TECHNIQUES FOR THE INTERACTIVE DEVELOPMENT OF NUMERICAL LINEAR ALGEBRA LIBRARIES FOR SCIENTIFIC COMPUTATION

Bret Andrew Marsolf, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1997
Kyle Gallivan, Advisor

The development of high-performance numerical algorithms and their effective use in application codes is an iterative process involving the refinement of the algorithms and their implementations that continues during the lifetime of the algorithm. Knowledge and expertise from the areas of numerical analysis, computer software, compilers, machine architecture, and applications are required during the development. To improve this process, the FALCON environment was developed to combine the analysis techniques from restructuring compilers with the algebraic techniques from numerical analysis. In this thesis, interactive techniques that were developed to extend the FALCON environment are described. These techniques allow the developer to improve the analysis of the algorithm, to restructure the algorithm using transformation patterns, to utilize additional information about structures within the data, and to control the generation of the target code. The experimental results show that the codes generated by the interactive techniques have better performance than those generated automatically. In addition, the environment was extended to support the generation of C++ code. When the C++ code generated by FALCON is compared to the code generated by other MATLAB translators, the C++ code is typically faster. However, when compared against the Fortran 90 code generated by FALCON, the C++ code is usually slower.

TECHNIQUES FOR THE INTERACTIVE DEVELOPMENT OF NUMERICAL LINEAR ALGEBRA LIBRARIES FOR SCIENTIFIC COMPUTATION

BY

BRET ANDREW MARSOLF

B.S.C.S., Rose-Hulman Institute of Technology, 1987
M.S., University of Illinois at Urbana-Champaign, 1991

THESIS

Submitted in partial fullfilment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

TECHNIQUES FOR THE INTERACTIVE DEVELOPMENT
OF NUMERICAL LINEAR ALGEBRA LIBRARIES
FOR SCIENTIFIC COMPUTATION

Bret Andrew Marsolf, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1997
Kyle Gallivan, Advisor

The development of high-performance numerical algorithms and their effective use in application codes is an iterative process involving the refinement of the algorithms and their implementations that continues during the lifetime of the algorithm. Knowledge and expertise from the areas of numerical analysis, computer software, compilers, machine architecture, and applications are required during the development. To improve this process, the FALCON environment was developed to combine the analysis techniques from restructuring compilers with the algebraic techniques from numerical analysis. In this thesis, interactive techniques that were developed to extend the FALCON environment are described. These techniques allow the developer to improve the analysis of the algorithm, to restructure the algorithm using transformation patterns, to utilize additional information about structures within the data, and to control the generation of the target code. The experimental results show that the codes generated by the interactive techniques have better performance than those generated automatically. In addition, the environment was extended to support the generation of C++ code. When the C++ code generated by FALCON is compared to the code generated by other MATLAB translators, the C++ code is typically faster. However, when compared against the Fortran 90 code generated by FALCON, the C++ code is usually slower.

To Kathy.

Acknowledgments

I am indebted to many people who have helped me over the years that I have spent here working on my degree. First, and foremost, I would like to thank my wife for the love and support that she has given me over these trying years. She has spent many an evening alone while I have gone back to work. Without her help, I never would have been able to finish.

Next, I would like to express my gratitude for my advisor, Dr. Kyle Gallivan, who recruited me to come work with him before I had even arrived at the university. Over the long years we have had many long and interesting conversations, during these I would have to answer questions not only about what was happening, but also about why it was happening. In answering the questions of why things happened, I learned far more about computing than I could have ever hoped to learn in class.

I would also like to thank the other members of my preliminary and final exam committees, Professors Padua, Polychronopoulos, Saied, and Saylor. They provided many thoughtful comments regarding my work and its potential direction. In addition, I would like to thank Professor Gallopoulos for his comments on my research.

I would to thank my friend and officemate Luiz De Rose for the help and encouragement he has given me over the years. We have helped each other through many trials and tribulations along the way towards our degrees. And, with the FALCON project, we were both able to succeed in this goal.

During the years that I have worked on my degree at CSRD, there have been many other students who have helped me along the way. In particular, I would like to thank Pete Shirley, George Frank, Brian Bliss, and Jose Moreira for their friendship.

I am grateful to all the support staff who have been part of CSRD over the past years. I would especially like to thank Donna Guzy and Sheila Clark for their help.

I would also like to thank my family. I am especially grateful to my parents for their support over these long years. I would also like to thank my brothers, who pestered me for years with the question "When will you be done?" And finally, I would like to say thanks to the rest of the family, both in-laws and out-laws, who have made it possible for Kathy and I get through these years.

This work was supported in part 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.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

Scientific applications rely upon algorithms provided by numerical libraries to effectively utilize high-performance computers; however, the tasks required to provide an application developer with efficient algorithms can be very complex. The development of these algorithms and their effective use in application codes is an iterative process involving the refinement of the algorithms and their implementations, and the tuning of parameters to match the machine architecture and the application context. This process continues during the lifetime of the algorithm, as the algorithm and its implementation are updated. Furthermore, when the algorithm is ported between machines, it may be necessary to modify the algorithm, to select different computational primitives, or to tune the parameters in order to optimize the performance. The development and implementation of an algorithm, therefore, requires a combination of knowledge and expertise from the areas of numerical analysis, computer software, compilers, machine architecture, and applications. The building of an environment which can help the algorithm designer utilize this knowledge can both speed the development process for libraries and improve their performance within application codes.

Most library algorithms are distributed to application users in one of two forms, either as an algorithm in a paper or as a library subroutine, each with its own disadvantages. An algorithm must be transformed, by hand, into code for a target machine. This is a tedious and time-consuming process which includes making numerous implementation decisions based upon the target machine's architecture, and the problem being solved, in order to achieve the best performance. This process must be repeated, by hand, for every algorithm that will be used on the machine. A library, on the other hand, has been encoded for a machine, or class of machines, with the implementation decisions already made. However, the library may not have been designed for the specific problem or machine being used, resulting in poor performance, or the data structures preferred for the application may not be supported by the library. The lack of good, portable, high-performance libraries using the best algorithms can make applications difficult to program, hard to port between machines, and decreases the performance of the code. The development of tools to help with the tasks required to develop, tune, and maintain algorithms can, therefore, both decrease the development time for

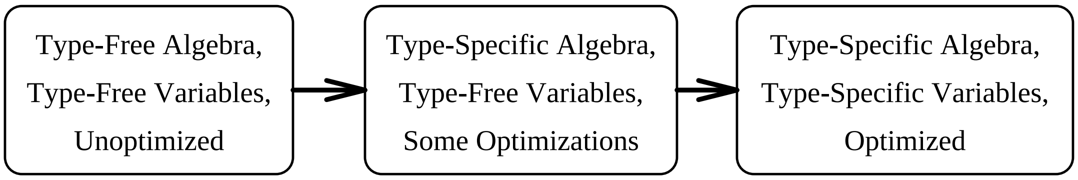

Figure 1.1: The stages of algorithm evolution during development.

the algorithms and improve the performance of the applications.

1.1 The Development Process

Regardless of the method, the development of high-performance programs can be a tedious and time consuming process involving the transformation of an algebraic algorithm into high-level code or code description. Multiple iterations of this process are usually required to refine the algorithms, data structures, and code, in order to generate high-performance software for a specific machine. This process may involve multiple people, from the algorithm developer to the application developer, performing these transformations by hand or with the help of tools.

As previously stated, the process of code generation involves taking an algebraic algorithm, encoding it in a target language, and then refining the algorithm, data structures, and the code. During this process, changes can be made to either representation, the algorithm or the code, with additional transformations sometimes being required to reflect the changes in the other representation. The amount of work required to transform the algorithm into the target language depends upon how close the operations in the algorithm map onto the target language. If there is a one-to-one mapping the encoding can proceed fairly quickly. However, the more complex the mapping between the algorithm and the target language, the slower the transformation.

The different phases an algorithm goes through during its development can be grouped into stages based on the representation being used and on the level of detail used for the implementation. As more details are added, the algorithm moves between the stages, as shown in Figure 1.1. During each stage the algorithm can be represented with an algebraic form and, at any point, it is possible to translate this algebraic form into a code form utilizing a traditional programming language. The code forms of the algorithm can also be grouped into stages that correspond to those presented in Figure 1.1.

The matrix structures used for the algorithms also evolve during the development process, with the algorithms first being developed using dense matrices. Next, the dense matrices are replaced with their structured dense equivalents. (The more common forms of structured dense matrices include upper and lower triangular matrices, diagonal matrices, band matrices, and Toeplitz matrices.) The next stage is to convert the algorithms from their structured dense matrix forms to forms utilizing general sparse matrices. A variety of data structures exist for representing sparse matrices, with a few of the more common methods being compressed by row, compressed by column, and the triplet form [DER86]. As these different types of matrix structures can be combined with each of the multiple algorithm representations just described, it becomes apparent that an algorithm may be represented in many different forms during its life-cycle.

During the process of incorporating an algorithm into an application, the development work on the algorithm can occur at three different levels of abstraction: the algorithm level, the subroutine level, and the application level. At the algorithm level, the algorithm developer refines the numerical properties of the algorithm and evaluates the performance bounds of the algorithm. Next, at the subroutine level, both the library developer and the library maintainer may be responsible for the algorithm. At this level, the mapping of the algorithm to the target language is refined by the library developer to improve performance for a specific target machine. This can be done both by changing the primitives used to implement the operations and by changing the data structures being used. The library maintainer, on the other hand, may be mapping the code from one machine to another machine without modifying the algorithm. At the application level, the application developer combines multiple algorithms into an application, attempting to improve the overall performance of the application through the selection of the appropriate algorithms and subroutines.

The different development tasks may be performed by different people spread over multiple groups working disjointly or, at the other extreme, they may all be performed by the same person. However, based upon the role, or roles, performed, the focus of the developer changes. For instance, the algorithm developer is more concerned with the numerical properties of the algorithm and the algebraic operations performed while the library developer is more concerned about the performance of the primitives used and about the selection of the data structures. Regardless of the focus of the individual developer, the main issues of performance and numerical correctness are a concern for all the developers.

1.2 Approaches to Algorithm Development

Several approaches are used to improve both the performance and the development process of algorithms and libraries, ranging from application-specific environments to restructuring compilers. These approaches, however, tend to focus on only one aspect of a library or its development and do not address all the issues involved with supporting an algorithm over its life-cycle.

One approach used to improve the development process is to provide the developer with a library of lower-level numerical operations; allowing the higher-level algorithms to be developed as a sequence of calls to this library. For instance, libraries such as LINPACK[DMBS79] and LAPACK[Dem89, ABD+90] were developed using the basic numerical operations in the BLAS libraries[DCHD90, DCHH88, LHKK79]. The extension of this idea to parallel machines has resulted in the development of both communication libraries, such as those based upon the MPI [Wal93] standard, and the development of numerical libraries such as PB-BLAS [CDW94]. This approach both speeds the implementation of an algorithm and can be used to generate a portable version that will run on any machine where the supporting library has been ported. However, this approach places several restrictions on the higher-level algorithm. First, only the primitives and data structures used by the lower-level library are available for the developer to use with the subroutines. Second, the granularity of the blocking or parallelism is mandated by the available subroutines, which may lead to poor performance if either the algorithm, the application, or the architecture does not map well with the library.

Problem solving environments provide the developer with additional tools for algorithm development. Along with libraries of numerical operations, these environments include graphical user interfaces; tools for loading, saving, and displaying data; and programming languages targeted to a specific application area. For instance, MATLAB[1] [The92] was designed for numerical linear algebra and //ELLPACK [HRC+90] was designed for differential equations. While making it easier to develop algorithms within the environment, these systems typically provide little help for transferring the algorithm to code outside of the environment and the numerical performance of the algorithm within the environment can be lacking.

Once the algorithm has been translated to the target language, restructuring compilers can be used to improve the performance of the code. For sequential machines, the developer will use the compiler provided by the vendor to optimize the code for specific target processor. To improve the performance on parallel machines, compiler research projects such as Polaris [PEH+93] and Parafrase-2 [PGH+89] have been developed to search for parallelism within sequential codes. While these compilers can improve the performance of the code on a specific machine through the examination of the numerical operations being performed, aggressive modifications to either the overall algorithm or to the library routines being used are usually impossible due to the compiler's lack of understanding of the essential algorithmic intent.

Another approach for improving algorithm performance has been the development of new languages that have been designed to exploit the architecture of a specific machine or class of machines. For example, CM Fortran [Thi91] was designed to utilize the data parallel capabilities of the CM-2 and CM-5. Similarly, Cedar Fortran [Hoe91] was designed to utilize the clusters of the Cedar architecture. More generic languages have also been developed such as HPF [Hig93] and pC++ [BBG+] which do not target a specific machine, but instead target a class of parallel machines. These types of languages make it easier to utilize the capabilities of the architecture, thereby achieving better performance, but they typically do not provide any additional understanding of the algorithms

[1]MATLAB is a trademark of The MathWorks, Inc.

being used or of the application being developed.

1.3 A Development Environment

Whereas the development approaches previously described tend to focus on one aspect of algorithm development, to effectively improve the development of high-performance libraries it is important to address the needs of the various developers with respect to the entire life-cycle of the algorithms. During the early stages of algorithm development, the developer wants to quickly make algebraic changes and evaluate their effects, i.e. rapid prototyping, and is not as concerned with implementation details. At this point, the developer tolerates the slower performance of the algorithm that results from using an interpreted environment or from the simple translation of the algorithm into code. However, as the development proceeds and the algorithm is refined, the developer wants more control over the implementation decisions. Now the developer wants to be able to select the data structures, algebraic primitives, and target subroutines. As new refinements are added to the algorithm and the algorithm is used on different machines, the developer needs to be able to reevaluate the decisions that were made previously and possibly make changes to both the algebraic algorithm and to the implementation.

To meet the changing needs of the algorithm developer, we have combined aspects of the development approaches described in the previous section to create FALCON [DGG+95], a development environment for numerical linear algebra algorithms. This system combines software techniques from the research in restructuring compilers with the algebraic techniques from numerical analysis to generate high-performance codes from algorithms that are described using the interactive array language from MATLAB. First, the environment analyzes the MATLAB algorithm and translates it into an intermediate form. While the algorithm is in this intermediate form, it can be restructured either automatically or interactively. Once the restructuring is complete, the intermediate form is then translated into a high-level language.

The choice of MATLAB for the input language was based on both technical and practical reasons. On the technical side, MATLAB's interactive environment and the *typeless* nature of its language, along with its wealth of numerical functionality, make it good candidate for prototyping algorithms. On the practical side, MATLAB has a significant user community and it is available on a range of machine platforms, from personal computers to parallel servers. By selecting MATLAB, a potential user community for FALCON already exists and numerical algorithms to be used for testing the system are freely available.

The MATLAB algorithm is translated into the intermediate form using compiler techniques developed by DeRose [DeR96]. While the typeless nature of the MATLAB language facilitates the prototyping of algorithms, it greatly hinders the translation into traditional programming languages that require variable declarations. DeRose's techniques analyze the algorithm and attempt to statically resolve the type, rank, and size of the variables. When the techniques fail, code must be inserted to complete the analysis at run time.

After the analysis, while the code is still in the intermediate form, both compiler and algebraic techniques can be applied to transform the algorithm, as well as to improve the analysis that was performed on the algorithm. These transformations can be performed either automatically or interactively.

After transformation, the intermediate form is translated into one of three output languages: Fortran 90, C++, and MATLAB. Fortran 90 was a natural choice as Fortran has traditionally been used for numerical programming and, with Fortran 90 supporting matrix operations and dynamic memory allocation, there is a fairly straight-forward mapping of MATLAB expressions to Fortran 90 expressions. C++ was chosen as an output language both because of its growing popularity, especially in the field of numerical analysis, and because of the flexibility that arises from its use of classes for defining the matrix operations. The third language choice, MATLAB, was chosen both because there are optimizations that can be made to improve the MATLAB code and because it allows the transformed code to be saved for later use in the environment.

1.4 Problem Overview and Thesis Outline

This thesis addresses the issues that arise during the development of numerical linear algebra algorithms. Its primary focus is on the restructuring of the algorithms within the development environment and it can be done interactively, providing the algorithm developer with control over both the algebraic form of the algorithm and the code generated. It also covers the refinement of the algorithm analysis, the addition of user-defined states to the algorithm to improve the code generation or code performance, and the addition of C++ as an output language.

Two main questions are addressed regarding the restructuring of the numerical algorithms. First, the current techniques used by restructuring compilers are examined to determine if these techniques can be extended to utilize additional algebraic information that may be known about the algorithm or the data. Second, the techniques which numerical analysts typically apply by hand are investigated to determine if they can be applied automatically, or semi-automatically, within a development environment. A secondary issue that arises within both of these questions is how to describe the algebraic information and the transformations such that the system can be modified and/or extended by the algorithm developer.

In turn, these two questions raise additional issues within the development environment. Can the restructuring system be used to refine the variable analysis within the program analysis phase and, conversely, what additional information can be utilized to refine the variable analysis to enable restructuring? Also, whereas the initial restructuring system was based on variable states defined within the environment, is it possible for the algorithm developer to define additional states to control the application of the restructuring transformations or the generation of specialized code?

The final issue to resolve is the addition of C++ as an output language to the development environment. In contrast to Fortran 90, C++ provides less direct support for matrix operations which are provided through a matrix class. While the use of a matrix class and overloaded operations can sometimes allow an algorithm to be expressed using fewer statements, the overhead of these statements can reduce the performance. With the generation of the C++ code, therefore, the goal is to determine a method for generating the code with fewer statements than Fortran 90, yet with performance close to Fortran 90.

Other research related to this work is discussed in Chapter 2 with an overview of the FALCON environment presented in Chapter 3. FALCON's approach to interactive program development is discussed in Chapter 4 including the description of the transformation language. Both traditional compiler transformations, in Chapter 5, and algebraic transformations, in Chapter 6, implemented in the transformation language are examined. A technique that allows the developer to define additional variable states which can be utilized by the transformation system is presented in Chapter 7 while further modifications to this technique which allow transformations of the variable representation are in Chapter 8. Additional techniques to improve the generated code are examined in Chapter 9; followed by a discussion of C++ code generation in Chapter 10. An idea for extending the system to support the parallel sparse codes is given in Chapter 11 with other areas of future work discussed in Chapter 12. Finally, the conclusions are presented in Chapter 13.

Chapter 2

RELATED WORK

As mentioned in the previous chapter, several approaches have been taken for the development and distribution of algorithms for high-performance computers. These include restructuring compilers, the development of architecture specific and parallel languages, and the development of numerical libraries for both sequential and parallel machines. In addition to these techniques, other techniques include the development of object-oriented matrix classes, both general-purpose and application-specific, and the use of algorithm descriptions. Further, interactive programming environments are being used for writing algorithms and applications and, in some cases, systems have been developed for the compilation of the algorithms. Each of these techniques will be examined in turn to see both what area of algorithm development they address and how the work relates to that being done for algorithm development within the FALCON environment.

2.1 Restructuring Compilers

Restructuring compilers are able to optimize the numerical operations that are present in the code for both scalar and array operations. For scalar operations, algebraic transformations described in [ASU86] include algebraic simplification by recognizing operations with one and zero, reduction in strength, as well as recognizing associative and commutative operations. For matrix operations, optimization research has included how to improve the evaluation of matrix products by examining the expression tree [MK73] and, more recently, the optimization of array expressions when the matrix is distributed across a data parallel architecture [GS91].

Most of these optimizations, however, are based on recognizing a single expression or recognizing a scalar expression contained within nested loops, where typically the loop corresponds to a matrix expression or a reduction operation [Hag94, PE95]. As a result, more aggressive algorithmic optimizations are usually impossible due to the lack of understanding of the essential algorithmic intent. This is, of course, due to the inevitable loss of key information inherent when expressing the algorithm in a particular programming language. Carr and Kennedy have worked on compiler optimizations to automatically block numerical algorithms for dense linear algebra on machines with memory hierarchies [CK92]. Even for this more specific restructuring task, they have encountered the fundamental difficulty of many high performance algorithms: the best block algorithms cannot always be derived solely based on the operations and computational primitives which are present in the code. The development of an algorithm to compute a QR factorization of a matrix based on block Householder reflectors is a typical example. In this case, the most effective combination of the reflectors involves additional operations that are not present in the original algorithm. Recently, Gallivan et al. have pointed out that it is often possible to move easily from standard algorithms with limited parallelism to highly parallel algorithms by using "higher level transformations" that exploit algebraic knowledge of the operations [GPS90]. It is this merger of high level transformations, algebraic knowledge, and more traditional compiler strategies applied to an algebraically expressive language that is to be exploited by the restructuring techniques within the FALCON environment.

2.2 Languages

Instead of developing tools to improve the performance of existing languages, an alternative approach is to develop new languages that have been designed to utilize features of the machine architecture, or class of machines, or to utilize information about a specific application domain. As was mentioned in the previous chapter, CM Fortran was designed to utilize the data parallel capabilities of the CM-2 and CM-5 and Cedar Fortran was designed to utilize the clusters of the Cedar architecture. More generic languages have also been developed, such as HPF and pC++, to target data parallel machines. These languages are very good at mapping dense arrays onto the machines for regular computations. However, their performance is typically much worse when dealing with sparse matrices or irregular computations. As a result, these languages provide little flexibility to the algorithm developer, with many of the implementation decisions having been decided by the selection of the language. By supporting multiple target languages, the FALCON environment attempts to provide the algorithm developer with more flexibility.

2.3 Numerical Libraries

Numerical libraries are used during algorithm development to provide *lower-level* operations to speed the development of the algorithm, to provide better performance, and to provide better portability. The libraries provide a wide range of functionality including: support for linear algebra on both sparse and dense matrices, parallelism, and high level numerical languages, with almost all of the functions being available in more than one package. As a result, the algorithm developer has the additional responsibility of deciding which is the best package to use before beginning the programming. The numerical libraries can be broken into three categories based upon the linear algebra operations provided: basic operations, general purpose, and sparse solvers.

The basic operation libraries provide the lower level arithmetic operations for dealing with matrices and vectors. Typical operations include dot-products, vector triads, matrix-vector multiplication, rank-k updates, and triangular solves. These libraries target both sparse and dense data structures, with a few examples being the Level 1 BLAS [LHKK79], the Level 2 BLAS [DCHH88], and the Level 3 BLAS [DCHD90].

The second set of linear algebra libraries are the general purpose libraries, which perform operations more complex than the arithmetic operations in the basic libraries. The operations performed include solving sets of linear equations, finding eigenvalues and eigenvectors, solving least square problems, and solving ordinary differential equations (ODE's). Most of these libraries operate on dense matrices, with LINPACK [DMBS79], EISPACK [SBD+76], and LAPACK [Dem89, ABD+90] being three of the most frequently used.

The third set of libraries are those that solve linear systems represented by sparse matrices. These are separated from the general purpose libraries because the algorithms for sparse matrices are typically more complex than the corresponding algorithms for dense matrices. This complexity comes from the use of sparse data structures and the additional data operations which sparse algorithms perform in order to reduce the number of arithmetic operations.

A matrix may be represented with multiple sparse matrix data structures, each with its own advantages and disadvantages. Depending upon the algorithm being used, a different data structure may be desired. In addition, the individual sparse matrix may lend itself to a certain type of data structure due to the location of the nonzero elements in the matrix. Representative sparse solvers are SPARSPAK [GL79, GL81], MA28 [DR79, DER86], and Y12M [ZWS81].

The traditional method of interfacing with a library is by placing subroutine calls within the application program and linking the application to the library module during the compilation process. With this method, the user is responsible for determining the correct subroutine name and the correct type and value for each of the subroutine's arguments, typically from the documentation for the library. The subroutines and data types supported by this type of library are static, only the data types and structures provided by the library designer can be used.

One difficulty with this method is that subroutines performing similar functions on different data types require different subroutine names (or entry points). As a result, the number of subroutines within the library grows not only with the number of functions supported but also by the number of data types and structures supported. This situation is usually improved by using a naming convention that differentiates the data types and structures supported by the subroutines. With the FALCON environment, the system can automatically perform this mapping from the algebraic operations to the subroutine calls, thereby speeding the development process. Furthermore, by being able to support multiple mappings, the developer can use FALCON to experiment with several libraries without needing to transform all the subroutines calls by hand.

2.4 Object-Oriented Matrix Classes

Matrix operations are provided in C++ through the use of matrix classes, that either implement the numerical operations directly or provide an interface to libraries such as those just described. These matrix classes can be divided into three different sets: general purpose matrix classes, specialized matrix classes, and those which support parallel systems.

The first set of classes are the general purpose matrix classes. These classes all support rectangular matrices with some of the classes supporting additional matrix structures. While the basic arithmetic operations for matrices are provided by all of these classes, there are differences in the more complex operations which are supported. A few of the matrix classes which are available include newmat07 [Dav94], MatClass [Bir93], and Matrix [TD94].

The second set of matrix classes are those that were defined for a specific problem type or interface. For instance, AMR++ [BLQ92] was designed to support adaptive mesh refinement and both LAPACK++ [DPW93a] and Lapack.h++ [Ver93] were designed to provide the functionality of the LAPACK library. These classes do not provide all of the functionalities of the general purpose matrix classes, such as support for multi-dimensional arrays or functions to compute trigonometric operations. They are used, however, for the special operations they do provide, such as matrix factorizations.

The third set of matrix classes are those which are defined for parallel systems, with most of the classes that are available designed to run on message passing parallel systems. This set includes matrix classes that are designed for both general and specific problem types, such as P++ [LQ93, BLQ92], ScaLAPACK++ [DPW93b], and OOSkit [Yan93].

While providing functionality that is similar to traditional subroutine calls, these object-oriented classes support overloading of subroutines and operators thereby simplifying the use of the library. Overloading allows multiple subroutines to have the same name but different argument types. The system will determine the correct subroutine to call by matching the argument types for the given call with the argument types for the available calls. In this way, the number of subroutine names required for the library is reduced, since all the subroutines supporting the same function on different data types can use the same name. Overloading is also supported for the arithmetic operators; allowing operators such as "+" and "–" to be used by data types defined in the class, in addition to those normally supported by the language. The method names in one matrix class, however, are not guaranteed to match the method names in a second matrix class. As a result, the code must be modified to utilize a different matrix class. By using a standardized input language, the FALCON environment attempts to maintain the portability of the algorithms and while providing the developer with the tools necessary to map the algorithm to multiple target languages.

Given the range of functions provided by the matrix classes, as well as the variations in the performance, it is difficult to determine the best matrix class to be used. Furthermore, when switching between different machines, the preferred class may not be available or the available classes may not contain the necessary functionality. Here, the long-term goal of the FALCON environment is to provide the developer with the tools to target multiple matrix classes so that such problems can be mitigated.

2.5 Algorithm Descriptions

In order to give application programmers more flexibility with the implementation of an algorithm, another approach is to distribute the algorithm description instead of distributing the final implementation. Two versions of this approach include algorithm templates and data-structure independent code.

2.5.1 Algorithm Templates

The approach in [BBC+93] is to distribute only the algebraic algorithms (called templates), instead of code, and allow the implementor to make the required optimizations. While this approach keeps the algorithms portable, it requires the developer to perform the transformation from algorithm to code by hand and, once transformed, the algorithm may no longer be portable. This approach forces the user to deal with all of the implementation decisions while allowing the numerical algorithm designer to ignore most architectural details. Clearly, this approach simplifies one researcher's design task by complicating someone else's. Within the FALCON environment, the tools are provided for performing the transformation from algorithm to implementation, while the developer is given discretion over the level of input to be provided.

2.5.2 Data-Structure Independent Code

The approach in [GS93] has concentrated on standardizing the interface to the library's algorithms and data structures. This allows both algorithms and data structures to be interchanged with almost no changes to the calling program. While this approach can help the developer of application programs try different algorithms, it constrains the developer to limited set of algorithms and data structures. With this approach it is the responsibility of the library designer to incorporate new algorithms or data structures into the system. Furthermore, the new algorithms and data structures must be written to fit predefined standard data access patterns. Again, the FALCON environment attempts to provide the developer with more flexibility and the capabilities for incorporating new features into the environment as they become available.

2.6 Programming Environments

Programming environments provide the developer with additional tools that support algorithm development. Not only do these systems provide better interfaces to the numerical operations, they also provide graphical tools for examining the results. These environments can be divided into two groups, those that support general programming and those that support parallel programming.

For instance, general programming environments such as MATLAB [The92] and Octave [Eat93] provide interactive environments for developing and solving linear algebra problems. A visual programming environment is provided by the Khoros [EH93] system, which has been combined with the ITPACK library of iterative methods for solving linear equations.

The second set of programming environments are designed for parallel programming. For example, CONLAB [JKR92] provides an interactive environment for developing matrix algorithms for distributed memory multicomputers and XENOOPS [JBV93] provides support for implementing parallel numerical applications in C++. Other examples include ESP [LFG93] which provides an environment that supports concurrent execution of object-oriented programs and MetaMP [Ott91, Ott93] which provides support for parallel object-oriented programming on message passing systems.

Typically, the interaction of the user with the environment is done in terminology appropriate to the application area rather than in terms of the constructs from traditional programming languages. Though substantial attention is usually paid to aiding a user in the preparation of the problem specification, the user is typically given little control over the underlying operations that will be used to solve the problem. The aim of these environments is to solve problems using high-level algorithms or problem descriptions, not to develop code.

2.7 Compilation of the MATLAB Language

One difficulty with using a programming environment like MATLAB is that the flexibility within the environment is provided by using an interpreter to execute the commands. The overhead incurred by the interpreter for loop control structures, the element-wise access of arrays, and dynamic memory allocation can have a significant impact on the performance of the algorithm. On the other hand, array-level operations are typically implemented using high-performance subroutines allowing codes consisting of mainly array operations to run efficiently.

In order to overcome the performance cost of the interpreter, one option is to translate the algorithm into a language that can be compiled. This is the approach that is being taken by the FALCON environment and is also being considered by others. The developers of MATLAB, The MathWorks, Inc., have developed a compiler, MCC [The95], that translates MATLAB algorithms into C programs. Another project, MATCOM [Yar96], translates MATLAB algorithms into C++ programs. The goal of both of these projects is the translation of the MATLAB code into an equivalent C or C++ program. With the FALCON environment, however, the goal is to be able to produce multiple equivalent programs, based upon transformations and optimizations selected by the algorithm developer.

The MCC compiler performs some variable analysis in an attempt to resolve the type and rank of variables, similar to the analysis performed by FALCON. It does allow the user to add variable properties through assertions inserted into the program and by command line arguments. The additional variable state can be used to indicate the type and rank of variables and to indicate whether the size of a matrix can change during the execution. If the type of any input variable within the program is unknown, the MCC compiler will generate two versions of the code. One version assumes all of the input variables are real and the other version assumes all of the variables are complex. If any of the input variables are complex when the algorithm is called, then the complex version is used with any real inputs variables being converted to complex. This simplifies the code generation, as the tests of the input variable types are all performed at the entry point to the code. However, as real variables may be promoted to complex types, the execution may be slowed as some complex operations require additional floating point operations to be performed. A new version of the MCC compiler that translates M-files into C++ code that can be used to build stand-alone application was also recently announced [The96].

In its default mode, the MATCOM compiler treats all variables as matrices and uses run-time tests to determine the actual rank and type of the variables. Alternatively, a command line argument can be used with the compiler to indicate that all variables within the program are scalar values. The matrix operations are implemented using a matrix class that contains run-time tests to determine the correct operation to be performed. Effectively, the matrix class replicates the tests that were performed by the MATLAB interpreter.

Though allowing the developer to provide assertions, both of these systems are designed to perform the translation automatically. On the other hand, the FALCON environment can provide the developer with interactive control over the translation process. Later, in Chapter 10, experiments are presented which compare the effectiveness of these different translation approaches.

Other related projects have used subsets of the MATLAB language or algebraic languages. For instance, CONLAB uses a subset of the MATLAB language, with extensions for expressing parallelism, synchronization, and communication. A compiler from CONLAB to C was developed by Drakenberg et al. [DJK]. A type inference system is alluded to by the authors in their papers, but it is not described. Also, some simplifications and modifications have been made to the source language to allow efficient C code to be produced, including the exclusion of all primitives for synchronization and communication, except for message passing.

Alternatively, the ObjectMath system [VHF92] combines Mathematica [Wol91] with object-oriented modeling for generating code. ObjectMath was designed to translate equation-based models to executable in either C++ or Fortran, applying transformations and simplifications to the formulae to generate a representation that can be solved efficiently. As compared to the other systems which focus on manipulating the code, this system is focused on the manipulation of the equations. A similar project is the Sinapse synthesis system [Kan93] that also generates mathematical-modeling software from the equation that define the system using Mathematica.

Chapter 3

OVERVIEW OF THE FALCON ENVIRONMENT

Before describing the algorithmic development techniques, it is first necessary to understand the environment within which they will be used. Chapter 1 contained a brief discussion of the FALCON environment, focusing mainly on the input and output languages. In this chapter, the basic systems contained within the environment, as well as some of the algorithms used for program analysis, are described.

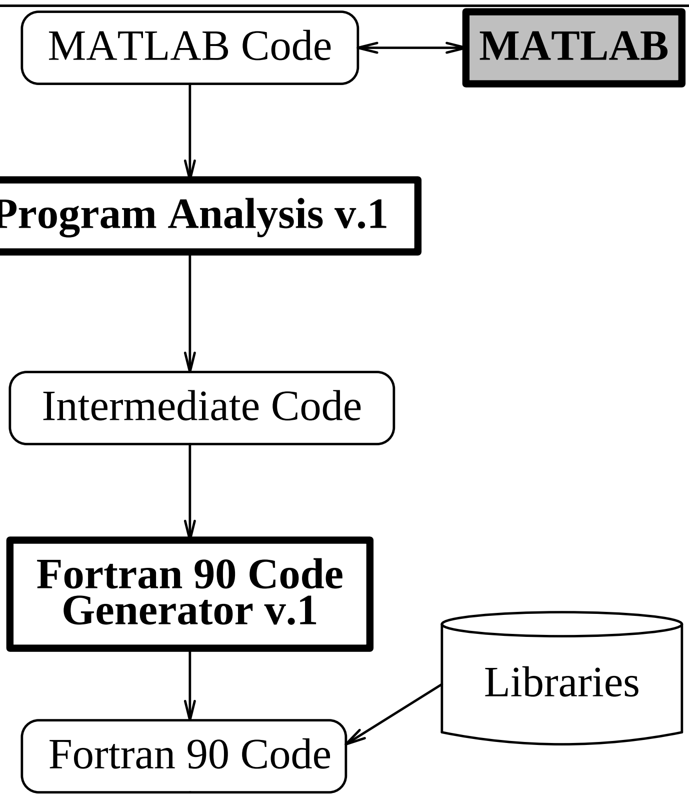

Figure 3.1: Overview of the first phase of the FALCON environment.

The development of the FALCON environment has proceeded through two phases. During the first phase, the research effort was focused on developing the automatic translation techniques for converting a MATLAB program into a Fortran 90 program. This version of FALCON consisted of two main components, the Program Analysis System and the Fortran 90 Code Generator, as shown in Figure 3.1.

The process starts with the algorithm in the form of a MATLAB program. The Program Analysis System, developed by DeRose [DeR96], analyzes the program in order to infer the variable properties, such as the type and rank. Whenever this information cannot be determined statically, the Program Analysis System generates code to make the determination at run time, similar to the operations performed by the interpreter in the MATLAB environment. Hence, after this transformation, the type and shape of each variable in the program are known. When finished, the Program Analysis System outputs the program in an intermediate form.

The target code is generated from the intermediate form using the Fortran 90 Code Generator. The generated code utilizes both the numerical capabilities of the Fortran 90 language and calls to numerical libraries, including public-domain libraries and libraries developed for the FALCON environment.

The second phase of development, which is the work described in this thesis, was the enhancement of the FALCON environment to support interactive restructuring and to provide additional code generation capabilities. The current FALCON environment consists of three main components, namely the Program Analysis System, the Interactive Restructuring System, and the Code Generation System which now contains three code generators, one for each of the output languages. Figure 3.2 presents an overview of the current FALCON environment and contains the flow of information between the main system components.

