine which of the transformations in the database apply to the code segment marked by the user. To make such a determination, it will use information automatically inferred from the code, such as dependence graphs, and properties of variables and expressions in the code. The system will then allow the user to select which applicable transformations to perform.

During code restructuring, the user will select the implementation to be used for each function invocation in the final version of the program. This selection process could be done automatically, based upon target machine characteristics. When the selection is done interactively, the user will be able to target specific libraries to be used in the implementation and to select a version of the implementation to be used for a specific function invocation. There will also be a facility that allows the user to query the system's database, which contains information about the library for particular target machines. This information will include: version descriptions; performance information and tuning recommendations; the type of processing used (parallel on all processors, parallel on a subset of processors, vector, scalar); and possible rewriting of the library routine in terms of other lower-level library routines.

There has been some work on interactive program restructuring and there are some commercially available interactive Fortran restructurers, such as FORGE [3]. Also, restructuring of high-level operators has been discussed in the literature [4, 12], although there are no widely-used systems that apply these types of transformations, as no system today includes capabilities for algebraic, control structure, and library selection transformations.

We are planning on implementing several restructuring techniques for complex transformations on the code. These transformations will be used either to improve the performance of the code or to enhance the numerical properties of the algorithm. These techniques can be divided into two groups: *algebraic restructuring*, based on the algebraic information, and *primitive-set restructuring*, based on the primitives being used.

Algebraic Restructuring This part of the system will use the algebraic rules defined for the variables, whether they are scalars, vectors, or matrices, to restructure the operations performed on the variables. To perform such manipulations, symbolic computation tools, such as Maple [10], can be employed. In some cases, applying these rules may be similar to the standard loop-based restructuring strategies used by conventional restructuring compilers, such as loop blocking. However, we also want to be able to handle special matrix classes and more complex operators. Our goal in applying the algebraic rules to matrices and vectors will be to achieve better restructuring than when the rules are only applied to scalar operations. For the interactive application of transformations, algebraic restructuring may provide a more convenient form for the user than loop-based restructuring, even when the effect on the resulting code is the same.

The main rules that will be considered for these transformations are the algebraic properties of associativity, distributivity, and commutativity for vectors and matrices. These properties for vectors and matrices are not as simple as for scalars. For instance, the multiplication operations are not commutative and, with matrices, the commuted operation may not even be possible. For example, $A * B$ may be possible whereas $B * A$ may not be possible. Furthermore, the order in which matrix operations are performed can have a significant effect on the number of operations to be performed [24]. Consider the multiplication of

$$
\begin{aligned}
x_i^T &= v_i^T A \\
Y_i &= 2 v_i x_i^T \\
A &= A - Y_i \\
x_j^T &= v_j^T A \\
Y_j &= 2 v_j x_j^T \\
A &= A - Y_j
\end{aligned}
$$

(a)

$$
\begin{aligned}
x_i^T &= v_i^T A \\
Y_i &= 2 v_i x_i^T \\
x_j^T &= v_j^T A \\
Y_j &= 2 v_j x_j^T \\
\alpha &= v_j^T v_i \\
Z_j &= 2 \alpha v_j x_i^T \\
A &= A - Y_i - Y_j - Z_j
\end{aligned}
$$

(b)

Fig. 3. Two examples of the application of two Householder transforms.

three vectors, $v_1 * v_2^t * v_3$, where each vector contains n elements. If the matrix operations are grouped as $(v_1 * v_2^t) * v_3$, the calculation takes $\mathcal{O}(n^2)$ floating point operations whereas, if the operations are grouped as $v_1 * (v_2^t * v_3)$, it only takes $\mathcal{O}(n)$ operations.

In addition to the simpler properties, the system will attempt to understand more complex operators, such as the Gauss transform and the Householder transform. These types of transformations are used to build matrix factorization algorithms: the Gauss transform for the LU factorization and the Householder transform for the QR factorization. By examining the properties of the transformations, we can reach an understanding of how these properties can be used for restructuring. For example, consider how the Householder QR factorization could be blocked. By examining the code, the application of two Householder transforms, H_i and H_j to the matrix A, could be viewed as $H_j(H_i A)$. Using the definition of the Householder transform, $H_i = (I - 2 v_i v_i^T)$, the operations normally required in the code would be as presented in Figure 3(a). This code requires the use of 6 $\mathcal{O}(n^2)$ operations. If, however, the transforms are combined before they are applied, $(H_j H_i)A$, then the operations presented in Figure 3(b) are required.

