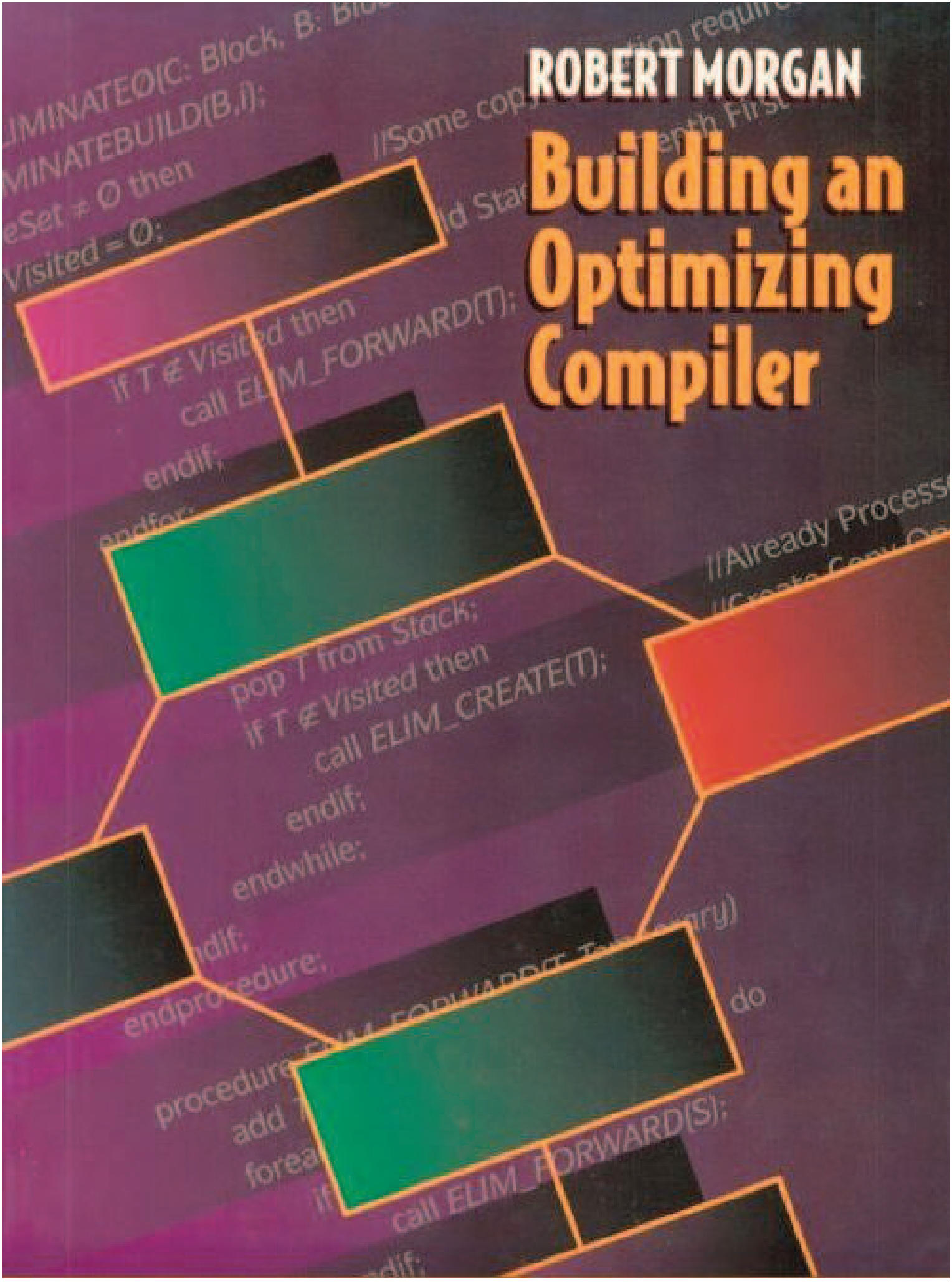

ROBERT MORGAN
Building an Optimizing Compiler
pop T from Stack;
if T ∉ Visited then
call ELIM_CREATE(T);
endif;
endwhile;

Building an Optimizing Compiler
by Bob Morgan
Digital Press
ISBN: 155558179x **Pub Date:** 12/01/97

Building an Optimizing Compiler
by Bob Morgan
Digital Press
ISBN: 155558179x **Pub Date:** 12/01/97

Table of Contents

Preface

Building compilers has been a challenging activity since the advent of digital computers in the late 1940s and early 1950s. At that time, implementing the concept of automatic translation from a form familiar to mathematicians into computer instructions was a difficult task. One needed to figure out how to translate arithmetic expressions into instructions, how to store data in memory, and how to choose instructions to build procedures and functions. During the late 1950s and 1960s these processes were automated to the extent that simple compilers could be written by most computer science professionals. In fact, the concept of "small languages" with corresponding translators is fundamental in the UNIX community.

From the beginning, there was a need for translators that generated efficient code: The translator must use the computer productively. Originally this constraint was due to computers' small memories and slow speed of execution. During each generation of hardware, new architectural ideas have been added. At each stage the compilers have also needed to be improved to use these new machines more effectively. Curiously, pundits keep predicting that less efficient and less expensive translators will do the job. They argue that as machines keep getting faster and memory keeps expanding, one no longer needs an optimizing compiler. Unfortunately, people who buy bigger and faster machines want to use the proportionate increase in size and speed to handle bigger or more complex problems, so we still have the need for optimizing compilers. In fact, we have an increased need for these compilers because the performance of the newer architectures is sensitive to the quality of the generated code. Small changes in the order and choice of the instructions can have much larger effects on machine performance than similar choices made with the complex instruction set computing (CISC) machines of the 1970s and 1980s.

The interplay between computer architecture and compiler performance has been legitimized with the development of reduced instruction set computing (RISC) architectures. Compilers and computer architecture have a mutually dependent relationship that shares the effort to build fast applications. To this end, hardware has been simplified by exposing some of the details of hardware operation, such as simple load-store instruction sets and instruction scheduling. The compiler is required to deal with these newly exposed details and provide faster execution than possible on CISC processors.

This book describes one design for the optimization and code-generation phases of such a compiler. Many compiler books are available for describing the analysis of programming languages. They emphasize the processes of lexical analysis, parsing, and semantic analysis. Several books are also available for describing compilation processes for vector and parallel processors. This book describes the compilation of efficient programs for a single superscalar RISC processor, including the ordering and structure of algorithms and efficient data structures.

The book is presented as a high-level design document. There are two reasons for this. Initially, I attempted to write a book that presented all possible alternatives so that the reader could make his or her own choices of methods to use. This was too bulky, as the projected size of the volume was several thousand pages—much too large for practical purposes. There are a large number of different algorithms and structures in an optimizing compiler. The choices are interconnected, so an encyclopedic approach to optimizing compilers would not address some of the most difficult problems.

Second, I want to encourage this form of design for large software processes. The government uses a three-level documentation system for describing software projects: The A-level documents are overview documents that describe a project as a whole and list its individual pieces. B-level documents describe the operation of each component in sufficient detail that the reader can understand what each component does and how it does it, whereas the C-level documents are low-level descriptions of each detail.

As a developer I found this structure burdensome because it degenerated into a bureaucratic device involving large amounts of paper and little content. However, the basic idea is sound. This book will describe the optimization and code-generation components of a compiler in sufficient detail that the reader can implement these components if he or she sees fit. Since I will be describing one method for each of the components, the interaction between components can be examined in detail so that all of the design and implementation issues are clear.

Each chapter will include a section describing other possible implementation techniques. This section will include bibliographic information so that the interested reader can find these other techniques.

Philosophy for Choosing Compiler Techniques

Before starting the book, I want to describe my design philosophy. When I first started writing compilers (about 1964), I noticed that much research and development work had been described in the literature. Although each of these projects is based on differing assumptions and needs, the availability of this information makes it easier for those who follow to use previous ideas without reinventing them. I therefore design by observing the literature and other implementations and choosing techniques that meet my needs. What I contribute is the choice of technique, the engineering of the technique to fit with other components, and small improvements that I have observed.