The translation process still starts with the algorithm in the form of a MATLAB program. However, the Program Analysis System was modified to support the additional capabilities of the current system. When finished, the Program Analysis System outputs the program in an intermediate form.

After the intermediate form has been generated, additional transformations can be performed on the code using the Interactive Restructuring System. Making use of a collection of transformation rules and the information gathered by the program analysis, the user can interactively transform the original program or, in some cases, the transformations can be applied automatically. It is within the Interactive Restructuring System that the techniques for restructuring the algorithms were developed. As shown in Figure 3.2, 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 or it might be possible to refine the analysis as additional information may be available. The restructuring techniques operate on the intermediate form, with the result of the transformations

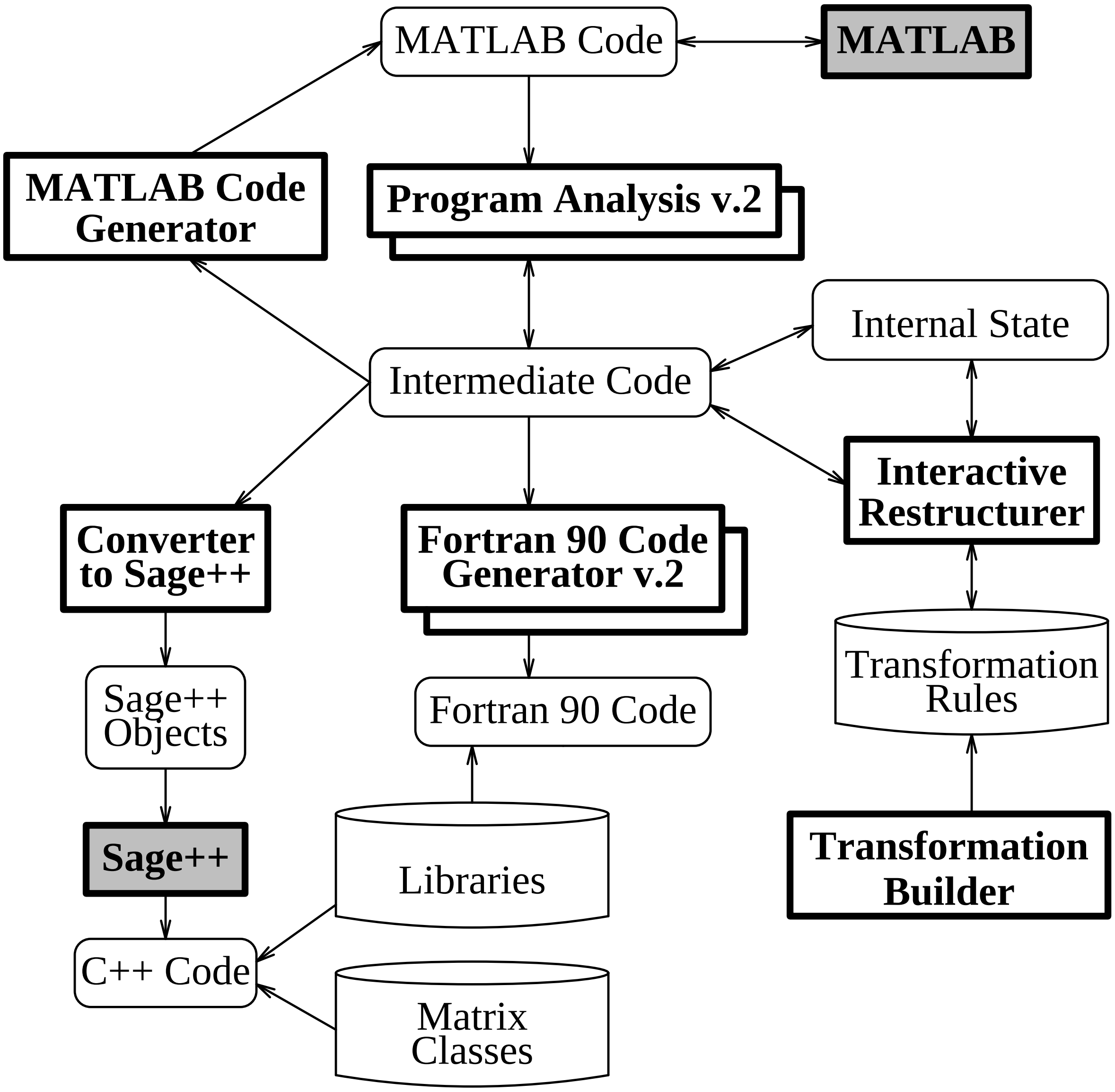

Figure 3.2: Overview of the current FALCON environment.

also being the intermediate form.

The target code is generated from the intermediate form using the code generator that has been selected. During this phase not only were the C++ and MATLAB code generators added to the FALCON environment, but the Fortran 90 code was also modified. Currently, the code generators only output sequential code. Parallel code could be created by passing the sequential code through a restructuring compiler, such as Polaris [PEH+93] or KAP [Kuc87], with the environment facilitating the translation into parallel form by using array operations whenever possible.

Having presented an overview of how the systems within the environment interact, the Program Analysis and Code Generation Systems will be discussed further in the remainder of this chapter, while discussion of the Interactive Restructuring System will follow in Chapter 4. In addition to describing the systems, an additional goal of this discussion is to differentiate the previous work that was done within the environment by DeRose from the new techniques that have been developed.

3.1 The Program Analysis System

The Program Analysis System reads in the MATLAB program and converts it into an abstract syntax tree (AST), which is the intermediate form for the environment. Before the AST is generated, all function calls to M-files, subroutines written in MATLAB's array language, 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. Then, as the AST is being generated, statements within the code are also transformed into three-address code, called *triplets*. Every expression is represented as a sequence of one or more statements of the form

$$\mathit{result} = \mathit{operand}_1 \ \mathit{operation} \ \mathit{operand}_2,$$

with temporary variables being inserted as necessary to store the results. By using triplets, the number of combinations of variable properties within a statement can be controlled, avoiding possible exponential growth.

During the transformation to the AST, the system analyzes the program to determine *variable properties*, such as type, rank (e.g vector, square matrix, scalar, etc.), and shape (i.e. the size of each dimension). Some work for recognizing structures (e.g. upper triangular, diagonal, Toeplitz) was begun, but never completed.

This work built upon type inference techniques developed for SETL [Sch75] and APL [Bud88, Chi86]. The inference techniques were also extended with techniques originally developed to analyze array accesses within Fortran loops [TP93]. The analysis utilizes a *high-level summary* of the behavior of the built-in MATLAB functions for the various combinations of input variable properties. This high-level information increases the accuracy of the analysis over the approach used in conventional high-level language compilers, where only the information from elementary operations is used.

3.1.1 Use-Definition Coverage

The MATLAB program is internally represented in Static Single Assignment (SSA) form [CFR+91], whereby a new version of the variable is used for each assignment statement. This is a convenient representation for implementing many of the analysis algorithms that are used within the environment. In the SSA representation, it is evident which definitions affect (or cover) a particular use of a scalar variable. Using this information, it is simple to perform variable renaming and privatization.

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

For local transformations, such as those within an expression, the use of the SSA has little effect. However, when trying to perform restructuring across a series of statements, the use of the SSA form can present difficulties if two assignments are made to the same variable. For instance, consider the following code where two assignments are made to the variable A:

$$A = B * C \tag{3.1}$$

$$A = A * D \tag{3.2}$$

In the program, both assignments are to the same variable. When using the SSA form, however, the two assignments are to different variables:

$$A_1 = B * C \tag{3.3}$$

$$A_2 = A_1 * D \tag{3.4}$$

In order to simplify the system, once the Program Analysis phase has resolved the variable properties, a *clean-up* algorithm is used to combine the versions of a variable that have the same properties. The versions of the variable remaining after the clean-up, now known to have different properties, are treated as different variables by the restructuring system.

3.1.2 Variable Property Inference

To generate variable declarations and to support data structure selection, the system infers variable properties using a forward/backward scheme [ASU86]. Although described separately here, the inference of the shape of variables works in coordination with the coverage analysis discussed in the previous subsection. The variable properties being determined by the inference algorithms are:

Type – The intrinsic type of the variable, with the possible choices being: COMPLEX, REAL, INTEGER, LOGICAL, or STRING.

Rank – The number of dimensions of the variable, with the possible choices being: SCALAR, VECTOR, or MATRIX.

Shape – The size of each dimension of the variable.

Before the inference is applied, the program is first converted into the AST without the variable information. The inference is then performed on the AST in two phases; first the static inference phase determines which variable properties are known for the program given the semantics of the statements and, possibly, using information about the input variables. Upon completion of the static inference phase, the dynamic phase inserts, into the AST, the code necessary to resolve any unknown variable properties at run time.

For the type inference, a *type algebra* is used similar to the one described in [Sch75] for SETL. This algebra operates on the type of the MATLAB objects and is implemented using tables for all operations and built-in functions. Each node in the AST contains attribute fields to store the 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. Again, in the case of rank and shape, code to apply dynamic analysis is generated whenever the static analysis fails.

The algorithms developed by DeRose work well for the propagation of known variable properties. The main difficulty the algorithms encounter, however, is the lack of information about the input data. Most of MATLAB's built-in functions and any MATLAB M-files are able to work regardless of the type, rank, or shape of the data, using run-time tests to determine the correct operations to perform. In order to provide some information about the input data, the algorithms require a test data set that contains the correct type and rank of the input variables. From the input data, the algorithms know the exact type of each variable, but use a less exact definition of the rank, assuming that any input dimension greater than one can also be one. As a result, an input scalar value is assumed to be a scalar, an input vector is considered to be either a scalar or a vector, denoted as NOT_MATRIX, and an input matrix may be of any rank, denoted as UNKNOWN.

An alternative approach to determining the properties of input values is to allow the developer to interactively set the type and rank of a variable. As part of the support for interactive development, this capability was also added to the environment. This addition, however, uses different assumptions about rank propagation than the original rank inference algorithm and, as a result, modifications to the algorithm were required. These modifications will be discussed later in Chapter 4.

3.1.3 Dynamic Analysis

In some cases, when the static information is insufficient, the developer 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 run time and, based on this determination, allocates the necessary space and selects the appropriate code sequence. An approach is used similar to that used in many systems: associating tags with each variable 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 data space.

Three shadow variables are employed by the system: one to keep track of the variable type and two to keep track of the size of the variable, one for each dimension. A shadow variable is not used for the rank, since the rank can be found by examining the size of the two dimensions, e.g., a value is recognized as a vector when one dimension is equal to one and the other dimension is greater than one. To simplify the analysis of the program and the subsequent code generation, any variable of unknown rank or size is allocated as a two-dimensional array.

Run-time tests of the shadow variables are used to perform the associated numerical operations correctly and efficiently. For example, when two variables, `A` and `B`, are multiplied, but the rank of neither is known, the following operations are possible:

- scalar `A` times `B` (each element of `B` is multiplied by `A`),
- `A` times scalar `B` (each element of `A` is multiplied by `B`),
- inner-product of vector `A` times vector `B`,
- outer-product of vector `A` times vector `B`,
- matrix-vector product of matrix `A` times vector `B`,
- vector-matrix product of vector `A` times matrix `B`, and
- matrix-matrix product of matrix `A` times matrix `B`.

If, in the same example, the type had been unknown instead, then there would be four possible operations:

- REAL `A` times REAL `B`,
- COMPLEX `A` times REAL `B`,
- REAL `A` times COMPLEX `B`, and
- COMPLEX `A` times COMPLEX `B`.

Clearly, expanding the code to perform the run-time tests cannot be done indiscriminately. For example, if `C` is a scalar of unknown type, it may be faster to assume throughout the program that it is a complex variable and generate code accordingly. However, if `C` 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, if `C` 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 rank and type combinations would grow exponentially. This problem is reduced by transforming the long expressions into triplets, or by distributing the loop; in some cases, however, the best strategy may be to assume that the variable is `complex`.

An additional issue to be considered when inserting run-time tests is whether all the tests are actually needed. After there are enough tests to guarantee that the correct numerical operation will be performed, the additional tests are typically used to select the algorithm that is optimized for a particular combination of operand ranks. Consider the previous example where `A` and `B` of unknown ranks are multiplied. Once it is known that neither `A` nor `B` is a scalar, then a matrix-matrix multiplication algorithm could be used for the other five possible combinations. The use of less tests could significantly reduce the code size and make it easier to maintain and modify the generated code, however, with a possible decrease in performance. Algorithms to reduce the number of tests are part of the interactive restructuring techniques and will be presented in Chapter 4.

3.2 The Code Generation System

The Code Generation System is responsible for converting the internal format, the AST, into the output format, one of the three target languages: Fortran 90, C++, and MATLAB's array language. The work by DeRose was only for the Fortran 90 code generator, with the C++ and MATLAB code generators added during the development of the interactive techniques. Therefore, this discussion will focus primarily on the Fortran 90 code generator, with the MATLAB code generator presented in Chapter 4 and the C++ code generator presented in Chapter 10.

In addition to the code generators, other important components of the Code Generation System are the libraries of numerical operations that are required to support the functionality of the MATLAB language. The importance of these libraries comes from both their role as the target of the code generators and the role their algorithms play in determining the numerical precision of the compiled code. Since the libraries serve as the target for the code generator, the discussion of the libraries will be presented first, followed by the discussion of the Fortran 90 code generator.

3.2.1 Target Libraries

MATLAB provides a wealth of functionality to the programmer, both through operators to be used in expressions and through function calls. The functions within MATLAB are provided in one of two forms, either as a function that is compiled into the environment, referred to as a *built-in* function, or as a subroutine written in the MATLAB language that is loaded and interpreted when needed. The arithmetic operators are always compiled into the environment.

As mentioned in the discussion of the Program Analysis System, functions provided as M-files are inlined into the program; therefore, supporting subroutines do not need to be provided for these type of functions. A support subroutine is necessary for any built-in function or operator not supported by the target language. In the case of the MATLAB code generator, all operations present in the input language, assuming it is a valid MATLAB program, are supported by the target language. Therefore, the MATLAB code generator does not need any support libraries outside of the MATLAB environment.

Using the MATLAB manuals, it was sometimes possible to determine the library, and in some instances the subroutine, MATLAB uses to support a particular functionality. In these cases, attempts were made to utilize the same subroutine. If the library could not be identified, then attempts were made to identify a suitable replacement library from those freely available within the public domain, with FALCON utilizing the following libraries to support the MATLAB functionality. The basic arithmetic operations for vectors and arrays are provided by the Level-1, Level-2, and Level-3 BLAS libraries. LINPACK and LAPACK are used to provide linear system solvers, with the eigenvalue and eigenvector solvers coming from EISPACK and the FFT routines coming from FFTPACK [Swa85].

All of these libraries are written in Fortran 77 and, as the subroutine interface is the same for Fortran 90, the Fortran 90 code generator has no difficulties with the interface. However, as the Fortran interface is different than the C++ subroutine interface, the C++ code generator must resolve the differences. Since Fortran 77 does not support dynamic allocation of data, all work arrays needed for the subroutines must be allocated before the subroutines are called. Therefore, in both the C++ and Fortran 90 code generators, additional temporary work arrays must sometimes be allocated or wrapper functions must be written to perform the allocations.

As implemented, the code generators provide a one-to-one translation between the function calls in the AST and the support libraries. One goal of the Interactive Restructuring System is to provide a means to select among multiple subroutines that support the same functionality or to allow different libraries to be targeted. This is discussed in Chapter 4.

3.2.2 Fortran 90 Code Generator

The Fortran 90 code generator was the first to be implemented within the FALCON environment. It is simpler than the C++ code generator in two ways; the Fortran 90 language provides more built-in features for array operations, allowing many MATLAB operations to be translated directly into Fortran 90, and because it has the same data layout and subroutine interface as the Fortran 77 target libraries. The code generator, as written by DeRose, provides a one-to-one translation from the AST to Fortran 90.

The work of the code generator can be broken down into two basic functions, generating declaration statements for the variables and translating expression trees in the AST into Fortran 90 statements. When generating declarations, the code generator must first determine if the size of the variable is known or unknown. When the size is unknown, the variable will be declared as allocatable and the shadow variables for the dimensions must be declared. If the type is unknown, then both a REAL and COMPLEX version of the variable will be declared and the shadow variable for the type must be declared. The code generator must also resolve the name of each variable instance with respect to both the SSA instances and to the *case insensitive* nature of Fortran. If the analysis, using the SSA method, has determined that multiple versions of a variable should be used (perhaps due to it having multiple types or ranks) an index number is appended to the variable name to differentiate the instances. A second problem arises from MATLAB recognizing variable names based on the case of the letters, while Fortran ignores the case. For instance, in MATLAB, the variables `A` and `a` are considered two separate variables while in Fortran they would be considered the same variable. If it is determined that two variable names are identical except for the case of the letters, then a flag is appended to one of the names to differentiate it.

When translating the expression trees into Fortran statements, the code generator utilizes the built-in Fortran 90 operators and functions for control structures, to perform scalar operations, and to perform the simpler operations on arrays and vectors such as addition and subtraction and for doing element-wise multiplication and division. The BLAS libraries are used for vector and matrix multiplication and the other target libraries are used to perform the higher-level operations, such as solves, as described in the previous section.

The translation of the dynamic memory allocation expressions is complicated by the manner in which Fortran 90 performs the dynamic data allocation within the scope of the current routine and frees the data upon its exit. Two attempts at writing a subroutine to perform dynamic memory allocations failed. In the first attempt, the compiler would not pass the unallocated matrix into the subroutine and, in the second attempt, the allocated data was freed when the subroutine returned. Instead of using a subroutine, the code generator inserts the dynamic allocation code for every allocation occurrence, increasing the size of the output program and making it more difficult to read, maintain, and modify the program.

The other code generators build upon the work done for the Fortran 90 code generator. However, they have been modified both to take advantage of language features present in their target language but not available in Fortran 90 and to provide code to support features present in Fortran 90 that are not present in their target language.

Chapter 4

A RESTRUCTURING SYSTEM FOR INTERACTIVE PROGRAM DEVELOPMENT

The third component of the FALCON environment is the Interactive Restructuring System. This system was designed to provide the algorithm developer with interactive techniques for use during the development of numerical algorithms by combining the software techniques from restructuring compilers with the transformations utilized in numerical analysis. The development of such techniques was the primary focus of this research.

The Interactive Restructuring System was built around pattern-based transformations so that the system could be tailored or enhanced to fit the individual needs of the developer. To provide this support, there are four main components within the system. First, there is the user interface that controls the application of transformations and can be used to examine properties of variables. Second, there is the transformation module for applying the transformation patterns and other restructuring algorithms and for propagating variable information. The database of transformation patterns is the third component. And fourth, the program for encoding the patterns into the internal format.

Another need for flexibility within the restructuring system comes from the desire to perform transformations at multiple levels. At one level there are the transformations that reorder the statements within the internal format, but only utilize operations that can be represented in MATLAB. However, at a second level, there are transformations that insert operations found only in the target language into the code being developed. During the design of the restructuring system, it was decided to develop a transformation system that supported both of these levels rather than develop two separate systems. Discussion of the restructuring system will begin with the patterns, followed by how the transformations are performed, and then finish with the user interface.

4.1 Pattern-Based Transformations

An integral part of the design of the Interactive Restructuring System is providing the algorithm developer with the ability to adapt existing transformations and to add new transformations. Compared to most restructuring compilers, this places a different emphasis upon how the transformations are implemented. Traditionally, the restructuring transformations are hard-coded into the compiler, with their use controlled either by flags passed to the compiler or by directives placed within the code. These commands can be used to control when a transformation is applied and to pass additional information or parameters to the transformation algorithm. An alternative approach is to develop a language for describing the transformations utilizing a pattern-matching system, which is the approach that was taken for both the Forbol [Wea94] and SORCERER [Par94] systems. However, in both of these systems, the languages were designed for a compiler writer to add passes to the compiler and were not designed for the user to modify the transformations.

The first step in providing the desired flexibility was to enable the transformations to be loaded into the environment. Though it is possible to dynamically link in subroutines, the approach chosen for FALCON is to load a description of the transformation and then execute the transformation. In order to define a language for describing a transformation, it is necessary to understand both the information that will be needed for the transformation, and the operations a transformation will be asked to perform. Most code transformations can be broken into two basic parts, the selection of a source code fragment and modifications to the selected code fragment.

The selection of the source code fragment is based upon a *code pattern* which utilizes various program characteristics, including properties of the variables, the operations performed, the sequence of statements, and the dependences between the variables. However, not all of these characteristics will be used every time and, in some cases, it is necessary to ignore specific characteristics. As it may be necessary to represent any possible operation from the source code in a pattern, it is necessary for the transformation language to support all of the source language.

Not only can the replacement part of the transformation include any possible operation from the source language, it can also contain operations that only appear in the target language. This is necessary to provide the developer with control over the code being generated, such as inserting calls to subroutines available in the target language but not in MATLAB.

In order to meet the needs of the system, the transformation has the basic form shown in Figure 4.1. In this figure, three of the keywords that have been added to the transformation language are used: `REPLACE`, `WITH`, and `END`. Unlike MATLAB, where keywords are not reserved, the keywords added to the transformation language are reserved. The block of statements between the `REPLACE` and `WITH` keywords are used to represent the source code fragment to be selected and is referred to as the REPLACE block. The block of statements between the `WITH` and `END` keywords are

```
REPLACE
      variable declarations
      transformation statements
WITH
      variable declarations
      transformation statements
END
```

Figure 4.1: Basic form of the transformation pattern.

used to represent the modification or replacement part of the transformation and is referred to as the WITH block. In both cases, the block of statements consists of two parts, variable declarations and transformation statements, each of which are discussed in turn.

The REPLACE block construct utilizes the same approach as the Forbol language `replace$block`, shown in Figure 4.2, for describing the transformation pattern. This approach is a logical choice for building transformations upon a pattern matching system, with the replace function from TXL [CC93], shown in Figure 4.3, having a similar format.

```
replace$block [pgm] {
      declarations
      statements
with
      declarations
      statements
};
```

Figure 4.2: Format of the replace block from Forbol.

```
function name
      replace [type]
            pattern
      with
            replacement
end function
```

Figure 4.3: Format of the replace function from TXL.

4.1.1 Variable Declarations

Whereas MATLAB has no variable declarations, the variable's type and rank are determined implicitly or at run time, the transformation language requires all variables (except for automatically-generated temporaries) to be explicitly declared. This is necessary because certain transformations only hold for specific variable type and rank combinations.

A variable declaration consists of the following four main fields:

type *name* (*index*) { *properties* },

with the first two fields required and the last two optional. Multiple variables may be declared on the same line by placing a comma after the first declaration and then appending the next declaration, excluding the type field. All variables on the same line are assumed to be of the same type.

The three variable types supported within the transformation language are `INTEGER`, `REAL`, and `COMPLEX`. These type names correspond to the types recognized by the Program Analysis System. The variable name may be any variable name supported by MATLAB, with the exception of the keywords reserved for the transformation language. The rank of the variable is determined by placing an index set, within parentheses, after the variable name. If the index set is missing, then the variable is assumed to be a scalar. If the index set contains a single value, then the variable is assumed to be a vector. And, if the index set contains two values, separated by a comma, the variable is assumed to be a matrix.

The final field in a declaration, *properties*, is used to set variable properties that are difficult to represent using the other fields. For instance, within MATLAB all vectors are assumed to be row vectors unless otherwise specified. To provide the developer more flexibility, this characteristic of the vector can be set within the pattern. If the keyword `ROW` is present within the property field, the vector is a row vector, and similarly, if the keyword `COLUMN` is present within the property field, the vector is a column vector. Also, although the index set can be used to set the rank of the variable to scalar, vector, or matrix, it is not possible to choose one of the other dynamic ranks. If the keyword `NOTSCALAR` is in the property field, the rank is set to NOT_SCALAR and if the keyword `NOTARRAY` is in the property field, then the rank is set to NOT_MATRIX. A final option is to define a variable that will match all ranks by placing the keyword `ANY` within the property field.

If a variable declared in the REPLACE block does not have a declaration in the WITH block, then the original declaration also holds for the WITH block. However, if a second declaration is present, then the properties of the variable will be modified to correspond to the new declaration. As a result, the variable declaration portion of the WITH block is optional.

In addition to declaring regular variables, user-provided routines can also be declared in the declaration portion of the WITH block for subsequent use in the statement portion of the block. These declarations are used when the algorithm developer wishes to replace MATLAB operations in the program with calls to specific routines. The declaration defines both the type of the routine call and the properties of the parameters, with the declaration format for both subroutine and function calls being provided in Figure 4.4.

`FUNCTION` *function name* (*argument list*){*dim1, dim2*}
`SUBROUTINE` *function name* (*argument list*)

Figure 4.4: Declaration of user-provided routines.

Within the routine declaration, the *argument list* is responsible for indicating the order of the input and output arguments, since the use of the routine in the MATLAB code will keep the arguments in two separate lists. For instance, consider a Fortran subroutine `find_range` that inputs a list of numbers, `values`, and returns the minimum and maximum values, `min` and `max`, with the arguments in that order:

```
find_range( values, min, max ).
```

In MATLAB, this call would be:

```
[ min, max ] = find_range( values ).
```

To generate the correct output code for using this call, it is necessary to indicate the order of the arguments as one input argument followed by the two output arguments. To accomplish this, input arguments are indicated by `P_IN` and output arguments are listed by `P_OUT`. Furthermore, to allow the output variables to be declared of the correct size dynamically, it is necessary to define the dimensions of each output variable, using the format {*dim1, dim2*}, after each variable. The declaration of the `find_range` subroutine would be as follows:

```
SUBROUTINE find_range( P_IN, P_OUT{1,1}, P_OUT{1,1} ),
```

with `min` and `max` being declared as scalars. Note that the same format for declaring the size of the output variable is also used for declaring the return value of a function at the end of the `FUNCTION` statement.

If the output arguments are not a constant size, however, additional information is needed to define the size. Since the size of an output argument is typically based on the size of an input argument, it is also possible to label an input argument and use its dimensions to denote the size of the output variables. First, an input argument is labeled by placing an identifier after the input flag, `P_IN`{*id*}. Next, a special function, `DIM(` *id*`,` *dim_val* `)`, is provided to take the first or second dimension, with *dim_val* equaling `1` or `2` respectively, of the input argument labeled with *id*. For example, the following argument list would declare the output argument to be the same size as the input argument:

```
P_IN{a}, P_OUT{DIM(a,1), DIM(a,2)}.
```

A case that cannot be handled by the previous strategy is when the routine argument is used for both input and output. To handle this case, the MATLAB call is written using two arguments, with the input argument placed within the MATLAB call and the output argument placed on the left-hand side of the assignment. A third argument declaration type, `P_IN_OUT`, is then used in the declaration of the routine. There is no need to define the size of the output argument as it is assumed to be the same size as the input argument. The argument can be labeled, using the same format as the regular input arguments, so that its size can be used to define the size of other output arguments.

4.1.2 Transformation Statements

Within both the REPLACE and WITH blocks, following the variable declaration (if any), is a sequence of transformation statements that are either MATLAB statements or new statements that have been added to the transformation language. The MATLAB statements supported within the transformation language are the same statements that are supported by the FALCON environment for the input code.

New statements were added to the transformation language to provide the developer with the means for matching code fragments in a more abstract manner and to provide additional methods for code modification. The three primary statements are the BODY, a WHERE block, and an IN block. At this time, only the format of these statements will be presented, with the discussion of their use delayed until Section 4.2, where the application of patterns will be discussed.

The BODY Statement

The BODY statement is used to match any sequence of statements between two statements in the REPLACE block, with the format of the BODY statement being presented in Figure 4.5. Note that the BODY is labeled with an identifier (called *id*) in the example. This allows multiple BODY statements to be used within a transformation pattern.

$statement_1$
`<BODY>(`*id*`)`
$statement_2$

Figure 4.5: Format of the BODY statement.

The WHERE Block

The WHERE block allows dependence information to be used as part of the criteria for selecting the source code fragment in the WITH block. The two forms of the WHERE block are presented in Figure 4.6. The dependence list contains one or more dependence statements of the following form:

$$id_1 \;\; operator \;\; id_2.$$

An *id* in a dependence statement is either a variable within the program or the identifier of a BODY. There are four possible dependence operators: `DOESNT_READ`, `DOESNT_WRITE`, `READS`, and `WRITES`.

MATLAB assignment statement	<BODY>(*id*)
<WHERE>	<WHERE>
dependence list	*dependence list*
<END>	<END>

Figure 4.6: The two forms of the WHERE block.

The IN Block

The IN block is used to modify the statements matched by a BODY, with the format of an IN block being presented in Figure 4.7. Within the IN block, only a single MATLAB expression can be used for *MATLAB expression*$_1$ and *MATLAB expression*$_2$.

```
IN
        <BODY>(id)
    REPLACE
        MATLAB expression1
    WITH
        MATLAB expression2
    END
```

Figure 4.7: Format of an IN block.

Additional Comparisons with Forbol

The Forbol language also provides constructs similar to the BODY construct and the IN block. The `BLOCK` variables in Forbol can be used to match any well-formed statement block, like the BODY construct, but the `BLOCK` variables can also be used to only match statements containing a particular pattern. Like the IN block, the `replace$exprs$in$block` is used to replace expressions within a block. The similarities between the two languages result from needing to provide the same basic functions to the users. However, due to the differences in the users of the two systems, the FALCON environment must provide additional capabilities that are typically not found in languages such as Forbol. With FALCON, the user can utilize algebraic properties about the operations and the structures within the data both to perform transformations on the algorithm and to improve the code that is generated, including the ability to select optimized data structures. (The utilization of the structures within the data will be discussed later in Chapters 7 and 8.)

4.2 Application of the Transformation Patterns–Matching

The application of a transformation pattern occurs in two steps, corresponding to the two parts of the transformation pattern. In the first step, the pattern from the REPLACE block is matched with a fragment of the source code. Next, if the match is valid, the modification of the source code is performed according to the WITH block.

The matching of the statements within the REPLACE block to the statements in the source code is performed using the internal format of the FALCON environment, the AST. The tree representing the source code, the *source*, is compared to the tree representing the REPLACE block, the *pattern*. A few of the node types that appear within the trees, however, either do not need to be or should not be compared. The majority of these nodes were inserted by the Program Analysis System and deal with the dynamic allocation of data. To avoid these nodes during the tree comparison, a second *search* tree is laid over the nodes within the AST for both the source and pattern trees. This approach both simplifies the code required to compare the trees and speeds up repeated comparisons as the search tree is rebuilt only when necessary.

For the pattern tree to match the source tree, several conditions must be satisfied. Most of the nodes must match exactly with an inexact match of the nodes being sufficient for a few exceptions. When present, dependence conditions must also be satisfied for the trees to match. Also, during the comparison, certain information must be maintained about how the nodes were matched in order for the subsequent modifications to be performed. This section will discuss how nodes are matched and Section 4.3 will cover the use of dependence information.

4.2.1 Exact Matches

Each node within the AST contains a *node type* that relates the node to a specific semantic construct within the source language. Whether or not a node requires an exact match is based upon the node type of the node in the pattern tree. All nodes corresponding to MATLAB statements must exactly match the nodes in the source tree, with the exception of nodes that represent variables. The node types for variables will be considered later during the discussion on inexact matches. Also, the nodes corresponding to the new statements added to the transformation language do not require an exact match.

For two nodes to be an exact match, three conditions must be satisfied:

1. The node types must be the same.
2. Either the nodes must refer to the same symbol or both nodes must not have a symbol reference.
3. If the nodes contain rank and intrinsic type information, this information must match.

Since the node types are represented as integers, the comparison of two node types is simple. The comparisons needed to satisfy the other two conditions, however, are more complex.

The comparison between symbols is based upon the assumption that there will be a one-to-one correspondence between the symbols in the pattern and the source. Once two symbols have been matched, this association is retained both for future comparisons and for subsequent use during code modification. Therefore, for a symbol in the pattern to match a symbol in the source, one of the following two conditions must be met:

1. Neither symbol has been previously matched.

2. The two symbols are already associated to one another through a previous match.

When comparing the rank and intrinsic type information of two nodes, the intrinsic type information must always match exactly while some inexact matches are allowed for the rank. Table 4.1 contains the possible ranks for the pattern and source nodes and indicates whether the combination of ranks will be considered a match or a failure.

Pattern Rank	Source Rank					
	UNKNOWN	SCALAR	NOT_MATRIX	VECTOR	NOT_SCALAR	MATRIX
UNKNOWN	match	fail	fail	fail	fail	fail
SCALAR	fail	match	match	fail	fail	fail
NOT_MATRIX	fail	match	match	match	fail	fail
VECTOR	fail	fail	match	match	match	fail
NOT_SCALAR	fail	fail	fail	match	match	match
MATRIX	fail	fail	fail	fail	match	match

Table 4.1: The matching of ranks when comparing nodes.

4.2.2 Inexact Matches

An *inexact match* refers to a comparison where the pattern node is considered to match the source node even though the node types are different. There are three specific cases where an inexact match may occur during the comparison of the source and pattern trees:

- When comparing *symbol references*[1] and indexed array references.
- When comparing a symbol reference in the pattern to a scalar constant in the source.
- When a BODY statement occurs within the pattern.

Inexact matches between symbol references and indexed array references were added to allow transformation patterns that operate on matrices to also operate on sub-matrices. Without these

[1] A *symbol reference* is used to refer to a variable reference that is not indexed.

matches, there would need to be a separate version of the transformation pattern for every possible combination of matrices and sub-matrices. For example, this allows the following expression in the source code:

```
A = B(1,1) * C,
```

to be matched by the following expression in the transformation pattern:

```
D = E * F.
```

Given that there are two types of variable references, symbol references and indexed array references, there are four possible match combinations to be examined. The discussion of these combinations can be simplified by the definition of a few terms:

$IDENT_{source}$ – refers to a symbol reference in the source program,

$IDENT_{pattern}$ – refers to a symbol reference in the transformation pattern,

$ARRAY(INDEX\_SET)_{source}$ – refers to an indexed array reference in the source program, and

$ARRAY(INDEX\_SET)_{pattern}$ – refers to an indexed array reference in the transformation pattern.

Therefore, the four possible comparisons are:

$$IDENT_{pattern} \text{ compared to } IDENT_{source},$$
$$ARRAY(INDEX\_SET)_{pattern} \text{ compared to } ARRAY(INDEX\_SET)_{source},$$
$$IDENT_{pattern} \text{ compared to } ARRAY(INDEX\_SET)_{source}, \text{ and}$$
$$ARRAY(INDEX\_SET)_{pattern} \text{ compared to } IDENT_{source}.$$

Given that the rank and intrinsic type of the variable references match, the remaining rules used to determine if the references match for each of these comparisons are presented in Figure 4.8. It is also necessary for the rank and intrinsic type of the variable references to match according to the criteria presented during the discussion on exact matches. If a match does occur, then the symbol in the source reference is associated with the symbol in the pattern reference, as with the exact matches. When a symbol reference is matched to an array reference, it is also necessary to store the index set used by the array reference.

A second type of inexact match can occur between a symbol reference in the pattern and a scalar constant in the source. This allows an expression in the source code such as

```
A = B + 1
```

to be matched by an expression in the transformation pattern such as

- $IDENT_{pattern}$ matches $IDENT_{source}$ if:

  neither has been matched before

  or

  $IDENT_{pattern}$ has already been matched with $IDENT_{source}$.

- $ARRAY_{pattern}$ matches $ARRAY_{source}$ if:

  neither has been matched before

  or

  $IDENT_{pattern}$ was matched with $IDENT_{source}$

  and

  $INDEX\_SET_{pattern}$ matches $INDEX\_SET_{source}$

  or

  $IDENT_{pattern}$ has already been matched with $ARRAY_{source}$

  and

  $INDEX\_SET_{pattern}$ matches $INDEX\_SET_{source}$.

  and

  $INDEX\_SET_{source}$ is a subset of the index set of the array reference in the source code which first matched the symbol in the pattern.

- $IDENT_{pattern}$ matches $ARRAY_{source}$ if:

  neither has been matched before

  or

  $IDENT_{pattern}$ has already been matched with $ARRAY_{source}$.

- $ARRAY_{pattern}$ never matches $IDENT_{source}$.