With this code there are now 8 $\mathcal{O}(n^2)$ operations and a new $\mathcal{O}(n)$ operation for the calculation of α. Although this new code requires additional operations and memory storage, in practice it provides better locality thus resulting in improved performance. These additional operations were generated by utilizing the definition of the transform; they would not have been apparent by just examining the code. This utilization of algebraic information is being performed not only by algorithm developers, but is also being explored by compiler writers [9].

The transformations will be based on patterns that the system can recognize in the code and on the replacements for these patterns. The developer will be able to select a segment of code and the system will indicate which patterns match the segment. The developer will then select which transformation to apply from those possible. For example, consider some replacement patterns for statements involving the MATLAB solve operation, "\", in Figure 4.

These patterns consist of two main parts: the pattern to be replaced and

REPLACE
  INTEGER n
  REAL $M(n,n), b(n), x(n)$
  $x = M \setminus b;$
WITH
  REAL $L(n,n), U(n,n)$
  REAL $y(n)$
  $[L,U] = lu(b);$
  $y = L^{-1} \times b;$
  $x = U^{-1} \times y;$
END

REPLACE
  INTEGER n
  REAL $M(n,n)\{DIAGONAL\}$
  REAL $b(n), x(n)$
  $x = M \setminus b;$
WITH
  $x = b ./ diag(M);$
END

Fig. 4. Sample restructuring patterns for MATLAB solve operation.

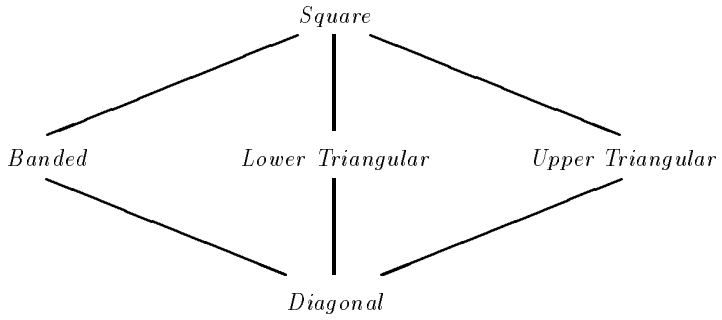

Fig. 5. Partial ordering of matrix types.

the resulting code. Within the pattern, it is important to match not only the operation, but also the characteristics of the operands. For instance, in the first pattern, the matrix M is a square matrix; but, for the second pattern, M must be a diagonal matrix. In order to reduce the number of patterns required, we will attempt to order the matrix types according to their characteristics.

The structured sparse matrix characteristics form a lattice through which properties are inherited from the original *square* matrix. For instance, operations on a *square* matrix will work on a *lower triangular* matrix, and operations on a *lower triangular* matrix will also work on a *diagonal* matrix. A partial ordering of these types is presented in Figure 5.

In order to support complex algebraic transformations, it is sometimes necessary to match code segments according to characteristics of the operations instead of exactly matching the operations. In Figure 6, a pattern for interchanging loops shows how certain dependence conditions can be placed on the matching of the loop body. In this example, the loops can be interchanged only if the data dependences do not contain a direction vector of the form <,> [26].

```
REPLACE
    INTEGER i,j
    do i = 1:n
        do j = 1:n
            <BODY>
            WHERE
                no-match-direction-vector("<, >", dependence(<BODY>))
            END
        end
    end
WITH
    INTEGER i,j
    do j = 1:n
        do i = 1:n
            <BODY>
        end
    end
END
```

Fig. 6. Pattern for loop interchange.

Primitive-set Translation Primitive-set translation can also be used to translate the code to the level of numerical operations that work best for the target machine and application [17]. Instead of dealing with the code only at a matrix operation level, this phase will be able to decompose the algorithms to matrix–vector operations and vector–vector operations; or, in some circumstances, it will form higher-level operations by combining multiple lower-level operations. This optimization technique should be guided by factors such as the machine architecture, availability of low-level libraries, and problem size, with the goal of achieving the best performance. This technique also allows the selection of library routines to implement the operations in the code.