One engineering rule of thumb must be added. It is easy to decide that one will use the latest techniques that have been published. This policy is dangerous. There are secondary effects from the choice of any

optimization or code-generation technique that are observed only after the technique has been used for some time. Thus I try to avoid techniques that I have not seen implemented at least twice in prototype or production compilers. I will break this rule once or twice when I am sure that the techniques are sound, but no more frequently.

In the course of writing this book, my view of it has evolved. It started out as a recording of already known information. I have designed and built several compilers using this existing technology. As the book progressed, I have learned much about integrating these algorithms. What started out as a concatenation of independent ideas has thus become melded into a more integrated whole. What began as simple description of engineering choices now contains some newer ideas. This is probably the course of any intellectual effort; however, I have found it refreshing and encouraging.

How to Use This Book

This book is designed to be used for three purposes. The first purpose is to describe the structure of an optimizing compiler so that a reader can implement it or a variation (compiler writers always modify a design). The book's structure reflects this purpose. The initial chapters describe the compilation phases and the interactions among them; later chapters describe the algorithms involved in each compilation phase.

This book can also be used as a textbook on compiler optimization techniques. It takes one example and describes each of the compilation processes using this example. Rather than working small homework problems, students work through alternative examples.

Practically, the largest use for this book will be informing the curious. If you are like me, you pick up books because you want to learn something about the subject. I hope that you will enjoy this book and find what you are looking for. Good reading.

Table of Contents

Building an Optimizing Compiler
by Bob Morgan
Digital Press
ISBN: 155558179x **Pub Date:** 12/01/97

Table of Contents

Dedication

I dedicate this book to some of the people who have inspired me. My mother and father, Florence and Charles William Morgan, taught me the concept of work. Jordan Baruch introduced me to the wonders of Computer Research. Louis Pitt, Jr., and Bill Clough have been instrumental in helping me understand life and the spirit. My wife, Leigh Morgan, has taught me that there is more than computers and books—there is also life.

Table of Contents

Chapter 1
Overview

What is an optimizing compiler? Why do we need them? Where do they come from? These questions are discussed in this chapter, along with how to use the book. Before presenting a detailed design in the body of the book, this introductory chapter provides an informal history of optimizing compiler development and gives a running example for motivating the technology in the compiler and to use throughout the rest of the book.

1.1 What Is an Optimizing Compiler?

How does a programmer get the performance he expects from his application? Initially he writes the program in a straightforward fashion so that the correct execution of the program can be tested or proved. The program is then profiled and measured to see where resources such as time and memory are used, and modified to improve the uses of these resources. After all reasonable programmer modifications have been made, further improvements in performance can come only from how well the programming language is translated into instructions for the target machine.

The goal of an optimizing compiler is to efficiently use all of the resources of the target computer. The compiler translates the source program into machine instructions using all of the different computational elements. The ideal translation is one that keeps each of the computational elements active doing useful (and nonredundant) work during each instruction execution cycle.

Of course, this idealized translation is not usually possible. The source program may not have a balanced set of computational needs. It may do more integer than floating point arithmetic or vice versa, or more load and store operations than arithmetic. In such cases the compiler must use the overstressed computational elements as effectively as possible.

The compiler must try to compensate for unbalanced computer systems. Ideally, the speed of the processor is matched to the speed of the memory system, which are both matched to the speed of the input/output (I/O) system. In modern reduced instruction set computer (RISC) systems this is not true: The processors are much faster than the memory systems. To be able to use the power of the processor, the compiler must generate code that decreases the use of the memory system by either keeping values in registers or organizing the code so that needed data stays in the memory cache.

An added problem is fetching instructions. A significant fraction of the memory references are references to instructions. One hopes that the instructions stay in one of the memory caches; however, this is not always the case. When the instructions do not fit in the cache, the compiler should attempt to generate as few instructions as possible. When the instructions do fit in the cache and there are heavy uses of data,

then the compiler is free to add more instructions to decrease the wait for data. Achieving a balance is a difficult catch-22.

In summary, the optimizing compiler attempts to use all of the resources of the processor and memory as effectively as possible in executing the application program. The compiler must transform the program to regain a balanced use of computational elements and memory references. It must choose the instructions well to use as few instructions as possible while obtaining this balance. Of course, all of this is impossible, but the compiler must do as well as it can.

1.2 A Biased History of Optimizing Compilers

Compiler development has a remarkable history, frequently ignored. Significant developments started in the 1950s. Periodically, pundits have decided that all the technology has already been developed. They have always been proven wrong. With the development of new high-speed processors, significant compiler developments are needed today. I list here the compiler development groups that have most inspired and influenced me. There are other groups that have made major contributions to the field, and I do not mean to slight them.

Although there is earlier work on parsing and compilation, the first major compiler was the Fortran compiler (Backus) for the IBM 704/709/7090/7094. This project marked the watershed in compiler development. To be accepted by programmers, it had to generate code similar to that written by machine language programmers, so it was a highly optimizing compiler. It had to compile a full language, although the design of the language was open to the developers. And the technology for the project did not exist; they had to develop it. The team succeeded beautifully, and their creation was one of the best compilers for about ten years. This project developed the idea of compiler passes or phases.

Later, again at IBM, a team developed the Fortran/Level H compilers for the IBM 360/370 series of computers. Again, these were highly optimizing compilers. Their concept of quadruple was similar to the idea of an abstract assembly language used in the design presented in this book. Subsequent improvements to the compilers by Scarborough and Kolsky (1980) kept this type of compiler one of the best for another decade.

During the late 1960s and throughout the 1970s, two research groups continued to develop the ideas that were the basis of these compilers as well as developing new ideas. One group was led by Fran Allen at IBM, the other by Jack Schwartz at New York University (NYU). These groups pioneered the ideas of reaching definitions and bit-vector equations for describing program transformation conditions. Much of their work is in the literature; if you can get a copy of the SETL newsletters (NYU 1973) or the reports associated with the SETL project, you will have a treat.

Other groups were also working on optimization techniques. William Wulf defined a language called Bliss (Wulf et al. 1975). This is a structured programming language for which Wulf and his team at Carnegie Mellon University (CMU) developed optimizing compiler techniques. Some of these

techniques were only applicable to structured programs, whereas others have been generalized to any program structure. This project evolved into the Production-Quality Compiler-Compiler (PQCC) project, developing meta-compiler techniques for constructing optimizing compilers (Leverett et al. 1979). These papers and theses are some of the richest and least used sources of compiler development technology.

Other commercial companies were also working on compiler technology. COMPASS developed compiler techniques based on p-graph technology (Karr 1975). This technology was superior to reaching definitions for compiler optimization because the data structures were easily updated; however, the initial computation of p-graphs was much slower than reaching definitions. P-graphs were transformed by Reif (Reif and Lewis 1978) and subsequent developers at IBM Yorktown Heights (Cytron et al. 1989) into the Static Single Assignment Form of the flow graph, one of the current flow graph structures of choice for compiler development.

Ken Kennedy, one of the students at NYU, established a compiler group at Rice University to continue his work in compiler optimization. Initially, the group specialized in vectorization techniques. Vectorization required good scalar optimization, so the group continued work on scalar optimization also. Some of the most effective work analyzing multiple procedures (interprocedural analysis) has been performed at Rice under the group led by Keith Cooper (1988, 1989). This book uses much of the flow graph structure designed by the Massive Scalar Compiler Project, the group led by Cooper.

With the advent of supercomputers and RISC processors in the later 1970s and early 1980s, new compiler technology had to be developed. In particular, instructions were pipelined so that the values were available when needed. The instructions had to be reordered to start a number of other instructions before the result of the first instruction was available. These techniques were first developed by compiler writers for machines such as the Cray-1. An example of such work is Richard Sites' (1978) paper on reordering Cray-1 assembly language. Later work by the IBM 801 (Auslander and Hopkins 1982) project and Gross (1983) at CMU applied these technques to RISC processors. Other work in this area includes the papers describing the RS6000 compilers (Golumbic 1990 and Warren 1990) and research work performed at the University of Wisconsin on instruction scheduling.