Figure 4.8: Rules for determining if two variable references can be matched.

```
C = D + F.
```

In order for this match to be allowed, several conditions must be met:

- The rank and intrinsic type of the symbol in the pattern must match the constant in the source code.

- The symbol in the pattern cannot have been previously associated with a symbol in the source code.

- If the symbol in the pattern has been previously associated with a constant, the value of the previous constant and this constant must be the same.

If all of these conditions are met, then the symbol in the symbol reference is associated with the value of the constant.

The third type of inexact match is the use of a BODY statement in the pattern to match statements in the source. This allows two statements within the source program to be found with any number of other statements between them. For example, this can be used to match on a `for` loop while not considering the body of the loop:

```
for k = 1:n
     < BODY >(a)
end.
```

Using the format of the BODY statement as described in Figure 4.5, its application is as follows. After $statement_1$ has been matched, the transformation system examines the next pattern statement. Upon identifying this statement as a BODY statement, the restructuring system modifies its internal state to indicate a BODY has been entered and then continues on to the next statement in the pattern, $statement_2$. It then attempts to match $statement_2$ with the next statement in the source. If this match fails, the source statement is labeled as the *starting* statement and it then attempts a match with the following statement in the source, continuing the process until either a match is made or all of the statements in the source have been exhausted. If a match of $statement_2$ is found, then the starting statement and the previous source statement, which was the last source statement that failed to match, are stored within the BODY statement. One limitation of the BODY statement is that it will only match nodes at the same control flow level. Nodes contained within a lower control flow level will be ignored and the search will end upon reaching a statement at a higher control flow level.

4.3 Use of Dependence Information in Transformation Patterns

In addition to matching the node type, rank, and intrinsic type information, the developer can also utilize dependence information for controlling the matching of transformation patterns. For example, the following sequence of statements could be used in a transformation pattern to find two updates to a variable where there are no accesses of the variable between the updates:

```
A = A * B
<BODY>(c)
<WHERE>
     c DOESNT_READ A
<END>
A = A * D.
```

The constraints are not limited to variable accesses, but also support array references. For instance, the previous example can be modified so that only accesses to `A(k)` are checked:

```
A(k) = A(k) * B
<BODY>(c)
<WHERE>
     c DOESNT_READ A(k)
<END>
A(k) = A(k) * D.
```

In the current system, the application of a transformation pattern with dependence constraints is a two-part process. First, the transformation pattern is compared to the source code to determine if the statements match. If a successful match is found between the statements, then the dependence information is checked. If the dependence constraints are not satisfied, then the match is rejected. In addition to tests of inner-loop dependences, such as those shown in the previous examples, cross-loop dependences can also be tested.

4.3.1 Cross-Loop Dependences

Cross-loop dependences can be used to test whether data references in the current loop iteration modify or access data from other loop iterations. In order to denote cross-loop dependences, the operands in the dependence list must refer to a specific loop iteration or range of iterations, with the left operand always referring to the current loop iteration. The loop index variable is treated as the value of the current loop iteration, allowing the right operand to refer to loop iterations relative to the loop index of the left operand; i.e., values less than the loop index refer to previous loop iterations and values greater than the loop index refer to future loop iterations. For instance, if the loop index is `k`, then the index range `(1:k-1)` would refer to all the previous loop iterations. For example, the dependence statement needed to test if, during the current loop iteration, the statements in the `<BODY>(c)` update anything written by the body, during a previous loop iteration, would be:

```
c DOESNT_WRITE c(1:k-1).
```

Or to make sure the statements in `<BODY>(c)` during the current loop iteration do not read anything written by a statements in `<BODY>(b)` in a future loop iteration, the dependence statement would be:

```
c DOESNT_READ b(k+1:n),
```

where `n` is the upper bound on the loop index. Additionally, inner-loop and cross-loop dependences can be freely mixed within a WHERE block.

4.3.2 Evaluation of Dependence Statements

Within the WHERE block, the dependence statements are evaluated individually to determine if the conditions are met and the comparison is considered a match only if all of the conditions are met.

The algorithm to evaluate a dependence statement is as follows:

1. For both the left and right operand, determine the list of data references:

   (i) Given the dependence operation and the position of the operand, determine if the write list or read list is needed for the operand. Table 4.2 shows how this is determined.

   (ii) Traverse the statements in the body, building the list of data references of the required reference type.

   (iii) Expand each of the data references in the list by replacing occurrences of the loop index in the array references with the loop index range associated with the dependence operand.

2. Compare the lists to determine if any reference in one of the lists intersects a reference in the other list.

3. The dependence condition is satisfied if:

   - the operand is `DOESNT_READ` or `DOESNT_WRITE` and no intersection was found, or
   - the operand is `READS` or `WRITES` and an intersection was found.

Operand Position	Dependence Operation			
	`DOESNT_READ`	`DOESNT_WRITE`	`READS`	`WRITES`
left	read	write	read	write
right	write	write	write	write

Table 4.2: Selection of the read list or write list for dependence test.

Two of the steps in this algorithm that warrant additional consideration are the expansion of the data references and the comparison of the lists for intersections.

Expansion of Data References

The list of data references is built using a simple traversal of the tree associated with the statements of the <`BODY`> block or the tree associated with the MATLAB assignment statement. If the dependence statement is for a cross-loop dependence, then each reference is examined to determine if it is an indexed array reference that is dependent upon the loop index.

To determine if an array index is dependent upon the loop index, each variable within the array index is replaced by the expression tree representing its definition. This replacement continues until all the variables remaining in the expression tree are either loop indices or variables that have been loaded. For example, consider the array reference `A(k)` in the the following code segment:

```
for i = 1:n
     x = 2 * i;
     y = 3 + i;
     k = x + y;
     A(k) = ···
end.
```

This array reference would be expanded to `A((2*i)+(3+i))`.

Once the variables within an array index have been replaced by the loop indices, the second phase of the variable expansion can proceed. The remaining loop indices are replaced with the loop index range associated with the dependence operand. For instance, if the reference to `A(k)` from the previous example was associated with the dependence operand `c(1:i-1)`, the array reference would now be expanded to `A((2*(1:i-1))+(3+(1:i-1)))`. This process is repeated until all of the array references have been expanded by the associated dependence operand.

Comparison of Reference Lists for Intersections

Once the reference lists have been expanded, the next step is determine if there are any intersections between variables in the two lists. Each reference from the left operand list is compared to the references in the right operand list. If the references are for different symbols, then the algorithm continues to the next pair of references. If the references are for the same symbol, however, the two references must be examined further to determine if an intersection exists. If an intersection is found, then the algorithm exits, indicating there is an intersection. If the system cannot determine if an intersection exists, then the algorithm exits, indicating the result is unknown. If no intersection is found, then the algorithms continues to the next pair of references. If all reference pairs are examined without finding an intersection, then the algorithm exits, indicating no intersection exists.

In order to determine if two references to the same variable intersect, it is necessary to consider the node type of the reference, the rank of the variable, and the type of the array index or indices. Symbol references, those without indices, will always intersect, regardless of whether the variable is a scalar, vector, or matrix. A problem occurs, however, in that the temporary variables created to form the triplets are almost always referenced as symbol references. This can result in many false intersections which do not occur for the source program variables. Therefore, temporary variables are ignored during the intersection calculations.

For a vector reference, the index sets are compared between the two references. Two index sets are known to not intersect if the maximum value of one of the index sets is less than the minimum value of the other index set. For a matrix reference, the index sets for both dimensions must intersect for the data references to intersect.

To determine if two index sets intersect, the index sets are converted into the form

lower bound:upper bound

for the different array index types. The possible array index types and the rules for their conversions are presented in Figure 4.9. After this conversion, the logical conditions,

$$\begin{aligned} \textit{upper bound}_{\textit{left}} &< \textit{lower bound}_{\textit{right}} \\ \textit{upper bound}_{\textit{right}} &< \textit{lower bound}_{\textit{left}}, \end{aligned}$$

can be evaluated by subtracting one side from the other and comparing the result to zero. Since the expression may contain variables with unknown values, the computation must be performed symbolically.

Index Type	Conversion
(:)	A colon indicates the entire index set and, therefore, no conversion is necessary as the index will always intersect.
(value)	When the index is a single value the lower and upper bounds are both set to the value. *lower bound = value* *upper bound = value*
(lower:upper)	When the index is a range, no conversion is necessary. *lower bound = lower* *upper bound = upper*
(lower:stride:upper)	If the *stride* is positive, then no conversion is necessary. *lower bound = lower* *upper bound = upper* However, if the *stride* is negative, then the lower and upper values are reversed. *lower bound = upper* *upper bound = lower* If the sign of the *stride* is unknown, then it is impossible to tell which is the lower and upper bound and the algorithm indicates the intersection is unknown. An alternative approach is to perform the test using both values in this case.

Figure 4.9: Rules for converting index types to allow the calculation of intersections.

Symbolic Evaluation of the Logical Conditions

In both cases, the logical expression being tested is of the form:

$$val_1 \quad < \quad val_2.$$

This expression is equivalent to testing if the difference of the two values is greater than zero:

$$val_2 - val_1 \quad > \quad 0.$$

The symbolic evaluation of the logical condition is performed in three steps:

1. Both operands are converted into an *expanded form*.

2. One expanded form is subtracted from the other form.

3. The resulting expanded form is compared to zero.

This expanded form is used instead of the expression trees to simplify both the subtraction and the comparison with zero.

One difficulty with the expression trees is that an algebraic expression can be represented by multiple trees due to the properties of associativity, distributivity, and commutativity. For instance, the expression `A+B+C` can be represented by the two expression trees in Figure 4.10. Notice that

Figure 4.10: Two expression trees for `A+B+C`.

since both are addition operations, and since the operation is associative, the expression can be represented as the sum of a list of operands, as shown in Figure 4.11. Since subtraction can be

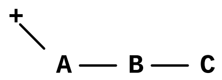

Figure 4.11: Expanded expression for `A+B+C`.

implemented as the addition of the negative of the right operand, the operands from subtraction statements can also be combined with addition statements:

$$\texttt{A + B - C} \rightarrow \texttt{A + B + (-C)}.$$

A similar approach can be used with multiplication and division statements, with division being considered as multiplication by the reciprocal,

$$\texttt{A * B / C} \rightarrow \texttt{A * B * (1/C)},$$

and the expression being represented as the multiplication of a list of operands, as shown in Figure 4.12.

As the expanded form is to be used for subtracting two expressions, it was decided to resolve distributed expressions into the sum of products form,

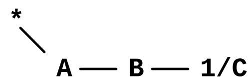

Figure 4.12: Expanded expression for A*B/C.

$$(A + B) * C + D \rightarrow (A * C) + (B * C) + D.$$

Within the expanded form, an operand within a list can itself be another operation with an associated operand list as shown in Figure 4.13. The operand list is also simplified by combining all

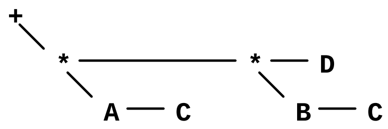

Figure 4.13: Expanded expression for (A*C)+(B*C)+D.

constants present in the list into a single operand.

The subtraction of the two expanded forms is done symbolically, with the result being a single expanded form. This resulting form is simplified as much as possible, using the concepts of the expanded form that were just described, and then compared to zero. If the exact value of the resulting form cannot be calculated, then attempts are made to find the minimum and maximum bounds on the value. For instance, a loop index is evaluated by examining the bounds on the loop. A value, or bound on the value, can be found only if all the remaining variables in the expanded form can be associated with a constant value. There are three possible outcomes from comparing the result with zero:

- If the resulting value is greater than zero, or if the minimum possible value is known to be greater than zero, then the logical expression is true.
- If the resulting value is less than zero, or if the maximum possible value is known to be less than zero, then the logical expression is false.
- If neither of the previous conditions can be satisfied, then the result is considered unknown.

The results from these logical comparisons are used to determine whether the two references intersect. The results of all the intersections are then combined to determine if the dependence condition is satisfied. If all of the dependence conditions within the *where* block are satisfied, then the pattern is considered to match the source.

4.3.3 Additional Comments on the Use of Dependence Information

Though this system for dependence information is similar to those utilized by restructuring compilers, it has very different design goals and, therefore, its implementation and operation are different. The goal of this system is to allow algorithm developers to write algebraic transformations to be applied to their code, not to allow compiler developers to implement general restructuring techniques for arbitrary code. As such, the language used to describe dependences is that which would be more familiar to the algorithm developers and not the syntax developed by compiler writers.

However, the dependence tests used in other restructuring compilers could be used to improve the dependence analysis within this system. The utilization of more advanced techniques, such as Banerjee's Test [Ban88], the GCD test [Ban88], or the Range Test [BE94], would improve the accuracy of the dependence system and increase the opportunities for the application of transformation patterns.

4.4 Application of Transformation Patterns–Code Replacement and Modification

After the transformation pattern has been matched to the source code, the *source*, the next step is to modify the source code based on the REPLACE block within the transformation pattern, the *transformation*. Typically, such modifications involve adding and deleting statements within the source code. However, it is also possible to use patterns to find statements within the code and not perform any modifications. As with the matching of the pattern, the application of the transformation is performed using the AST form for both the transformation and the source.

The code modifications can occur at two levels, at the statement level and at the expression level. The WITH block of the basic transformation pattern is used for modifications at the statement level while the IN block is used for modifications at the expression level.

4.4.1 Statement-Level Modifications

Statement-level modifications are used to replace the statements in the source code by the statements in the pattern. The algorithm for performing this replacement contains three steps.

1. The AST for the new source code is built using the transformation and the information maintained about the match.

2. The program analysis information in the new source code is updated.

3. The matched source code is replaced with the new source code.

Building the New Source Code

The AST for the new source code is built by traversing the AST for the transformation and, based upon the transformation statement associated with the node type, either copying the node or performing a transformation operation. Only three types of transformation operations are currently supported: two for statement modifications, the BODY statement and assignments to user-provided routines, and the third is for expression-level modifications, the IN block. All other statements are MATLAB statements that will be copied into the AST for the new source code. In this section, only the statement-level modifications are discussed, while the expression-level modifications are discussed in Section 4.4.2.

When copying the MATLAB statements from the pattern to the AST for the new source, the MATLAB operators remain the same but the variables in the pattern must be modified to reflect the associated variables in the source code. For example, if the variable `A` in the REPLACE block of the transformation pattern was matched to the variable `Z` in the source code, then when the WITH block of the transformation pattern is copied into the new source code, every occurrence of variable `A` must be replaced with `Z`. In addition, the SSA index for the variable must also be updated to reflect the correct SSA index in the source code. Alternatively, if the variable is a new variable added by the declaration portion of the WITH block of the transformation pattern, a temporary variable will be allocated in the source code and the associated variables in the new source code will be replaced with the temporary variable.

If the statement in the transformation is a BODY statement, then the source statements which the body matched are copied into the new source code. In this case, the new statements do not need to have their variables updated because they already refer to the variables in the source code.

Lastly, if the statement in the transformation is an assignment to a user-provided routine, then the MATLAB assignment must be converted into a routine call. Two main tasks are performed during this conversion. First, a statement to copy the values of the input argument to the output argument must be inserted before the routine call, for every routine argument declared with type `P_IN_OUT`. And second, a routine call must be inserted into the new source code, with the input and output argument lists merged in the correct order. As with the copying of the MATLAB statements, the variables within the pattern code must be changed to the associated source code variables.

Updating the Program Analysis Information

Once the AST for the new source code has been built, each new statement is examined twice; the first time to recalculate the program analysis information and, the second time, to determine if any data space needs to be allocated for output variables. This is necessary both to allow additional transformations to be performed and to support the subsequent code generation.

The program analysis information of the new code is first updated using the static inference techniques, developed by DeRose [DeR96], for all statements corresponding to MATLAB statements. For the statements that are calls to the user-provided routines, however, new static analysis techniques were developed. These new techniques use the declaration of the routine arguments, as previously described, to determine the size of the output variables based on the size of the input variables. This is accomplished by traversing the list of arguments and, for each output argument, finding the argument declaration. Once the declaration has been found, the size of the dimensions will either be a constant or the dimension of an input argument declaration. If it is the second case, then the input argument corresponding to the declaration is found and used to define the size of the dimension for the output argument. It is not necessary to calculate the intrinsic type of the routine arguments as the variable declarations within the pattern are always typed.

The information from the static inference techniques is then used to update the information needed for the dynamic analysis. As part of this update, the rank of all variables must be made exact, so each variable with a rank of NOT_SCALAR is promoted to MATRIX and each variable with a rank of NOT_MATRIX is promoted to VECTOR.

Once this information is updated, each statement that generates a result is examined to determine if the size of the output is known. If the size is unknown, then code to dynamically allocate the output is inserted into the program.

Replacing the Matched Source Code

The replacement of the existing source code with the new source code is performed by first cutting the matched statements out of the AST. The new statements are then placed in the AST at the position where the matched statements were removed. Also, special flag statements are inserted at either end of the new code, to indicate these statements were created through a replacement. Next, a pointer is added from the flag statement at the beginning of the new code to the matched statements that were cut. The matched statements are saved in this manner to allow for the future implementation of code to undo a transformation. After the new code has been correctly inserted, the source search tree is also rebuilt.

4.4.2 Expression-Level Modifications

The only construct for performing expression-level modifications is the IN block, whose format is presented in Figure 4.7. When an IN block is encountered in the transformation, the BODY statement within the block is handled the same as was described for the statement-level modifications, with the source statements that are matched to the BODY being copied and placed into the new source AST. However, after the copies of the source statements have been made, a second pass is made through these new statements looking for any occurrence of the first MATLAB expression, $expression_1$. If this expression is found, it is then replaced by the second MATLAB expression, $expression_2$.

Before the search is performed, copies of $expression_1$ and $expression_2$ are made so that any variables that appear within these expressions in the transformation pattern can be changed to the associated variables that were matched in the source code. This speeds up the matching process, since both the source and target expressions will be using the same variables.

After this second pass is completed, the program analysis information is updated and the matched source code is replaced using the techniques as described for statement-level modifications.

4.5 The User Interface

For the restructuring techniques to be applied interactively, a user interface from which the algorithm developer can examine and modify the program is necessary. To meet this need, a graphical user-interface was designed to support the operations performed by the developer using the Motif widgets for X Windows [JR93].

4.5.1 Displaying the Information

At various times during the restructuring process, the developer may want access to three items of information:

- The current version of the source program.
- The transformation patterns that may be applied.
- The information known about variables.

When the developer chooses interactive restructuring, a main window is created which displays the first two items, the source program and transformation pattern, side-by-side. Using this window, the developer is able to scroll through each item independently. An example of this window, named the *main* window, is provided in Figure 4.14 for a short sample program and a transformation pattern. In this window, the source code is displayed on the left side and the transformation pattern is displayed on the right side. (The method for generating the displayed codes will be discussed in Section 4.5.2.) In addition to the statements, buttons and menus for controlling the restructuring are present in a row along the top of the window.

Note that the only transformation patterns that will be displayed are those that were given at the start of the execution. Currently, there is no means to load additional transformation patterns during the interactive restructuring.

The first control button, `symbols`, is used to create a second window for displaying information about the variables in the program. The *symbol* window contains the name of each variable and

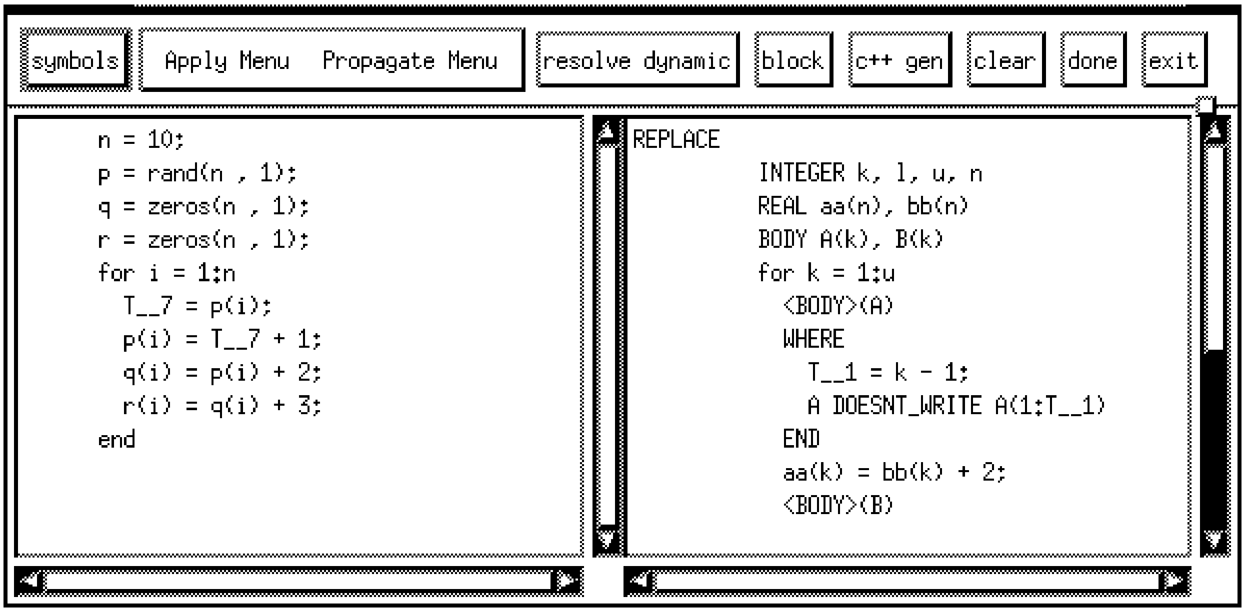

Figure 4.14: Main window for displaying source code and transformation pattern statements.

three properties of the variable: the intrinsic type, the rank, and a user-defined field that will be described later in Chapter 7. Figure 4.15 contains a sample window showing the variables from the previous example.

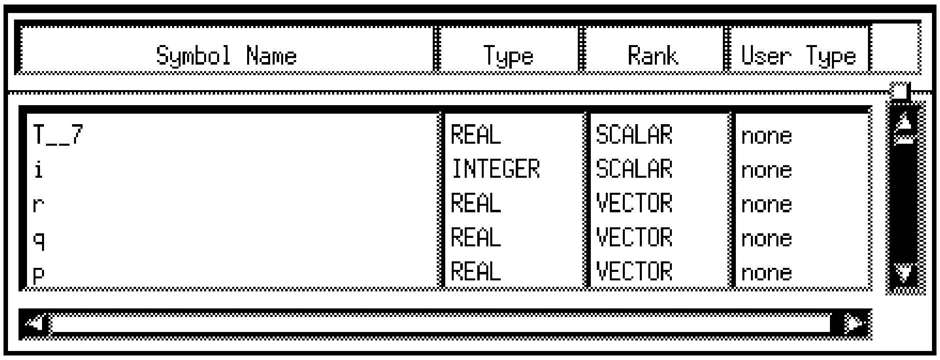

Figure 4.15: Window for displaying the symbol information.

4.5.2 Displaying MATLAB Code

During interactive restructuring process, the developer needs to view both the source program and the transformation patterns. At this point in the FALCON environment, however, these two items are maintained in the internal AST format. In the initial FALCON environment, the only system capable of converting the AST back into human-readable code was the Fortran 90 code generator.

However, as some MATLAB operations require multiple statements in Fortran 90, this can make the code and the transformation pattern difficult to read and understand. Since the source code is in MATLAB, it is assumed the developer would be familiar with MATLAB and the MATLAB source program. It was decided, therefore, that MATLAB would be the preferred language for representing the code and the transformation patterns.

The system for converting the AST back to the MATLAB language, the MATLAB code generator, is based upon the Fortran 90 code generator. The primary differences between the code generators is that the MATLAB version does necessarily need to generate the dynamic statements, nor does it need to generate variable declarations, but it does need to support the new transformation statements.

Since MATLAB handles the dynamic assignments within the source program, the dynamic code inserted by the Program Analysis System is unnecessary in a MATLAB version of the program. However, as the output language targeted by the developer may require the dynamic statements, it could be misleading to the developer not to represent these statements. Therefore, it was decided to implement a flag within the code generator to indicate when the dynamic statements should be displayed. At this time the flag is set such that the dynamic statements are always displayed, but in the future this flag could be reset if MATLAB is the target language.

A similar argument could be used with respect to variable declarations and whether they should be represented in the source program. In this case, however, it was felt that it was better to omit the declarations from the source window and, instead, provide a separate window for the variable information. This is in part due to the variable declarations not being represented as nodes in the AST but are instead maintained in a separate symbol table.

As new statements were added to the MATLAB language to support the transformation patterns, support was also added to the MATLAB code generator to display these statements. This allows the code generator to display the transformation patterns correctly, along with the BODY statements, the WHERE blocks, and the IN blocks. In contrast to the source program, where the declarations are not displayed, in the transformation patterns the explicit variable declarations are represented as statements in the AST and, therefore, it was decided to display these statements. This both avoids the need for a variable display window for each transformation pattern and it allows the developer to easily differentiate between similar transformation patterns that have variables with different intrinsic types or ranks.

At this time, the primary use of the MATLAB code generator is in support of the interactive restructuring system, not to generate output code. Future work on the system could optimize the MATLAB code generated.

4.5.3 Applying Restructuring Techniques

Along with viewing the program information, the other main operation performed by the developer is the application of the restructuring techniques. Since the only technique that has been described so far is the application of a transformation pattern to the source program, that is the only technique that will be discussed at this time.

The transformation patterns can be applied in two different manners, one being a very selective approach and the other being a very broad-based approach. With the selective approach, the developer applies a single pattern to a selected portion of the source code. On the other hand, with the broad-based approach, every pattern is applied to all of the source code.

To utilize the selective approach, the developer must first choose a transformation pattern and the source code where it is to be applied. The transformation pattern is chosen by examining the available transformations in the main window and selecting any statement within the desired transformation. This will highlight the entire transformation, with only one transformation being selected at a time. The transformation pattern is applied to a contiguous block of the statements in the source code. This block is chosen by pressing the mouse button while the cursor is on the first statement and not releasing the mouse button until the cursor is over the last statement. Only one block of source code statements can be selected at a time, upon a new selection the previous selection is disregarded. After the source and transformation have been selected, the pattern is applied by choosing the `apply selected pattern` option of the `Apply Menu`. (The `Apply Menu` is the second button of the main window presented in Figure 4.14.) The application of the transformation occurs as was previously described. It should be noted that the selection of the source code by the developer does not guarantee the transformation will actually be performed, as it is still necessary for the source code to match the transformation pattern.

By choosing the `apply all patterns` option of the `Apply Menu`, all transformation patterns will be applied to all statements in the source code. One at time, each transformation is applied to all the statements in the source program. As a result, the order of the transformation patterns can determine which transformations are actually performed, with the first transformation to match being applied. Also, the results of the previous transformations will be tested for matches with the subsequent transformation patterns. After the application of the transformation pattern, or patterns, the display of the source code is updated to reflect any changes and the display of the variable information is also updated.

It is possible to have the transformation patterns applied automatically without using the user interface. If the `-match` flag is placed on the command line when the compilation is started, each of the given transformation patterns will automatically be applied, in order, to all of the source code statements. (This is the same as choosing the `apply all patterns` option in the `Apply Menu`.)

After all the transformation patterns have been applied, the environment will continue with the code generation.

Chapter 5

TRADITIONAL RESTRUCTURING COMPILER TRANSFORMATIONS

In the previous chapter, the idea of using pattern-based transformations for interactive restructuring was presented, including details of how the system is implemented within the FALCON environment. Using the transformation language, patterns can be written to describe a variety of transformations. The transformation patterns supported by the system can be roughly divided into two groups: those similar to transformations performed by traditional restructuring compilers and those similar to transformations performed by numerical analysts which are based on more algebraic information. In this chapter, the transformations based on traditional restructuring compiler techniques will be examined, while in Chapter 6, the algebraic transformations will be discussed. The transformations in this chapter are divided into two classes, traditional code restructuring transformations and routine selection techniques.

5.1 Traditional Transformations

The initial design of the Interactive Restructuring System was to support algebraic transformations and was not aimed at replacing traditional restructuring compilers. However, due to the capabilities that were necessary in the restructuring system to support the algebraic transformations, a certain number of traditional compiler transformations are also possible. These transformations are included in the discussion as they can be used to enhance other interactive techniques and, in some cases, are useful for exposing the underlying algebraic algorithm within a code.

These types of transformations operate primarily on the statements, recognizing control structures and variable types, and rarely utilize algebraic knowledge about the operations being performed. For instance, transformations such as loop unrolling and loop fusing can be described using simple transformation patterns, as shown in Figure 5.1 and Figure 5.2.

```
REPLACE
    INTEGER i,l,u
    BODY B1

    for i = l:u
        <BODY>(B1)
    end
WITH
    for i = l:2:u
        <BODY>(B1)
        IN
            <BODY>(B1)
        REPLACE
            i
        WITH
            i+1
        END
    end
END
```

Figure 5.1: Transformation pattern for loop unrolling.

```
REPLACE
    INTEGER i,l,u
    BODY B1, B2

    for i = l:u
        <BODY>(B1)
    end
    for i = l:u
        <BODY>(B2)
    end
WITH
    for i = l:u
        <BODY>(B1)
        <BODY>(B2)
    end
END
```

Figure 5.2: Transformation pattern for loop fusion.

It should be noted, however, that the implementation of the transformations with such simple patterns may not always guarantee correct results. For instance, in the loop unrolling pattern, it is assumed that the number of iterations is always even and in the loop fusion example it is assumed that no data dependences will be violated when the bodies of the two loops are merged. These patterns are useful only when the developer is willing to check such conditions by hand, but in general, the dependence conditions should be placed in the pattern. An example of this is the loop splitting transformation pattern shown in Figure 5.3, where the loop is split into two loops following the rank-1 update.

Another transformation that can be used to simplify the body of loops is the removal of conditional statements that are used for initial conditions. For example, the transformation pattern for handling an `if` statement that is executed only during the first loop iteration is presented in Figure 5.4. In the resulting code, the initial iteration is executed before entering the loop and the loop body no longer needs to perform the test of the initial condition.

These types of transformations, by themselves, do not typically have much effect on the performance of the code. Instead, they are generally used so that subsequent transformations can be applied.

```
REPLACE
    INTEGER k, l, u, n
    REAL C(n,n), d(n), e(n)
    BODY A(k), B(k)

    for k = 1:u
        <BODY>(A)
        <WHERE>
            A DOESNT_READ B(1:k-1)
            A DOESNT_WRITE A(1:k-1)
        <END>
        C = C + d * e';
        <WHERE>
            C DOESNT_READ B(1:k-1)
        <END>
        <BODY>(B)
        <WHERE>
            B DOESNT_READ A(k+1:u)
            B DOESNT_READ C
        <END>
    end
WITH
    for k = 1:u
        <BODY>(A)
        C = C + d * e';
    end
    for k = 1:u
        <BODY>(B)
    end
END
```

Figure 5.3: Transformation pattern for loop splitting.

```
REPLACE
    INTEGER k, l, u
    BODY B1, B2, B3, B4

    for k = l:u
        <BODY>(B1)
        if( k == 1 )
            <BODY>(B2)
        else
            <BODY>(B3)
        end
        <BODY>(B4)
    end
WITH
    k = l;
    if( k <= u )
        <BODY>(B1)
        <BODY>(B2)
        <BODY>(B4)
    end
    for k = l+1:u
        <BODY>(B1)
        <BODY>(B3)
        <BODY>(B4)
    end
END
```

Figure 5.4: Transformation pattern for removing initial condition.

5.2 Subroutine Replacement

The code generators within the FALCON environment translate each MATLAB operation to a specific subroutine from the libraries described in Chapter 3. This translation, however, does not make full use of the libraries as it is limited to only recognizing unary and binary operations. Also, when there are multiple possible routines for supporting the operation, the selection of the routine is hard-coded into the code generator. These choices were made within the code generators so that the resulting code would utilize many of the same routines as the MATLAB environment, when it could be determined which routine MATLAB uses. By doing this, the numerical results of the compiled code are typically very close to those from the MATLAB execution.

During the interactive development, however, the developer may wish to modify the translation process, even if it will result in slightly different numerical results. For instance, the developer may wish to call a single routine that will perform multiple operations or target a routine from a different library for performing the same operation.

5.2.1 Using a Single Routine for Multiple Operations

Since the FALCON environment deals with expressions by dividing them into triplets, an expression such as

$$y = \alpha Ax + \beta b$$

will be divided into the following operations:

$$\begin{aligned} t_1 &= \alpha * A \\ t_2 &= t_1 * x \\ t_3 &= \beta * b \\ y &= t_2 + t_3, \end{aligned}$$

where upper-case letters represent matrices, lower-case letters represent vectors, and Greek letters represent scalars. This execution will utilize a single call to the BLAS matrix-vector multiplication routine, `dgemv`, to calculate t_2. However, this approach ignores the capabilities of the `dgemv` routine that can compute not only

$$y = A * x$$

but can also compute

$$y = \alpha * A * x + \beta * y.$$

This allows the first expression to be calculated by copying b to y and then utilizing a single call to `dgemv`.

Several BLAS routines support this ability to compute more than a single operation, with Table 5.1 containing some of the more common operations. Transformation patterns can be written to recognize these expressions and replace the expressions with the call to the appropriate subroutines. For instance, the transformation pattern to recognize the expression from the previous example and replace it with a call to `dgemv` is shown in Figure 5.5.

BLAS Routine	Supported Expressions
dgemv	$y = \alpha * A * x + \beta * y$
	$y = \alpha * A^T * x + \beta * y$
dger	$A = \alpha * x * y^T + A$
dgemm	$C = \alpha * A * B + \beta * C$
	$C = \alpha * A^T * B + \beta * C$
	$C = \alpha * A * B^T + \beta * C$
	$C = \alpha * A^T * B^T + \beta * C$

Table 5.1: Expression supported by BLAS routines.

```
REPLACE
    INTEGER n
    REAL A(n,n), x(n), b(n), y(n), alpha, beta

    y = alpha * A * x + beta * b;
WITH
    SUBROUTINE dgemv(P_IN, P_IN, P_IN, P_IN, P_IN, P_IN, P_IN, ...
       P_IN, P_IN, P_IN_OUT, P_IN)

    y = dgemv('N', DIM(A,1), DIM(A,2), alpha, A, DIM(A,1), x, 1, ...
       beta, b, 1);
END
```

Figure 5.5: Transformation pattern for $y = \alpha Ax + \beta b$.

It is hoped that the use of a single subroutine call to perform multiple operations would provide better performance. However, in practice, the variation in performance is small, with some transformations actually slowing the performance. For example, consider the following matrix-matrix multiplication expression:

$$D = \alpha * A * B + \beta * C.$$

A sample program to calculate this expression using matrices of size 400×400 is shown in Figure 5.6. The transformation pattern for performing this operation with a single call to `dgemm` is shown in Figure 5.7. The MATLAB program was compiled using the FALCON environment both with and without the `dgemm` transformation. The compiled code was run five times on Sun SPARCstation

```
n = 400;
A = rand(n);
B = rand(n);
C = rand(n);
alpha = rand(1);
beta = rand(1);
t1 = cputime;
D = alpha * A * B + beta * C;
t2 = cputime;
TIME = t2 - t1
```

Figure 5.6: MATLAB program for testing `dgemm`.

```
REPLACE
    INTEGER n
    REAL A(n,n), B(n,n), C(n,n), D(n,n), alpha, beta

    D = alpha * A * B + beta * C;
WITH
  SUBROUTINE dgemm( P_IN, P_IN, P_IN, P_IN, P_IN, P_IN, ...
     P_IN{a}, P_IN, P_IN{b}, P_IN, P_IN, P_IN_OUT, P_IN )
  D = dgemm('N', 'N', DIM(A,1), DIM(A,2), DIM(B,2), ...
     alpha, A, DIM(A,1), B, DIM(B,1), beta, C, DIM(A,1));
END
```

Figure 5.7: Transformation pattern for `dgemm` expression.

10, with the best run without the transformation taking 11.40 seconds of CPU time and the best run with the transformation taking 12.15 seconds of CPU time, a slowdown of 6.5%. The `dgemm` subroutine was then optimized slightly to avoid testing the value of the β argument and the code with the transformation was run again, with the best time being 11.25 seconds of CPU time, for an improvement of 1.3% over the original code. These tests indicate that the Fortran 90 code generated is calculating the expression in nearly the same time as the BLAS subroutines and that this transformation is not necessary. However, if an optimized version of the BLAS subroutines is available, or if the performance of the target language was less than the BLAS, then such transformations could prove useful.