These translations can be viewed in two ways: in an algebraic sense and in a library sense. In the algebraic sense, primitive levels will be switched via the combination or decomposition of matrix and vector operations. In the library sense, primitive levels will be switched via modifying the code so that it maps to the subroutines in a specific library. The end results, however, will be similar regardless of which of the two views is used.

When mapping an operation to the subroutines in a library, the subroutines may be at the same primitive level, at a lower level, or at a higher level. Switching to a lower primitive level will involve taking an operation and decomposing it into multiple operations. This will be achieved by decomposing one, or both, of the operands into substructures and then applying the operation with these substructures. For instance, matrix multiplication can be performed by: using the Level-3 BLAS subroutine `DGEMM` once; using the Level-2 BLAS subroutine `DGEMV` in a loop; or, using the Level-1 BLAS subroutine `DDOT` in a doubly nested loop. The code can be transformed using patterns similar to those used for the

```
REPLACE
 INTEGER n
 REAL A(n,n), B(n,n), C(n,n)
 C = A * B;
WITH
 INTEGER i
 SUBROUTINE DGEMV(P_IN,P_IN,P_IN,P_IN,P_IN,P_IN,P_IN,P_IN,P_IN,P_OUT,P_IN)
 for i = 1:DIM(B,2)
   C(1,i) = DGEMV('N',DIM(A,1),DIM(A,2),1.D0,A,DIM(A,1),B(1,i),1,0.D0,1);
 end
END
```

Fig. 7. Pattern for mapping matrix multiplication to level-2 of the BLAS.

algebraic restructuring, as shown in Figure 7 for the Level-2 BLAS subroutine.

As with algebraic restructuring, it will be necessary for the code developer to specify which transformation to use when multiple patterns match the operation. It will be possible for the developer to select the patterns from a specific library to use for all operations, or to examine the code and select a pattern for each specific operation. For each library the system will support, replacement patterns must be developed to map the MATLAB operations to subroutines in the library.

Moving to a higher primitive level, therefore, would involve finding multiple primitives that can be combined into a single operation. However, for the transformation system to do this, it needs to search the program for multiple statements that fit a known pattern, typically a loop of operations, or to react to the user's directions.

2.3 Support for Parallelism

In the initial implementation, the primary support for parallelism will be the ability to place data dependence assertions within the code using directives. Such directives can be used to supply information, such as interprocedural analysis and the independence of data accesses. For instance, the calls to built-in functions in MATLAB are known to be side-effect free by the semantics of the language. However, if the calls to the built-in functions are translated to subroutine calls for a specific library, the semantics of the calls may no longer guarantee that the operation is side-effect free. Consider the example in Figure 8, where the matrix multiplication `C = A * B` has been decomposed into a series of independent matrix-vector products to allow parallel execution. In this example, the left side shows the original code, and the right side shows the code after it has been translated to use a level-2 BLAS primitive. A traditional compiler would not be able to determine if the loop is parallel without performing analysis of the `DGEMV` code. However, our system will be able to assert that the calls can be performed in parallel since the functionality of the original code was parallel.

```
for i = 1:n                          for i = 1:n
    C(:,i) = A * B(:,i);                 call DGEMV('N',n,n,1.0D0,A,n,
end                                        B(1,i),1,0.0D0,C(1,i),1);
                                     end
```

Fig. 8. Two implementations showing the loss of dependence information.

```
for i = 1:n                          ind = nonzeros(A);
    B(i) = A(i) + B(i);              len = size(ind);
end                                  for i = 1:len
                                         B(ind(i)) = A(ind(i)) + B(ind(i));
                                     end
```

Fig. 9. Two vector additions showing the loss of dependence information.

A second example of dependence information involves the use of sparse data structures. The code in Figure 9 shows a code segment for adding two vectors together and the resulting code when one of the vectors, `A`, is changed to a sparse vector. A traditional compiler would not be able to determine that the indices contained in `ind` vector of the transformed code are unique. Our system, however, would be able to assert that the indices are unique because it understands the semantics of the `nonzeros` function.

3 Current Status and Experimental Results

Presently we have a compiler that can parse MATLAB code into the AST and generate Fortran 90 for most operations and built-in functions. This current version of the compiler supports: the static analysis for type, rank, and shape; the utilization of shadow variables for dynamic inference; the generation of code for the dynamic allocation of variables; and the interactive replacement of MATLAB statements using simple patterns. Future work on the compiler includes: the implementation of the structural inference; an optimization of the dynamic analysis for indexed variables; and the expansion of the pattern matching system.