In the 1970s and early 1980s, register allocation was a difficult problem: How should the compiler assign the values being computed to the small set of physical registers to minimize the number of times data need to be moved to and from memory? Chaitin (1981, 1982) reformulated the problem as a graph-coloring problem and developed heuristics for coloring the graphs that worked well for programs with complex flows. The PQCC project at Carnegie Mellon developed a formulation as a type of bin-packing problem, which worked best with straight-line or structure procedures. The techniques developed here are a synthesis of these two techniques using some further work by Laurie Hendron at McGill University.

1.3 What Have We Gained with All of This Technology?

Considering this history, all the technology necessary to build a high-performance compiler for modern RISC processors existed by about 1972, certainly by 1980. What is the value of the more recent research? The technology available at those times would do the job, but at a large cost. More recent research in optimizing compilers has led to more effective and more easily implemented techniques for optimization. Two examples will make this clearer. The Fortran/Level H compiler was one of the most

effective optimizing compilers of the late 1960s and early 1970s. It used an algorithm to optimize loops based on identifying the nesting of loops. In the late 1970s Etienne Morel developed the technique called Elimination of Partial Redundancies that performed a more effective code motion without computing anything about loops (Morel and Renvoise 1979).

Similarly, the concepts of Static Single Assignment Form have made a number of transformation algorithms similar and more intuitive. Constant propagation, developed by Killdall (1973), seemed complex. Later formulations by Wegman and Zadeck (1985) make the technique seem almost intuitive.

The new technology has made it easier to build optimizing compilers. This is vital! These compilers are large programs, prone to all of the problems that large programs have. When we can simplify a part of the compiler, we speed the development and compilation times and decrease the number of bugs (faults, defects) that occur in the compiler. This makes a cheaper and more reliable product.

1.4 Rules of the Compiler Back-End Game

The compiler back end has three primary functions: to generate a program that faithfully represents the meaning of the source program, to allocate the resources of the machine efficiently, and to recast the program in the most efficient form that the compiler can deduce. An underlying rule for each of these functions is that the source program must be faithfully represented.

Unfortunately, there was a time when compiler writers considered it important to get most programs right but not necessarily *all* programs. When the programmer used some legal features in unusual ways, the compiler might implement an incorrect version of the program. This gave optimizing compilers a bad name.

It is now recognized that the code-generation and optimization components of the compiler must exactly represent the meaning of the program as described in the source program and in the language reference manual for the programming language. This does not mean that the program will give exactly the same results when compiled with optimization turned on and off. There are programs that violate the language definition in ways not identifiable by a compiler. The classic example is the use of a variable before it is given a value. These programs may get different results with optimization turned on and turned off.

Fortunately, standards groups are becoming more aware of the needs of compiler writers when describing the language standards. Each major language standard now describes in some way the limits of compiler optimization. Sometimes this is done by leaving certain aspects of the language as “undefined” or “implementation defined.” Such phrases mean that the compiler may do whatever it wishes when it encounters that aspect of the language. However, be cautious—the user community frequently has expectations of what the compiler will do in those cases, and a compiler had better honor those expectations.

What does the compiler do when it encounters a portion of the source program that uses language

facilities in a way that the compiler does not expect? It must make a conservative choice to implement that facility, even at the expense of runtime performance for the program. Even when conservative choices are being made, the compiler may be clever. It might, for example, compile the same section of code in two different ways and generate code to check which version of the code is safe to use.

1.5 Benchmarks and Designing a Compiler

Where does the compiler writer find the set of improvements that must be included in an optimizing compiler? How is one variant of a particular optimization chosen over another? The compiler writer uses information about the application area for the target machine, the languages being compiled, and good sense to choose a particular set of optimizations and their organization.

Any application area has a standard set of programs that are important for that area. Sorting and databases are important for commercial applications. Linear algebra and equation solution are important for numeric applications. Other programs will be important for simulation. The compiler writer will investigate these programs and determine what the compiler must do to translate these programs well. While doing this, the compiler writer and his client will extract sample code from these programs. These samples of code become *benchmarks* that are used to measure the success of the compiler.

The source languages to be compiled are also investigated to determine the language features that must be handled. In Fortran, an optimizing compiler needs to do strength reduction since the programmer has no mechanism for simplifying multiplications. In C, strength reduction is less important (although still useful); however, the compiler needs to compile small subroutines well and determine as much information about pointers as possible.

There are standard optimizations that need to be implemented. Eliminating redundant computations and moving code out of loops will be necessary in an optimizing compiler for an imperative language. This is actually a part of the first criterion, since these optimizations are expected by most application programmers.

The compiler writer must be cautious. It is easy to design a compiler that compiles benchmarks well and does not do as well on general programs. The Whetstone benchmark contained a kernel of code that could be optimized by using a trigonometric identity. The SPEC92 benchmarks have a kernel, EQNTOT, that can be optimized by clever vectorization of integer instructions.

Should the compiler writer add special code for dealing with these anomalous benchmarks? Yes and no. One has to add the special code in a competitive world, since the competition is adding it. However, one must realize that one has not really built a better compiler unless there is a larger class of programs that finds the feature useful. One should always look at a benchmark as a source of general comments about programming. Use the benchmark to find general improvements. In summary, the basis for the design of optimizing compilers is as follows:

1. Investigate the important programs in the application areas of interest. Choose compilation techniques that work well for these programs. Choose kernels as benchmarks.
2. Investigate the source languages to be compiled. Identify their weaknesses from a code quality

point of view. Add optimizations to compensate for these weaknesses.

3. Make sure that the compiler does well on the standard benchmarks, and do so in a way that generalizes to other programs.

1.6 Outline of This Book

Before developing a compiler design, the writer must know the requirements for the compiler. This is as hard to determine as writing the compiler. The best way that I have found for determining the requirements is to take several typical example programs and compile them by hand, pretending that you are the compiler. No cheating! You cannot do a transformation that cannot be done by some compiler using some optimization technique.

This is what we do in Chapter 2 for one particular example program. It is too repetitious to do this for multiple examples. Instead, we will summarize several other requirements placed on the compiler that occur in other examples.

Then we dig into the design. Each chapter describes a subsequent phase of the compiler, giving the theory involved in the phase and describing the phase in a high-level pseudo-code.

We assume that the reader can develop detailed data structures from the high-level descriptions given here. Probably the most necessary requirement for a compiler writer is to be a “data structure junkie.” You have to love complex data structures to enjoy writing compilers.

1.7 Using This Book as a Textbook

This compiler design can be used as a textbook for a second compiler course. The book assumes that the reader is familiar with the construction of compiler front ends and the straightforward code-generation techniques taught in a one-term compiler course. I considered adding sets of exercises to turn the book into a textbook. Instead, another approach is taken that involves the student more directly in the design process.

The example procedure in Figure 1.1 is used throughout the book to motivate the design and demonstrate the details. As such, it will be central to most of the illustrations in the book. Students should use the three examples in Figures 1.2-1.4 as running illustrations of the compilation process. For each chapter, the student should apply the technology developed therein to the example. The text will also address these examples at times so the student can see how his or her work matches the work from the text.

```
SUBROUTINE MAXCOL(A,N,LARGE,VALUE)
   DOUBLE PRECISION VALUE(N), A(N,N)
   INTEGER N, LARGE(N)
   INTEGER I, J
   DO I = 1, N
      LARGE(I) = 1
      VALUE(I) = DABS(A(1,I))
      DO J = 2, N
         IF (DABS(A(J,I).GT.VALUE(I)) THEN
            VALUE(I) = DABS(A(J,I))
            LARGE(I) = J
         ENDIF
      ENDDO
   ENDDO
END
```

Figure 1.1 Running Exercise Throughout Book

```
SUBROUTINE MATMUL(A,B,C,N)
   DOUBLE PRECISION A(N,N), B(N,N), C(N,N)
   INTEGER N, I, J, K
   DO I = 1, N
      DO J = 1, N
         C(I,J) = 0.0
      ENDDO
   ENDDO
   DO I = 1, N
      DO J = 1, N
         DO K = 1, N
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
         ENDDO
      ENDDO
   ENDDO
END
```