5.2.2 Targeting Specific Library Calls

The second type of routine replacement is used when there are multiple routines that can perform the same operation. The FALCON code generators translate each operation into routine calls based on hard-coded rules. The developer, however, may wish to utilize other routines to perform the operations. These can be alternative routines in the libraries already targeted by the FALCON environment, for instance there is a Gaussian elimination routine in both LINPACK and LAPACK, or these can be routines from a library provided by the developer.

Furthermore, the developer is not constrained to using routines that have a one-to-one correspondence with the operations. For instance, a matrix-matrix multiplication operation can be represented using the `dgemm` routine from the Level 3 BLAS, a loop of `dgemv` routines from the Level 2 BLAS, or a doubly-nested loop of `ddot` routines from the Level 1 BLAS. The transformation patterns for performing these library substitutions are presented in Figure 5.8.

To show the effects of using these different subroutine replacements, a short MATLAB program was written to multiply two matrices, each of size 200×200, as shown in Figure 5.9. This MATLAB program was compiled using the FALCON environment, both without any transformations and then with each of the three transformation patterns in Figure 5.8. The resulting programs were then compiled using the VAST90 compiler with the -O3 optimization level and run on a Sun SPARCstation 10. The results of these tests are presented in Table 5.2, with the CPU execution time being presented for the best out of five runs for each test.

Program	Execution Time (seconds)
No Transformations	1.52
Level 1 BLAS	7.26
Level 2 BLAS	1.41
Level 3 BLAS	1.52

Table 5.2: Execution times of matrix multiplication tests.

Most of the tests executed in approximately the same time, with the Level 2 BLAS code being slightly faster than the Level 3 BLAS code. As the Level 3 BLAS code utilizes the same BLAS subroutine as the no transformations code, `dgemm`, the comparison of the execution time from these two tests would indicate whether the transformation incurs any overhead in the execution. Since the times for these two codes are the same, no overhead was occurred by the transformation. The Level 1 BLAS program, however, was significantly slower than the other codes, taking over 4.5 times longer to run.

Upon examination of the transformation pattern for the Level 1 BLAS, in Figure 5.8, it was noted that the arguments to the inner product subroutine are both array sections. To pass the array argument, `a(i,:)`, a copy of the data must be made in order for the elements to be contiguous. The copying of the data can be avoided by passing only the first element of the array section and using the correct stride to access the subsequent elements. The transformation pattern to generate this alternative approach is shown in Figure 5.10. Using this alternative transformation, the execution time using the same inner product routine was dropped from 7.26 seconds to 2.80 seconds, which is approximately 1.8 times slower than the other methods.

Level 1 BLAS:

```
REPLACE
    INTEGER n
    REAL a(n,n), b(n,n), c(n,n)
    c = a * b;
WITH
    INTEGER i,j
    FUNCTION ddot( P_IN, P_IN, P_IN, P_IN, P_IN ){1,1}
    for i = 1:DIM(a,1)
        for j = 1:DIM(b,2)
            c(i,j) = ddot( DIM(a,2), a(i,:), 1, b(:,j), 1 );
        end
    end
END
```

Level 2 BLAS:

```
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{a}, ...
       P_IN, P_IN{b}, P_IN, P_IN, P_OUT{DIM(a, 1), 1}, P_IN)
    for i = 1:DIM(b,2)
        c(:,i) = dgemv('N', DIM(a,1), DIM(a,2), ...
           1.D0, a, DIM(a,1), b(:,i), 1, 0.D0, 1);
    end
END
```

Level 3 BLAS:

```
REPLACE
    INTEGER n
    REAL a(n,n), b(n,n), c(n,n)
    c = a * b;
WITH
  SUBROUTINE dgemm( P_IN, P_IN, P_IN, P_IN, P_IN, P_IN, P_IN{a}, ...
     P_IN, P_IN{b}, P_IN, P_IN, P_OUT{DIM(a,1),DIM(b,2)}, P_IN )
  c = dgemm('N', 'N', DIM(a,1), DIM(a,2), DIM(b,2), ...
     1.D0, a, DIM(a,1), b, DIM(b,1), 0.D0, DIM(a,1));
END
```

Figure 5.8: Matrix multiplication using the three BLAS levels.

```
n = 200;
A = rand(n);
B = rand(n);
t1 = cputime;
D = A * B;
t2 = cputime;
TIME = t2 - t1
```

Figure 5.9: MATLAB test program for matrix multiplication.

```
REPLACE
    INTEGER n
    REAL a(n,n), b(n,n), c(n,n)

    c = a * b;
WITH
    INTEGER i,j
    FUNCTION ddot( P_IN, P_IN, P_IN, P_IN, P_IN ){1,1}
    for i = 1:DIM(a,1)
        for j = 1:DIM(b,2)
            c(i,j) = ddot( DIM(a,2), a(i,1), DIM(a,1), b(1,j), 1 );
        end
    end
END
```

Figure 5.10: Alternative transformation pattern for the Level 1 BLAS

Chapter 6

BASIC ALGEBRAIC TRANSFORMATIONS

The second group of transformation patterns to be discussed are those which utilize basic algebraic information about the operations being performed to restructure the expressions. Traditional compilers utilize the algebraic properties of associativity, commutativity, and distributivity to optimize scalar expressions. With the transformation patterns, similar rules can be used to optimize expressions involving vectors and matrices.

The restructuring of vector and matrix expressions using algebraic properties may sometimes have the same effect as the application of traditional restructuring techniques. However, during the interactive application, the algebraic transformations may provide a more convenient form for the user than the traditional loop-based techniques, even when the results are the same. For example, the conversion of an algebraic algorithm from a row-sweep to a column-sweep algorithm can typically also be represented as a loop-interchange. It is the support for such an overlap that allows the system to be molded to meet the developer's needs rather than forcing the developer to use a single type of transformation.

When applying the transformations, the developer should realize there are usually trade-offs between performance and reliability. Certain transformations that can be used to increase performance may also decrease the numerical stability of an algorithm, with the reverse also being true. During the development, therefore, the developer may need to test the performance and the stability as the transformations are applied to verify both qualities are maintained within reasonable limits. This is one instance where the rapid prototyping capabilities of the environment can be used effectively.

The transformations that can be built using the algebraic information are based on several different approaches. These approaches include using the same operations but changing the order, as well as changing the operations being performed. Both of these approaches will be discussed in this chapter.

6.1 Optimizing the Order of Operation Evaluation

In the MATLAB environment, the execution order for the operations in an expression is based upon the usual precedence rules, followed by a left-to-right ordering of operations with the same precedence. This approach is very predictable and is similar to those used by almost all compilers. For an expression with scalar values, the ordering typically has little effect on the performance as the number of numerical operations being performed is not affected by the ordering of the operations. (This ignores the effects on performance due to differences in the amount of temporary storage required and due to differences in register allocation.) When expressions contain vectors and matrices, however, the ordering of the operations can have a significant effect on the number of operations.

As an example, consider the evaluation of the following expression:

$$xy^T A,$$

with the two vectors, x and y, being n elements long and the matrix A being $n \times n$. This computation will require $\mathcal{O}(n^3)$ operations if evaluated from left-to-right, but would require only $\mathcal{O}(n^2)$ operations if evaluated from right-to-left. Two expressions similar to this arise during the development of the QR factorization using Householder transforms. The updates of the Q and R matrices are performed by multiplying by the Householder transform, H, defined as:

$$H_i = I - \frac{2}{\|v_i\|} v_i v_i^T.$$

Figure 6.1 shows the MATLAB code for the body of the factorization loop with this definition distributed within both updates. The two update operations, of Q and R, can be optimized using

```
for i = 1:n-1
   na = norm(R(i:n,i));
   v = zeros(n,1);
   v(i:n) = R(i:n,i);
   v(i) = v(i) - na;
   nv = v'*v;
   R = R - ((2/nv) * v * v') * R;
   Q = Q - Q * ((2/nv) * v * v');
end
```

Figure 6.1: Loop body from QR factorization.

the transformations presented in Figure 6.2. This QR program was translated into Fortran 90 using the FALCON environment, both with and without the transformations. The Fortran 90 codes were compiled using VAST 90 and were run five times on a SPARCstation 10 to find the best time for

```
REPLACE
    INTEGER n
    REAL x(n), A(n,n), B(n,n), alpha

    B = ( alpha * x * x' ) * A;
WITH
    B = alpha * x * ( x' * A );
END
```

```
REPLACE
    INTEGER n
    REAL x(n), A(n,n), B(n,n), alpha

    B = A * ( alpha * x * x' );
WITH
    B = ( A * alpha * x ) * x';
END
```

Figure 6.2: Transformation patterns for optimizing QR factorization.

factoring a 101×101 matrix. Without the optimization, the execution time was 29.10 seconds and with the optimization, the execution time was 3.10 seconds, for a speedup of 9.4.

6.2 Use of Equivalent Operations

Instead of reordering the operations, an alternative approach for improving the computation of an expression is to replace the operations with an equivalent operation or operations. Some replacements can be effective for improving the performance, such as replacing expensive MATLAB operations with less costly operations, and other replacements may be used to improve the numerical stability of the code. As with most transformations, care must be taken to not destroy the numerical stability when the operations being performed are modified.

6.2.1 Decomposing Operations

One class of these transformations is the decomposition of a matrix operation into lower-level operations. For instance, the following matrix-matrix multiplication operation,

$$A = B * C,$$

can be decomposed into multiple matrix-matrix operations, into matrix-vector operations, and into vector-vector operations:

$$\begin{aligned} A(i:l, j:k) &= B(i:l,:) * C(:, j:k), \\ A(:, j) &= B * C(:, j), \text{ and} \\ A(i, j) &= B(i,:) * C(:, j). \end{aligned}$$

The transformation patterns that would generate the last two operations are presented in Figure 6.3. These transformation patterns were applied to the matrix-multiplication program in Figure 5.6 using the FALCON environment. The resulting codes, along with the original code, were compiled using VAST 90 and run on a SPARCstation 10. Each code was run five times, with the best CPU

Matrix-Vector Product

```
REPLACE
    INTEGER n
    REAL a(n,n), b(n,n), c(n,n)
    c = a * b;
WITH
    INTEGER i
    for i = 1:DIM(b,2)
        c(:,i) = a * b(:,i);
    end
END
```

Vector-Vector Product

```
REPLACE
    INTEGER n
    REAL a(n,n), b(n,n), c(n,n)
    c = a * b;
WITH
    INTEGER i,j
    for i = 1:DIM(a,1)
        for j = 1:DIM(b,2)
            c(i,j) = a(i,:) * b(:,j);
        end
    end
END
```

Figure 6.3: Transformation patterns for decomposing matrix-matrix multiplication.

Program	Execution Time (seconds)
No Transformations	1.52
Vector-Vector Product	7.55
Matrix-Vector Product	1.43

Table 6.1: Execution times for matrix multiplication after algebraic transformations.

execution time for each code presented in Table 6.1. These results are similar to those presented in Table 5.2, when the matrix-multiply subroutine was replaced. As in those previous results, the matrix-vector multiplication provides slightly better performance than the original code and the vector-vector multiplication is much worse. In this case, however, it is not possible to optimize the routine call for the vector-vector product, as done in the previous chapter, since the pattern only changes the operation and does not insert the subroutine call. On the other hand, it is possible to use a separate transformation which recognizes the vector-vector product and maps it to the optimized subroutine call.

6.2.2 Combining Operations

In addition to decomposing operations, transformation patterns can also be used to combine operations into a single primitive. For instance, this could be used for recognition of a loop of matrix-vector products as a matrix multiplication or for recognizing that a loop of scalar multiplications can be replaced by a vector-vector product. The MATLAB code segment in Figure 6.4 was found in an Incomplete Cholesky factorization program. This loop can be replaced by the following vector-vector product:

$$s = L(j, 1 : j - 1) * L(j, 1 : j - 1)^T.$$

```
s = 0;
for k = 1:j-1;
  s = s + L(j,k)*L(j,k);
end
```

Figure 6.4: MATLAB code segment from Incomplete Cholesky factorization program.

The transformation pattern to perform this modification is shown in Figure 6.5. The FALCON

```
REPLACE
    INTEGER i, j, k, n
    REAL s, A(n,n)
    BODY B
    s = 0;
    <BODY>(B)
    <WHERE>
      B DOESNT_WRITE s
    <END>
    for j = 1:k
        s = s + A(i,j) * A(i,j);
    end
WITH
    s = 0;
    <BODY>(B)
    if( k > 0 )
       s = A(i,1:k) * A(i,1:k)';
    end
END
```

Figure 6.5: Transformation pattern for Incomplete Cholesky factorization.

environment was used to generate Fortran 90 code for both the original program and for the program in which the loop was replaced, along with the replacement of a similar loop later in the program. The Fortran 90 programs were compiled with VAST 90 and each program was run five times on a SPARCstation 10. The best CPU execution time from the five runs for each test are shown in Table 6.2. Notice how the execution time more than doubled after the transformations

Program	Execution Time (seconds)
No Transformations	0.68
After Transformations	1.79

Table 6.2: Execution times for Incomplete Cholesky factorization.

were applied. As was encountered in previous examples, the overhead of passing array sections to functions can be a serious detriment to performance. If the additional transformation, shown in Figure 6.6, is applied to avoid the copy of the array sections, the performance drops to 0.65 seconds, for a very small improvement over the original code of about 4.5%. However, this transformation

```
REPLACE
    INTEGER i, k, n
    REAL s, A(n,n)
    s = A(i,1:k) * A(i,1:k)';
WITH
    FUNCTION ddot( P_IN, P_IN, P_IN, P_IN, P_IN ){1,1}
    s = ddot( k, A(i,1), DIM(A,1), A(i,1), DIM(A,1) );
END
```

Figure 6.6: An additional optimization for the Incomplete Cholesky factorization.

has a dramatic effect upon the performance of the code within the MATLAB interpreter. The MATLAB code, and the optimized MATLAB code, were both evaluated within the MATLAB environment on a SPARCstation 10, with each test being run five times. The best result of the original program took 56.10 seconds of CPU time while the best result of the optimized program took only 10.08 seconds, for a speedup of 5.57.

6.2.3 Replacing Expensive Operations

A third type of operation replacement involves the recognition of expensive primitives when less costly alternatives are available. This type of replacement would typically have the largest effect on operations such as matrix inverses and matrix solves.

One such instance is when the matrix inverse is used in an expression to solve multiple vectors. Consider the following expression:

$$H = (S^T * S + \nu * I)^{-1} * S^T,$$

which was extracted from a synthetic aperture radar code. The number of solves to be performed is determined by the number of rows in S, as S^T is multiplied by the inverse of the expression in parentheses. An alternative approach is to transform the multiplication by the inverse into a MATLAB solve operation:

$$H = (S^T * S + \nu * I)\backslash S^T.$$

This modification can be performed using the transformation pattern in Figure 6.7. The MATLAB loop containing the inverse was translated into Fortran 90 using the FALCON environment, both with and without the transformation. These codes were then compiled using VAST 90 and each code was run five times on a SPARCstation 10 to find the best CPU execution time. In both of

```
REPLACE
    INTEGER n
    REAL A(n,n), B(n,n), C(n,n)
    A = inv(B)*C';
WITH
    A = B \ C';
END
```

Figure 6.7: Transformation pattern for converting an inverse to a solve.

the tests that were run, the matrix solved was of size 64×64 and inside a loop that ran for 64 iterations. In the first test, only two vectors were solved and, in the second test, 64 vectors were solved, with the results of these tests presented in Table 6.3.

Program	Execution Time (seconds)	
	2 vectors	64 vectors
Using Inverses	92.62	99.35
Using Solves	3.47	98.50

Table 6.3: Experiments replacing inverses with solves.

As expected, when a small number of vectors were solved, the use of the solve operations was significantly faster. When the number of solves being performed was equal to the size of the matrix being inverted, the two solution methods took roughly the same time. However, if the number of vectors being solved becomes greater than the size of the matrix, then the inversion of the matrix becomes the faster method. It should also be noted that the current method for performing repeated solves within the code generated by the FALCON environment is to call the matrix-vector solve routine multiple times. This could be improved by performing a factorization of the system and then using the factored system to perform the subsequent solves.

Chapter 7

UTILIZATION OF STRUCTURE INFORMATION

One limitation of the FALCON environment, as it has been described, is that it is constrained to the data structures and data types that are supported by MATLAB. This limits the data structures to scalars, dense vectors, and dense rectangular matrices. There are, however, some MATLAB operations that generate specific matrix structures which could be utilized in optimizing the code. These structures include those based both on the location of the nonzero elements, such as triangular matrices and banded matrices, and those based on the element values, such as symmetric matrices. In fact, for certain expensive operations, the MATLAB environment performs run-time tests on the operands to recognize these types so that optimized algorithms can be utilized.

In the FALCON environment, similar run-time tests were placed in the support routines to replicate the functionality of MATLAB. In addition, DeRose began an effort to recognize the specialized structures at compile time within the FALCON environment by examining the semantics of the MATLAB program. However, this effort never reached the stage where it could be used to influence the generation of code. Furthermore, even if it had been completed, it still would have been constrained to the few matrix structures that are known to be generated by MATLAB-supplied functions.

An alternative approach is to use the concepts developed through the transformation patterns to recognize matrix structures within dense matrices and perform optimizations and code generation based on these matrix structures. Not only does this approach allow the structures used by MATLAB to be recognized, but it also supports any matrix type for which the developer can both write a pattern to recognize the matrix type and provide the routines to operate on the type. Extending this idea to additionally support vectors and scalars, the developer is given the power to create *user-defined types*.

In this chapter, the discussion will focus on utilizing user-defined types that can be represented with MATLAB's data structures. In the following chapter, these ideas will be extended to include the definition of the data representation.

7.1 The Use of User-Defined Types

One of the main goals of the system is to provide tools to develop better codes from numerical algorithms. In this part of the system, that goal translates into providing tools that allow the developer to include additional information about the matrices and to utilize that information during subsequent transformations and code generation.

For example, the restructuring system's ability to recognize triangular matrices can significantly reduce the time necessary to perform matrix multiplication and matrix solves. MATLAB recognizes the usefulness of this and includes run-time tests during solve operations to determine if the matrix is triangular, thereby reducing an $\mathcal{O}(n^3)$ operation to an $\mathcal{O}(n^2)$ operation. However, this run-time test can itself require an $\mathcal{O}(n^2)$ data accesses. Therefore, if the matrix is known to be triangular at compile time, the triangular solve can be called without performing the run-time test. Furthermore, once the structure of the matrix is known, all operations with the matrix can be optimized while MATLAB only utilizes the test for expensive operations such as solves and does not use it for operations such as matrix multiplication.

It is important that the system recognize the matrix structures not just when they are first generated, but to also recognize how they are propagated through the code and how different matrix structures interact. For instance, the addition of two lower triangular matrices will generate another lower triangular matrix, while adding a lower triangular matrix and a diagonal matrix will generate a lower triangular matrix. The ability to recognize the hierarchy that exists between the matrix structures can also improve their use within the system.

In order to utilize user-defined types effectively, the FALCON environment must be able to perform the following four tasks:

1. the recognition of user-defined types,
2. the propagation of user-defined types,
3. transformations using the user-defined types, and
4. the generation of code using the user-defined types.

The definition of the user-defined type, therefore, must provide the necessary information to allow the environment to perform these tasks. This was accomplished by extending the concept of the transformation patterns in two ways. First, a *structure* definition construct was added to the transformation pattern language to support the recognition and propagation of user-defined types. Second, the transformation patterns were extended to include the use of user-defined types during variable declaration and matching to support the transformations and code generation.

7.2 User-Defined Type Definition

The STRUCTURE definition was added to the transformation language to both describe how the user-defined type can be identified and to describe how the type is propagated. The format of the STRUCTURE definition is provided in Figure 7.1. This definition contains two blocks, first is the INHERITS block and the second is the ASSERTIONS block. The INHERITS block contains a list of user-defined types from which this type will inherit characteristics and will be examined further during the discussion on propagation of user-defined types. The ASSERTIONS block consists of assertions identified by the `ASSERT` construct. This construct is very similar to the transformation patterns used in the previous chapters, with the exception that the `WITH` part of the construct can only contain variable declarations. This implies that the `ASSERT` will only modify variable attributes and cannot modify the statements within the code. The `ASSERT` constructs within the ASSERTIONS block are used both to recognize the user-defined types and to define the propagation of the user-defined types.

```
STRUCTURE(structure name)
    INHERITS
        parent structure
        ...
    ASSERTIONS
        ASSERT
            variable declarations
            MATLAB statements
        WITH
            variable declarations
        END
        ...
END
```

Figure 7.1: Format of the structure definition.

7.2.1 Patterns for Recognizing User-Defined Types

The `ASSERT` construct contains two sections, the ASSERT section and the WITH section. The ASSERT section is used to identify a user-defined type by defining a pattern that will recognize the use of specific variable types as operands to a MATLAB operation. The WITH section is used to update the state of the variables when the ASSERT section is matched. To utilize the user-defined types, the variable declarations were extended to allow an optional user-defined type field to be placed after the dimensions of the variable:

variable(*dimensions*){`TYPE=`*structure name*}.

The type field can be added to variable declarations in both sections of the ASSERT construct. When the type field is present in the ASSERT section, the pattern-matching algorithm will find a match only when the variable is of the correct user-defined type. When the pattern in the ASSERT section matches the source code and there is a type field in the WITH section, then the state of the matched variables will be updated using the variable declarations present in the WITH section. Two example ASSERT patterns are presented in Figure 7.2, one for recognizing a diagonal matrix and a second for recognizing a lower triangular matrix.

```
ASSERT
   INTEGER n
   REAL A(n), B(n,n)
   B = diag(A);
WITH
   REAL B(n,n){TYPE=DIAG}
END
```

```
ASSERT
   INTEGER n
   REAL A(n,n), B(n,n)
   B = tril(A);
WITH
   REAL B(n,n){TYPE=LOWERT}
END
```

Figure 7.2: ASSERT patterns for recognizing two matrix structures.

The application of the ASSERT patterns is controlled by the user interface, from which the patterns can be applied both individually or all at the same time. To perform an individual application, the pattern is chosen by selecting any line within the pattern and the source code lines are selected using the mouse, as was previously described for applying a transformation pattern. The ASSERT pattern is then applied to the selected code statements by choosing the `apply selection` option from the `Apply Menu`. Alternatively, all ASSERT patterns can be applied to all source code statements by choosing the `apply all assertions` option from the `Apply Menu`. This will loop over the ASSERT patterns, applying each pattern in turn to all source code statements. Because the patterns are applied in the order in which they are found, an ASSERT pattern may try to match on a user-defined type that will not be set until later patterns are applied. As a result, it is sometimes necessary to apply the assertions more than once to recognize all user-defined types.

7.2.2 Propagation of User-Defined Types

When a variable with a user-defined type is used as an operand to an arithmetic operation or passed to a MATLAB function, it is possible that the result will also be a user-defined type. One method for recognizing these operations is to write ASSERT patterns to match on all possible operations and operand combinations. However, an alternative approach is to use rules to denote how user-defined types are propagated through the operations. Within the FALCON environment, the propagation of user-defined types is accomplished using both generic type propagation rules and ASSERT patterns from the user-defined type specifications.

The basic algorithm for propagating the user-defined types is:

```
For each statement
    If the statement performs an assignment then
        If any operand has a user-defined type
            Apply assertions from user-defined types of operands
            If none of the assertions matched then
                Apply generic rules
            End If
        End If
    End If
End For
```

Notice that by applying the assertions from the specification of the user-defined types before the generic rules, the type propagation can be customized for each type. Since the assertions are used for customizing the generic rules, the generic rules will be discussed first followed by the discussion of the assertions.

Generic Propagation Rules

The generic rules for propagating user-defined types were developed by examining matrix structures defined by the location of the nonzero elements. The arithmetic operations and MATLAB built-in functions were evaluated to determine if the result of the operation would preserve a nonzero structure if all operands or arguments had the same nonzero structure. For instance, the addition of two lower-triangular matrices will generate a lower triangular matrix. The MATLAB arithmetic operations that were determined to be structure-preserving are listed in Table 7.1 and the MATLAB

Operation	Description
$A + B$	Addition
$A - B$	Subtraction
$A.*B$	Element-wise multiplication
$A./B$	Element-wise division
$A.\wedge\beta$	Element-wise exponentiation
$-A$	Unary minus
$+A$	Unary plus

Table 7.1: Structure-preserving operations used for type propagation.

built-in functions that were determined to be structure-preserving are listed in Table 7.2. In addition to the items in these lists, however, other operations and functions may preserve the type for a specific user-defined type. For the type to be propagated through these operations, the type specification must contain assertions for the operation. Or, when these operations do not correctly propagate the type, an assertion must be present to prevent the incorrect propagation.

Function	Description
abs(A)	Absolute value of each element
ceil(A)	Round each element towards $+\infty$
fix(A)	Round each element towards zero
floor(A)	Round each element towards $-\infty$
full(A)	Dense version of a matrix
imag(A)	Imaginary part of each element
real(A)	Real part of each element
round(A)	Round each element
sign(A)	Sign of each element
sparse(A)	Sparse version of a matrix
sqrt(A)	Square-root of each element

Table 7.2: Structure-preserving built-in functions used for type propagation.

Assertions

The `ASSERT` patterns used during the propagation are the same as those used during the user-defined type recognition. As a result, all appropriate uses for an assertion may occur during the recognition phase and the assertion may not contribute any additional information during the propagation phase. However, due to the propagation of the type information to other variables, an assertion may match during the propagation phase when it did not previously match during the recognition phase. During the application of the assertions, the user-defined types must match exactly.

The assertions can be used to expand the list of operations and routines that propagate a specific type. For instance, the assertion in Figure 7.3 could be used to propagate lower triangular matrices through matrix multiplication, which would not occur using the generic rules. In addition, the

```
ASSERT
    INTEGER n
    REAL A(n,n), B(n,n){TYPE=LOWERT}, C(n,n){TYPE=LOWERT}
    A = B * C;
WITH
    REAL A(n,n){TYPE=LOWERT}
END
```

Figure 7.3: Assertion for propagating lower triangular matrices through matrix multiplication.

assertions can be used to propagate types when operations have operands of different types. As an example, the assertion in Figure 7.4 propagates the lower triangular type when a lower triangular matrix is multiplied with the transpose of an upper triangular matrix.

```
ASSERT
    INTEGER n
    REAL A(n,n), B(n,n){TYPE=LOWERT}, C(n,n){TYPE=UPPERT}
    A = B * C';
WITH
    REAL A(n,n){TYPE=LOWERT}
END
```

Figure 7.4: Assertion for propagating lower triangular matrices through matrix multiplication with mixed operands.

Inheritance

The discussion on the propagation has covered both operations where the operands have the same type and assertions where the types are explicitly listed. By understanding simple relationships between the user-defined types, however, the propagation algorithm can also handle operations with mixed types without the need for explicitly-typed assertions. To accomplish this, the FALCON environment supports the concept of inheritance between user-defined types.

Inheritance between two types is denoted by the child type specifying the user-defined types that are its parent. This is done by listing the type names within the INHERITS block of the structure definition. For example, a diagonal matrix type could inherit from both lower-triangular and upper-triangular matrix types, as shown in Figure 7.5. A type is allowed to inherit from multiple parents and will also implicitly inherit all types which the parent definitions inherit. This inheritance of types forms hierarchy among the types, with the root of the hierarchy being the MATLAB data structures, i.e. scalars, vectors, and rectangular matrices. The inheritance relationships for the diagonal matrix type in Figure 7.5 are shown in Figure 7.6.

```
STRUCTURE(DIAG)
    INHERITS
        LOWERT
        UPPERT
END
```

Figure 7.5: Inheritance structure for a diagonal matrix type.

The inheritance of types allows the application of the generic propagation rules to operands of different types when the types have an inherits relationship. The types present in the expression must follow a strict ordering of the inheritance by which all types are either the ancestors or descendants of all other types present. The *root* type is chosen to be that type which is the ancestor of all the other types present in the expression. The rules are then applied, treating all the operands as if they were the root type.

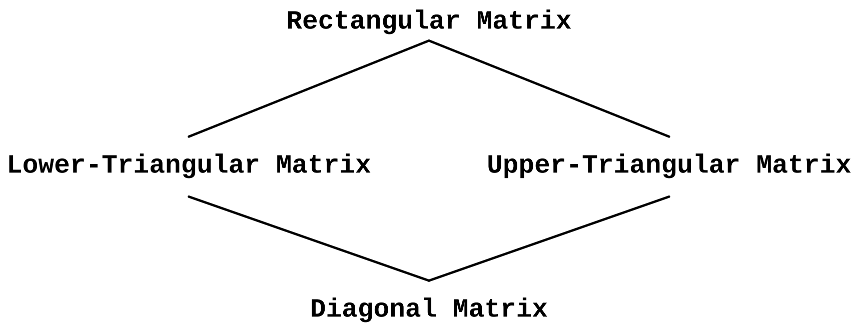

Figure 7.6: Inherits relationship for the diagonal matrix type.

7.3 Transformation Patterns Using User-Defined Types

Having developed a method for defining user-defined types and algorithms for recognizing and propagating the types, the next step is to use the types for applying transformations and performing code generation. Both of these tasks are performed using `REPLACE` constructs of the same format as described in Chapter 4 for defining transformation patterns, with the construct consisting of two sections, a REPLACE block and a WITH block. Similar to the `ASSERT` construct, an optional user-defined type field can be used in the variable declarations of both sections:

variable(*dimensions*){`TYPE=`*structure name*}.

It is this inclusion of the user-defined type fields that allow the developer to target the transformation patterns to specific types.

The previous discussion on transformation patterns presented two alternative uses of the patterns, using the patterns for traditional restructuring compiler techniques and using the patterns to perform basic algebraic transformations. When considering the user-defined types, the transformation patterns are also used for two very similar roles. First, they are used to perform transformations based on the algebra of the specified types and, second, they are used to generate code for operations that make use of the characteristics of the type.

7.3.1 Performing Transformations

In Chapter 6, the transformations were based on the algebraic properties of the data structures supported by MATLAB. However, using the user-defined types, additional data structures and matrix types are not only recognized, but transformations based on the algebraic properties of these types can be performed. In fact, the developer can utilize the user-defined type concept to encode state information within the system to support algebraic transformations without using the type information to generate code.

One possible type of transformation is the replacement of a general operation with a more specific operation based on characteristics of the user-defined type. For instance, the solve of a vector by a diagonal matrix can be replaced with an element-wise division by the diagonal, as shown in Figure 7.7.

```
REPLACE
    INTEGER n
    REAL a(n), B(n,n){TYPE=DIAG}, c(n)
    a = B \ c;
WITH
    a = c ./ diag(B);
END
```

Figure 7.7: Transformation pattern for optimizing a diagonal matrix solve.

Another possible type of transformation is the elimination of operations that are unnecessary due to the properties of the user-defined type. For example, if a matrix is symmetric, there is no need to take the transpose of the matrix. In this case, the transpose operation can be replaced by an assignment to the variable, with the transformation pattern to perform this shown in Figure 7.8. Similarly, matrix-matrix multiplication involving an identity matrix does not need to be performed and can be replaced by an assignment statement with the transformation patterns for this being shown in Figure 7.9.

```
REPLACE
    INTEGER n
    REAL A(n,n), B(n,n){TYPE=SYMMETRIC}
    A = B';
WITH
    A = B;
END
```

Figure 7.8: Transformation pattern for eliminating the transpose of a symmetric matrix.

7.3.2 Generating Code

The final issue with using user-defined types is that the provided code generators do not recognize these types. As a result, the code that is output by the code generators is based upon the MATLAB data structure associated with the variable. In order to generate code to take advantage of the types, transformation patterns must be applied to update the internal format before it is sent to the code generator.

To achieve this, transformation patterns must be written to convert the MATLAB operations

```
REPLACE
    INTEGER n
    REAL A(n,n), B(n,n)
    REAL I(n,n){TYPE=IDENTITY}
    A = B * I;
WITH
    A = B;
END
```

```
REPLACE
    INTEGER n
    REAL A(n,n), B(n,n)
    REAL I(n,n){TYPE=IDENTITY}
    A = I * B;
WITH
    A = B;
END
```

Figure 7.9: Transformation patterns for eliminating the multiplication by an identity matrix.

into subroutine calls for the user-defined type. Given the number of MATLAB operations and built-in functions, the transformations are typically only written for those MATLAB operations that are being used. It is felt that over time, the transformations for more of the operations would be added as they are needed. If a transformation is not provided for an operation, then the code generator will perform the default action.

An example program that utilizes several user-defined types is the Successive-Over-Relaxation (SOR) iterative solver. The MATLAB code fragment containing the primary computation statements is presented in Figure 7.10. The first two lines of this code fragment contains four operations

```
M =  w * tril(A,-1) + ...
   diag(diag(A));
N = -w * triu(A, 1) + ...
   (1.0 - w) * diag(diag(A));
b = w * b;
for iter = 1:max_it
   x_1 = x;
   x   = M \ (N*x+b);
   error = norm(x-x_1) / norm(x);
   if ( error <= tol ), break, end
end
```

Figure 7.10: MATLAB code fragment from a successive-over-relaxation iterative solver.

```
STRUCTURE(DIAG)
INHERITS
   UPPERT
   LOWERT
ASSERTIONS
   ASSERT
      INTEGER n
      REAL A(n,n), B(n,n)
      B = diag(diag(A));
   WITH
      REAL B(n,n){TYPE=DIAG}
   END
END
```

Figure 7.11: Structure definition for a diagonal matrix type.

that can be used to define specialized types. These operations are the `tril()` routine, the `triu()` routine, and the two occurrences of the `diag(diag())` routines. Applying the assertions for the diagonal matrix in Figure 7.11 and for the lower-triangular matrix in Figure 7.12, the matrix `M` is found to be assigned to a lower-triangular matrix in the first statement. Also, applying the assertions for the diagonal matrix in Figure 7.11 and for the upper-triangular matrix in Figure 7.13, the

```
STRUCTURE(LOWERT)
INHERITS
ASSERTIONS
   ASSERT
      INTEGER n
      REAL A(n,n), B(n,n)
      B = tril(A,-1);
   WITH
      REAL B(n,n){TYPE=LOWERT}
   END
END
REPLACE
   INTEGER n
   REAL X(n,n){TYPE=LOWERT}
   REAL y(n), z(n)
   z = X \ y;
WITH
   SUBROUTINE dtrsv(P_IN, P_IN,...
       P_IN, P_IN, P_IN{X}, P_IN,...
       P_IN_OUT, P_IN)
   z = dtrsv('L', 'N', 'N',...
       DIM(X,1), X, DIM(X,1), y, 1);
END
```

Figure 7.12: Structure definition for a lower-triangular matrix type.

```
STRUCTURE(UPPERT)
INHERITS
ASSERTIONS
   ASSERT
      INTEGER n
      REAL A(n,n), B(n,n)
      B = triu(A,1);
   WITH
      REAL B(n,n){TYPE=UPPERT}
   END
END
REPLACE
   INTEGER n
   REAL X(n,n){TYPE=UPPERT}
   REAL y(n), z(n)
   z = X * y;
WITH
   SUBROUTINE dtrmv(P_IN, P_IN,...
       P_IN, P_IN, P_IN{X}, P_IN,...
       P_IN_OUT, P_IN)
   z = dtrmv('U', 'N', 'N',...
       DIM(X,1), X, DIM(X,1), y, 1);
END
```

Figure 7.13: Structure definition for an upper-triangular matrix type.

matrix `N` is found to be assigned to an upper-triangular matrix in the second statement. Notice that determining the types of `N` and `M` required both the recognition of the types by assertions and the propagation of type by the generic propagation rules using inheritance. Once these types have been propagated, the calculation of the expression

```
x = M \ ( N*x + b );
```

within the loop body can be optimized with the level-2 BLAS primitives for triangular matrices: using the `dtrsv` routine for the lower-triangular solve and using the `dtrmv` routine for the upper-triangular matrix-vector multiplication.