Currently, the primitive–set translation system supports translating simple MATLAB operations into subroutine calls for a specific library. The transformation system is being modified to support other combining rules for mapping the MATLAB operations to subroutine calls. The new rules allow one MATLAB operation to be mapped to multiple subroutine calls as well as multiple MATLAB operations to be mapped to a single subroutine call.

Figure 10 shows an example of the code generated by the transformation system which used the replacement pattern from Figure 7. In this figure, the original MATLAB code segment is presented, followed by the code segment that would utilize a level-2 BLAS library. Other libraries, such as a level-3 BLAS or a sparse library, can be supported.

MATLAB code:

```
D = rand(4);
F = D * D;
```

Translated Fortran 90 code:

```
call RANDOM_NUMBER(D)
do T__3 = 1,4
   call DGEMV('N', 4, 4, 1.0D0, D, 4,
      D(1, T__3), 1, 0.0D0, F(1, T__3), 1)
end do
```

Fig. 10. Example of code translation for matrix multiplication, using level-2 BLAS.

			Fortran 90	
Algorithm	Iterations	MATLAB	Generated	Hand coded
CG	103	11.2	4.8	4.6
QMR	54	12.3	4.8	4.4
SOR	40	15.9	6.8	2.0

Table 1. Execution times in seconds using optimized MATLAB code.

We used two sets of programs to test the performance of the compiler. The first set contains three MATLAB code segments provided in Netlib. These programs were written for general use; hence, they were not completely optimized for specific matrix structures. However, for our tests, we hand-optimized these MATLAB programs, and also wrote a Fortran 90 version of the same algorithms for comparison with the compiler generated codes. The main characteristic of these programs is that none of the variables are referenced or assigned using indices (i.e. only array operations on whole arrays are present).

These code segments correspond to the following algorithms presented in [5] for the iterative solution of linear systems: the Conjugate Gradient method (CG), the Quasi-Minimal Residual method (QMR), and the Successive Overrelaxation method (SOR). The programs were tested using a 420×420 stiffness matrix from the Harwell-Boeing Test Set. To control the time required for the experiments, the tolerance was set to 10^{-5} for the CG and QMR algorithms, and 10^{-2} for the SOR. The experiments were conducted on a SPARCstation 10. The times are presented in Table 1.

When comparing the timing results presented in Table 1, we observe that the compiled programs are almost three times faster than the interpreted programs. Since both versions use BLAS and LINPACK routines to perform the numerical operations on the matrices, this difference in performance is attributed to the overhead of the interpretation of the MATLAB code. However, since most of the hand optimizations used to improve the performance of the MATLAB execution will be done automatically by the compiler with the structural inference, we can expect an even greater improvement in performance, when comparing with programs that were written for general use.

When comparing the hand-coded Fortran 90 programs with the compiler generated versions, several observations can be made. First, in the CG case, the performance of the compiled version was very close to the performance of the

hand-coded program. This 5% difference is attributed to the overhead of the runtime inference. Second, in the QMR program we observed a difference in performance on the order of 10%. This larger performance difference is because, in the current version of the static inference mechanism, operations (such as square root), that rely upon the value (in addition to the type) of the input variable to determine the resulting type, are always considered to result in a `complex` variable. Since the QMR program had a square root operation, some of the variables in the compiled program were defined to be of type `complex`. The operations with these complex variables generated a second source of overhead, due to the larger number of computations required by complex operations. We are extending our inference mechanism for this kind of function, by trying to keep track of each variable's sign.

Finally, in the SOR case, we observe that the hand-coded version is almost four times faster than the compiled code. The main reason for this difference is because the hand-written Fortran 90 code takes advantage of certain matrix structures to call specialized routines to perform operations on these matrices. The compiled version, as well as MATLAB, uses more expensive routines that try to detect such structures at runtime only when it is profitable. Otherwise, they operate on full matrices. We observe that, after the structural inference process is complete, the compiler will be able to detect some specialized matrix structures, and the performance of the generated code will be closer to the hand-optimized code.

The second set of test programs were performed using code segments that contain indexed variables. A brief description of these programs is presented in Table 3. These programs were run on a SPARCstation 10, and on a Convex C-240 compiling for vector optimization.