Figure 1.2 Matrix Multiply Example

```
INTEGER FUNCTION MONOTONE(A,N)
   DOUBLE PRECISION A(N)
   INTEGER C(N), CMAX
   INTEGER I, J, N
   C(N) = 1
   CMAX = 1
   DO I = N - 1, 1, -1
      C(I) = 1
      DO J = I + 1, N
         IF ((X(I) <= X(J)).AND.(C(I) <= C(J)+1) THEN
            C(I) = C(J)+1
         ENDIF
      ENDDO
      IF (CMAX <= C(I)) THEN
         CMAX = C(I)
      ENDIF
   ENDDO
   MONOTONE = CMAX
END
```

Figure 1.3 Computing the Maximum Monotone Subsequence

```
INTEGER FUNCTION BINARYSEARCH(A,N,L,U,KEY)
   DOUBLE PRECISION A(N), KEY
   INTEGER L, U, N
   INTEGER M
   IF (U < L) THEN
      BINARYSEARCH = 0
   ELSE
      M = (L+U)/2
      IF (A(M) = KEY) THEN
         BINARYSEARCH = M
      ELSIF (A(M) < KEY) THEN
         BINARYSEARCH = BINARYSEARCH(A,N,L,M-1,KEY)
      ELSE
         BINARYSEARCH=BINARYSEARCH(A,N,M+1,U,KEY)
      ENDIF
   ENDIF
END
```

Figure 1.4 Recursive Version of a Binary Search

Figure 1.2 is a version of the classic matrix multiply algorithm. It involves a large amount of floating point computation together with an unbalanced use of the memory system. As written, the inner loop consists of two floating point operations together with three load operations and one store operation. The problem will be to get good performance from the machine when more memory operations are occurring than computations.

Figure 1.3 computes the length of the longest monotone subsequence of the vector $\boldsymbol{A}$. The process uses dynamic programming. The array $C(I)$ keeps track of the longest monotone sequence that starts at position I. It computes the next element by looking at all of the previously computed subsequences that can have $X(I)$ added to the front of the sequence computed so far. This example has few floating point operations. However, it does have a number of load and store operations together with a significant amount of conditional branching.

Figure 1.4 is a binary search algorithm written as a recursive procedure. The student may feel free to translate this into a procedure using pointers on a binary tree. The challenge here is to optimize the use of memory and time associated with procedure calls.

I recommend that the major grade in the course be associated with a project that prototypes a number of the optimization algorithms. The implementation should be viewed as a prototype so that it can be implemented quickly. It need not handle the complex memory management problems existing in real optimizing compilers.

1.8 References

Auslander, M., and M. Hopkins. 1982. An overview of the PL.8 compiler. *Proceeding of the ACN SIGPLAN '82 Conference on Programming Language Design and Implementation*, Boston, MA.

Backus, J. W., et al. 1957. The Fortran automatic coding system. *Proceedings of AFIPS 1957 Western Joint Computing Conference (WJCC)*, 188-198.

Chaitin, G. J. 1982. Register allocation and spilling via graph coloring. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Boston, MA. Published as *SIGPLAN Notices* 17(6): 98-105.

Chaitin, G. J., et al. 1981. Register allocation via coloring. *Computer Languages* 6(1): 47-57.

Cooper, K., and K. Kennedy. 1988. Interprocedural side-effect analysis in linear time. *Proceedings of the SIGPLAN 88 Symposium on Programming Language Design and Implementation*, Altanta, GA. Published as *SIGPLAN Notices* 23(7).

Cytron, R., et al. 1989. An efficient method of computing static single assignment form. *Conference Record of the 16th ACM SIGACT/SIGPLAN Symposium on Programming Languages*, Austin, TX. 25-35.

Gross, T. 1983. Code optimization of pipeline constraints. (Stanford Technical Report CS 83-255.) Stanford University.

Hendron, L. J., G. R. Gao, E. Altman, and C. Mukerji. 1993. A register allocation framework based on hierarchical cyclic interval graphs. (Technical report.) McGill University.

Karr, M. 1975. P-graphs. (Report CA-7501-1511.) Wakefield, MA: Massachusetts Computer Associates.

Kildall, G. A. 1973. A unified approach to global program optimization. *Conference Proceedings of Principles of Programming Languages I*, 194-206.

Leverett, B. W., et al. 1979. An overview of the Production-Quality Compiler-Compiler project. (Technical Report CMU-CS-79-105.) Pittsburgh, PA: Carnegie Mellon University.

Morel, E., and C. Renvoise. 1979. Global optimization by suppression of partial redundancies. *Communications of the ACM* 22(2): 96-103.

New York University Computer Science Department. 1970-1976. *SETL Newsletters*.

Reif, J. H., and H. R. Lewis. 1978. Symbolic program analysis in almost linear time. *Conference Proceedings of Principles of Programming Languages V, Association of Computing Machinery*.

Scarborough, R. G., and H. G. Kolsky. 1980. Improved optimization of Fortran programs. *IBM Journal of Research and Development* 24: 660-676.

Sites, R. 1978. Instruction ordering for the CRAY-1 computer. (Technical Report 78-CS-023.) University of California at San Diego.

Wegman, M. N., and F. K. Zadeck. 1985. Constant propagation with conditional branches. *Conference Proceedings of Principles of Programming Languages XII*, 291-299.

Wulf, W., et al. 1975. *The design of an optimizing compiler*. New York: American Elsevier.

Chapter 2
Compiler Structure

The compiler writer determines the structure of a compiler using information concerning the source languages to be compiled, the required speed of the compiler, the code quality required for the target computer, the user community, and the budget for building the compiler. This chapter is the story of the process the compiler writer must go through to determine the compiler structure.

The best way to use this information to design a compiler is to manually simulate the compilation process using the same programs provided by the user community. For the sake of brevity, one principle example will be used in this book. We will use this example to determine the optimization techniques that are needed, together with the order of the transformations.

For the purpose of exposition this chapter simplifies the process. First we will describe the basic framework, including the major components of the compiler and the structure of the compilation unit within the compiler. Then we will manually simulate an example program.

The example is the Fortran subroutine in Figure 2.1. It finds the largest element in each column of a matrix, saving both the index and the absolute value of the largest element. Although it is written in Fortran, the choice of the source language is not important. The example could be written in any of the usual source languages. Certainly, there are optimizations that are more important in one language than another, but all languages are converging to a common set of features, such as arrays, pointers, exceptions, procedures, that share many characteristics. However, there are special characteristics of each source language that must be compiled well. For example, C has a rich set of constructs involving pointers for indexing arrays or describing dynamic storage, and Fortran has special rules concerning formal parameters that allow increased optimization.

```
SUBROUTINE MAXCOL(A,N,LARGE,VALUE)
    DOUBLE PRECISION VALUE(N), A(N,N)
    INTEGER N, LARGE(N)
    INTEGER I, J
    DO I = 1, N
       LARGE(I) = 1
       VALUE(I) = DABS(A(1,I))
       DO J = 2, N
          IF (DABS(A(J,I).GT.VALUE(I))
             VALUE(I) = DABS(A(J,I))
             LARGE(I) = J
          ENDIF
       ENDDO
    ENDDO
END
```

Figure 2.1 Finding Largest Elements in a Column

2.1 Outline of the Compiler Structure

This book is a simplification of the design process. To design a compiler from scratch one must iterate the process. First hypothesize a compiler structure. Then simulate the compilation process using this structure. If it works as expected (it won‘t) then the design is acceptable. In the process of simulating the compilation, one will find changes one wishes to make or will find that the whole framework does not work. So, modify the framework and simulate again,. Repeat the process until a satisfactory framework is found. If it really does not work, scrap the framework and start again.

There are two major decisions to be made concerning the structure: how the program is represented and in what order the transformations are performed. The source program is read by the compiler front end and then later translated into a form, called the intermediate representation (IR), for optimization, code generation, and register allocation. Distinct collections of transformations, called *phases*, are then applied to the IR.