The FALCON environment was used to translate the SOR routine into Fortran 90 twice, first in its original format using the dense arrays and a second time using the type definitions. Both of the resulting Fortran 90 programs were compiled using Vast 90 and then run five times on a SPARCstation 10. The best CPU execution time for the original program was 22.40 seconds while the program generated using the type definitions ran in 6.52 seconds, for a speedup of 3.4. As was expected, the use of the BLAS primitives optimized for the triangular matrices provided significantly better performance.

In addition to optimizing the operations within the loop body, three other operations during the initialization of the matrices can be optimized using the structure information:

`w * tril(A,-1)`,
`-w * triu(A,1)`, and
`(1.0 - w) * diag(diag( A ))`.

However, as these operations are performed only once, the performance effects are very small. The patterns for these optimizations were created, as shown in Figure 7.14, and a new version of the Fortran 90 code was generated using FALCON. The code was then compiled using Vast 90 and then run five times on a SPARCstation 10, with the best CPU execution time being 6.35 seconds. This only provided a 2.6% improvement over the previous time of 6.52 seconds.

7.4 Comments on Loading User-Defined Types

The implementation of the FALCON environment has limited support for including transformation patterns during the code generation process. The files containing all the patterns the developer may wish to use must be provided on the command line when the system is started. Currently, there is no means for the developer to reference additional patterns once the session has been started.

When inheritance is used with user-defined types, however, the FALCON environment knows that additional user-defined types are necessary to complete the definition of a type. Instead of forcing the developer to include the type hierarchy, the FALCON environment will automatically load the structure definition for all ancestors of a requested type, if the ancestor type has not already been loaded. To accomplish this, the FALCON environment expects the user-defined types to be stored in files that have the same name as the type and that the suffix of the file is `.PAT`. For instance, the type definitions from the example in the previous section would be stored in the files `DIAG.PAT`, `LOWERT.PAT`, and `UPPERT.PAT`. To use all three files, it is only necessary for the developer to reference `DIAG.PAT` and the environment will automatically include the other two. As the system will be loading the structure definitions from these files, all transformation patterns for these types must also be included in the same files.

Transformation pattern for a scalar times a lower-triangular matrix:

```
REPLACE
   INTEGER n
   REAL A(n,n){TYPE=LOWERT}, alpha, B(n,n){TYPE=LOWERT}
   A = alpha * B;
WITH
   SUBROUTINE tscale(P_IN, P_IN, P_OUT{DIM(b,1),DIM(b,2)},...
      P_IN, P_IN, P_IN{b}, P_IN)
   A = tscale('L', DIM(B,1), DIM(B,1), alpha, B, DIM(B,1));
END
```

Transformation pattern for a scalar times an upper-triangular matrix:

```
REPLACE
   INTEGER n
   REAL A(n,n){TYPE=UPPERT}, alpha, B(n,n){TYPE=UPPERT}
   A = -alpha * B;
WITH
   SUBROUTINE tscale(P_IN, P_IN, P_OUT{DIM(b,1),DIM(b,2)},...
      P_IN, P_IN, P_IN{b}, P_IN)
   A = tscale('U', DIM(B,1), DIM(B,1), -alpha, B, DIM(B,1));
END
```

Transformation pattern for a scalar times a diagonal matrix:

```
REPLACE
   INTEGER n
   REAL X(n,n){TYPE=DIAG},Y(n,n){TYPE=DIAG},alpha
   Y = alpha * X;
WITH
   INTEGER k
   for k = 1:DIM(X,1)
      Y(k,k) = alpha * X(k,k);
   end
END
```

Figure 7.14: Transformation patterns for additional optimizations to the SOR program.

Chapter 8

TRANSFORMING THE STRUCTURE REPRESENTATION

The ability to recognize and utilize matrix structures within the existing MATLAB data structures, as described in the previous chapter, may not fully realize the performance potential of certain structures. To achieve this additional performance, it is also necessary to utilize data representations other than the standard MATLAB data structures.

The developers of MATLAB had already recognized the disadvantages of using their data structures for certain problem classes. In particular, the use of dense matrices for all problems can be very ineffective when the data is sparse. This not only causes slower performance, but the memory requirements for storing the dense matrices can prevent large problems from even being attempted. To overcome this, the developers incorporated a sparse data representation into MATLAB.

Within MATLAB, a matrix will be treated as sparse when it is assigned to the result of the `sparse()` primitive, or when assigned to an expression with sparse operands. MATLAB does not examine the values within a matrix to determine if the matrix is sparse. The sparse representation chosen for MATLAB is the compressed sparse row format (CSR). This is a common format in which the nonzero value, and associated column number, for each element in a row are stored contiguously along with a pointer to the start of each row. The sparse implementation of MATLAB's built-in functions are provided to operate on the sparse matrices using the CSR format.

During the design of the FALCON environment, it was decided that the sparse format would not be given direct support. The primary reasons for excluding the sparse format were two-fold, first there was concern about the amount of time needed to provide complete support to the sparse format within the system. The second reason, however, is that by just reproducing the `sparse` primitive from MATLAB, support for sparse representations would be limited to a single sparse data representation.

Examining the literature on sparse numerical analysis, one realizes that a large number of sparse data representations have been developed. These include representations for general use, as well as those that have been developed for use within a specific application or problem domain. For instance, the use of six sparse storage formats for iterative linear system solvers are discussed in [BBC+93] and five sparse storage formats for direct linear system solvers are discussed in [DR79]. Examining these formats, one realizes there are advantages and disadvantages to each and that, due to these, the developer should have control over the format choice.

However, direct support for multiple sparse storage formats within the FALCON environment would be very difficult. Either a special code generator would be necessary for each supported format or the code to generate each operation would need a test to determine the storage format. In some instances, multiple statements may need to be changed to support operations on certain formats. And, to experiment with a new storage format, the algorithm developer would have to request that the FALCON environment be modified. It was decided, therefore, that a better solution would be one which provided the developer with the ability to add storage formats to the FALCON environment.

Furthermore, though this discussion has so far focused on the traditional dense and sparse matrix storage format, there may also be cases for which the algorithm developer wishes to use other data representations. For instance, storing the matrix by rows rather than by columns or only storing the lower triangular half of a symmetric matrix. A method flexible enough to allow for defining sparse storage formats should also allow for defining most data representations.

8.1 Definition of Data Representations

The `STRUCTURE` definition in the previous chapter was used to define the recognition and propagation of a user-defined type within MATLAB's data representations. The type definitions were then used to perform transformations and code generation based on the type. Upon examination, this construct already provides most of the information that would be necessary to implement new data representations, with the exception of allocating and referencing the new data elements.

8.1.1 Allocation of Data Storage

Most data storage formats require the allocation of several data items, which may be of different data types and sizes. To support this within the `STRUCTURE` definition, a new block labeled `DECLARATION` was added. Within this block the data fields are declared and their sizes are defined, with the format being shown in Figure 8.1. Notice that with the DECLARATION block being added, the definition becomes similar to a class definition in the C++ language.

The *field declaration* used for the declaration of the data field consists of three parts:

type name (*size*)

```
STRUCTURE(structure name)
     DECLARATION(dummy variable)
          field declaration
          ...
     INHERITS
          ...
     ASSERTIONS
          ...
END
```

Figure 8.1: Structure definition with data field declarations.

with the first two parts required and the last one optional. The *type* can be any of the types used for variable declarations in transformation patterns. The *name* is the name of the field and must be unique within the the structure, i.e. two fields cannot have the same name, and can be any valid MATLAB variable name. The last field, the *size*, is used for determining how many elements to allocate for the field. If the *size* field is not present, then the field will be a scalar value.

In general, the size of the fields will be based on the size of the variable being replaced. To reference those sizes for the field declarations, a dummy variable is placed after the `DECLARATION` statement. References to this dummy variable within the *size* declaration will then be translated to the respective sizes of the variable being replaced, for instance using the `DIM` primitive to reference the size of a dimension. Simple arithmetic expressions are allowed for the size declaration, as shown in the following definition that allocates a vector `A` to contain the same number of elements as the matrix being replaced, with `X` being the dummy variable:

```
REAL A(DIM(X,1)*DIM(X,2)).
```

The *size* field may contain either one operand, denoting a vector, or two operands separated by a comma, denoting an array.

Just being able to reference the size of the matrix dimensions, however, is not sufficient for the efficient allocation of storage for data representations such as sparse matrices. In this case, it is generally necessary to also know the number of nonzero elements in the array. To support this, another primitive called `NNZ` has been added to the transformation language to reference the number of nonzero elements in a matrix. Figure 8.2 illustrates the data declarations to allocate the storage for a compressed sparse row format, similar to that used by MATLAB. Notice how the fields to store values, `VALS`, and column indices, `COL_NUMS`, are defined to equal the number of nonzero entries in the matrix. The field that points to the start of each row, `ROW_PTRS`, is defined to be one more than the number of rows in the matrix. The extra element in this field is necessary as the end of each row is found by looking at the start of the next row.

```
DECLARATION(A)
    REAL VALS(NNZ(A))
    INTEGER COL_NUMS(NNZ(A))
    INTEGER ROW_PTRS(DIM(A,1)+1)
```

Figure 8.2: Field declarations for the compressed sparse row format.

Adding the NNZ primitive required several modifications to the FALCON environment. This value is not known at compile time and must be calculated at run time by examining the elements of the matrix. Routines had to be written and added to the support libraries to perform this calculation. To store the value once it had been calculated, a fourth shadow variable, *?*__NNZ, was added to the generated code.

The code generators also needed to be modified to support the declaration of the fields. The code used to generate the variable declarations needs to check each variable to determine if it is an user-defined type that has a field declaration. If so, then the standard matrix or vector declaration will not be generated, but instead a declaration for each of the fields will be generated, along with the shadow variables. For example, the Fortran 90 data declarations for a variable M using the compressed sparse row format are shown in Figure 8.3. Notice how the resulting names are a combination of the original variable name and the field name, separated by two underscores.

```
INTEGER, ALLOCATABLE :: M__ROW_PTRS(:)
INTEGER, ALLOCATABLE :: M__COL_NUMS(:)
DOUBLE PRECISION, ALLOCATABLE :: M__VALS(:)
INTEGER M__D1, M__D2, M__T, M__NNZ
DATA M__T, M__D1, M__D2, M__NNZ /0, 0, 0, 0/
```

Figure 8.3: Fortran 90 declarations for the compressed sparse row format.

8.1.2 Referencing New Data Fields

To utilize the data representation, it is necessary to generate code using the fields allocated for the data. The code generation is accomplished by using transformation patterns to replace the original statements containing the variable with new statements containing the fields. Within the transformation patterns, the references to the fields are indicated by the use of a USER_FIELD construct that has been added to the transformation language.

The USER_FIELD construct contains two operands:

USER_FIELD(*variable name*, '*field name*').

The first operand, *variable name*, refers to the variable that was matched in the REPLACE block of the transformation pattern. The second operand, *field name*, is the data field that is being referenced. Note that the field name is placed in quotes, indicating that it is being stored as a literal string. For example, a reference to the ROW_PTRS field from the previous section would be:

```
USER_FIELD(M,'ROW_PTRS').
```

In the current implementation, the code generator blindly attaches the string which represents the field name to the variable. At no time is the field name tested to verify that it is a valid field for the matched variable. As a result, if the field name is wrong in the transformation pattern, then the generated code will also be incorrect, generating a reference to an undeclared variable.

Also notice that this implementation does not allow for the referencing of elements within a data field. Additional work is necessary to understand the issues involved in performing such accesses and subsequent work on the FALCON environment would be required.

8.1.3 Modifications to the Routine Declarations

The transformation pattern contains a routine declaration for routines that are being inserted by the WITH block. Within the declaration, the input/output state of each argument is defined and a declaration of the size of the output arguments is defined relative to the input arguments, as described in Chapter 4. Previously, the size declaration involved two values, one for each dimension. Now, as the allocation of the field declarations can be based not only on the dimensions, but also on the number of nonzeros, the size declaration must also include a third value for the nonzeros.

At this time, the values used for the sizes must be either a constant or one of the two size primitives, DIM or NNZ. This works well for functions where the size of the results are the same size as an input variable, but causes difficulties when the resulting size is unknown. In particular, functions such as sparse matrix factorizations and sparse matrix multiplications are difficult to handle as the resulting size is not known until after the operation has been applied. The following argument declarations illustrate the case where the output argument is the same size as the input argument:

```
P_IN{A}, P_OUT{DIM(A,1),DIM(A,2),NNZ(A)}.
```

When the allocation of the data fields requires the number of nonzero entries within a source variable, this value is found in one of two ways. If the source variable also has data fields, then an NNZ shadow variable will contain the number of nonzero entries. If this is not the case, then the source variable will be passed to a routine that will traverse all the elements, counting the number of nonzero elements.

8.2 Using a Sparse Matrix Data Representation

To evaluate the effectiveness of modifying the data representation, it was decided to generate a sparse matrix type similar to that used by MATLAB. This would allow the performance of the generated code to be compared not only to the generated code without the new types, but also to the MATLAB code using its sparse data types.

The capabilities of the user-defined types allow the sparse matrix type to take advantage of additional structure information. As in the previous chapter where triangular and diagonal matrices were utilized, it is also possible to utilize triangular and diagonal sparse matrices. Through the type inheritance, these matrix types form a relationship, shown in Figure 8.4, similar to that for the dense matrices.

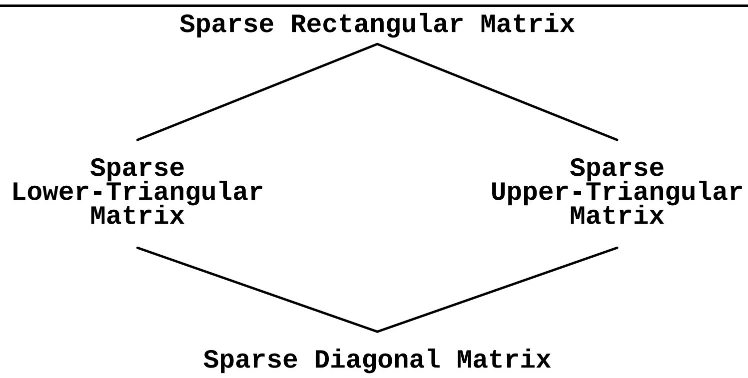

Figure 8.4: Inherits relationships for the sparse matrix types.

User-defined types were created for the four matrix types:

- `DSPARSE` – for the diagonal sparse matrix,
- `LSPARSE` – for the lower-triangular sparse matrix,
- `USPARSE` – for the upper-triangular sparse matrix, and
- `SPARSE` – for the rectangular sparse matrix.

All of these types use a compressed sparse row format (CSR) similar to that used by MATLAB, with the `STRUCTURE` definition for each of these types being shown in Figure 8.5. Each structure definition also includes transformation patterns for the translation of the MATLAB operations into subroutine calls that support the CSR format. The primary operations that need to be replaced are the multiplication of a matrix times a vector, the solve using a triangular matrix, and the solve using a diagonal matrix. For example, the transformation pattern for replacing the multiplication

Diagonal Sparse Matrix

```
STRUCTURE(DSPARSE)
DECLARATION(A)
   REAL VALS(DIM(A,1))
   INTEGER COL_NUMS(DIM(A,1))
   INTEGER ROW_PTRS(DIM(A,1)+1)
INHERITS
   LSPARSE
   USPARSE
ASSERTIONS
   ASSERT
      INTEGER n
      REAL a(n), B(n,n)
      B = diag(a);
   WITH
      REAL B(n,n){TYPE=DSPARSE}
   END
   ASSERT
      INTEGER n
      REAL a(n,n), B(n,n)
      B = diag(diag(a));
   WITH
      REAL B(n,n){TYPE=DSPARSE}
   END
END
```

Lower-Triangular Sparse Matrix

```
STRUCTURE(LSPARSE)
DECLARATION(A)
   REAL VALS(NNZ(A))
   INTEGER COL_NUMS(NNZ(A))
   INTEGER ROW_PTRS(DIM(A,1)+1)
INHERITS
   SPARSE
ASSERTIONS
   ASSERT
      INTEGER n
      REAL A(n,n){TYPE=SPARSE},B(n,n)
      INTEGER x
      B = tril(A,x);
   WITH
      REAL B(n,n){TYPE=LSPARSE}
   END
   ASSERT
      INTEGER n
      REAL A(n,n){TYPE=SPARSE},B(n,n)
      B = tril(A,-1);
   WITH
      REAL B(n,n){TYPE=LSPARSE}
   END
END
```

Upper-Triangular Sparse Matrix

```
STRUCTURE(USPARSE)
DECLARATION(A)
   REAL VALS(NNZ(A))
   INTEGER COL_NUMS(NNZ(A))
   INTEGER ROW_PTRS(DIM(A,1)+1)
INHERITS
   SPARSE
ASSERTIONS
   ASSERT
      INTEGER n, x
      REAL A(n,n){TYPE=SPARSE}
      REAL B(n,n)
      B = triu(A,x);
   WITH
      REAL B(n,n){TYPE=USPARSE}
   END
END
```

Rectangular Sparse Matrix

```
STRUCTURE(SPARSE)
DECLARATION(A)
   REAL VALS(NNZ(A))
   INTEGER COL_NUMS(NNZ(A))
   INTEGER ROW_PTRS(DIM(A,1)+1)
INHERITS
ASSERTIONS
   ASSERT
      INTEGER n
      REAL A(n,n), B(n,n)
      B = sparse(A);
   WITH
      REAL B(n,n){TYPE=SPARSE}
   END
END
```

Figure 8.5: Definitions of sparse matrix types.

```
REPLACE
   INTEGER n
   REAL A(n,n){TYPE=SPARSE}, x(n), y(n)
   y = A * x;
WITH
   SUBROUTINE amux(P_IN, P_IN, P_OUT{DIM(a,1)}, ...
      P_IN{a}, P_IN, P_IN)
   y = amux(DIM(A,1), x, USER_FIELD(A,'VALS'), ...
      USER_FIELD(A,'COL_NUMS'), USER_FIELD(A,'ROW_PTRS'));
END
```

Figure 8.6: Transformation pattern for a sparse matrix times a vector.

of a matrix times a vector with primitive to perform the operation using a sparse matrix is shown in Figure 8.6.

In addition to the transformations on operations with the matrix types, for each of these types there is a transformation pattern that is used to initialize the data fields. These transformations correspond to the assertions that are used to recognize the matrix types and to describe a routine to be used for translating the values from the source matrix into the target data fields. For instance, to generate a sparse matrix from a dense matrix, the dense matrix must be traversed and the nonzero values extracted. The transformation pattern used for converting a dense matrix to a sparse matrix is presented in Figure 8.7.

```
REPLACE
   INTEGER n
   REAL A(n,n), B(n,n){TYPE=SPARSE}
   B = sparse(A);
WITH
   SUBROUTINE dense_to_sparse( P_IN{a}, P_IN, P_IN, ...
      P_OUT{DIM(a,1),DIM(a,2),NNZ(a)}, P_OUT, P_OUT )
   [ USER_FIELD(B,'VALS'), USER_FIELD(B,'COL_NUMS'), ...
     USER_FIELD(B,'ROW_PTRS')] = dense_to_sparse( A, ...
                                 DIM(A,1), DIM(A,2) );
END
```

Figure 8.7: Transformation pattern for converting a dense matrix to a sparse matrix.

The sparse matrix type definitions were used to generate code using the FALCON environment for three iterative linear system solvers from [BBC+93]. The solvers included the Conjugate Gradient method (CG), the Quasi-Minimal Residual method (QMR), and the Successive Overrelaxation method (SOR). For all three programs, a 420×420 matrix from the Harwell-Boeing test set, BCSSTK06, is used as input to the programs.

To evaluate the effectiveness of defining the data representation, four different test programs were created for each iterative solver:

- an M-file with dense matrices to be interpreted using the MATLAB environment,
- an M-file with sparse matrices to be interpreted using the MATLAB environment,
- a Fortran 90 program generated by FALCON with dense matrices, and
- a Fortran 90 program generated by FALCON with the sparse matrix types applied using the user interface.

The Fortran 90 codes were compiled using Vast 90 and the code with the sparse matrix types was linked using the library of sparse primitives. All the tests were run on a Sun SPARCstation 10, with each individual test being run five times and the best CPU execution time being selected. The results from all the tests are shown in Table 8.1.

Program	FALCON Fortran 90		MATLAB M-file	
	Dense	Sparse	Dense	Sparse
CG	14.24	1.83	51.58	11.11
QMR	8.06	1.74	56.05	5.30
SOR	21.86	0.84	63.18	9.27

Table 8.1: Performance comparison using the sparse matrix types.

For both the MATLAB and Fortran 90 programs, switching from dense matrices to sparse matrices provided significant performance improvements. With the MATLAB M-files, the speedups ranged from 4.6 to 10.6, and with the Fortran 90 programs, the speedups were from 4.6 to 26.0. When comparing the MATLAB performance to the Fortran 90 performance, the speedups generated by the equivalent Fortran 90 programs range from 2.9 to 7.0 when using dense matrices and from 3.0 to 11.0 when using sparse matrices. It was expected that the performance of the programs using sparse matrices would be better than the equivalent program using dense matrices, though it is interesting to note that the sparse MATLAB programs are faster than the dense Fortran 90 programs. The importance of these tests, however, was to determine if the FALCON environment could generate effective code using the data representations for user-defined types. This goal has been achieved as the range of speedups, when comparing the Fortran 90 programs relative to the MATLAB programs, are similar for both the sparse matrix codes and the dense matrix codes.

8.3 Additional Comments

In this chapter, the experiments have focused on the use of the CSR sparse matrix format, but the system is in no way limited to this data representation. The examples just as easily could have been written using the compressed sparse column format or any number of the other sparse matrix formats. The CSR format was chosen for two reasons, the first being that since it was the format chosen for MATLAB, a comparison between the two implementations could be performed. Second, support routines were already available to perform the mathematical operations using the CSR format. Additionally, the system is not limited to just the general sparse matrix data formats. Other matrix types, such as banded systems or symmetric matrices, can also be used just as easily.

In Chapter 4, the similarities between the FALCON transformation language and the Forbol transformation language were examined. In contrast, the capability of FALCON to define characteristics of matrix structures and the relationships between such structures, which is not supported by Forbol, illustrates the different approach taken by FALCON as compared to other pattern-based transformation languages.

Chapter 9

CODE IMPROVEMENT TECHNIQUES DEVELOPED FOR THE FALCON ENVIRONMENT

In addition to the Interactive Restructuring System that was built upon the transformation patterns, several other enhancements and modifications have been made to the FALCON environment in order to improve program development. Two of these modifications involve methods for reducing the number of dynamic statements that are present in the code. The third enhancement of the system is the support for compiling routines separately, instead of always needing to compile the entire program. Each of these modifications are discussed in this chapter.

9.1 Assertion of Variable Properties

As mentioned previously, the static analysis phase of the FALCON environment cannot always determine the exact rank of a variable, especially for those variables that are loaded from data files. This difficulty arises from the polymorphic nature of many statements in MATLAB, whereby the statements operate correctly regardless of whether the operands are scalars, vectors, or matrices. For variables input from data files, the static analysis assumes that the loaded variable will always have the same number of dimensions, but that any dimension can be either a size of one or a size greater than one. As a result, a scalar value will always be treated as a scalar, but a vector can be either a vector or a scalar, and a matrix may actually be any of the three, depending upon whether any of its dimensions have a size of one.

During the dynamic analysis phase, code is inserted into the AST to deal with the possible rank combinations. Not only does this make the generated code difficult to understand and modify, but it also makes the application of interactive restructuring more difficult. Each algebraic operation may be represented in the code by several instances that deal with the possible combinations of operand ranks making it difficult to identify the underlying algebraic algorithm being performed.

One approach to improving this problem would be to modify the static analysis to maintain the ranks of loaded variables. However, as the developer may not always want this feature, or want it for only selected variables, this approach was not taken. Instead, it was decided to provide the developer with the ability to interactively set the rank of any variable in the program. It was felt that this approach was best at providing the developer with the most control over the code generation.

9.1.1 Updating the Information

The user interface, as described in Chapter 4, has the means for displaying the variables and three properties for each variable: the intrinsic data type, the rank, and the user-defined type. Instead of having the developer edit the fields in this window, it was decided to have the developer select a variable and then perform the updates to the variable on a separate screen.

Upon the selection of a variable in the symbol information window, a symbol update window is generated, as shown in Figure 9.1. This window contains the variable name, followed by three

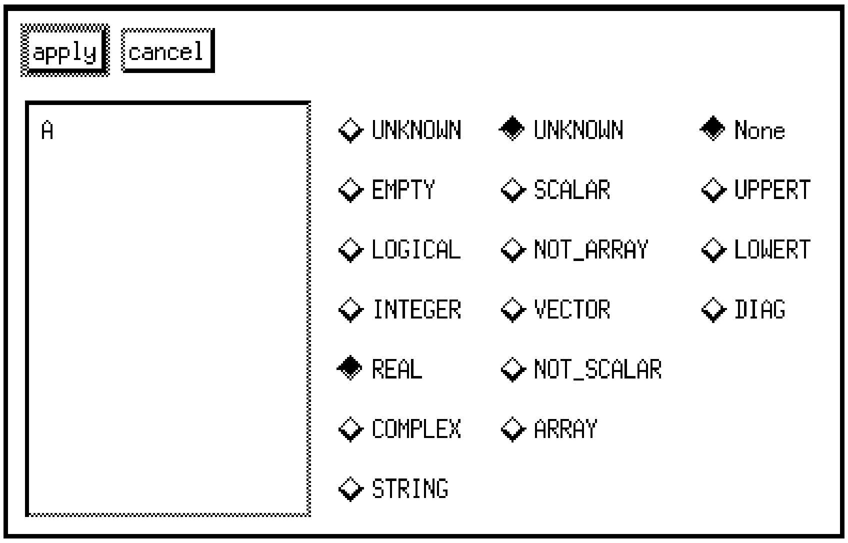

Figure 9.1: Window for updating the symbol information.

columns that contain the possible intrinsic data types, the possible ranks, and the possible user-defined types. In each column, the current value of the property is marked and the developer can modify the property by selecting a different value. After making any changes to this window, the new information can be saved with the `apply` button or disregarded with the `cancel` button.

Notice that the last column in the figure, for the user-defined types, begins with the value `None`, followed by three types, `UPPERT`, `LOWERT`, and `DIAG`. The value `None` indicates the variable does not have a user-defined type and is always present in the symbol update window. The other three values are the specific user-defined types that are being used during the current restructuring session. Whereas the values in the other columns are hard-coded into the environment, this column is always generated at run time to reflect the current possibilities.

When the properties of a variable are updated, the changes are not automatically propagated through the source code. The propagation is only performed at the command of the developer, by selecting the `propagate symbols` choice from the `Propagate Menu` on the main window. This was done to allow the developer to update multiple symbols and then propagate the information using a single pass of the propagation algorithm, instead of running the algorithm each time a variable is updated. This is especially useful when the developer wants to update several variables that are being loaded from a data file.

9.1.2 Propagation of Asserted Information

The propagation of the variable properties that are asserted by the the developer is similar to the static inference in the Program Analysis System developed by DeRose, which was described in Chapter 3. In this case, however, the inference is based upon knowing the rank and intrinsic type of a variable, but does not require knowing the size of the dimensions as was required by DeRose's algorithm.

The modifications are apparent when considering how DeRose's algorithm handles variables that are loaded from files. For instance, when a matrix `A` and a vector `b` are loaded from a file, both dimensions of `A`, and the first dimension of `b`, are assumed to be unknown, with the second dimension of `b` being set to one. As a result, `A` is given the rank UNKNOWN and `b` is given the rank NOT_MATRIX. If these two variables are multiplied,

```
c = A * b,
```

then the result, `c`, is given the same rank as `b`, NOT_MATRIX. In contrast, if the rank of `A` is asserted to be a MATRIX and the rank of the `b` is asserted to be VECTOR, the new algorithm would give `c` the rank of VECTOR even though the sizes of its dimensions are unknown.

Furthermore, as the new algorithm is applied, tests are made to determine if the resulting information eliminates the need for dynamic statements that may have been inserted. Using the same example,

```
c = A * b,
```

the dynamic analysis phase would have inserted five different multiplication statements into the code to handle the different possible rank combinations. The new analysis algorithm will replace the five multiplication statements with a single statement. The dynamic analysis would have also inserted statements to perform the dynamic allocation of the result. Instead of recalculating the size of this result and rebuilding the allocation statements, a search is made through the dynamic operations being replaced in order to find the statement that would have performed the operation for the equivalent ranks. Once this operation has been found, the dynamic allocation statements associated with it can be copied to the new operation. This avoids the need to rerun the analysis to calculate dynamic data allocations.

9.1.3 Effects of Asserting Variable Types

To determine the effects of asserting variables properties, experiments were performed using the three iterative solvers that were used in the previous chapter: the Conjugate Gradient method (CG), the Quasi-Minimal Residual method (QMR), and the Successive Overrelaxation method (SOR). For all three programs, a 420×420 matrix from the Harwell-Boeing Test set, BCSSTK06, was used as input to the programs. In order to control the time each program takes to run, the stopping criteria was set so that the MATLAB runs would take roughly the same amount of time.

Two versions of each program were generated using the FALCON environment, one without any modification and the second using variable assertions. Each of these programs is an iterative solver used to solve for x in the following equation:

$$Ax = b.$$

To perform the solve, the programs load A, b, and an initial guess for x from a data file. The variable assertions were used to set A to a matrix and the other two variables to vectors and this information was then propagated through the program. The programs were then compiled using Vast 90 and each was run five times on a SPARCstation 10. Table 9.1 contains both the lines of code generated for each method and the best CPU execution time observed for each program.

Program	Original Program		Using Symbol Assertions			
	Lines of Code	Execution Time	Lines of Code	Reduction in Lines	Execution Time	Performance Degradation
CG	565	14.24	398	30%	14.41	1.2%
QMR	1119	8.06	790	29%	8.22	2.0%
SOR	582	22.27	388	33%	22.37	0.5%

Table 9.1: Comparison of programs with and without variable assertions.

As can be seen in the table, the assertion of variable ranks reduced the number of lines of code between 29% and 33%. What is unusual, however, is that for all three programs, the version with assertions ran slightly slower than the original program. These variations are only between 0.5% and 2.0%, but the initial assumption was that the versions with assertions would be faster due to performing less conditional statements. This performance degradation was not observed when additional tests were performed with smaller problems, so this effect may be due to variation in the cache usage. Regardless, the advantages of producing 30% fewer lines far outweigh the slight performance degradation in cases where the code is being restructured interactively or will subsequently be modified.

9.2 Resolution of Dynamic Statements within Loops

Though the method just described can be used to resolve the rank of a variable, knowing the ranks of the variables is not enough to guarantee that the rank of the result from every operation will be known. When a reference to a variable uses indices, the analysis system may not be able to determine if the size of the indexed section is one or greater than one. In this case, the system will still need to have dynamic analysis to utilize the different possible ranks.

For instance, in loop-based algorithms it is common for the loop index to be used as one of the bounds in the array or vector index. The bounds of the reference typically go either from the first element to the loop index,

$$\texttt{1} \; : \; \textit{loop index},$$

or from the loop index to the last element,

$$\textit{loop index} \; : \; \textit{size of dimension}.$$

As a result, when the loop index goes from `1` to the *size of dimension*, the size of the reference section will also vary from `1` to the *size of dimension*. If the reference is to a vector, then during one loop iteration, the reference will be a scalar and, in the remaining loop iterations, the reference will be a vector. Similarly, if the reference is for a matrix, the reference may be a scalar during one loop iteration and a matrix during the other loop iterations.

During the static analysis phase, the system will detect that the size of the reference section varies and will generate dynamic code to handle the two cases, first when the size is one and second when the size is greater than one. In such codes, however, the size of the reference section will be one for a single loop iteration and will be greater than one for the remaining loop iterations. Each loop iteration, though, will incur the overhead to determine which operation to perform.

When many loop iterations are being performed, however, the benefits of using the optimized operation for a single loop iteration may be outweighed by the overhead required to perform the test. If the code that performs the operation when the size of the index is greater than one will also work when the size of the index is one, then the test can be removed and the more general operation can always be used.

9.2.1 Identifying the Dynamic Statements

An algorithm to perform transformations such as those just discussed was developed and included in the Interactive Restructuring System. This algorithm is applied to the entire program when the `resolve dynamic` button is selected from the main window. It is not possible to limit the application of the algorithm to a subset of the program.

The algorithm traverses the AST representation of the program and, based upon the statement type that is encountered, performs one of two actions:

- If the statement is not part of the tests inserted by the dynamic analysis, then rank of the operands will be updated.
- Else, if the statement is part of the dynamic tests, then the dynamic statements will be evaluated to determine if the tests can be replaced.

The first action, the update of the ranks, is based upon two assumptions:

1. The ranks of variables should be preserved even if the size of the dimension is unknown.
2. Index expressions that need to be evaluated at run time to determine their exact size will be treated as if their sizes are greater than one.

The first assumption, the preservation of rank, was also used during the propagation of asserted variable properties. It is based on the idea that rank of the result of an operation can be determined by the ranks of the operands even when the sizes are unknown. For instance, when the operation is a matrix times a vector, the result will always be a vector, even though the result can be either a scalar or a vector when the sizes are not known. As with the assertion of variable properties, a new version of the static inference was created based on this assumption.

The second assumption, regarding indexed expressions, is used so that the algorithm does not have to stop when array expressions are encountered. When the size of an index expression is not known at compile time, the static analysis phase must assume that the expression can evaluate to both one and greater than one. As a result, it is not able to statically determine the rank of the array reference containing an expression. The four possible index expressions and how each is treated by this algorithm are shown in Table 9.2.

The second action is only applied when the statements are known to be part of a dynamic assignment. First, the dynamic assignment is examined to determine if it contains tests of the operand ranks to select the operation. If no tests are present, then the dynamic assignment is ignored. If the tests are present, then the algorithm attempts to find a *probable* path through the tests.

Index Expression	Assumed Size
value or *variable*	1
:	> 1
lower : *upper*	> 1
lower : *stride* : *upper*	> 1

Table 9.2: Assumed sizes for index expressions.

To understand how the probable path can be calculated, it is necessary to first examine the structure of the dynamic assignments. The dynamic analysis phase will insert tests that compare the dimension of an operand to one, with a true clause for when the values are equal, and a false clause for when the values are different. A tree of these comparisons is built to allow the correct operation to be selected. To determine the probable path, therefore, the operand dimension is compared to one using symbolic computations. If the operand dimension equals one, then the path is known to be the true clause. If the maximum value of the operand dimension is known to be greater than one, then the probable path is the false clause. If neither of these conditions is met, but the operand dimension is based on a loop index, then the probable path is the false clause. If none of these conditions are met, then no probable path is found.

When a probable path is found, then the tree of comparisons can be replaced with the operation from the probable path. In addition, the dynamic allocation code corresponding to the operation from the probable path must also be retained with the operation.

9.2.2 Example of Identifying Dynamic Statements

A Gaussian Elimination program was used to test the effectiveness of the algorithm for optimizing dynamic statements, with the computational loop from the program shown in Figure 9.2. The dynamic resolution algorithm was applied to the program and it correctly identified that the multiplication in the last line of the loop body could always be performed as an outer product.

The FALCON environment was used to generate both the original program and the code using the dynamic resolution algorithm. Both programs were compiled using Vast 90 and tests were performed on a SPARCstation 10. The tests were conducted using two different matrices, a 200×200 matrix that was generated with random values and a 420×420 matrix, from the Harwell-Boeing Test set, BCSSTK06. Each program was run five times for each matrix with the results being presented in Table 9.3.

As can be seen from the results, there is no performance difference for the 200×200 matrix and for the 420×420 matrix the difference is less than 1%. The transformation does remove 18 lines of code from the resulting program, roughly 6%. Again, the goal of this type of transformation is not the performance improvement. Instead, this transformation is another attempt to make

```
for j = 1:n-1
    piv = j;
    val = abs(A(j,j));
    for i = j+1:n
        if val < abs(A(i,j))
            val = abs(A(i,j));
            piv = i;
        end
    end
    if piv ~= j
        temp = A(piv,:);
        A(piv,:) = A(j,:);
        A(j,:) = temp;
    end
    A(j+1:n,j) = A(j+1:n,j) / A(j,j);
    A(j+1:n,j+1:n) = A(j+1:n,j+1:n) - ...
         (A(j+1:n,j)*A(j,j+1:n));
end
```

Figure 9.2: Computational loop from Gaussian Elimination.