We can classify the programs in Table 3 into three groups, depending upon certain characteristics of the MATLAB code and its execution. The first group consists of programs that spend most of their time executing built-in functions (i.e. QL, 3D-Surface, and Cholesky). In this case, the performance improvements are similar to those of the previous test, and are attributed mainly to the overhead of interpretation. We place in the second group the program that requires dynamic allocation of matrices inside of a loop (i.e. Adaptive Quadrature). In this case, for the data set used, we observed an improvement for the serial execution on the order of 20. However this improvement varies considerably, depending upon the number of reallocations that are required by the program, that is in turn dependent upon the function that is being used. Finally, the third group contains loop-based programs that require element-wise access of arrays (i.e. Dirichlet, Crank-Nicholson, Finite Difference, and Inverse Hilbert). In this case, we observed a very good performance improvement (close to 50 on the workstation, and up to 140 on the vector supercomputer). This performance improvement is attributed, in part, to the loop control structure of the compiled code, that is far more efficient than the interpreted code (especially for doubly-nested loops), and to the overhead of the indexed assignments within the interpreted code.

Problem	Source	Problem size
QL method for finding eigenvalues	a	50×50
Generation of a 3D-surface	b	$41 \times 21 \times 11$
Incomplete Cholesky factorization	b	200×200
Adaptive quadrature using Simpson's rule	a	1 Dim. (7)
Dirichlet solution to Laplace's equation	a	26×26
Crank-Nicholson solution to the heat equation	a	321×321
Finite difference solution to the wave equation	a	321×321
Computation of the inverse Hilbert matrix	c	180×180

Source:
a - From [23]; b - Colleagues; c - The MathWorks Inc, as a M-File (*invhilb.m*).

Table 2. List of test programs for indexed variables

	SPARCstation 10			Convex C-240		
Algorithm	MATLAB	F 90	Speedup	MATLAB	F 90	Speedup
QL method	8.8	3.17	2.8	NA	NA	NA
3D-Surface	17.1	2.95	5.8	NA	NA	NA
Inc. Cholesky Fact.	2.1	0.35	6.0	6.3	0.49	12.9
Adaptive Quadrature	1.4	0.07	20.0	4.5	0.06	68.8
Dirichlet Method	20.7	0.61	33.9	71.8	0.99	72.5
Crank-Nicholson	61.7	1.46	42.2	237.7	2.50	95.0
Finite Difference	25.2	0.55	45.8	98.4	0.82	120.0
Inv. Hilbert matrix	5.8	0.12	48.3	21.2	0.15	141.3

Table 3. Execution times (in seconds) for programs using indexed assignments.

4 Conclusions

This system creates an environment that is useful for researchers in computational sciences and engineering for the rapid prototyping and development of numerical programs and libraries for scientific computation, an environment that takes advantage of both the power of interactive array languages and the performance of compiled languages. This environment provides capabilities for interactive and automatic transformations at both the operation-level and the function- or algorithmic-level.

In order to generate code from the interactive array language, this system combines static and runtime inference methods for type, shape, and structural inference. Research is also being performed into the application of high-level optimizations, that take into consideration the semantics of the array language to improve the performance of the target code.

As shown by our preliminary results, by compiling the interpreted language we can generate a code that executes faster than the interpreted code, especially for loop-based programs with element-wise access. Also, we observe that in some cases the performance of the compiled code is very close to the performance of the hand-optimized Fortran 90 program.

Finally, exploitation of parallelism will be enhanced, because the utilization

of a high-level array language (Fortran 90) as output will facilitate the generation of code for a parallel machine by a parallelizing compiler. Moreover, the possibility of adding directives for parallelization, and data dependence assertions, will further improve parallelism.