2.1.1 Source Program Representation

The source program must be stored in the computer during the translation process. This form is stored in a data structure called the IR. Past experience has shown that this representation should satisfy three requirements:

1. The intermediate form of the program should be stored in a form close to machine language, with only certain operations kept in high-level form to be “lowered” later. This allows each phase

to operate on all instructions in the program. Thus, each optimization algorithm can be applied to all of the instructions. If higher-level operators are kept in the IR, then the subcomponents of these operations cannot be optimized or must be optimized later by specialized optimizers.

2. Each phase of the compiler should retain all information about the program in the IR. There should be no implicit information, that is, information that is known after one phase and not after another. This means that each phase has a simple interface and the output may be tested by a small number of simulators. An implication of this requirement is that no component of the compiler can use information about how another component is implemented. Thus components can be modified or replaced without damage to other components.

3. Each phase of the compiler must be able to be tested in isolation. This means that we must write support routines that read and write examples of the IR. The written representation must be in either a binary or textual representation.

The assembly program also has program labels that represent the places to which the program can branch. To represent this concept, the intermediate representation is divided into *blocks* representing straight-line sequences of instructions. If one instruction in a block is executed, then all instructions are executed. Each block starts with a label (or is preceded by a conditional branching instruction) and ends with a branching instruction. Redundant branches are added to the program to guarantee that there is a branch under every possible condition at the end of the block. In other words, there is no fall-through into the next block.

The number of operation codes is large. There is a distinct operation code for each instruction in the target machine. Initially these are not used; however, the lowering process will translate the set of machine-independent operation codes into the target machine codes as the compilation progresses. There is no need to list all of the operation codes here. Instead the subset of instructions that are used in the examples is listed in Figure 2.2.

Now the source program is modeled as a directed graph, with the nodes being the blocks. There is a directed edge between two blocks if there is a possible branch from the first block to the second. A unique node called *Entry* represents the entry point for the source program. The entry node has no predecessors in the graph. Similarly, a unique node called *Exit* represents the exit point for the source program, and that node has no successors. In Figure 2.3 the entry node is node B0, and the exit node is node B5.

```
iLDC    c => T          Load constant c into temporary T
i2i     T1 => T2        Copy integer temporary T1 into T2
iSLD    (T1) => T2      Load integer from memory location T1 into T2
iCMPGT  T1,T2 => T3     Place true in T3 if and only if T1 > T2
iBCOND  T,B1,B2         If T is true, branch to block B1; otherwise branch to block B2
iSUB    T1,T2 => T3     integer T3 = T1 - T2
iMUL    T1,T2 => T3     integer T3 = T1 * T2
iADD    T1,T2 => T3     integer T3 = T1 + T2
iSST    (T1),T2         Store value T2 into integer location T1
dSLD    (T1) => SF1     Load double precision value in address T1 into SF1
dSST    (T1),SF1        Store value in double precision temporary SF1 into address T1
dABS    SF1 => SF2      Place absolute value of value in SF1 into SF2
dCMPLE  SF1,SF2 => SF3  Place true in SF3 if and only if SF1 <= SF2
dBCOND  SF1,B1,B2       If SF1 is true, branch to B1; otherwise, branch to block B2
d2d     SF1 => SF2      Copy value in SF1 into SF2
BR      B               Unconditionally branch to block B
```

Figure 2.2 Operation Codes Used In Examples

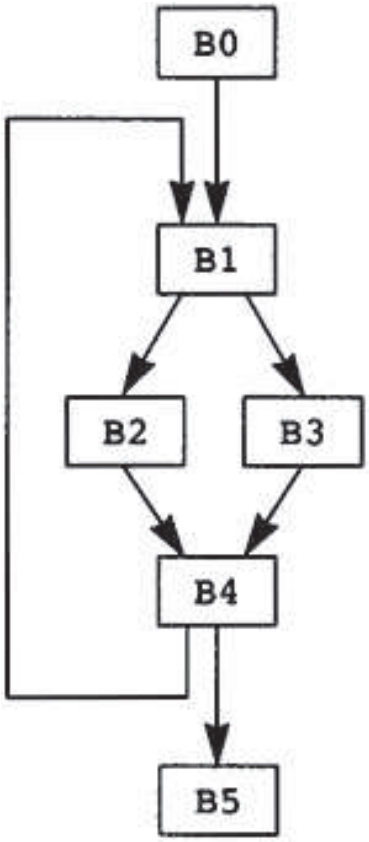


Figure 2.3 Example Flow Graph

The execution of the source program is modeled by a path through the graph. The path starts at the entry node and terminates at the exit node. The computations within each node in the path are executed in order of the occurrence of nodes on the path. In fact, the computations within the node are used to determine the next node in the path. In Figure 2.3, one possible path is B0, B1, B2, B4, B1, B3, B4, B5. This execution path means that all computations in B0 are executed, then all computations in B1, then B2, and so on. Note that the computations in B1 and B4 are executed twice.

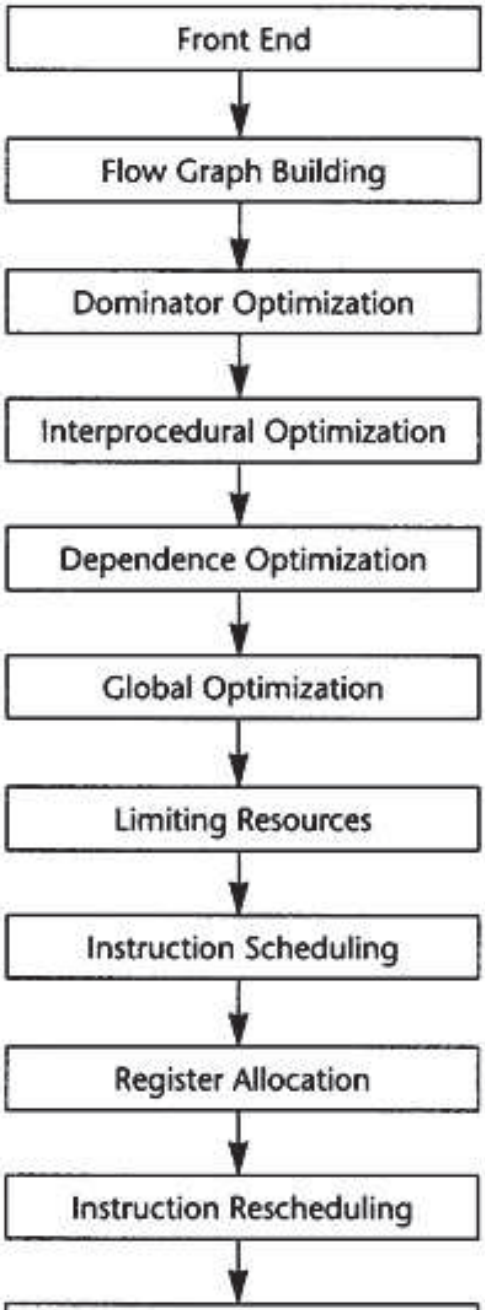
2.1.2 Order of Transformations

Since the compiler structure is hard to describe linearly, the structure is summarized here and then reviewed during the remainder of the chapter. The rest of the book provides the details. The compiler is divided into individual components called phases as shown in Figure 2.4. An overview of each of the phases is presented next.

The compiler front end is language specific. It analyzes the source file being compiled and performs all lexical analysis, parsing, and semantic checks. It builds an abstract syntax tree and symbol table. I will not discuss this part of the compiler, taking it as a given, because most textbooks do an excellent job of describing it. There is a distinct front end for each language, whereas the rest of the compiler can be shared among compilers for different languages as long as the specialized characteristics of each language can be handled.

After the front end has built the abstract syntax tree, the initial optimization phase builds the flow graph, or intermediate representation. Since the intermediate representation looks like an abstract machine language, standard single-pass code-generation techniques, such as used in 1cc (Frazer and Hanson 1995), can be used to build the flow graph. Although these pattern-matching techniques can be used, the flow graph is sufficiently simple that a straightforward abstract syntax tree walk generating instructions on the fly is sufficient to build the IR. While building the flow graph some initial optimizations can be performed on instructions within each block.