Program	Lines of Code	Execution Time	
		200×200	420×420
Original Program	285	3.38	28.40
Transformed Program	267	3.38	28.24

Table 9.3: Identifying dynamic statements in a Gaussian Elimination program.

the code more readable and to clean up the code so that it might be easier to apply additional transformations.

9.3 Support for Separate Compilation

The third modification added to the FALCON environment was the support necessary to translate a MATLAB M-file into a subroutine rather than always needing to operate on a complete MATLAB program. This capability is useful both because it allows a large program to be translated as individual routines and then linked together, and because it is necessary when trying to develop library routines instead of programs.

As the FALCON environment inlines function calls into the source program, the generated code consists of one large routine with calls to numerical primitives. This can result in a routine consisting of several thousand lines of code; this routine is not only slow to compile but can be difficult for the compiler to optimize (primarily due to tables not being large enough). By breaking the program up into several routines, the compilation may be both faster and better.

Since one of the goals of the FALCON environment is the development of libraries, it is important for the system to be able to generate routines. An alternative approach to compiling a routine is to write a simple driver script to set the correct argument types and then call the routine to be translated. The code in the resulting program can then be modified in order to convert the program into a routine call. However, this approach requires the developer to modify the code by hand and does not follow a basic theme of the system, which is to provide the developer with tools to perform the modifications.

The original environment was restricted to complete programs due to the lack of information that is known about function parameters in MATLAB. Since neither the data type nor rank is known for a parameter, the correct operations can only be determined at run time. As a result, the code generated by the dynamic analysis phase would be a reproduction of the MATLAB interpreter. To overcome this problem, it was decided that only complete programs would be compiled with input data files providing some initial information about the variables.

To effectively translate subroutines by themselves, therefore, it is necessary to acquire additional information about the data type and rank for each parameter. To accomplish this, a simple text interface was developed to query the developer for the data type and rank of each input parameter. The developer is not queried regarding the output parameters since the analysis can determine the information for these.

For each input parameter, the developer is given three possible data types, *integer*, *real*, and *complex*, from which one must be chosen. If the data type is chosen to be *integer*, then the rank of the parameter is assumed to be *scalar*. If the selected data type is one of the other two, then the developer is prompted to select one of four data ranks: *scalar*, *row vector*, *column vector*, or *matrix*. This information is collected before the static analysis is performed, allowing the information to be propagated through the entire subroutine.

Though the rank and data type is known for each input parameter, the lack of size information prevents the original static analysis algorithm from correctly propagating the ranks. Again, this modification makes use of the alternative static analysis algorithm that was developed to operate solely on the basis of ranks and ignores the size of the dimensions. Also, since the size of the dimensions are unknown within the subroutine, these must be passed in as arguments, which the FALCON environment automatically adds to the subroutine declaration.

Additional modifications were necessary to support the way in which MATLAB treats subroutine parameters differently than Fortran 90. To support the side-effect free nature of MATLAB M-files, it is necessary to first create a copy of any input argument before it is modified within the subroutine. In this way, all modifications will be made to the copy of the data, which will be destroyed when the routine exits, and the data passed into the routine will not be changed. Also, since the output parameters are allocated before the routine is called, the routine is not allowed to reallocate any output parameters. If the dynamic analysis system needs to perform an allocation on a variable that corresponds to an output parameter, then a copy of the output parameter will be created and all allocations and modifications will occur on this copy. Immediately prior to the return from the routine, the data from the copy of the output parameter will be transferred to the output parameter.

Chapter 10

GENERATION OF C++ CODE

Another enhancement that was added to the FALCON environment was the ability to generate C++ code. This capability was added both to reflect the way in which the development of numerical software is evolving and to examine the issues that arise from developing numerical software with an object-oriented approach.

Traditionally, numerical software has been written in Fortran; it contains language constructs that lend themselves to representing numerical formulas, while at the same time remaining simple enough to support many compiler optimizations. As a result, loop-based numerical programs written in Fortran and compiled using optimizing compilers frequently achieve a significant percentage of the peak performance on many sequential machines. Given this capability, many numerical libraries have been written in Fortran, with only the lowest-level primitives sometimes being written in assembler.

Though the performance of Fortran is very good, improvements in language design, such as data structures and data classes, have been added to Fortran very slowly. As a result, some developers have turned to other languages, such as object-oriented languages, that provide additional constructs to help with program development and maintenance. Now, with Fortran 90, some of these constructs are being added to the Fortran languages.

The increasing use of object-oriented languages for numerical programming is reflected in the conferences that have been formed to discuss this issue. Among these has been both OON-SKI, a conference on object-oriented numerics, first held in 1993 [Rog93], and POOMA, a conference on parallel object-oriented methods, first held in 1994 [Los94]. One of the primary languages being discussed at these conferences is C++, whose use has been growing rapidly both in the the general software community and in the numerical software community. This is in part to due to C++ being based upon C, which already had a substantial user community, but also due to the features that were added. C++ was designed, according to Stroustrup [Str91], to:

– be a better C

– support data abstraction

– support object-oriented programming

It is the software development advantages from the second two items, data abstraction and object-oriented programming, that have drawn developers to C++. Therefore, as more numerical programmers are utilizing C++, it was felt that the FALCON environment should also support C++.

Upon deciding to support C++ for numerical programming, one realizes there are several interesting issues to be addressed. In Fortran 90, the matrix data structures and the arithmetic operations on matrices are built into the language. By contrast, the matrix data structures and operations in C++ are defined in data classes supplied by the program developer. In developing classes to support the required operations, trade-offs are made with regards to readability, performance, and flexibility. As a result, the decisions made by the developer of the matrix class can have a significant impact on the performance of the generated code. Furthermore, based on the design decisions for the matrix classes, there may or may not be possibilities for the FALCON environment to optimize the primitives that will be utilized.

An important issue in a C++ class is the trade-off between using overloaded operators for readability as compared to using method invocations for performance. For instance, to perform a matrix-vector multiplication, a class can overload the multiplication operator so that the operation can be written as follows:

```
y = A * x;
```

The problem with this approach is that the actual implementation of the code will generate a temporary variable to hold the result of the multiplication and then this temporary variable will be copied to the variable on the left-hand side. An alternative is to create a method for the left-hand side, that both multiplies the two operands and directly assigns the values:

```
y.m_times_v( A, x );
```

in this case, the temporary storage has been eliminated, but the readability of the code has decreased.

Another issue with matrix classes is whether or not the values within the matrix class can be directly accessed. In Figure 10.1, consider the two subroutines designed to implement a matrix-vector multiplication similar to that in the previous example. Notice how in the first implementation each element access is actually a method invocation that must be translated into a data access. For some classes, the method may be replaced at compile time through the use of inlining, while in other cases an actual routine may need to be invoked at run time. For the second case, since the values are accessed directly, no method invocations are necessary.

Implementation Using Value Access Methods:

```
for( i = 1; i <= A.rows(); i++ )
{
   temp = 0;
   for( j = 1; j <= A.cols(); j++ )
      temp += A.element(i,j) * x.element(j);
   y.element(i) = temp;
}
```

Implementation Using Direct Value Access:

```
for( i = 0; i < A.rows(); i++ )
{
   temp = 0;
   ij = i * A.cols();
   for( j = 0; j < A.cols(); j++, ij++ )
      temp += A.values[ij] * x.values[j];
   y.values[i] = temp;
}
```

Figure 10.1: Two implementations of a matrix-vector multiplication.

The issue of what type of a routine would need to be added to perform the operation is also related to how the data is accessed. Either of the two implementations discussed could be encapsulated in a subroutine that would require all three variables as arguments,

```
m_times_v( A, x, y );
```

or an alternative approach is to modify the vector class and insert the routine so that it can be invoked either as an overloaded operator or through a method invocation, as shown during the initial discussion of this example. Each of these implementations have different performance implications and is based on different assumptions about data encapsulation.

Only through an understanding of these issues can effective C++ code can be generated. It is also important to realize that one method is not necessarily better than another but, depending upon the needs of the developer, the preferred method may change. Though the current effort to incorporate C++ code generation into the FALCON environment has focused primarily on creating readable code with good performance, future efforts may look at allowing the developer to place different priorities on these goals.

10.1 Generation of C++ Code

The generation of the C++ code might have been accomplished following the same approach DeRose took with Fortran 90. The Fortran 90 Code Generator traverses the internal representation of the program in lexicographic order, outputting into a file the correct source code for each statement and expression as it is encountered. This allows the file containing the Fortran 90 program to subsequently be compiled by Fortran 90 compilers and restructurers. An alternative approach, however, is to directly translate the internal format of FALCON into the internal format of another restructuring tool. This allows subsequent restructuring to occur without the need to first generate the source code that would then need to be parsed. In turn, the restructuring tool would be responsible for either generating the source code or performing the compilation. With the availability of the Sage++ library [BBG+93], a class library for building Fortran 90 and C++ restructuring tools, it was decided to take this second approach for the C++ code generation.

Whereas the Fortran 90 code is generated in a single step, this alternative approach requires that the C++ code be generated in two steps. First, the program is translated from the internal format of FALCON into the internal format of Sage++ and, second, the Sage++ tools are used to generate the source code. Once the code has been translated into the Sage++ format, however, additional optimizations could be applied before the source code is generated.

10.1.1 Introduction to Sage++

Sage++ is an object-oriented toolkit for building program transformation and preprocessing tools, with support for four languages: Fortran 90, C, C++, and pC++, a parallel version of C++. The toolkit provides the parsers needed to translate the source code into its internal representation, tools for transforming the internal format, and the code generators to translate the internal representation back into source code. Only the last two Sage++ systems, the transformation tools and code generators, are utilized by the FALCON environment.

The Sage++ internal format is built using objects from five C++ classes. These classes are organized within the following hierarchy:

Project – contains of one or more *files*.
  File – contains of one or more *statements*.
    Statement – may contain other *statements* or *expressions*.
      Expression – may contain other *expressions* or *symbols*.
        Symbol – defines a variable.

The *statements* within the *file* are used to represent global data declarations and routine definitions. The *statement* defining a routine will also contain other *statements* that represent the variable declarations local to that particular routine and the code statements within the routine. Derivative classes are built from the *statement* class to represent the various language constructs. It is the responsibility of these classes to verify that the semantics of the language constructs are met when *statement* objects are created.

The FALCON environment interfaces with the Sage++ system by utilizing the Sage++ transformation tools to create objects representing *statements* and inserting these objects into a Sage++ *file*. Before the *statements* can be inserted, however, the Sage++ system must be initialized with objects from the *project* and *file* classes. To build these objects, a template C++ program was parsed using the Sage++ system with the results being stored in a file. Prior to generating new *statements*, the FALCON environment instructs the Sage++ system to first load the stored data, thus creating the required *project* and *file* objects within the system. The FALCON environment then translates the AST tree into *statements* objects and inserts the objects into the existing *file* object. Once the translation is complete, the `unparse()` function of the *file* class is used to generate the C++ source code.

10.1.2 Translation from FALCON to Sage++

A C++ Code Generator was developed to translate the FALCON internal format, the AST, into the internal format of Sage++. Though this code generator is based upon the Fortran 90 code generator developed by DeRose, the differences in the target languages coupled with the different approaches to the code generation necessitated substantial modifications.

As previously mentioned, the Fortran 90 code generator traverses the AST in lexicographic order, generating the source code for each statement, expression, symbol, and value as it is encountered. The C++ code generator, on the other hand, builds a Sage++ object to represent the entire program. To build the program, symbols and values are combined to form expressions, statements are built from these expressions, and statements are combined into other statements. Since the AST was developed in conjunction with the Fortran 90 code generator, it closely represents the semantics of Fortran 90 and, as a result, the code generator needs to understand very little of the semantics in order to generate the correct code. The ordering of the nodes in the tree inherently contains this information. In contrast, the C++ code generator must examine the information present in the AST to determine the correct operation to be performed and, based upon the semantics of that operation, create the correct C++ object.

For example, the generation of the code to represent the MATLAB loop constructs is straightforward in the Fortran 90 code generator and more complex in the C++ code generator. Consider the MATLAB `for` loop construct,

`for` *loop index* `=` *lower bound*:*stride*:*upper bound*`,` and AST representation of this construct in Figure 10.2. By performing an in-order traversal of

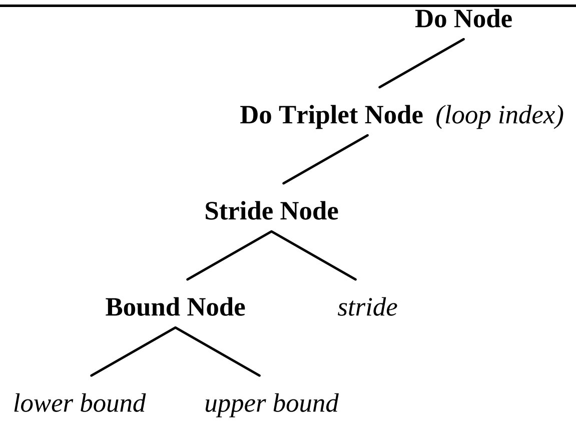

Figure 10.2: AST representation of the `for` loop construct.

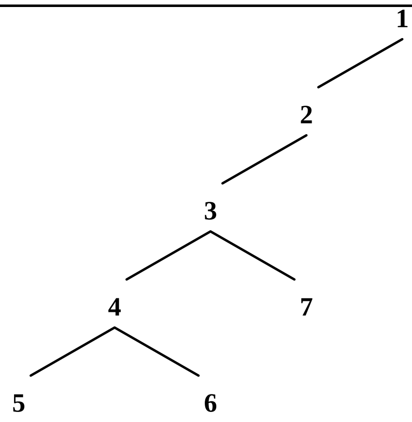

Figure 10.3: In-order traversal of the `for` loop construct.

the nodes in the AST, as shown in Figure 10.3, the Fortran 90 code generator is able to generate the correct code by performing the operations at each node as shown in Table 10.1. Note that the

Node	Operation
1	Output `DO`
2	Output *loop index* `=`
5	Output *lower bound*
6	Output *upper bound*
7	Output *stride*

Table 10.1: Translation of the `for` loop to Fortran 90.

only information required to perform the operation at each node is the information present in that node. In contrast, to perform the operations required by the C++ code generator, as shown in Table 10.2, information must be shared between the nodes. This sharing of information is necessary

Node	Operation
5	Build expression object for *loop index* = *lower bound*
6	If *stride* is positive then
	Build expression object for *loop index* $\leq$ *upper bound*
	Else
	Build expression object for *loop index* $\geq$ *upper bound*
7	Build expression object for *loop index* += *stride*
1	Build loop object using *lower bound* expression, *upper bound* expression, and *stride* expression.

Table 10.2: Translation of the `for` loop to C++.

not only to build the expression objects, but also to select the conditional operator based on the sign of the *stride*.

Whereas Fortran 90 provides built-in matrix operations, these same operations are provided in C++ through the use of a matrix class. Though the class structure can support most of the necessary operations, a few of the constructs are difficult to reproduce in this manner. As a result, modifications to the code generator were performed to support a few of the matrix operations.

For instance, consider the vector constructor from MATLAB, which may contain symbols, vector expressions, and vector sections:

```
v1 = [ v2 a:b v3(c:d) ].
```

Within Fortran 90, this can be represented as a single statement through the use of an *array constructor*. Using the same example, the generated Fortran 90 code would be:

```
v1(1,:) = (/ v2, (/ (a + T__12, T__12 = 0,int((b-a))) /), v3(c:d) /).
```

Since this construct is not provided by C++, a different approach is required in the C++ code generator.

One alternative is to first construct a single vector that represents all the elements on the right-hand side and then assign this vector to the left-hand side. A second alternative is to individually assign each expression on the right-hand side to the appropriate section of the vector on the left-hand side. The second approach was chosen for the C++ code generator since it avoids the need to allocate a temporary array, even though this approach may require additional statements.

Additional modifications to the C++ code generator were required based upon differences in the implementation of the matrix operations within the matrix class as compared to Fortran 90. Before these modifications are explored, however, the matrix class will first be discussed.

10.1.3 Development of a Matrix Class

With the decision to support the generation of C++ code from the FALCON environment, it was known that a matrix class, or classes, would be required to provide the necessary matrix operations. Given that a number of matrix classes are freely available, the long term goal is to have the FALCON environment support multiple matrix classes and allow the developer to select the class. Initially, however, it was decided to support only a single matrix class.

The next decision to be made was whether to utilize an existing matrix class with the FALCON environment or to develop a new matrix class. One problem with the use of an existing matrix class is the difficulty in determining if all the necessary functions are provided. Of special concern is the support for all the MATLAB array reference types, including the use of a vector expression with a stride as an index:

$$v(lower{:}stride{:}upper).$$

For instance, according to their documentation, neither the C++ Matrix Class from Tisdale [Tis93], the MatClass from Birchenhall [Bir93], nor newmat07 from Davies [Dav94] support this type of reference. A matrix class that did appear to meet most of the computational needs of the system, including support for array indexing with a stride, was the LAPACK++ matrix class from Dongarra, Pozo, and Walker [DPW93a].

It was decided, however, that by developing a matrix class in conjunction with the C++ code generator, it could both be tailored to meet the needs of the FALCON environment and to allow the exploration of other issues, such as the improvement of the allocation and reallocation of data. The development of the matrix class also required decisions to be made with respect to functionality and performance.

Supported Data Types

Given the capabilities of the C++ classes, it is possible to develop a single class that supports all of the MATLAB data structures, which is the approach taken by the MATCOM compiler [Yar96]. This approach, however, would lose the rank and type information that had been found by the inference techniques and would require significantly more run-time decisions. It was decided, therefore, to have multiple types that support scalars, vectors, and matrices for both real and complex values. The data types that were developed are as follows:

Real Scalars – Real scalars are implemented using the `double` type defined within the language for double precision variables. In certain situations, a scalar integer, `int`, will also be used.

Complex Scalars – Complex scalar values are implemented using the `double_complex` class provided in the standard library.

Real Vectors – This class was developed to perform operations on vectors of double precision elements when the length of the vector is known.

Complex Vectors – This class was developed to perform operations on vectors of double precision complex elements when the length of the vector is known.

Real Matrices – This class was developed to perform operations on 2-dimensional matrices of double precision elements. In addition, this class is used to support real vectors of unknown size. This was done to replicate the approach taken with the generation of the Fortran 90 code.

Complex Matrices – This class was developed to perform operations on 2-dimensional matrices of double precision complex elements. In addition, this class is used to support complex vectors of unknown size. This was done to replicate the approach taken with the generation of the Fortran 90 code.

The basic functions of the matrix classes are the same, regardless of the data types, and can be divided into three areas: matrix allocation, referencing values, and numerical operations.

Matrix Allocation

Matrices in MATLAB are dynamically allocated when needed and allowed to grow as they are accessed. To support this type of functionality, two types of allocation methods are needed within the matrix classes. The first allocation method is used for allocating the entire matrix. This is accomplished in two steps, first any existing data space is deallocated and then the data space is allocated using the new size.

The second allocation method is that which is used for extending the matrix. This allocation is performed with three slightly different steps. First, the new data space is allocated, next the data in the old data space is copied into the new data space, and then the old data space is deallocated.

Matrix Reference Types

MATLAB provides four types of references that must be supported by the matrix class:

A – Reference to the entire matrix.
$A(i, j)$ – Reference to a single element.
$A(l_i : u_i, l_j : u_j)$ – Reference to a contiguous sub-matrix.
$A(l_i : s_i : u_i, l_j : s_j : u_j)$ – Reference to sub-matrix, with strides between the elements.

However, only the first two reference types are supported by the C++ language. Therefore, the matrix class must contain methods to support the referencing of sub-matrices.

One possibility for implementing the sub-matrix references is to restrict their use to simple assignment statements where the right hand side consists of a single variable reference. With this approach, only assignment statements need to support references to sub-matrices. This approach, however, would require more temporary variables and additional copying of data.

A second approach is to create a *reference* to the matrix that incorporates the additional information regarding the requested sub-matrix. During the subsequent use of the reference, this information is utilized to correctly access the necessary elements. This is the approach that was taken by other classes, such as LAPACK++ and the C++ Matrix class, and can reduce the need to copy the data. As the colon operator is not easily supported by the C++ language, the different matrix references are identified by the number of parameters in the reference. For instance, a matrix reference with two parameters refers to a single element, a reference with four parameters refers to a contiguous sub-matrix, and a reference with six parameters refers to a sub-matrix that has strides between the elements. The reference types for both dimensions must be the same, i.e. it is not possible to use a single value for the first dimension, i, and a range of values for the second dimension, $j : k$. The reference with the three parameters would be $A(i, j, k)$ and the matrix class would not be able to differentiate $A(i : j, k)$ from $A(i, j : k)$; instead, the reference would need to be $A(i, i, j, k)$.

Numerical Operations with Matrices

All the numerical operations provided by MATLAB should be supported by the matrix classes. Currently, the matrix classes support arithmetic operations such as addition, subtraction, multiplication, and division, in addition to trigonometric functions such as cosine, sine, and tangent. The logical operations supported by the matrix classes include testing whether elements are equal or not equal. Other functions for finding maximum and minimum values are also supported. The classes support these operations not only between two variables from the same class, but also between variables from different classes.

Where possible, the numerical operations within the matrix classes are implemented using publicly available libraries, such as the BLAS. In order to facilitate the use of such libraries, the actual data elements are stored in column-major order, which is the format used by Fortran. This allows the C++ code to make use of the same optimized routines which the Fortran 90 code utilizes. Before the data can be passed to one of these routines, however, the matrix reference must first be checked to see if it refers to a sub-matrix with strides between the elements. As some of the routines require that data elements be located in contiguous storage, the sub-matrix elements may need to be copied into a contiguous temporary space before being passed to the routine.

10.2 Performance of the Generated C++ Code

To evaluate the performance of the C++ code generated by the FALCON environment, its performance was compared against the performance of other execution methods for MATLAB files. In analyzing such a comparison, however, one must realize there are multiple differences between the execution methods and that each of these will play a role in determining the performance of the code. Furthermore, the differences in the MATLAB files being tested also play a role in the performance, as some features in the files lend themselves to better performance under certain execution methods.

10.2.1 Execution Methods

Including the FALCON generated C++ code, five different execution methods were used for the performance evaluation. These tests utilized three different MATLAB compilers and generated code in three different languages in addition to the MATLAB language. A short description regarding the use of each execution method is presented.

FALCON Fortran 90 – The M-file script was translated into Fortran 90 code using the FALCON environment. The Fortran 90 code was compiled using the VAST Fortran 90 compiler, that in turn uses the Sun Fortran 77 compiler `f77`, at the `-O3` optimization level, and then executed. To perform the numerical operations, versions of the BLAS, LINPACK, LAPACK, and EISPACK libraries must be provided.

FALCON C++ – The M-file script was translated into C++ using the FALCON environment. The C++ code was compiled using the GNU C++ compiler `g++`, at the `-O` optimization level, and then executed. To perform the numerical operations, versions of the BLAS, LINPACK, LAPACK, and EISPACK libraries must be provided.

MATLAB C-MEX File – The M-file script was translated into C file using the MathWorks `mcc` compiler. The C-MEX file was compiled using the GNU C compiler `gcc`, at the `-O3` optimization level, and then executed from within the MATLAB environment. All numerical libraries used during the execution are provided by MATLAB.

MATLAB M-file – The M-file script was run within MATLAB, where the file is interpreted. This is the original execution method for the M-file. All numerical libraries used during the execution are provided by MATLAB.

MATCOM C++ – The M-file script was translated into C++ using the MATCOM C++ translator. The C++ code was compiled using the GNU C++ compiler `g++`, at the `-O` optimization level, and then executed. In addition to the libraries provided by MATCOM, a version of the BLAS libraries must also be provided.

10.2.2 Test Programs

To compare the performance of the execution methods, twelve MATLAB programs were collected. These programs implement methods 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.

AQ – A numerical approximation of the integral:

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

This code uses the Simpson's rule in conjunction with adaptive quadrature [Mat92].

CG – An iterative method for the solution of linear systems that uses the preconditioned Conjugate Gradient method with a diagonal preconditioner [BBC$^+$93]. The input data is a 420×420 stiffness matrix from the Harwell-Boeing Test Set (BCSSTK06).

CRN – A numerical approximation method for the solution of parabolic differential equations (the heat equation) [Mat92]. The problem size is a 321×321 grid.

DIR – An iterative method for the solution of Laplace's equations [Mat92]. The problem size is a 41×41 grid.

FDF – A numerical approximation method for the solution of hyperbolic differential equations (the wave equation) [Mat92]. The problem size is a 451×451 grid.

GAL – A numerical approximation method to solve the Poisson equations in two dimensions using the Galerkin method [Gar94]. The problem size is a 40×40 grid.

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

OEC – 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 [Gar94]. The program executes 6240 steps.

ORK – A numerical approximation method to solve ordinary differential equations. This program solves the orbit of a comet around the sun using a 4^{th} order Runge-Kutta method [Gar94]. The program executes 3200 steps.

QMR – A second iterative method for the solution of linear systems that uses the Quasi-Minimal Residual method [BBC$^+$93]. 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 Over-relaxation method [BBC$^+$93]. 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 surface are $51 \times 31 \times 21$.

[1]Code was provided by a colleague.

The numerical operations in the three iterative solvers, CG, QMR, and SOR, are performed at the matrix level and do not require element-wise access of the matrices. The other nine programs require element-wise access.

10.2.3 Execution Times

The performance tests were conducted on a SPARCstation 10 running in multi-user mode. Each program was run five times for each execution method, with the best CPU execution time being selected. The results from these tests are presented in Table 10.3. As the execution methods utilize similar underlying computational functions, such as the BLAS and LINPACK, the primary differences in execution time come from the overhead incurred by the different methods and differences in the program analysis. The performance of the FALCON C++ execution method will be compared, in turn, to each of the other four execution methods.

Program	FALCON		MATLAB		MATCOM
	C++	Fortran 90	C-MEX File	M-File	C++
AQ	20.30	25.71	25.70	59.47	†
CG	15.00	14.24	58.42	46.98	44.91
CRN	2.27	0.32	1.83	61.27	63.26
DIR	1.38	0.27	4.42	75.63	52.35
FDF	1.43	0.28	1.28	49.23	52.41
GAL	1.47	0.73	1.30	55.17	33.77
ICN	3.67	1.59	5.08	53.90	31.25
OEC	1.13	0.18	17.37	72.62	23.08
ORK	1.90	0.22	39.53	53.48	25.61
QMR	30.29	8.06	60.93	55.30	87.61
SOR	10.35	21.86	66.08	62.25	116.70
3D	16.18	10.47	29.42	65.12	‡

† - A segmentation fault occurs when executing the code.
‡ - The code generated by MATCOM contains an infinite loop.

Table 10.3: Execution time, in seconds, for the execution methods.

Comparison with FALCON Fortran 90

A comparison of the FALCON C++ execution times and the FALCON Fortran 90 execution times is presented in Figure 10.4. In this figure, the speedup is calculated as t_{F90}/t_{C++} when the C++ code is faster and the slowdown is calculated as t_{C++}/t_{F90} when the Fortran 90 code is faster. The Fortran 90 execution times are faster for ten of the twelve programs, with the biggest slowdowns occurring for the programs with element-wise computations. Of the two programs where the C++ code is faster, the AQ program illustrates the advantage of the improved memory allocation strategy.

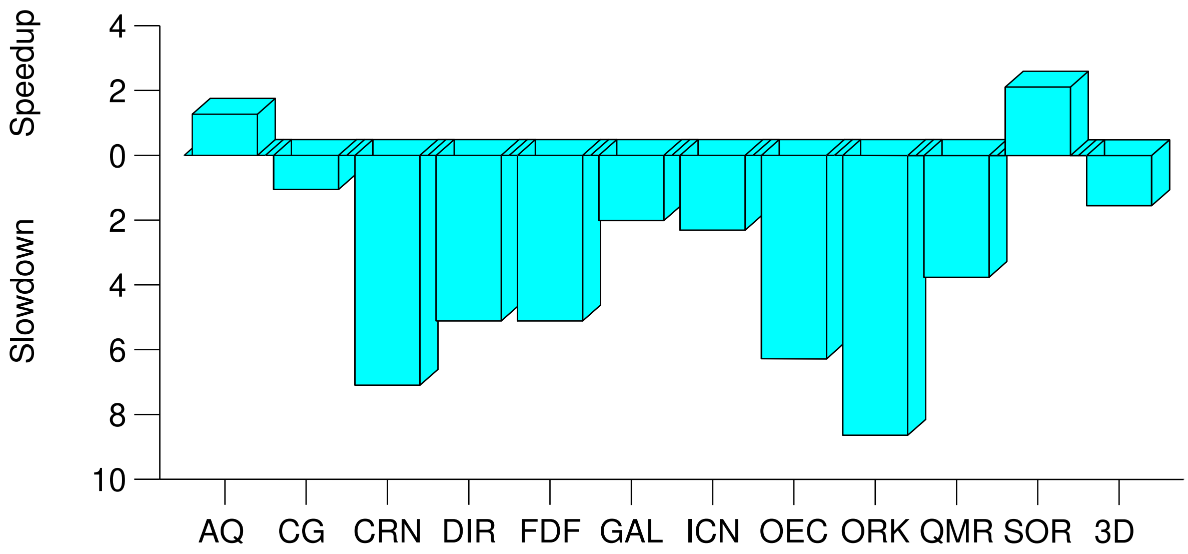

Figure 10.4: Comparison of C++ times with Fortran 90 times.

Since both the Fortran 90 and C++ codes were generated using the same program analysis and utilize the same libraries, the differences in the performance are related to the overheads incurred by the language choices. To determine why the C++ code is usually slower than the Fortran 90 code, three of the programs, CRN, OEC, and QMR, were explored in more detail.

Each loop iteration of the CRN program requires the solution of a tridiagonal linear system, using the code shown in Figure 10.5. Each of these vector accesses is performed by the invocation

```
for k = 2:n,
   mult = Va(k-1)/Vd(k-1);
   Vd(k) = Vd(k) - mult*Vc(k-1);
   Vb(k) = Vb(k) - mult*Vb(k-1);
end
X(n) = Vb(n)/Vd(n);
for k = (n-1):-1:1,
   X(k) = (Vb(k) - Vc(k)*X(k+1))/Vd(k);
end
```

Figure 10.5: Tridiagonal system solver from CRN.

of a method from the vector class. However, in order to support references to sub-arrays defined with strides, the access method is required to test for a stride during each access. To determine the amount of overhead the accesses are incurring, the C++ code was modified, by hand, to replace the access method with a reference to the values. When the modified code was run, the execution time dropped to 0.67 seconds, a reduction of 71%. In turn, the slowdown when compared to the

Fortran 90 code, was reduced from a factor of 7.1 to a factor of 2.1.

In each loop iteration of the OEC program, several calculations are performed using vectors of length two. As a result, the overhead required to invoke methods to perform the calculations is high relative to the amount of time that is spent doing the numerical operations. As two of the operations are vector-vector updates, $a = a + \alpha b$, a transformation pattern was written to replace the updates with a direct call to the BLAS primitive, as shown in Figure 10.6. When this transformation was applied, the execution time dropped to 0.63, a reduction of 44%, and the slowdown, when compared to the Fortran 90 code, was reduced from a factor of 6.3 to a factor of 3.5.

```
REPLACE
   INTEGER n
   REAL a(n), b(n){ROW}, alpha
   a = a + alpha * b;
WITH
   SUBROUTINE daxpy(P_IN,P_IN,P_IN,...
      P_IN,P_IN_OUT,P_IN)
   a = daxpy(DIM(b,2),alpha,b,1,a,1);
END
```

Figure 10.6: Transformation pattern to optimize the OEC program.

Upon examination of the QMR program, it was determined that the Fortran 90 code generator was utilizing the BLAS `dgemv` primitive to multiply the transposed matrix times the vector, while the C++ code was first transposing the matrix and then performing the multiplication. As this multiplication occurs during every loop iteration, a severe performance degradation was occurring. A transformation pattern for the purpose of recognizing the multiplication by the transposed matrix was generated and applied to the C++ code. With the improved C++ code, the execution time dropped to 8.73 seconds, a reduction of 71%. This reduced the slowdown, when compared to the Fortran 90 code, from a factor of 3.8 to a factor of 1.1.

After these tests, it is clear that the overhead to invoke the C++ methods for small data accesses has a significant performance effect. As Fortran 90 has these type of operations built into the language, it has noticeable performance advantage for such problems.

Comparison with MATLAB C-MEX

A comparison of the FALCON C++ execution times and the MATLAB C-MEX execution times is presented in Figure 10.7. In this figure, the speedup is calculated as t_{CMEX}/t_{C++} when the C++ code is faster and the slowdown is calculated as t_{C++}/t_{CMEX} when the CMEX code is faster. The C++ execution times are faster for nine of the twelve programs, with the speedups ranging from

1.3 to 20.9. Of the three programs where the CMEX execution was faster, the C++ program was slower by 12% to 24%.

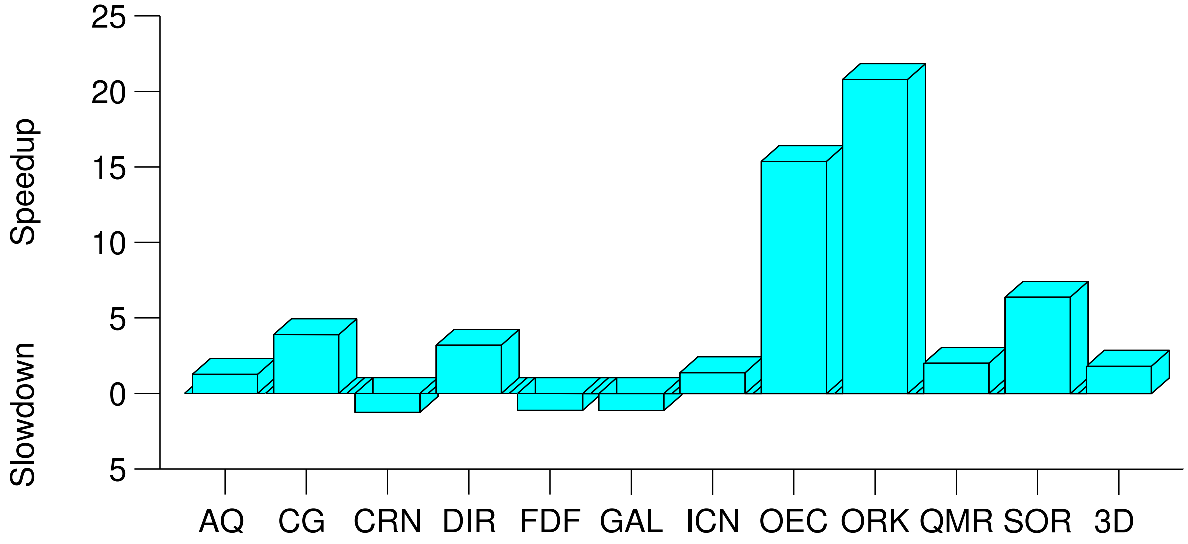

Figure 10.7: Comparison of C++ times with MATLAB C-MEX times.

The CRN program, where the CMEX program has the biggest advantage, was already examined during the comparison with Fortran 90. That analysis determined that by improving the element-wise access of the arrays, the performance of the C++ program could be improved 71%, which would be 2.7 times faster than the CMEX program. As such, it appears that the CMEX programs have better support than the C++ code for the element-wise access of arrays.

In contrast, when accesses are made to short vectors, as in the OEC program, the C++ program performs much better than the CMEX program, with the C++ program being 15.4 times faster. For instance, the `mcc` compiler used to generate the CMEX file implements both of the vector-vector updates, $a = a + \alpha b$, as matrix-matrix updates using doubly-nested loops. Also, to calculate the norm of the vector, the CMEX file packages the vector and passes it back to a MATLAB library to perform the operation. Once the operation is performed, the results must be imported back into the CMEX file. The code required to calculate the norm of a vector, `RO_ = norm(v)`, is shown in Figure 10.8. For a short vector, this overhead will dominate the time needed to calculate the norm.