References

1. Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1985.
2. Amarasinghe, S. P., Anderson, J. M., Lam, M. S., and Lim, A. W. An Overview of a Compiler for Scalable Parallel Machines. In *Languages and Compilers for Parallel Computing* (August 1993), U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., Springer-Verlag, pp. 253–272. 6th International Workshop, Portland, Oregon.
3. Applied Parallel Research. *FORGE Explorer User's Guide*. Placerville, California, 1995. Version 2.0.
4. Backus, J. Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM 21*, 8 (August 1978), 613–641.
5. 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.
6. Blume, W., and Eigenmann, R. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. In *Proceedings of Supercomputing '94* (November 1994), pp. 528–537.
7. Bodin, F., Beckman, P., Gannon, D., Narayana, S., and Yang, S. Distributed pC++: Basic Ideas for an Object Parallel Language. In *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference* (April 1993), pp. 1–24.
8. Budd, T. *An APL Compiler*. Springer-Verlag, 1988.
9. Carr, S., and Kennedy, K. Compiler Blockability of Numerical Algorithms. In *Proceedings, Supercomputing '92* (November 1992), pp. 114–124.
10. Char, B. W., Geddes, K. O., Gonnet, G. H., Leong, B. L., Monagan, M. B., and Watt, S. M. *Maple V Language Reference Manual*. Springer-Verlag, New York, 1991.
11. Ching, W.-M. Program Analysis and Code Generation in an APL/370 Compiler. *IBM Journal of Research and Development 30:6* (November 1986), 594–602.
12. Cook Jr., G. O. ALPAL A Tool for the Development of Large-Scale Simulation Codes. Tech. rep., Lawrence Livermore National Laboratory, August 1988. Technical Report UCID-21482.
13. Cousot, P., and Halbwachs, N. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th Anual ACM Symposium on Principles of Programming Languages* (1978), pp. 84–97.
14. 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.

15. Dai, D. L., Gupta, S. K. S., Kaushik, S. D., Lu, J. H., Singh, R. V., Huang, C.-H., Sadayappan, P., and Johnson, R. W. EXTENT: A Portable Programming Environment for Designing and Implementing High-Performance Block-Recursive Algorithms. In *Proceedings of Supercomputing '94* (November 1994), pp. 49–58.
16. DeRose, L., Gallivan, K., Gallopoulos, E., Marsolf, B., and Padua, D. An Environment for the Rapid Prototyping and Development of Numerical Programs and Libraries for Scientific Computation. In *Proc. of the DAGS'94 Symposium: Parallel Computation and Problem Solving Environments* (Dartmouth College, July 1994), F. Makedon, Ed., pp. 11–25.
17. Gallivan, K., and Marsolf, B. Practical Issues Related to Developing Object–Oriented Numerical Libraries. In *OON-SKI'94 Proceedings of the Second Annual Object-Oriented Numerics Conference* (April 1994), pp. 93–106.
18. Gerlek, M. P., Stoltz, E., and Wolfe, M. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-driven SSA Form. *ACM TOPLAS* (to appear).
19. High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.
20. Houstis, E. N., Rice, J. R., Chrisochoides, N. P., Karathanasis, H. C., Papachiou, P. N., Samartzis, M. K., Vavalis, E. A., Wang, K. Y., and Weerawarana, S. //ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD Machines. In *Proceedings 1990 International Conference on Supercomputing* (1990), pp. 96–107.
21. Kuck and Associates, Inc. *KAP User's Guide*, 4th ed. Savoy, IL 61874, 1987.
22. The Math Works, Inc. *MATLAB, High-Performance Numeric Computation and Visualization Software. User's Guide*, 1992.
23. Mathews, J. H. *Numerical Methods for Mathematics, Science and Engineering*, 2nd ed. Prentice Hall, 1992.
24. Muraoka, Y., and Kuck, D. J. On the Time Required for a Sequence of Matrix Products. *Communications of the ACM 16*, 1 (January 1973), 22–26.
25. Padua, D., Eigenmann, R., Hoeflinger, J., Petersen, P., Tu, P., Weatherford, S., and Faigin, K. Polaris: A New-Generation Parallelizing Compiler for MPP's. Tech. rep., Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, June 1993. CSRD Report No. 1306.
26. Padua, D., and Wolfe, M. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM 29*, 12 (December 1986), 1184–1201.
27. Polychronopoulos, C., Girkar, M., Haghighat, M. R., Lee, C.-L., Leung, B., and Schouten, D. Parafrase-2: A New Generation Parallelizing Compiler. In *Proceedings of 1989 Int'l. Conference on Parallel Processing, St. Charles, IL* (August 1989), vol. II, pp. 39–48.
28. Schwartz, J. T. Automatic Data Structure Choice in a Language of a Very High Level. *Communications of the ACM 18* (1975), 722–728.
29. Tu, P., and Padua, D. Automatic Array Privatization. In *Languages and Compilers for Parallel Computing* (August 1993), U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., Springer-Verlag, pp. 500–521. 6th International Workshop, Portland, Oregon.

This article was processed using the LaTeX macro package with LLNCS style