The Dominator Optimization phase performs the initial global optimizations. It identifies situations where values are constants, where two computations are known to have the same value, and where instructions have no effect on the results of the program. It identifies and eliminates most redundant computations. At the same time it reapplies the optimizations that have already occurred within a single block. It does not move instructions from one point of the flow graph to another.



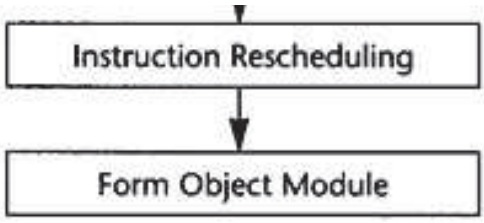


Figure 2.4 Compiler Structure

The Interprocedural Optimization phase analyzes the procedure calls within this flow graph and the flow graphs of all of the other procedures within the whole program. It determines which variables might be modified by each procedure call, which variables and expressions might be referencing the same memory location, and which parameters are known to be constants. It stores this information for other phases to use.

The Dependence Optimization phase attempts to optimize the time taken to perform load and store operations. It does this by analyzing array and pointer expressions to see if the flow graph can be transformed to one in which fewer load/stores occur or in which the load and store operations that occur are more likely to be in one of the cache memories for the RISC chip. To do this it might interchange or unroll loops.

The Global Optimization phase lowers the flow graph, eliminating the symbolic references to array expressions and replacing them with linear address expressions. While doing so, it reforms the address expressions so that the operands are ordered in a way that ensures that the parts of the expressions that are dependent on the inner loops are separated from the operands that do not depend on the inner loop. Then it performs a complete list of global optimizations, including code motion, strength reduction, and dead-code elimination.

After global optimization, the exact set of instructions in the flow graph has been found. Now the compiler must allocate registers and reorder the instructions to improve performance. Before this can be done, the flow graph is transformed by the Limiting Resources phase to make these later phases easier. The Limiting Resources phase modifies the flow graph to reduce the number of registers needed to match the set of physical registers available. If the compiler knows that it needs many more registers than are available, it will save some temporaries in memory. It will also eliminate useless copies of temporaries.

Next an initial attempt to schedule the instructions is performed. Register allocation and scheduling conflict, so the compiler attempts to schedule the instructions. It counts on the effects of the Limiting Resources phase to ensure that the register allocation can be performed without further copying of values to memory. The instruction scheduler reorders the instructions in several blocks simultaneously to decrease the time that the most frequently executed blocks require for execution.

After instruction scheduling, the Register Allocation phase replaces temporaries by physical registers. This is a three-step process in which temporaries computed in one block and used in another are assigned first, then temporaries within a block that can share a register with one already assigned, and finally the temporaries assigned and used in a single block. This division counts on the work of the Limiting Resources phase to decrease the likelihood that one assignment will interfere with a later assignment.

It is hoped that the Register Allocation phase will not need to insert store and load operations to copy temporaries into memory. If such copies do occur, then the Instruction Scheduling phase is repeated. In this case, the scheduler will only reschedule the blocks that have had the instructions inserted.

Finally, the IR is in the form in which it represents an assembly language procedure. The object module is now written in the form needed by the linker. This is a difficult task because the documentation of the form of object modules is notoriously inaccurate. The major work lies in discovering the true form. After that it is a clerical (but large) task to create the object module.

2.2 Compiler Front End

To understand each of the phases, we simulate a walk-through of our standard example in Figure 2.1 for each phase, starting with the front end. The front end translates the source program into an abstract syntax tree. As noted earlier, I will not discuss the operation of the front end; however, we do need to understand the abstract syntax tree. The abstract syntax tree for the program in Figure 2.1 is given in Figure 2.5.

There is a single tree for each procedure, encoding all of the procedure structure. The tree is represented using indentation; the subtrees of each node are indented an extra level. Thus the type of a node occurs at one indentation and the children are indented slightly more. I am not trying to be precise in describing the abstract syntax tree. The name for the type of each node was chosen to represent the node naturally to the reader. For example, the nodes with type “assign” are assignment nodes.

The “list” node represents a tree node with an arbitrary number of children, used in situations in which there can be an arbitrary number of components, such as blocks of statements. The “symbol” node takes a textual argument indicating the name of the variable; of course, this will actually be represented as a pointer to the symbol table.

The “fetch” node differentiates between addresses and values. This compiler has made a uniform assumption about expressions: Expressions always represent values. Thus the “assign” node takes two expressions as operands—one representing the address of the location for getting the result and the other representing the value of the right side of the assignment. The “fetch” node translates between addresses and values. It takes one argument, which is the address of a location. The result of the “fetch” node is the value stored in that location.

```
Procedure
   symbol("MAXCOL")
   list
      symbol("a")
      symbol("n")
      symbol("large")
      symbol("value")
   list
      declare("a",double,array,parameter,bounds=("n","n"))
      declare("n",integer,scalar,parameter)
      declare("large",integer,array,parameter,bounds=("n"))
      declare("value",double,array,parameter,bounds=("n"))
      declare("i",integer,scalar,local)
      declare("j",integer,scalar,local)
   list
      doloop
         symbol("i")
         intconst(1)
         fetch
            symbol("n")
         list
            assign
               subscript
                  symbol("large")
                  list
                     fetch
                        symbol("i")
               intconst(1)
            assign
               subscript
                  symbol("value")
                  list
                     fetch
                        symbol("i")
               absvalue
                  fetch
                     subscript
                        symbol("a")
                     list
                        intconst(1)
                        fetch
                     symbol("i")
            doloop
               symbol("j")
               intconst(2)
               fetch
                  symbol("n")
               list
                  ifthen
                     greater_than
                        absvalue
                           fetch
                              subscript
```

```
greater_than
   absvalue
      fetch
         subscript
            symbol("a")
            list
                fetch
                  symbol("j")
                fetch
                  symbol("i")
   fetch
      subscript
         symbol("value")
         list
            fetch
               symbol("i")
list
   assign
      subscript
         symbol("value")
         list
            fetch
               symbol("i")
      absvalue
         fetch
            subscript
               symbol("a")
            list
               fetch
                  symbol("j")
               fetch
                  symbol("i")
   assign
      subscript
         symbol("large")
         list
            fetch
               symbol("i")
      fetch
         symbol("j")
```

Figure 2.5 Abstract Syntax Tree for MAXCOL

Note that this tree structure represents the complete structure of the program, indicating which parts of the subroutine are contained in other parts.

2.3 Building the Flow Graph

The abstract syntax tree is translated into the flow graph using standard code generation techniques described in introductory compiler books. The translation can be done in two ways. The more advanced method is to use one of the tree-based pattern-matching algorithms on the abstract syntax tree to derive

the flow graph. This technique is not recommended here because of the RISC nature assumed for the target machine. Complex instructions will be generated later by pattern matching the flow graph. Instead, the abstract syntax tree should be translated into the simplest atomic instructions possible. This procedure allows more opportunity for optimization.

Thus, the translation should occur as a single walk of the abstract syntax tree. Simple instructions should be generated wherever possible. Normal operations such as addition and multiplication can be lowered to a level in which each entry in the program flow graph represents a single instruction. However, operations that need to be analyzed later (at the equivalent of source program level) are translated into higher-level operations equivalent to the source program construct. These will later be translated into lower-level operations after completion of the phases that need to analyze these operations. The following four classes of operations should be kept in higher-level form:

1. A fetch or store of a subscript variable, *A* [*i,j,k*] is kept as a single operation, with operands being the array name and the expressions for the subscripts. Keeping subscripted variables in this form rather than linearizing the subscript expression allows later dependence analysis to solve sets of linear equations and inequalities involving the subscripts.

2. Extra information is kept with normal load and store operations also. This information is needed to determine which store operations can modify locations loaded by load operations. This is particularly important in languages involving pointers. Extra analysis, called *pointer alias analysis*, is needed to determine which storage locations are modified. Loads and stores of automatic variables, that is, variables declared within a routine whose values are lost at the end of the routine, are not generated. Instead these values are handled as if they were temporaries within the program flow graph.