Another difference between the generation of the CMEX code and the C++ code is that the FALCON program analysis system is able to infer more information about the variable properties than the `mcc` compiler. The differences in the analysis systems, as well as their performance implications, have already been addressed in [DeR96] and will not be discussed here.

It has been shown that the FALCON C++ code is generally faster than the CMEX code, with

```
Mprhs_[0] = &v;
Mplhs_[0] = 0;
mccCallMATLAB(1, Mplhs_, 1, Mprhs_, "norm", 21);
R0_ = mccImportReal(0, Mplhs_[ 0 ],
      " (tst_orbec, line 21): R0_");
```

Figure 10.8: Calculation of a norm in a CMEX file.

the FALCON C++ code again having the most difficulties with programs that have element-wise access.

Comparison with MATLAB M-File

A comparison of the FALCON C++ execution times and the MATLAB M-File execution times is presented in Figure 10.9. In this figure, the speedup is calculated as t_{MFile}/t_{C++}. The C++ execution times are faster for all twelve programs, with the speedups ranging from 1.8 to 64.3.

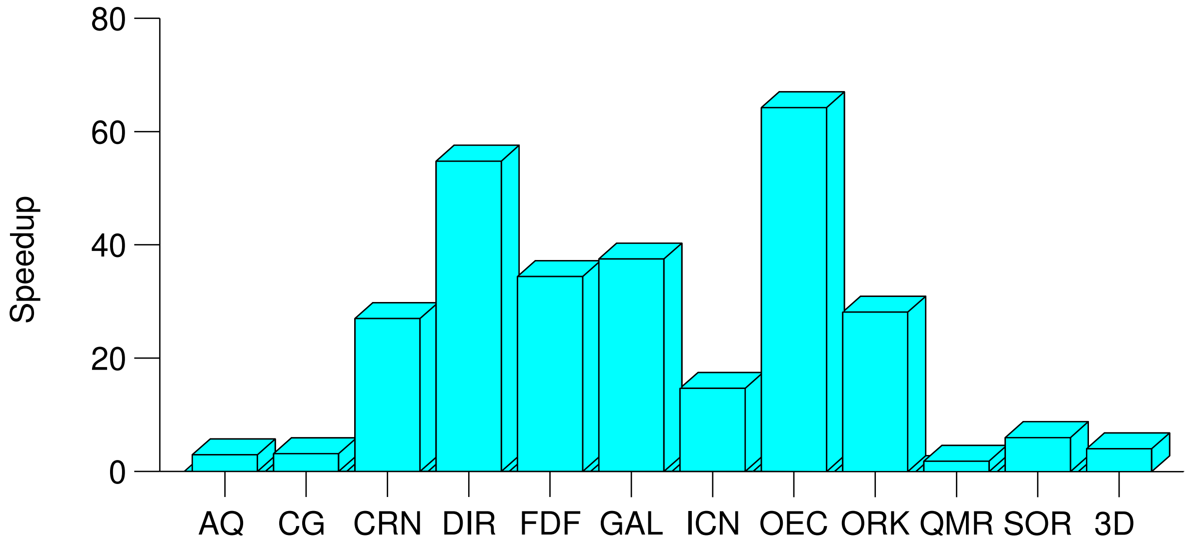

Figure 10.9: Comparison of C++ times with MATLAB M-File times.

The program where the least speedup occurred was already examined during the comparison with the Fortran 90 code. There it was determined that the C++ code could be improved by using the BLAS `dgemv` primitive to perform the multiplication of the transposed matrix times the vector rather than performing the matrix transpose during every loop iteration. As this improvement reduced the C++ execution time to 8.73 seconds, it would also increase the speedup from 1.8 to 6.33.

The speedups for the other two iterative solvers, CG and SOR, are also fairly small. In both of these codes, most of the time is spent in BLAS primitives performing matrix-vector multiplications or triangular solves. Since both the C++ code and the M-File code call numerical libraries to perform these operations, the numerical operations will take a similar amount of time in this case. In this case, the speedup comes from reducing the loop overhead and improving the performance of the scalar and vector operations.

It has been shown that the C++ code provides better performance than the MATLAB M-Files for all of the test cases. The biggest performance improvements are found for the programs with element-wise or short vector accesses.

Comparison with MATCOM C++

A comparison of the FALCON C++ execution times and the MATCOM C++ execution times is presented in Figure 10.10. In this figure, the speedup is calculated as t_{MATCOM}/t_{C++}. The FALCON execution times are faster for all ten programs where the MATCOM code executed successfully, with the speedups ranging from 2.9 to 37.9.

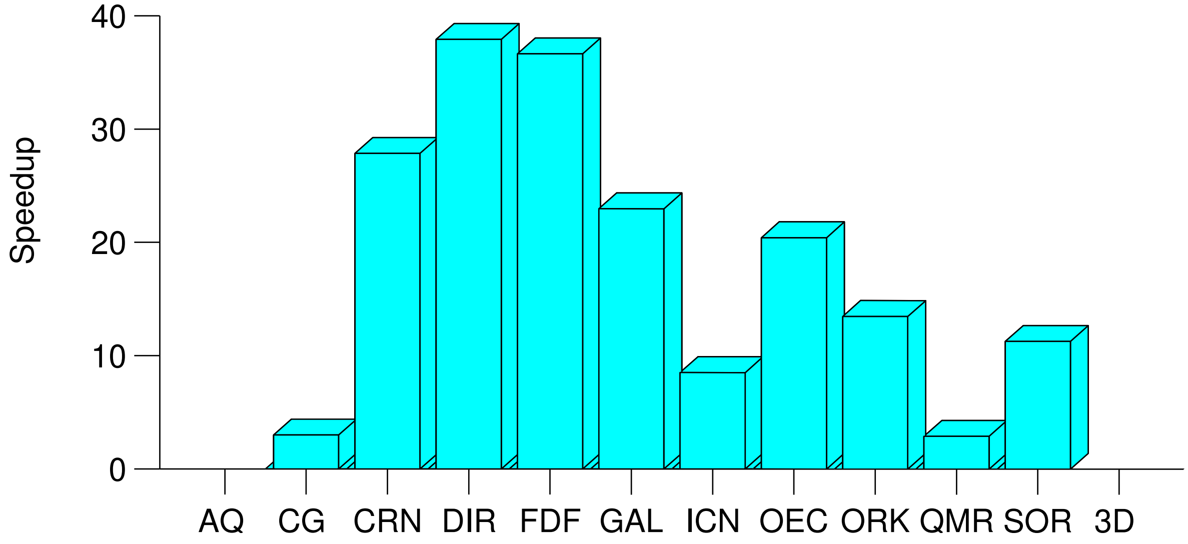

Figure 10.10: Comparison of C++ times with MATCOM C++ times.

As both of these execution methods utilize the same language, this comparison highlights the difference between the utilization of extensive program analysis to determine variable properties, the FALCON approach, and the use of a single powerful matrix class to perform these determinations at run time, the MATCOM approach. Not only does the FALCON code execute faster than the MATCOM code for the ten programs, the compiled MATCOM code is slower than the MATLAB M-Files for four of the ten programs. This indicates that the run-time decisions of the MATCOM matrix class are sometimes even slower than the run-time decisions of the MATLAB interpreter.

For eight of the programs, the FALCON code runs at least 8.5 times faster than the MATCOM code. These programs include those that operate on both entire matrices and element-wise accesses. Of the two remaining programs, if the FALCON version of the QMR program is improved using the `dgemv` optimization, its speedup increases to 10.0. This clearly shows that utilizing the program analysis system of FALCON to eliminate run-time decisions provides a distinct performance advantage.

Another issue with the MATCOM compiler is that it failed to generate correct code for two of the tests and some of the codes required modification before the MATCOM compiler would work. For both the AQ and 3D programs, the MATCOM compiler generates C++ code that can be compiled by the `g++` compiler. However, when executing the AQ program, a segmentation fault occurs; when executing the 3D code, the program enters an infinite loop.

Also, several changes had to be made to the programs with regards to how data is loaded. MATCOM does not support the standard MATLAB syntax for loading data from a file:

`load` *file_name*.

Instead, the variables to be loaded must be appear after the file name in the order in which they occur in the file:

`load` *file_name variable_1 variable_2 variable_3* $\cdots$.

Furthermore, if the file name in the load command does not contain the `.mat` suffix, and there is both a `file.m` and `file.mat` present, MATCOM will incorrectly attempt to load the data from `file.m`.

In addition, MATCOM does not appear to support the overloading of certain MATLAB functions. For instance, several of the programs defined a variable `error` within the calculations. However, when references to the `error` symbol were encountered during compilation, the MATCOM compiler complains that the MATLAB `error` function does not have enough parameters. Nor does MATCOM support the `cputime` function for calculating execution time.

10.3 Comparison of C++ Code to Fortran 90 Code

One of the primary interests in using C++ was the hope that the object-oriented nature of C++ would lead to code that was easier to read and maintain. In examining the test codes, this appears to be true for certain areas, such as data allocation, but there are some MATLAB functionalities that the Fortran 90 language supports that are difficult to represent in C++. As a first step in comparing the languages, Table 10.4 contains the number of lines of required to represent the MATLAB test programs for both languages.

Program	Lines of Code		
	Fortran 90	C++	Reduction
AQ	358	221	38%
CG	538	360	33%
CRN	232	138	41%
DIR	276	135	51%
FDF	171	91	47%
GAL	456	354	22%
ICN	384	278	28%
OEC	208	122	41%
ORK	344	263	24%
QMR	1080	737	32%
SOR	551	394	28%
3D	263	163	38%

Table 10.4: Comparison of lines of code from the C++ and Fortran 90 programs.

As can be seen from this table, the C++ program contains fewer lines for all twelve of the programs, with the differences in lines ranging from 24% to 51% fewer lines. To better understand the difference in the C++ and Fortran 90 codes, two specific code constructs will be examined, the way matrices are allocated and the way vectors are initialized. In the first example, an advantage of C++ will be shown and in the second example, an advantage of Fortran 90 will be discussed.

10.3.1 Matrix Allocation Code

A primary difference in the matrix allocation code results from the different approaches to memory allocation in the two languages. Because Fortran 90 allocates memory within the scope of a subroutine, it is difficult to allocate memory within a subroutine and return this memory to the calling subroutine. As a result, the Fortran 90 code generator includes all the code for memory allocation within the main program. Within the C++ code, however, all the memory allocation takes place within the matrix class. As a result, the C++ code generator can represent all the memory allocation functions with a single statement.

For example, consider the following indexed assignment statement from the AQ MATLAB program:

```
SRmat(p,:)  = SR0vec;
```

The Fortran 90 implementation of this statement requires 28 lines of code, as shown in Figure 10.11. In contrast, the C++ implementation of this assignment requires only two statements, as shown in Figure 10.12. In this case, the ability of the C++ class to encapsulate the memory allocation code within a method makes the resulting code far simpler.

```
if (p .gt. SRmat__30__D1 ) then
   if (ALLOCATED(SRmat__30)) then
      T0__D1 = SRmat__30__D1
      T0__D2 = SRmat__30__D2
      ALLOCATE(T0__R(T0__D1, T0__D2))
      T0__R = SRmat__30
      DEALLOCATE(SRmat__30)
      SRmat__30__D1 = MAX(SRmat__30__D1, p)
      ALLOCATE(SRmat__30(SRmat__30__D1, SRmat__30__D2))
      SRmat__30(1:T0__D1, 1:T0__D2) = T0__R
      SRmat__30(1:T0__D1, T0__D2+1:SRmat__30__D2) = 0.
      SRmat__30(T0__D1+1:SRmat__30__D1, :) = 0.
      DEALLOCATE(T0__R)
   else
      SRmat__30__D1 = p
      SRmat__30__D2 = 6
      ALLOCATE(SRmat__30(SRmat__30__D1, SRmat__30__D2))
      SRmat__30 = 0.
   end if
else
   if (.not. ALLOCATED(SRmat__30)) then
      SRmat__30__D1 = p
      SRmat__30__D2 = 6
      ALLOCATE(SRmat__30(SRmat__30__D1, SRmat__30__D2))
      SRmat__30 = 0.
   end if
end if
SRmat__30(p , :) = SR0vec
```

Figure 10.11: Fortran 90 implementation of the indexed allocation.

```
SRmat__30.allocateIndex(p, 6, &(SRmat__30$d1), &(SRmat__30$d2));
SRmat__30(p, p, 1, 6) = SR0vec;
```

Figure 10.12: C++ implementation of the indexed allocation.

10.3.2 Vector Initialization

In contrast to the matrix allocation code, the Fortran 90 code benefits from the support for array operations, such as vector initialization, which is built into the language. This allows an entire vector to be initialized with a single statement. For the C++ code, the vector initialization is implemented with a separate assignment for each element of the vector.

For example, consider the following matrix initialization from the 3D program:

```
amat=[a(1,kk) b(1,ii) c(1,jj);1 0 0;0 1 0];
```

In the Fortran 90 code, this initialization is performed as three vector initializations and requires a total of seven statements, as shown in Figure 10.13. For the C++ code, each element has a separate assignment statement and a total of sixteen statements are required, as shown in Figure 10.14. In this case, the additional features built into the Fortran 90 language make the code simpler and easier to understand.

```
IT__1 = 1
amat(1,:) = (/ a(1 , kk), b(1 , ii), c(1 , jj) /)
IT__1 = 2
amat(2,:) = (/ 1, 0, 0 /)
IT__1 = 3
amat(3,:) = (/ 0, 1, 0 /)
IT__1 = 4
```

Figure 10.13: Fortran 90 implementation of vector initialization.

```
IT__1 = 1;
IT__3 = 1;
amat(IT__1, (IT__3)++) = a(1, kk);
amat(IT__1, (IT__3)++) = b(1, ii);
amat(IT__1, (IT__3)++) = c(1, jj);
IT__1 = 2;
IT__3 = 1;
amat(IT__1, (IT__3)++) = 1;
amat(IT__1, (IT__3)++) = 0;
amat(IT__1, (IT__3)++) = 0;
IT__1 = 3;
IT__3 = 1;
amat(IT__1, (IT__3)++) = 0;
amat(IT__1, (IT__3)++) = 1;
amat(IT__1, (IT__3)++) = 0;
IT__1 = 4;
```

Figure 10.14: C++ implementation of vector initialization.

Chapter 11

SUPPORT FOR SPARSE PARALLEL CODES

The previous discussion in Chapter 8, on generating code for sparse matrices, focused on the replacement of the dense primitives with sparse primitives based on a global matrix structure. In addition to this approach, there are also techniques for improving the performance of codes for general sparse problems by extracting both local and global structure information from a sparse matrix instance. This information can be used to improve the performance of the primitives through the utilization of specialized methods for the component parts which result from the matrix decomposition.

In the context of the design of computational primitives for the multiplication of a sparse matrix and a vector, Agarwal, Gustavson and Zubair have advocated an analysis procedure that decomposes a sparse matrix into a sum of several sparse matrices, each of which possess a sparsity structure that can be exploited to enhance performance [AGZ92]. In that work, the extraction and exploitation of sparsity structure was limited to the scope of the matrix-vector product primitive. When placed in the context of, say, one iteration of a sparse linear system solver, it is clear that the structural information can be propagated to the other computation primitives that surround the matrix-vector product routine, to enhance code efficiency.

Also, Bik and Wijshoff have developed tools for the selection of sparse data structures, based on the reference patterns, and for the extraction of features from a sparse matrix [BW94]. These tools can be used to select the appropriate control structures and to generate efficient Fortran code. However, in their work, no algebraic information was used to aid in code generation.

11.1 Matrix Decomposition using Problem Specific Information

In order to utilize the matrix decompositions, it is first necessary to define the algebraic relationship between the matrix and its decomposition, of which there are two main relationships.

1. If no matrix element is in more than one component and every element is in a component, then the original matrix is equal to the sum of the component matrices.

$$A = \sum A_k \tag{11.1}$$

2. If no matrix element may be in more than one component and every element is in at least one component, then the original matrix is equal to the merge of the component matrices.

$$A = \amalg A_k \tag{11.2}$$

Where:

$\amalg_{k=1}^{m} A_k = A_1 \vee A_2 \vee \cdots \vee A_m$

and $\vee$ is an arithmetic OR, defined as follows:

Let Υ be the undefined value, then

$A = A_1 \vee A_2$ is defined as

$a_{i,j} = a_{1_{i,j}} \vee a_{2_{i,j}} \forall(i,j)$ such that

If $a_{1(i,j)} = a_{2(i,j)}$ then $a_{(i,j)} \leftarrow a_{1(i,j)}$

else if $a_{1(i,j)} = \Upsilon$ then $a_{(i,j)} \leftarrow a_{2(i,j)}$

else if $a_{2(i,j)} = \Upsilon$ then $a_{(i,j)} \leftarrow a_{1(i,j)}$

else an invalid state.

Using these composition rules, several matrix decompositions can be defined. For example, a non-overlapping dense block decomposition could be defined as:

$$\begin{aligned} A &= \sum A_k, \text{ where} \\ A_k(il_k : iu_k, jl_k : ju_k) &= A(il_k : iu_k, jl_k : ju_k) \\ A_k &= 0 \text{ elsewhere.} \end{aligned}$$

Since this is non-overlapping, no element can appear in more than one component. If multiple component parts could contain the same element, then the composition operator would be a merge rather than a sum and the value of the remaining elements would be Υ instead of 0.

The block description just defined is sufficient to represent all other sparse matrix structures through the selection of suitable blocks. This is guaranteed to be true because each of the nonzero elements in any sparse structure could be represented with a block of size 1×1. For example, in Figure 11.1 a lower triangular matrix is shown and in Figure 11.2 a tri-diagonal matrix is shown.

Instead of just considering dense blocks, this decomposition can also support *nearly dense* blocks, where some of the elements within the blocks are allowed to be zero. By allowing the constraint of

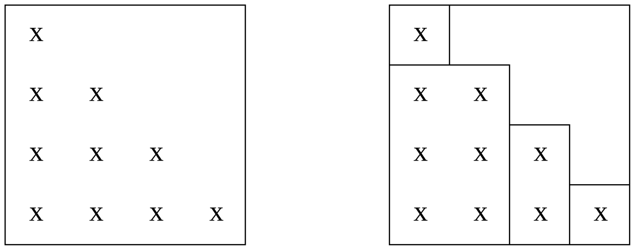

Figure 11.1: A lower triangular matrix and its non-overlapping block decomposition.

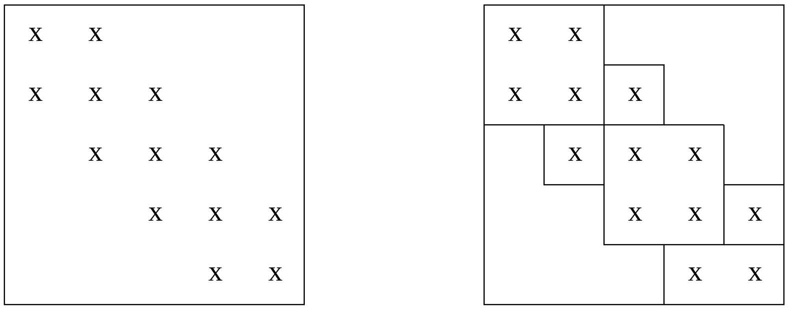

Figure 11.2: A tridiagonal matrix and its non-overlapping block decomposition.

dense blocks to be loosened, the number of blocks required to define the matrix may be reduced and the block sizes may be increased. However, zero elements contained with the blocks will generate additional operations and require additional storage space. The number of extra zeros to allow, therefore, is dependent upon the matrix structure, the operations being performed, and the machine architecture. In Figure 11.3 the *nearly dense* block decomposition of the lower triangular matrix is shown. In this example at most one zero element occurs within each dense block and only three evenly-sized blocks are required to represent the matrix, as compared to the four unevenly-sized blocks in Figure 11.1.

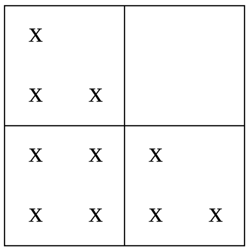

Figure 11.3: A lower triangular matrix and its *nearly dense* block decomposition.

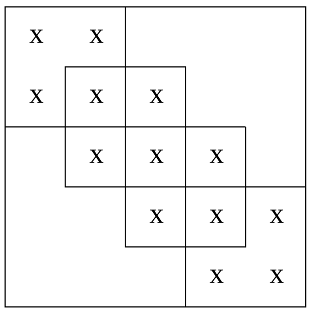

Figure 11.4: A tridiagonal matrix and its overlapping block decomposition.

In contrast to the non-overlapping block decompositions, some structures are more easily represented using overlapping blocks. Consider in Figure 11.4, where the tridiagonal matrix is represented with overlapping blocks. In this case, only four blocks are needed, while six non-overlapping blocks were required. Though it may be simpler to represent matrices with overlapping blocks, the subsequent use of the decomposed blocks may be more complicated, requiring extra synchronization, and redundant arithmetic operations will occur with such use. For these reasons, only non-overlapping decompositions will be considered further.

Given that such decompositions are possible, there must be a means by which the decomposition can be determined. One possibility is to determine the decomposition by inspection. This may be a reasonable approach when there are only a few component structures in the matrix or when the matrix has an overall global structure. In the case of a general sparse matrix, however, this approach will become very tedious.

However, an analysis engine can be used to collect characteristics of the input data and supply this information to the code generator system automatically. The tool of Bik and Wijshoff, previously mentioned, is able to detect several well-known forms in $\mathcal{O}(n + nnz)$ time. In addition, some statistical information about the nonzero structure is computed, comparable with the information that is gathered by the tool of Saad (SPARSKIT) [Saa90]. Although originally developed for a compiler that transforms dense programs into semantically equivalent sparse code at programming level, the tool of Bik and Wijshoff can also be used to control code generation at this higher level.

11.2 Operator Decomposition

Once the matrix has been decomposed, the next step is the modification of the operations with the matrix to use the component parts. To better understand the issues involved with decomposing the operations, consider one operation, a matrix-vector multiplication, $p = A * q$.

1. Replace A with its block decomposition.

$$p = (\sum A_k) * q$$

2. Use the distributive properties of multiplication and addition to distribute the matrix-vector multiplications.

$$p = \sum (A_k * q)$$

3. Use the definition of the block decomposition to eliminate multiplication operations where the component matrix elements are defined to be zero.

$$p_k(il_k : iu_k) \quad = \quad A_k(il_k : iu_k, jl_k : ju_k) * q(jl_k : ju_k)$$

$$
\begin{aligned}
p_k &= 0 \text{ elsewhere} \\
p &= \sum p_k
\end{aligned}
$$

It is important to realize that even though the matrix decomposition was defined to have non-overlapping values, the resulting vectors, p_k do not maintain this property. However, in contrast to the overlapping matrix definition which uses the merge operator for the composition, in this case the overlapping values are combined with the sum operator.

In order to achieve the best results from these decompositions, however, is important to deal with more than a single operation. When a sequence of operations can be decomposed together, it allows the overhead of the decomposition to be amortized over more operations, which can be especially important for parallel machines.

For example, consider the following sequence which contains three operations.

$p = r + \beta * p$
$q = A * p$
$\alpha = p^T * q$

As in the previous example, the matrix-vector multiplication can be decomposed by the block matrix decomposition. This changes the code sequence to the following.

$p = r + \beta * p$
For $k = 1 : m$
    $q_k(il_k : iu_k) = A_k(il_k : iu_k, jl_k : ju_k) * p(jl_k : ju_k)$
End For
$q = \sum q_k$
$\alpha = p^T * q$

This version of the algorithm has now defined a decomposition for the p vector which can be used to only calculate the portion of p that is necessary for the associated loop iteration.

For $k = 1 : m$
    $p_k(jl_k : ju_k) = r(jl_k : ju_k) + \beta * p(jl_k : ju_k)$
    $q_k(il_k : iu_k) = A_k(il_k : iu_k, jl_k : ju_k) * p_k(jl_k : ju_k)$
End For
$p = \amalg p_k$
$q = \sum q_k$
$\alpha = p^T * q$

The final step, therefore, is to decompose the inner product, $p^T * q$. However, the index sets which were used to compute the two operands do not align for the inner product. The operand p was calculated for indices $jl_k : ju_k$ and operand q was calculated for indices $il_k : iu_k$. Given that the cost of computing a new set of values for q is much greater than that for p, the index set for p can be modified to compute the values for both the matrix-vector operation and the inner-product operation.

For $k = 1 : m$
  $[ijl, iju] = il_k : iu_k \cup jl_k : ju_k$
  $p_k([ijl, iju]) = r([ijl, iju]) + \beta * p([ijl, iju])$
  $q_k(il_k : iu_k) = A_k(il_k : iu_k, jl_k : ju_k) * p_k(jl_k : ju_k)$
  $\alpha_k = p_k(il_k : iu_k)^T * q(il_k : iu_k)$
End For
$p = \amalg p_k$
$q = \sum q_k$
$\alpha = \sum \alpha_k$

As illustrated by this example, the decomposition of multiple operations becomes complicated as more of the operands are decomposed. The code to perform the operations becomes burdensome and difficult to write by hand. Furthermore, the code transformations must be repeated for each use, making the process even more tedious. To make the process simpler, it would be convenient to have a code generation system which could take the algorithm, decompose the matrices, and generate the subsequent code for the operations.

In order to generate parallel code, the dependencies within the decomposed operations must also be defined. For instance, when a matrix-vector operation is decomposed, each of the component multiplications can be done in parallel. However, when a solve is decomposed, certain operations must be synchronized. As an example, consider the algorithm for doing a lower triangular solve, $Ly = b$, in parallel using the block decomposition of the matrix. (This example assumes there are k blocks which contain diagonal elements, and that all off-diagonal blocks only contain rows from a single diagonal block. If the matrix decomposition does not obey the condition, it is simple to decompose the off-diagonal block into multiple blocks such that the condition is met.)

$y_1 = b$
Parallel For $i = 1 : k$
  For $j = 1 : i - 1$
    If $A_{i,j}$ is nonzero then
      Wait for y_j
      $y_i = y_i - A_{i,j} * y_j$
    End If
  End For

$y_i = A_{i,i} \backslash y_i$
Broadcast y_i
End Parallel For

For a code generation system to perform the automatic code transformations, however, the rules for the composition and decomposition of matrix and vector operations must be well defined. These definitions must also include information about how operations may be combined and the order in which they can be executed, as well as the synchronization required between operations or sequences of operations.

11.3 The Underlying Calculus

The calculus to support an automatic transformation system must achieve three goals: define the decomposition of operations, define how decomposed operations can be organized, and define how decomposed results can be combined. This is accomplished by defining two levels of rules for the transformation system. At the lowest level, the rules define how the operators perform when using decomposed operands. The second level of rules define how the decomposed results are combined in order to form the final results. With these two levels of rules, as well as the standard algebraic rules, the calculus is defined.

In order to understand the rules, examples will be presented illustrating how a few of the rules were derived. Following are the definitions of the operands and operations that will be used within the calculus.

- Operand types: scalars, vectors, transposed vectors, and matrices.
- Arithmetic Operations: $+$, $-$, $*$, and $/$ (or the equivalent solves).
- Composition Operations: sum and merge.

For a single operation, p *op* q, there are four possible combinations which use a single composition operation:

1. p *op* $\sum(q_k)$,
2. $\sum(p_k)$ *op* q,
3. p *op* $\amalg(q_k)$, and
4. $\amalg(p_k)$ *op* q.

Now, consider the multiplication operator and how it performs for some of the possible operand combinations, using the sum composition. Each operation will be transformed into two statements, the first statement to compute a *partial* value using the decomposed component and a second statement to compute a *final* value by combining the *partial* values.

1. $\alpha = \beta * \sum \gamma_k \Rightarrow$
$$\alpha_k = \beta * \gamma_k \; \forall \; k$$
$$\alpha = \sum \alpha_k$$

2. $p = q^T * \sum r_k \Rightarrow$
$$p_k = q^T * r_k \; \forall \; k$$
$$p = \sum p_k$$

3. $p = A * \sum q_k \Rightarrow$
$$p_k = A * q_k \; \forall \; k$$
$$p = \sum p_k$$

In these examples, notice how the distributive properties of multiplication and addition allow the composition operator to be a sum.

Performing similar operations with the addition operator, however, leads to different results.

1. $\alpha = \beta + \sum \gamma_k \Rightarrow$
$$\alpha_1 = \beta + \gamma_1$$
$$\alpha_k = \gamma_k \; \forall \; k > 1$$
$$\alpha = \sum \alpha_k$$

2. $p = q + \sum r_k \Rightarrow$
$$p_1 = q + r_1$$
$$p_k = r_k \; \forall \; k > 1$$
$$p = \sum p_k$$

3. $A = B + \sum C_k \Rightarrow$
$$A_1 = B + C_1$$
$$A_k = C_k \; \forall \; k > 1$$
$$A = \sum A_k$$

Here, the addition operator cannot be distributed over the decomposition and, therefore, requires different operations with the decomposed components. Yet, given the right operations, the composition operator remains a sum.

From these examples, it is apparent that the decomposed operations can be selected, based on the distributive property of the operation, such that the composition will remain a sum. In fact, these rules can be abstracted as follows.

1. Op is distributive and commutative.

   (a) $p\ op\ \sum q \rightarrow \sum(p\ op\ q)$

   (b) $\sum p\ op\ q \rightarrow \sum(p\ op\ q)$

2. Op is distributive and not commutative.

   (a) If op is right distributive:
   $p\ op\ \sum q \rightarrow \sum(p\ op\ q)$

   (b) If op is left distributive:
   $\sum p\ op\ q \rightarrow \sum(p\ op\ q)$

3. Op is associative and not commutative and not distributive.

   (a) $p\ op\ \sum q \rightarrow (p\ op\ \sum_{i \leq k} q_i) + \sum_{i>k} q_i$

   (b) $\sum p\ op\ q \rightarrow \sum_{i<k} p_i + (\sum_{i \geq k}(p_i)\ op\ q)$

4. Op is associative and commutative and not distributive.

   Let:

   $$\begin{aligned} S &= \text{set of all possible instances} \\ S_1 &\subseteq S \\ S_2 &= S - S_1 \end{aligned}$$

   (a) $p\ op\ \sum q \rightarrow (p\ op\ \sum_{i \in S_1} q_i) + \sum_{i \in S_2} q_i$

   (b) $\sum p\ op\ q \rightarrow \sum_{i \in S_2} p_i + (\sum_{i \in S_1}(p_i)\ op\ q)$

Therefore, given that such rules are based on the algebraic properties of the operation, the properties must be enumerated for each operation. This can be accomplished using a simple table format as follows for the basic operation form $p + \sum q$. As the '+' operation is commutative, the table is also valid for the basic form $\sum(p) + q$.

Let

A = associativity, and

C = commutativity.

$+ \sum$	*scalar*	*vector*	*vector*^{T}	*matrix*
scalar	A,C	A,C	A,C	A,C
vector	A,C	A,C		
vector^T	A,C		A,C	
matrix	A,C			A,C

An empty slot in the table indicates the operation is not valid.

Given that the code transformations are based on the algebraic properties of associativity, distributivity, and commutativity, the transformed code will generate the correct results in true arithmetic. However, when floating point operations are reorganized within computer programs, variations in numerical results can occur.

Having defined the calculus for the compositions, it is important to note that the calculus only defines the high-level code transformations. In order to actually generate code, several issues which were mentioned in the previous section, must also be resolved. First, strategies must be developed to determine if the redundant computations which can arise when coordinating multiple operation decompositions should be performed. For sequential machines, the redundant computations are unnecessary, but for parallel machines, the performance trade-off between communicating the values versus recomputing the values must be examined. The second issue is that in order to generate parallel code, additional information, such as dependency analysis, is necessary. As was shown in the triangular solve example, it is sometimes necessary to generate synchronization and data communication primitives.

11.4 Implementation of the Decompositions

Using the calculus just defined, it would be possible to incorporate the matrix decomposition into a code development environment, such as FALCON. The following high-level algorithm illustrates how the decomposition of the sparse matrices could be implemented.

1. Find an operation involving a matrix operand.
2. Decompose the operation using the matrix decomposition.
3. Backtrack to the definition of the other operand.
4. Propagate the decomposed result.

This algorithm requires two other procedures, a backtrack algorithm and a propagate algorithm. These are defined as follows:

- Backtrack Algorithm
  This algorithm starts with a required result index set and a target result variable defined as *target* = *left op right*.

  If (decomposition(*left*) *op* decomposition(*right*) = *target* index set) then
  Decompose operation using *target* index set
  Backtrack *left*

Backtrack *right*
Propagate *target*
End If

- Propagate Algorithm

  This algorithm starts with a source variable which has been decomposed for a given index set and a list of all the uses of the source variable definition.

```
For each use of source
    If ( ( result = source op other_operand) can be decomposed ) then
        Decompose ( result = source op other_operand)
        Backtrack other_operand
        Propagate other_operand
        Propagate result
    End If
End For
```

Within the algorithms, certain tests must be performed to determine if an operation can be decomposed. These tests are based not just on the calculus which defines the decompositions, but also on the comparison of the index sets which are utilized in the operations. As was shown in the previous example, when combining multiple operations, it is necessary to resolve any differences in index sets which may arise. If the differences cannot be resolved, then the operation cannot be decomposed.

11.5 Example Decompositions

To illustrate the effects of these transformations on actual codes, two examples from the solution of linear systems are considered. The first example will be the Conjugate Gradient (CG) iterative solver and the second example is Gaussian elimination without pivoting. The original code for the CG algorithm, as presented in Figure 11.5, was taken from [BBC+93]. This code was modified to eliminate the conditional in the loop and a diagonal preconditioner has been incorporated.

Within each loop iteration of this algorithm there are two matrix operations. The first operation performs the solve with the preconditioner and the second operation is the matrix-vector multiplication. Consider the decomposition of these operations when the matrix A is a block tridiagonal matrix.

Given that the preconditioner is the diagonal of the A, the preconditioner can be decomposed using the same block structure. Therefore, the preconditioner solve can be performed by blocks,

$M = diag(A)$
$x = 0$
$r = b - A * x$
$\rho = 1$
$p = 0$
Loop
  $\rho_1 = \rho$
  Solve $Mz = r$
  $\rho = r^T * z$
  $\beta = \rho/\rho_1$
  $p = z + \beta * p$
  $q = A * p$
  $\alpha = \rho/p^T * q$
  $x = x + \alpha * p$
  $r = r - \alpha * q$
  Test for convergence
End Loop

Figure 11.5: Original code for the Conjugate Gradient algorithm.

---

using a divide operation for the solve. Assuming the matrix has m diagonal blocks, the solve operation can be decomposed as follows.

For $k = 1 : m$
  $z_k(il_k : iu_k) = r_k(il_k : iu_k)./M_k(il_k : iu_k)$
End For
$z = \sum z_k$

Having decomposed z, it is also possible to decompose the inner product used for calculating ρ.

For $k = 1 : m$
  $\rho_k = r(il_k : iu_k)^T * z_k(il_k : iu_k)$
End For
$\rho = \sum \rho_k$

Given the block tridiagonal decomposition of the matrix A, the matrix-vector multiplication can be decomposed as follows. (Again, assuming there are m diagonal blocks.)

For $k = 1 : m$
  $q_k(il_k : iu_k) = A_{k,k-1}(il_k : iu_k, jl_{k-1} : ju_{k-1}) * p(jl_{k-1} : ju_{k-1})$
  $q_k(il_k : iu_k) = q_k(il_k : iu_k) + A_{k,k}(il_k : iu_k, jl_k : ju_k) * p(jl_k : ju_k)$
  $q_k(il_k : iu_k) = q_k(il_k : iu_k) + A_{k,k+1}(il_k : iu_k, jl_{k+1} : ju_{k+1}) * p(jl_{k+1} : ju_{k+1})$
End For
$q = \sum q_k$

Using the decomposition of p in the matrix-vector multiplication, the calculation of p can also be decomposed.