3. Subroutine calls are kept in terms of the expression representing the name of the procedure and the expression representing the arguments.Methods for passing the arguments, such as call-by-value and call-by-reference, are not expanded. This allows more detailed analysis by the interprocedural analysis components later in the compiler.

4. Library routines are handled differently than other procedure calls. If a library procedure is known to be a pure function it is handled as if it were an operator. This allows the use of identities involving the library routines. Other procedure calls may be used in other parts of the analysis of the program, for example, calls on **malloc** are known to return either a null pointer or a pointer to a section of memory unreferenced in other parts of the program.

A straightforward translation will result in the flow graph shown in Figure 2.6. It is shown here to describe the process of translation. It is not actually generated, since certain optimizations will be performed during the translation process. Note that the temporaries are used in two distinct ways. Some temporaries, such as T5, are used just like local variables, holding values that are modified as the program is executed. Other temporaries, such as T7, are pure functions of their arguments. In the case of T7, it always holds the constant 1. For these temporaries the same temporary is always used for the result of the same operation. Thus any load of the constant 1 will always be into T7. The translation process must guarantee that an operand is evaluated before it is used.

To guarantee that the same temporary is used wherever an expression is computed, a separate table called the *formal temporary table* is maintained. It is indexed by the operator and the temporaries of the operands and constants involved in the instruction. The result of a lookup in this table is the name of the temporary for holding the result of the operation. The formal temporary table for the example routine is shown in Figure 2.7. Some entries that will be added later are listed here for future reference.

What is the first thing that we observe about the lengthy list of instructions in Figure 2.6? Consider block B1. The constant 1 is loaded six times and the expression *I* - 1 is evaluated three times. A number of simplifications can be performed as the flow graph is created:

- If there are two instances of the same computation without operations that modify the operands between the two instances, then the second one is redundant and can be eliminated since it will always compute the same value as the first.

```
B0:   PROLOG
      iLDC      1             => T7      /constant 1 always in T7
      i2i       T7            => T5      /I = 1, first copy into value
      iSLD      (T2)          => T8      /fetch value of N. T2 address of N
      iCMPGT    T5,T8         => T9      /is I > N
      iBCOND    T9,B5,B1                 /no iterations to execute
B1:   iLDC      1             => T7      /constant 1
      iSUB      T5,T7         => T10     /I - 1
      iLDC      4             => T11     /constant 4
      iMUL      T11,T10       => T12     /4*(i-1)
      iADD      T3,T12        => T13     /address(LARGE(I)), T3 address of LARGE
      iLDC      1             => T7      /constant 1
      i2i       T7            => T14     /put in register for LARGE(I)
      iSST      (T13),T14                /store the value into LARGE(I)
      iLDC      1             => T7      /constant 1
      iSUB      T5,T7         => T10     /I - 1
      iLDC      8             => T15     /constant 8
      iMUL      T15,T10       => T16     /8*(I-1)
      iADD      T4,T16        => T17     /address(VALUE(I))
      iLDC      1             => T7      /constant 1 - compute address(A(1,I))
      iLDC      1             => T7      /constant 1
      iSUB      T7,T7         => T18     /1 - 1
      iLDC      1             => T7      /constant 1
      iSUB      T5,T7         => T10     /I - 1
      iSLD      (T2)          => T8      /fetch N
```

```
        iLDC    1           => T7     /constant 1
        iSUB    T7,T7       => T18    /1 - 1
        iLDC    1           => T7     /constant 1
        iSUB    T5,T7       => T10    /I - 1
        iSLD    (T2)        => T8     /fetch N
        iMUL    T8,T10      => T19    /N*(I-1)
        iADD    T19,T18     => T20    /N*(I-1) + (1-1)
        iLDC    8           => T15    /constant 8
        iMUL    T15,T20     => T21    /8*(N*(I-1) + (1-1))
        iADD    T1,T21      => T22    /address(A(1,I))
        dSLD    (T22)       => SF2    /value A(1,I)
        dABS    SF2         => SF3    /DABS(A(1,I))
        d2d     SF3         => SF1    /copy into value of VALUE(I)
        dSST    (T17),SF1             /store the value of VALUE(I)
        iLDC    2           => T23    /constant 2 - Initialize index
        i2i     T23         => T6     /setup value of J
        iSLD    (T2)        => T8     /fetch N
        iCMPGT  T6,T8       => T24    /is J > N
        iBCOND  T24,B4,B2             /yes - no iterations
B2:     iLDC    1           => T7     /1
        iSUB    T6,T7       => T25    /J - 1
        iLDC    1           => T7     /constant 1
        iSUB    T5,T7       => T10    /I - 1
        iSLD    (T2)        => T8     /fetch N
        iMUL    T8,T10      => T19    /N*(I-1)
        iADD    T19,T25     => T26    /N*(I-1) + (J-1)
        iLDC    8           => T15    /constant 8
        iMUL    T15,T26     => T27    /8*(N*(I-1) + (J-1))
        iADD    T1,T27      => T28    /address(A(J,I))
        dSLD    (T28)       => SF4    /value A(J,I)
        dABS    SF4         => SF5    /DABS(A(J,I))
        iLDC    1           => T7     /constant 1
        iSUB    T5,T7       => T10    /I-1
        iLDC    8           => T15    /constant 8
        iMUL    T15,T10     => T16    /8*(I-1)
        iADD    T4,T16      => T17    /address(VALUE(I))
        dSLD    (T17)       => SF1    /value of VALUE(I)
        dCMPLE  SF5,SF1     => SF6    /comparison
        dBCOND  SF6,B3,B6             /skip update of variables
B6:     iLDC    1           => T7     /constant 1
        iSUB    T5,T7       => T10    /I-1
        iLDC    8           => T15    /constant 8
        iMUL    T15,T10     => T16    /8*(I-1)
        iADD    T4,T16      => T17    /address(VALUE(I))
        iLDC    1           => T7     /1
        iSUB    T6,T7       => T25    /J-1
        iLDC    1           => T7     /constant 1
        iSUB    T5,T7       => T10    /I-1
        iSLD    (T2)        => T8     /fetch N
        iMUL    T8,T10      => T19    /N*(I-1)
        iADD    T19,T25     => T26    /N*(I-1)+(J-1)
        iLDC    8           => T15    /constant 8
        iMUL    T15,T26     => T27    /8*(N*(I-1)+(J-1))
        iADD    T1,T27      => T28    /address(A(J,I))
        dSLD    (T28)       => SF4    /value A(J,I)
        dABS    SF4         => SF5    /DABS(A(J,I))
        d2d     SF5         => SF1    /update value of VALUE(I)
        dSST    (T17),SF1             /store it back in memory
        iLDC    1           => T7     /constant 1
        iSUB    T5,T7       => T10    /I-1
```

```
        dABS      SF4           => SF5     /DABS(A(J,I))
        d2d       SF5           => SF1     /update value of VALUE(I)
        dSST      (T17),SF1                /store it back in memory
        iLDC      1             => T7      /constant 1
        iSUB      T5,T7         => T10     /I-1
        iLDC      4             => T11     /constant 4
        iMUL      T11,T10       => T12     /4*(i-1)
        iADD      T3,T12        => T13     /address(LARGE(I))
        i2i       T6            => T14     /put in register for LARGE(I)
        iSST      (T13),T14                /store the value into LARGE(I)
        BR        B3

B3:     iLDC      1             => T7      /increment by 1
        iADD      T6,T7         => T29     /J+1
        i2i       T29           => T6      /update value of J
        iSLD      (T2)          => T8      /fetch N
        iCMPGT    T6,T8         => T24     /is J > N
        iBCOND    T24,B4,B2

B4:     iLDC      1             => T7      /constant 1 to increment I
        iADD      T5,T7         => T30     /I+1
        i2i       T30           => T5      /first store into fetch location
        iSLD      (T2)          => T8      /get value of N
        iCMPGT    T5,T8         => T9      /is I > N
        iBCOND    T9,B5,B1                 /more iterations to perform

B5:     EPILOG                             /return from subroutine
```