For $k = 1 : m$

$p_k(jl_{k-1} : ju_{k+1}) = z(jl_{k-1} : ju_{k+1}) + \beta * p(jl_{k-1} : ju_{k+1})$

End For

$p = \Pi p_k$

With both p and q decomposed, the calculation of the inner product can also be decomposed.

For $k = 1 : m$

$\alpha_k = p_k(il_k : iu_k)^T * q_k(il_k : iu_k)$

End For

$\alpha = \rho / \sum \alpha_k$

Figure 11.6 contains the code which results when placing the decomposed operations back into the CG algorithm.

As a second example, Gaussian Elimination without pivoting will be considered, with Figure 11.7 containing the original version of the algorithm for dense matrices. This algorithm is very simple, requiring only two operations, the scaling of a vector and a rank-1 update. In the case of a block tridiagonal matrix, the operations will involve a limited number of blocks per iteration. As is shown in Figure 11.8, the scaling operation will update two blocks and the rank-1 update will update four blocks. The original Gaussian elimination code can be rewritten to perform the operations on a block level.

First, the scaling operation can be decomposed into two block updates.

$A_{k,k}(i+1 : iu_k, i) = A_{k,k}(i+1 : iu_k, i)/A_{k,k}(i,i)$

If $il_{k+1} < n$ then

$A_{k+1,k}(il_{k+1} : iu_{k+1}, i) = A_{k+1,k}(il_{k+1} : iu_{k+1}, i)/A_{k,k}(i,i)$

End If

Next, the rank-1 update can be decomposed into four rank-1 updates.

$A_{k,k}(i+1 : iu_k, i+1 : ju_k) = A_{k,k}(i+1 : iu_k, i+1 : ju_k) -$
$A_{k,k}(i+1 : iu_k, i) * A_{k,k}(i, i+1 : ju_k)$

If $il_{k+1} < n$ then

$A_{k,k+1}(i+1 : iu_k, jl_{k+1} : ju_{k+1}) = A_{k,k+1}(i+1 : iu_k, jl_{k+1} : ju_{k+1}) -$
$A_{k,k}(i+1 : iu_k, i) * A_{k,k+1}(i, jl_{k+1} : ju_{k+1})$

$A_{k+1,k}(il_{k+1} : iu_{k+1}, i+1 : ju_k) = A_{k,k+1}(il_{k+1} : iu_{k+1}, i+1 : ju_k) -$
$A_{k+1,k}(il_{k+1} : iu_{k+1}, i) * A_{k,k}(i, i+1 : ju_k)$

$r = b - A * x$
$\rho = 1$
$p = 0$
Loop
    $\rho_1 = \rho$
    For $k = 1 : m$
        $z_k(il_k : iu_k) = r(il_k : iu_k)./M(il_k : iu_k)$
        $\rho_k = r(il_k : iu_k)^T * z_k(il_k : iu_k)$
    End For
    $z = \sum z_k$
    $\rho = \sum \rho_k$
    $\beta = \rho/\rho_1$
    For $k = 1 : m$
        $p_k(jl_{k-1} : ju_{k+1}) = z(jl_{k-1} : ju_{k+1}) + \beta * p(jl_{k-1} : ju_{k+1})$
        $q_k(il_k : iu_k) = A_{k,k-1}(il_k : iu_k, jl_{k-1} : ju_{k-1}) * p_k(jl_{k-1} : ju_{k-1})$
        $q_k(il_k : iu_k) = q_k(il_k : iu_k) + A_{k,k}(il_k : iu_k, jl_k : ju_k) * p_k(jl_k : ju_k)$
        $q_k(il_k : iu_k) = q_k(il_k : iu_k) + A_{k,k+1}(il_k : iu_k, jl_{k+1} : ju_{k+1}) * p_k(jl_{k+1} : ju_{k+1})$
        $\alpha_k = p_k(il_k : iu_k)^T * q_k(il_k : iu_k)$
    End For
    $p = \amalg p_k$
    $q = \sum q_k$
    $\alpha = \rho / \sum \alpha_k$
    $x = x + \alpha * p$
    $r = r - \alpha * q$
    Test for convergence
End Loop

Figure 11.6: Decomposed code for the Conjugate Gradient algorithm.

$$A_{k+1,k+1}(il_{k+1} : iu_{k+1}, jl_{k+1} : ju_{k+1}) = A_{k,k+1}(il_{k+1} : iu_{k+1}, jl_{k+1} : ju_{k+1}) - A_{k+1,k}(il_{k+1} : iu_{k+1}, i) * A_{k,k+1}(i, jl_{k+1} : ju_{k+1})$$

    End If

The decomposed operations can now be placed back into the algorithm, as shown in Figure 11.9. Using standard restructuring techniques with dependency analysis, it is possible to reorganize the operations to develop a blocked form of the algorithm. The code for the blocked algorithm is presented in Figure 11.10.

11.6 Experimental Results

To test the usefulness of automatically decomposing operations, the algorithms from the previous section were encoded by hand in Fortran. Tests were then performed using block tridiagonal matrices of varying size, where the density within the blocks is 1. An example of such a matrix is shown in Figure 11.11.

For $i = 1 : n-1$
  $A(i+1:n, i) = A(i+1:n, i)/A(i,i)$
  $A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i) * A(i, i+1:n)$
End For

Figure 11.7: Original code for Gaussian elimination.

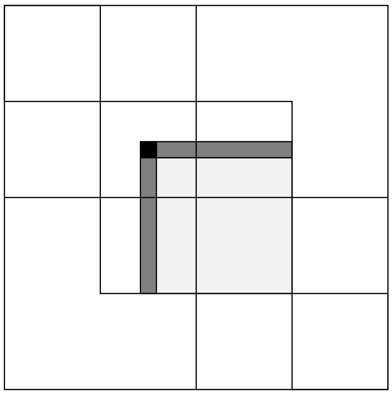

Figure 11.8: The values modified during one iteration of Gaussian elimination on a block tridiagonal matrix.

For $i = 1 : n-1$
  $k = (i \text{ modulo } m) + 1$
  $A_{k,k}(i+1:iu_k, i) = A_{k,k}(i+1:iu_k, i)/A_{k,k}(i,i)$
  If $il_{k+1} < n$ then
    $A_{k+1,k}(il_{k+1}:iu_{k+1}, i) = A_{k+1,k}(il_{k+1}:iu_{k+1}, i)/A_{k,k}(i,i)$
  End If
  $A_{k,k}(i+1:iu_k, i+1:ju_k) = A_{k,k}(i+1:iu_k, i+1:ju_k) -$
    $A_{k,k}(i+1:iu_k, i) * A_{k,k}(i, i+1:ju_k)$
  If $il_{k+1} < n$ then
    $A_{k,k+1}(i+1:iu_k, jl_{k+1}:ju_{k+1}) = A_{k,k+1}(i+1:iu_k, jl_{k+1}:ju_{k+1}) -$
      $A_{k,k}(i+1:iu_k, i) * A_{k,k+1}(i, jl_{k+1}:ju_{k+1})$
    $A_{k+1,k}(il_{k+1}:iu_{k+1}, i+1:ju_k) = A_{k,k+1}(il_{k+1}:iu_{k+1}, i+1:ju_k) -$
      $A_{k+1,k}(il_{k+1}:iu_{k+1}, i) * A_{k,k}(i, i+1:ju_k)$
    $A_{k+1,k+1}(il_{k+1}:iu_{k+1}, jl_{k+1}:ju_{k+1}) = A_{k,k+1}(il_{k+1}:iu_{k+1}, jl_{k+1}:ju_{k+1}) -$
      $A_{k+1,k}(il_{k+1}:iu_{k+1}, i) * A_{k,k+1}(i, jl_{k+1}:ju_{k+1})$
  End If
End For

Figure 11.9: Modified code for Gaussian elimination using a block-tridiagonal decomposition with the block size $= m$.

For $i = 1 : n$ step m
    $k = (i \text{ modulo } m) + 1$
    For $j = il_k : iu_k - 1$
        $A_{k,k}(j+1 : iu_k, j) = A_{k,k}(j+1 : iu_k, j)/A_{k,k}(j,j)$
        $A_{k,k}(j+1 : iu_k, j+1 : ju_k) = A_{k,k}(j+1 : iu_k, j+1 : ju_k) -$
            $A_{k,k}(j+1 : iu_k, j) * A_{k,k}(j, j+1 : ju_k)$
    End For
    If $il_{k+1} < n$ then
        For $j = il_k : iu_k - 1$
            $A_{k,k+1}(j+1 : iu_k, jl_{k+1} : ju_{k+1}) = A_{k,k+1}(j+1 : iu_k, jl_{k+1} : ju_{k+1}) -$
                $A_{k,k}(j+1 : iu_k, j) * A_{k,k+1}(j, jl_{k+1} : ju_{k+1})$
        End For
        For $j = jl_k : ju_k - 1$
            $A_{k+1,k}(il_{k+1} : iu_{k+1}, j) = A_{k+1,k}(il_{k+1} : iu_{k+1}, j)/A_{k,k}(i,i)$
            $A_{k+1,k}(il_{k+1} : iu_{k+1}, j+1 : ju_k) = A_{k,k+1}(il_{k+1} : iu_{k+1}, j+1 : ju_k) -$
                $A_{k+1,k}(il_{k+1} : iu_{k+1}, j) * A_{k,k}(j, j+1 : ju_k)$
        End For
        $A_{k+1,k+1}(il_{k+1} : iu_{k+1}, jl_{k+1} : ju_{k+1}) = A_{k,k+1}(il_{k+1} : iu_{k+1}, jl_{k+1} : ju_{k+1}) -$
            $A_{k+1,k}(il_{k+1} : iu_{k+1}, jl_k : ju_k) * A_{k,k+1}(il_k : iu_k, jl_{k+1} : ju_{k+1})$
    End If
End For

Figure 11.10: Blocked code for Gaussian elimination using a block-tridiagonal decomposition with the block size $= m$.

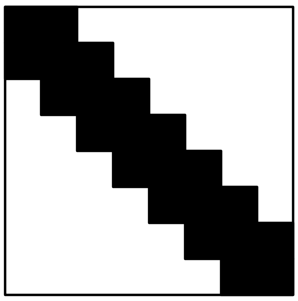

Figure 11.11: The block tridiagonal form

Three versions of the Conjugate Gradient algorithm were developed:

Dense – This version of the code is the original algorithm using dense matrices.

Sparse – This version of the code implements the same algorithm as the first version, but uses sparse data structures and sparse primitives to perform the operations.

BTF – This version of the code exploits the structure of the block tridiagonal matrix with decomposed operations.

Each version of the algorithm was compiled using the Sun Fortran 77 compiler, `f77`, with the optimization flags `-fast -O3` and run five times on a SPARCstation 10. The best CPU execution time for each algorithm is presented in Table 11.1. These results show that as the size of the matrix increases, the decomposed operations in the BTF version provide a significant improvement over the dense version. Even with the additional operations required by the BTF version, it still performs within a factor of two of the general sparse version.

Method	block50 (nnz=1300)	block100 (nnz=2800)	block500 (nnz=14800)	block1000 (nnz=29800)
Dense	0.03	0.20	22.21	175.83
Sparse	0.02	0.08	2.08	8.64
BTF	0.03	0.13	4.26	13.55

Table 11.1: Performance of the Conjugate Gradient algorithm.

The second set of experiments were performed with four versions of Gaussian Elimination:

Dense – This code is the original version of the algorithm using dense matrices.

Sparse – This version of the code implements the original algorithm using compressed sparse row data structures.

BTF – This version of the code exploits the structure of the block tridiagonal matrix with decomposed operations.

ma28 – This code is standard sparse solver based on the compressed sparse row data structure [DR79]. This code is not optimized for the block tridiagonal matrix since it will perform tests to find the pivot element and check for fill elements.

Each of these codes was compiled using the Sun Fortran 77 compiler, `f77`, with the optimization flag `-O3` and run five times on a SPARCstation 10. The best CPU execution time for each algorithm is presented in Table 11.1. The BTF version yields not only a substantial gain over the dense implementation, but also an improvement over the two sparse row-wise implementations.

Method	block50 (nnz=1300)	block100 (nnz=2800)	block500 (nnz=14800)	block1000 (nnz=29800)
Dense	0.01	0.04	0.55	2.53
Sparse	0.01	0.02	0.09	0.18
BTF	0.01	0.03	0.04	0.07
ma28	0.02	0.06	0.35	0.68

Table 11.2: Performance of Gaussian elimination.

Chapter 12

FUTURE WORK

The purpose of this research has been to develop techniques for the interactive development of numerical programs. This thesis has documented the development of the tools to implement such techniques and the infrastructure necessary to support such tools. Additional work on this research may be considered in two main areas, the increased utilization of algebraic information and improvements in the generated code.

12.1 Increased Utilization of Algebraic Information

One of the primary goals for the development environment was to provide the developer with tools to exploit the algebraic information known about the operations. By utilizing an array language with a rich set of built-in functions, a large amount of algebraic information can be deduced from the code using the semantics of the language. This information can be used both to improve the application of traditional compiler restructuring techniques and to develop new techniques for the recognition and utilization of algebraic constructs which are not directly represented in the input language.

12.1.1 Improvements to Traditional Techniques

Many traditional compiler restructuring techniques were developed for use with loop-based Fortran codes. These techniques are typically applied to operations on array elements using the algebraic rules for scalar values, but do not always recognize the overall vector or matrix operation being performed. As a result, dependency analysis techniques based solely on the scalar accesses fail to recognize when transformations to the matrix operations would allow the dependency conditions to be met.

One particular algebraic operation that confounds the techniques is the application of a permutation:

$$A = P * A.$$

If there are two updates being performed to the matrix A, L_1 and L_2,

$$\begin{aligned} A &= L_1 + A \\ A &= L_2 + A, \end{aligned}$$

standard techniques would recognize that the updates could be combined:

$$A = (L_2 + L_1) + A.$$

However, if there is a permutation between the two updates,

$$\begin{aligned} A &= L_1 + A \\ A &= P * A \\ A &= L_2 + A, \end{aligned}$$

the techniques do not typically recognize that the operations can still be combined as

$$A = (L_2 + (P * L_1)) + (P * A).$$

Therefore, by recognizing the matrix operations being performed, especially for operations such as permutations, the application of traditional blocking techniques can be improved.

This work has already been started within the transformation system, focusing primarily on the dependency analysis for identifying the conflicts that need to be resolved. To resolve the conflicts, the code will need to identify operations such as permutations and to understand how to propagate the permutation through the other operands to allow the operations to be combined. In addition, it is necessary to understand the semantics of the MATLAB operations and its built-in functions in order to decompose the operations in order to resolve conflicts and combine operations for the blocking.

12.1.2 Support for Additional Algebraic Operators

Though the array language from MATLAB provides a large number of array operations and built-in functions, there are still a number of algebraic operations that must be represented using multiple statements. As a result, it is difficult to utilize the properties of these operations within the transformation system. If, on the other hand, the sequence of statements could be replaced by a construct that represents the algebraic operation, it would be much simpler to incorporate the properties into the transformation system.

In selecting the method for implementing this capability, attention must be paid to how much control the developer will want. If there is only a small, known set of operations that need to be added, these could be hard-coded into the transformation system. However, if there is a large, unknown set of operations, it would be better to use an implementation which allows the developer to describe the operations to be recognized, along with a description of the properties of the operations.

The hard-coded implementation would probably both be faster to implement and would execute faster, but would limit the flexibility of the developer. The use of operator descriptions, on the other hand, would provide the developer with the flexibility to add new operators to the system, but would require a more complex implementation and the description may be more limited in scope. Since a primary goal of this work has been to develop tools that provide control to the developer, it was felt that the selection of the implementation method for the algebraic operations should also be based on this goal. Therefore, the preferred implementation would be the one which utilizes descriptions of the operators.

The description of an algebraic operator could be similar to that used for the structure definitions, as both must meet similar needs:

- Recognition of the item.
- Description of the item properties.
- Implementation details for the representation of the item.

However, in instances where the structure definitions modified the state of the symbol using the ASSERT constructs, the operator definitions would need to modify the code, using REPLACE constructs.

To better understand what is needed to describe an algebraic operator, consider the matrix product operator.

$$B = \prod_{i=1}^{n} A_i$$

Given that MATLAB does not support arrays with more than two dimensions, it is not possible to have a data structure that explicitly contains the A_i's. However, it is possible to have a loop that generates the A_i's based on a column or row of another matrix:

$$A_i = I + A(:,i) * I(i,:)$$

where I is the identity matrix. Using this definition to generate the A_i's, the product operation could be represented in MATLAB as follows:

```
B = I
for i = 1:n
     Ai = I + A(:,i)*I(i,:);
     B = Ai * B;
end
```

Now consider the information which is needed represent the product operation:

- the loop index,
- the loop bounds, and
- the expression used for calculating the matrix during each loop iteration.

This operation could then be represented with a function call as follows:

```
B = PROD( 1, n, i, I, I + A(:,i)*I(i,:)  );
```

where the first two arguments represent the loop bounds, the third argument is the loop index, the fourth argument is the initial value, and the fifth argument is the expression for generating the matrix. Note that the arguments of the function must store the expressions, not the values of the expressions, in order to allow the PROD function to be transformed back into a loop at a later date.

A slightly modified version of the REPLACE construct could be used to identify the product operator, as shown in Figure 12.1. This example contains two new functionalities, the ability to relate a symbol to an expression and the ability to declare a function symbol as an operator type. Likewise, a transformation can be written for converting the product operator PROD back to a loop iteration.

```
REPLACE
    INTEGER n, indx, lower, upper
    REAL Ai(n,n), B(n,n), A0(n,n)
    EXPRESSION <ITERATE>
    B = A0;
    for indx = lower:upper
        Ai = <ITERATE>;
        B = Ai * B;
    end
WITH
    OPERATOR PROD
    B = PROD( indx, lower, upper, A0, <ITERATE> );
END
```

Figure 12.1: Recognition of the product operation.

Properties of the product operator can now be described using transformation patterns written with the PROD operator. For instance, the decomposition of the operator into two sections could be described as:

```
B0 = PROD( indx, lower, middle, A0, <ITERATE> );
B = PROD( indx, middle+1, upper, B0, <ITERATE> );
```

With these additions, the FALCON environment should be able to support the definition and use of user-defined algebraic operators.

12.2 Improvements in the Generated Code

The other main focus for future work is in the area of code generation. This includes both improving certain constructs that are currently being generated, such as dynamic memory allocations, and adding new code generation capabilities to the system, such as the ability to target multiple C++ matrix classes.

12.2.1 Replacement of Dynamic Memory Allocations

The current compiler uses dynamic memory allocation for all variables of unknown size. These allocations are inserted immediately prior to the use of the variable and then attempts are made to move the allocations outside of loop constructs, using an algorithm developed by DeRose. Also, no attempt is made to free memory once it is no longer used. The current approach was chosen since it guarantees correct code and follows the overall approach used by MATLAB.

However, as Fortran 77 does not have dynamic memory allocation, other approaches for memory allocation have traditionally been used for numerical programs. These approaches have included the use of statically allocated work arrays that are decomposed to fit the current problem size and the reshaping of arrays as they are passed into subroutines. In addition, once the data in the work space is no longer needed, the space is reused, possibly for a totally different purpose.

Within the generated code, improvements can be made both to reduce the overall amount of memory used and to reduce the number of dynamic allocations required. With the inclusion of live variable analysis [ASU86] within the system, several methods can be used to reduce the amount of memory used during the execution. First, when it is known that a temporary variable is no longer needed, this variable can be reused rather than allocating a new temporary variable. To perform this transformation, the inference system must prove that the two temporary variables are of the same size, shape, and rank. If this can be proven, then the symbol for the second variable can be replaced by the symbol for the first variable. Initialization of the data may also need to be performed before the space can be reused.

The second use for live variable analysis is to free dynamically allocated space once it is no longer needed. This can be used for both temporary variables and for variables that are present in the original program. After the space is freed, it can be reused for future allocations, thereby reducing the total amount of memory needed to execute the program. These two methods which utilize live variable analysis can be used together or separately.

Furthermore, the methods used for allocating space within Fortran 77 programs could also be used within the generated code to reduce or eliminate dynamic allocations. For instance, the developer could be queried to determine either the exact size of the variables that will be used or to incorporate a maximum size for each of the variables. This would allow the data space to be allocated statically and would remove the need for the dynamic allocations during the computation. The current system, however, assumes that the variables are always allocated to the correct size and additional changes to the system would be required to correctly utilize data space that has been allocated larger than necessary.

Another possibility when using the separate compilation of functions is to pass a work space into the function. This work space can then be allocated to the temporary variables as needed. This approach could be incorporated with the methods for using live variable analysis, described previously, to determine when the space can be reused. A further improvement would be to have the system calculate the amount of work space needed, based on the problem size.

A final improvement in the use of memory is to eliminate unnecessary data copies for M-file parameters. To support the side-effect free nature of M-files, the code generated by the current system copies the input parameters to temporary variables for use within the function code and uses temporary variables that are copied to the output variables for the results. Additional analysis could be used to determine whether or not the input variables would be modified and only create the copy when necessary. Furthermore, under most circumstances, the temporary variable used for the output data could be replaced by the output parameter.

12.2.2 Including Additional C++ Matrix Classes

One of the goals of the FALCON environment is to provide the developer with as much control as possible over the generated code. The ability to target the code to multiple numerical libraries is one example of this goal. With regards to the C++ code generation, however, this goal is not met as the developer is limited to the matrix class that was developed with the FALCON environment.

The support necessary for multiple matrix classes is very similar to that required to support the structure representations. In both cases it is necessary to describe the data representation, how the data type is propagated, and the functions which are called to perform operations on the data type. Therefore, it should be possible to make modifications to the transformation language which are similar to those required for the structure representation in order to support the description of a C++ matrix class.

The proposed format for the class description is shown in Figure 12.2. This format is similar to that used for the structures, with the declaration of the fields being replaced by the constructors for the class. The file containing the class description would also contain the transformation patterns for converting the MATLAB operations into the necessary class operations.

```
CLASS( class name )
    DECLARATION( dummy variable )
        constructor
        ...
    ASSERTIONS
        ...
END
```

Figure 12.2: Proposed format for matrix class description.

In some cases, however, the Interactive Restructuring System may fail to transform the operation. This may be due to the developer not providing the necessary transformation for the operation or it may be that the operation is not supported by the matrix class. In either case, it will be the responsibility of the system to notify the developer; this allows the developer to make the necessary modifications by hand.

Chapter 13

CONCLUSIONS

This thesis has examined the development of numerical algorithms and studied the methods used for converting the algorithms into code. Techniques have been developed for the interactive development of numerical algorithms within the FALCON environment; this allows the developer to improve the analysis, restructure the algorithm, and control the generation of the target code.

Instead of hard-coding the algorithm and code transformations within the system, a language for describing the transformations was developed. The use of a transformation language allows the developer to both modify the transformations that have been developed and to incorporate additional transformation into the system. The language can be used to describe both traditional compiler transformations and to describe transformations based on the algebraic properties of the matrix operations. Furthermore, the developer can use the language to target library subroutines to replace selected MATLAB constructs.

Additionally, the transformation language allows the identification and utilization of specialized matrix structures using a description provided by the developer. This structure information can both be propagated through the algorithm and used to replace the general matrix operations with specialized subroutines. In addition to the standard data representation, the structure description allows the developer to provide a specialized data representation; this representation can then be utilized to generate sparse matrix code. Both the recognition of the matrix structures and the utilization of specialized data structures were shown to provide significant performance improvements in tests performed with iterative linear system solvers.

Two additional code generators have been added to the FALCON environment. A MATLAB code generator has been added, thereby allowing the developer to interact with the environment using the MATLAB language. Also, a C++ code generator was added thereby allowing the generation of code which utilizes a matrix class developed for the system. Tests of the C++ code generated found that is was much faster than both the original MATLAB code and the C++ code produced by the MATCOM compiler. The C++ code was faster than the C code generated by the MCC compiler for nine of the twelve test programs and was close to the Fortran 90 code generated by the FALCON environment for six of the twelve test programs.

Additional techniques and tools have been added to the FALCON environment which allow the developer to improve the generated code. These techniques include the ability to incorporate additional information about the variable properties and the ability to eliminate dynamic tests within loops for certain arithmetic operations. Further, the ability to compile subroutines separately from the main program was added.

A strategy was described for generating parallel sparse codes by decomposing a sparse matrix structure into component matrices. The calculus to support the decompositions was introduced and algorithms for performing the transformations were presented. Also described was further research which could be done to increase the utilization of algebraic information and to improve the generated code.

The combination of these techniques provide the developer with the tools for the interactive development of the numerical algorithms which utilize both traditional compiler transformations and incorporate algebraic information. Furthermore, these techniques provide the basis for developing capabilities in the areas of sparse code generation, the selection of sparse data structures, and the generation of code for distributed memory machines.

BIBLIOGRAPHY

[ABD$^+$90] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing '90*, pages 2–11, November 1990.

[AGZ92] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A High Performance Algorithm Using Pre-Processing for the Sparse Matrix-Vector Multiplication. In *Proceedings of Supercomputing '92*, pages 32–41, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer, Boston, MA, 1988.

[BBC$^+$93] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM Publications, 1993.

[BBG$^+$] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. To appear in: *Proceedings of the Supercomputing '93 Conference*, Portland, Oregon, November 15–19, 1993.

[BBG$^+$93] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, and S. Srinivas. Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools. Technical report, Indiana University, November 1993.

[BE94] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. In *Proceedings of Supercomputing '94*, pages 528–537, November 1994.

[Bir93] C. R. Birchenhall. *A Draft Guide to MatClass: A matrix class for C++*, 1993.

[BLQ92] Dinshaw Balsara, Max Lemke, and Daniel Quinlan. AMR++, A C++ Object-Oriented Class Library for Parallel Adaptive Mesh Refinement Fluid Dynamics Applications. *Adaptive, Multilevel, and Hierarchical Computational Strategies*, 157:413–433, 1992.

[Bud88] Timothy Budd. *An APL Compiler*. Springer-Verlag, 1988.

[BW94] Aart J. C. Bik and Harry A. G. Wijshoff. Nonzero Structure Analysis. In *Proceedings of the International Conference on Supercomputing*, pages 226–235, 1994.

[CC93] James R. Cordy and Ian H. Carmichael. The TXL Programming Language Syntax and Informal Semantics Version 7. Technical Report 93-355, Department of Computing and Information Science, Queen's University, Kingston, Canada, June 1993.

[CDW94] Jaeyoung Choi, Jack J. Dongarra, and David W. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subprograms. In *Proceedings of SHPCC '94: Scalable High-Performance Computing Conference*, pages 534–541, May 1994.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, October 1991.

[Chi86] Wai-Mee Ching. Program analysis and code generation in an apl/370 compiler. *IBM Journal of Research and Development*, 30(6):594–602, November 1986.

[CK92] Steve Carr and Ken Kennedy. Compiler Blockability of Numerical Algorithms. In *Proceedings, Supercomputing '92*, pages 114–124, November 1992.

[Dav94] Robert B. Davies. Writing a matrix package in C++. In *OON-SKI'94 Proceedings of the Second Annual Object-Oriented Numerics Conference*, pages 207–213, April 1994.

[DCHD90] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[DCHH88] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[Dem89] James Demmel. LAPACK: A Portable Linear Algebra Library for Supercomputers. In *1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design (CACSD)*, pages 1–7, December 1989.

[DER86] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Monographs on Numerical Analysis. Oxford University Press, New York, 1986.

[DeR96] Luiz Antonio DeRose. *Compiler Techniques for MATLAB Programs*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[DGG+95] L. DeRose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. A MATLAB Compiler and Restructurer for the Development of Scientific Libraries and Applications. In *Languages and Compilers for Parallel Computing*, pages 269–288. Springer-Verlag, 1995. (8th International Workshop on Languages and Compilers for Parallel Computing, August 1995, Columbus, Ohio).

[DJK] Peter Drakenberg, Peter Jacobson, and Bo Kagstrom. A CONLAB Compiler for a Distributed Memory Multicomputer. To appear in: *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, Virginia, March 22–24, 1993.

[DMBS79] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK: Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[DPW93a] Jack J. Dongarra, Roldan Pozo, and David W. Walker. *LAPACK++: A Design Overview of Object-Oriented Extensions for High-Performance Linear Algebra*. University of Tennessee, 1993.

[DPW93b] Jack J. Dongarra, Roldan Pozo, and David W. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 257–264, April 1993.

[DR79] I. S. Duff and J. K. Reid. Some Design Features of a Sparse Matrix Code. *ACM Transactions on Mathematical Software*, 5(1):18–35, March 1979.

[Eat93] John W. Eaton. *Octave*, 1993.

[EH93] H. Carter Edwards and Linda J. Hayes. Visual programming of iterative methods. In *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 163–170, April 1993.

[Gar94] Alejandro L. Garcia. *Numerical Methods for Physics*. Prentice Hall, 1994.

[GL79] Alan George and Joseph W. H. Liu. The Design of a User Interface for a Sparse Matrix Package. *ACM Transactions on Mathematical Software*, 5(2):139–162, June 1979.

[GL81] Alan George and Joseph W-H Liu. *Computer Solutions of Large Sparse Positive Definite Systems.* Prentice-Hall Series in Computational Mathematics. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[GPS90] K. A. Gallivan, P. J. Plemmons, and A. H. Sameh. Parallel Algorithms for Dense Linear Algebra Computations. *SIAM Review*, 32(1):54–135, March 1990.

[GS91] John R. Gilbert and Robert Schreiber. Optimal Expression Evaluation for Data Parallel Architectures. *Journal of Parallel and Distributed Computing*, 13(1):58–64, September 1991.

[GS93] William Gropp and Barry Smith. Simplified Linear Equation Solvers Users Manual. Technical Report ANL-93/8-REV 1, Argonne National Laboratory, June 1993.

[Hag94] Mohammad Reza Haghighat. *Symbolic Analysis for Parallelizing Compilers.* PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[Hig93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.

[Hoe91] Jay Hoeflinger. Cedar Fortran Programmer's Handbook. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, October 1991. CSRD Report No. 1157.

[HRC+90] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, Ko Yang Wang, and S. Weerawarana. //ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD Machines. In *Proceedings 1990 International Conference on Supercomputing*, pages 96–107, 1990.

[JBV93] Wouter Joosen, Stijn Bijnens, and Pierre Verbaeten. Object parallelism in XENOOPS. In *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 42–54, April 1993.

[JKR92] Peter Jacobson, Bo Kagstrom, and Mikael Rannar. Algorithm Development for Distributed Memory Multicomputers Using CONLAB. *Scientific Programming*, 1:185–203, 1992.

[JR93] Eric F. Johnson and Kevin Reichard. *Power Programming Motif.* MIS: Press, 2nd edition, 1993.

[Kan93] Elaine Kant. Synthesis of Mathematical Modeling Software. *IEEE Software*, 10(3):30–41, MAY 1993.

[Kuc87] Kuck and Associates, Inc., Savoy, IL 61874. *KAP User's Guide*, 4th edition, 1987.

[LFG93] Sharon Lamb, Geoffrey Furnish, and Mark Gray. A distributed environment for plasma simulation. In *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 155–162, April 1993.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[Los94] Los Alamos National Laboratory. *POOMA '94 Parallel Object-Oriented Methods and Applications*, Santa Fe, New Mexico, December 1994. Proceedings are located at http://www.acl.lanl.gov/Pooma94/.

[LQ93] Max Lemke and Daniel Quinlan. P++, a parallel C++ array class library for architecture-independent development of numerical software. In *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 268–269, April 1993. Extended Abstract Only.

[Mat92] John H. Mathews. *Numerical Methods for Mathematics, Science and Engineering.* Prentice Hall, 2nd edition, 1992.

[MK73] Yoichi Muraoka and David J. Kuck. On the Time Required for a Sequence of Matrix Products. *Communications of the ACM*, 16(1):22–26, January 1973.

[Ott91] Steve W. Otto. MetaMP: A Higher Level Abstraction for Message-Passing Programming. Technical Report CS/E 91-003, Oregon Graduate Institute, March 1991.

[Ott93] Steve W. Otto. Parallel array classes and lightweight sharing mechanisms. In *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 285–302, April 1993.

[Par94] Terence J. Parr. An Overview of SORCERER: A Simple Tree-Parse Generator. Technical report, Army High Performance Computing Research Center, University of Minnesota, 1994.

[PE95] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. In *Proceedings of* 9^{th} *ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[PEH+93] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A New-Generation Parallelizing Compiler for MPP's. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, June 1993. CSRD Report No. 1306.

[PGH+89] Constantine Polychronopoulos, Milind Girkar, Mohammad Reza Haghighat, Chia-Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: A New Generation Parallelizing Compiler. In *Proceedings of 1989 Int'l. Conference on Parallel Processing, St. Charles, IL*, volume II, pages 39–48, August 1989.

[Rog93] Rogue Wave Software in cooperation with SIAM. *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference*, Sunriver, Oregon, April 1993.

[Saa90] Youcef Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Technical Report 1029, Center for Supercomputing Research and Development, University of Illinois, August 1990.

[SBD+76] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines - EISPACK Guide Second Edition*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1976.

[Sch75] J. T. Schwartz. Automatic Data Structure Choice in a Language of a Very High Level. *Communications of the ACM*, 18:722–727, 1975.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 2nd edition, 1991.

[Swa85] Paul N. Swarztrauber. *FFTPACK*, April 1985.

[TD94] M. M. Tiller and J. A. Dantzig. FEMLIB: A framework for simulation and optimization. In *OON-SKI'94 Proceedings of the Second Annual Object-Oriented Numerics Conference*, pages 366–378, April 1994.

[The92] The Math Works, Inc. *MATLAB, High Performance Numeric Computation and Visualization Software. Reference Guide*, 1992.

[The95] The MathWorks, Inc. *MATLAB Compiler*, 1995.

[The96] The MathWorks, Inc. Develop C++ applications with new C++ Math Library and Compiler 1.1. *MATLAB News & Notes*, page 10, Summer 1996.

[Thi91] Thinking Machines Corporation, Cambridge, Massachusetts. *CM5 Technical Summary*, October 1991.

[Tis93] E. Robert Tisdale. *C++ Matrix Class*. UCLA, 1993.

[TP93] Peng Tu and David Padua. Automatic Array Privatization. In U. Banerjee, D. Gelemter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 500–521. Springer-Verlag, August 1993. 6th International Workshop, Portland, Oregon.

[Ver93] Al Vermeulen. Eigenvalues in Lapack.h++: Object oriented linear algebra software. In *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 314–317, April 1993.

[VHF92] Lars Viklund, Johan Herber, and Peter Fritzson. The Implementation of ObjectMath - a High-Level Programming Environment for Scientific Computing. In *Compiler Construction - 4th International Conference, CC'92*, pages 312–318. Springer-Verlag, 1992.

[Wal93] David W. Walker. The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers. Technical Report TM-12512, Oak Ridge National Laboratory, October 1993.

[Wea94] Stephen Andrew Weatherford. High-Level Pattern-Matching Extensions to C++ for Fortran Program Manipulation in Polaris. Master's thesis, University of Illinois at Urbana-Champaign, 1994.

[Wol91] Stephen Wolfram. *Mathematica–A System for Doing Mathematics by Computer.* Addison-Wesley Publishing Company, second edition, 1991.

[Yan93] Lan Yang. Object-oriented techniques in the design and construction of paralle numerical application software. In *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 303–313, April 1993.

[Yar96] Keren Yaron. *MATCOM, A MATLAB to C++ Translator*, March 1996.

[ZWS81] Zahari Zlatev, Jerzy Wasniewski, and Kjeld Schaumburg. *Y12M: Solution of Large and Sparse Systems of Linear Algebraic Equations*, volume 121 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

VITA

Bret Marsolf was born on the 19th of April, 1965 in Wichita, Kansas. He graduated from Rosedale High School in Rosedale, Indiana in 1983. He spent the next four years at the Rose-Hulman Institute of Technology in Terre Haute, Indiana. There, in 1987, he graduated *summa cum laude* with a Bachelor of Science in Computer Science degree. For the next year he worked as a Software Design Engineer at Applied Computing Devices in Terre Haute, Indiana.

Since 1988 he has been at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign working towards a Doctor of Philosophy in Computer Science. While there he has worked as a graduate research assistant and in 1991 obtained a Master of Science degree in Computer Science.