Figure 2.6 Initial Program Representation

```
T1    Address(A)                    constant
T2    Address(N)                    constant
T3    Address(LARGE)                constant
T4    Address(VALUE)                constant
T5    Value(I)
T6    Value(J)
T7    Constant 1                    constant
T8    Value of N
T9    I > N
T10   I - 1
T11   Constant 4                    constant
T12   4*(I-1)
T13   Address(LARGE(I))
T14   Value(LARGE(I))
T15   Constant 8                    constant
T16   8*(I-1)
T17   Address(VALUE(I))
SF1   Value(VALUE(I))
T18   1 - 1
T19   N*(I-1)
T20   N*(I-1) + (1-1)
T21   8*(N*(I-1) + (1-1))
T22   Address(A(1,I))
SF2   Value(A(1,I))
SF3   DABS(A(1,I))
T23   Constant 2                    constant
T24   J > N
T25   J - 1
T26   N*(I-1) + (J-1)
T27   8*(N*(I-1) + (J-1))
T28   Address(A(J,I))
SF4   Value(A(J,I))
SF5   DABS(A(J,I))
SF6   DABS(A(J,I)) >= VALUE(I)
T29   J + 1
T30   I + 1
T31   0 > N                         added during value numbering
T32   8*N*(I-1)                     added during value numbering
T33   Address(A(1,I)) simplified    added during value numbering
T34   2 > N                         added during value numbering
T35   T28 + 8                       added during strength reduction
T36   T13 + 4                       added during strength reduction
T37   T17 + 8                       added during strength reduction
T38   8*N                           added during strength reduction
T39   T33 + 8*N                     added during strength reduction
```

Figure 2.7 Initial Formal Temporary Table

• *Algebraic identities* can be used to eliminate operations. For example, *A* * 0 can be replaced by 0. This can only occur if the side effects of computing A can be ignored. There is a large collection of algebraic identities that may be applied; however, a small set is always applied with

the understanding that new algebraic identities can be added if occasions occur where the identities can improve the program.

- *Constant folding* transforms expressions such as 5 * 7 into the resulting number, 35. This frequently makes other simplifications possible. The arithmetic must be done in a form that exactly mimics the arithmetic of the target machine.

These transformations usually remove about 50 percent of the operations in the procedure. The rest of the analysis in the compiler is therefore faster since about half of the operations that must be scanned during each analysis have been eliminated. The result of these simplifications is given in Figure 2.8.

2.4 Dominator Optimizations

The preliminary optimization phase takes the program represented as a program flow graph as input. It applies global optimization techniques to the program and generates an equivalent program flow graph as the output. These techniques are global in the sense that the transformations take into account possible branching within each procedure.

There are two global optimization phases in this compiler. The initial phase performs as much global optimization as possible without moving computations in the flow graph. After interprocedural analysis and dependence optimization phases have been executed, a more general global optimization phase is applied to clean up and improve the flow graphs further. The following global optimization transformations are applied.

- If there are two instances of a computation *X* * *Y* and the first one occurs on all paths leading from the *Entry* block to the second computation, then the second one can be eliminated. This is a special case of the general elimination of redundant expressions, which will be performed later. This simple case accounts for the largest number of redundant expressions, so much of the work will be done here before the general technique is applied.
- *Copy propagation* or *value propagation* is performed. If an *X* is a copy of *Z*, then uses of *X* can be replaced by uses of *Z* as long as neither *X* nor *Z* changes between the point at which the copy is made and the point of use. This transformation is useful for improving the program flow graph generated by the compiler front end. There are many compiler-generated temporaries such as loop counters or components of array dope information that are really copy operations.

```
B0:   PROLOG
      iLDC      1           => T7     /constant 1 always in T7
      i2i       T7          => T5     /I = 1, first copy into value
      iSLD      (T2)        => T8     /fetch value of N
      iCMPGT    T7,T8       =>T31     /is I > N
      iBCOND    T31,B5,B1             /no iterations to execute

B1:   iLDC      1           => T7     /constant 1
      iSUB      T5,T7       => T10    /I - 1
      iLDC      4           => T11    /constant 4
      iMUL      T11,T10     => T12    /4*(i-1)
      iADD      T3,T12      => T13    /address(LARGE(I))
      i2i       T7          => T14    /put in register for LARGE(I)
```

```
        iSUB    T5,T7       => T10    /I - 1
        iLDC    4           => T11    /constant 4
        iMUL    T11,T10     => T12    /4*(i-1)
        iADD    T3,T12      => T13    /address(LARGE(I))
        i2i     T7          => T14    /put in register for LARGE(I)
        iSST    (T13),T14             /store the value into LARGE(I)
        iLDC    8           => T15    /constant 8
        iMUL    T15,T10     => T16    /8*(I-1)
        iADD    T4,T16      => T17    /address(VALUE(I))
        iSLD    (T2)        => T8     /fetch N
        iMUL    T8,T10      => T19    /N*(I-1)
        iMUL    T15,T19     => T32    /8*N*(I-1)
        iADD    T1,T32      => T33    /address(A(1,I))
        dSLD    (T33)       => SF2    /value A(1,I)
        dABS    SF2         => SF3    /DABS(A(1,I))
        d2d     SF3         => SF1    /copy into value of VALUE(I)
        dSST    (T17),SF1             /store value into VALUE(I)
        iLDC    2           => T23    /constant 2 - Initialize index
        i2i     T23         => T6     /setup value of J
        iSLD    (T2)        => T8     /fetch N
        iCMPGT  T23,T8      => T34    /is 2 > N
        iBCOND  T34,B4,B2             /yes - no iterations

B2:     iLDC    1           => T7     /1
        iSUB    T6,T7       => T25    /J - 1
        iSUB    T5,T7       => T10    /I - 1
        iSLD    (T2)        => T8     /fetch N
        iMUL    T8,T10      => T19    /N*(I-1)
        iADD    T19,T25     => T26    /N*(I-1)+(J-1)
        iLDC    8           => T15    /constant 8
        iMUL    T15,T26     => T27    /8*(N*(I-1)+(J-1))
        iADD    T1,T27      => T28    /address(A(J,I))
        dSLD    (T28)       => SF4    /value A(J,I)
        dABS    SF4         => SF5    /DABS(A(J,I))
        iMUL    T15,T10     => T16    /8*(I-1)
        iADD    T4,T16      => T17    /address(VALUE(I))
        dSLD    (T17)       => SF1    /value of VALUE(I)
        dCMPLE  SF5,SF1     => SF6    /comparison
        dBCOND  SF6,B3,B6             /skip update of variables

B6:     iLDC    1           => T7     /constant 1
        iSUB    T5,T7       => T10    /I - 1
        iLDC    8           => T15    /constant 8
        iMUL    T15,T10     => T16    /8*(I-1)
        iADD    T4,T16      => T17    /address(VALUE(I))
        iSUB    T6,T7       => T25    /J - 1
        iSLD    (T2)        => T8     /fetch N
        iMUL    T8,T10      => T19    /N*(I-1)
        iADD    T19,T25     => T26    /N*(I-1)+(J-1)
        iMUL    T15,T26     => T27    /8*(N*(I-1)+(J-1))
        iADD    T1,T27      => T28    /address(A(J,I))
        dSLD    (T28)       => SF4    /value A(J,I)
        dABS    SF4         => SF5    /DABS(A(J,I))
        d2d     SF5         => SF1    /update value of VALUE(I)
        dSST    (T17),SF1             /store it back in memory
        iLDC    4           => T11    /constant 4
        iMUL    T11,T10     => T12    /4*(i-1)
        iADD    T3,T12      => T13    /address(LARGE(I))
        i2i     T6          => T14    /put in register for LARGE(I)
```

```
        sSST    (T17),SP1               /store it back in memory
        iLDC    4           => T11      /constant 4
        iMUL    T11,T10     => T12      /4*(i-1)
        iADD    T3,T12      => T13      /address(LARGE(I))
        i2i     T6          => T14      /put in register for LARGE(I)
        iSST    (T13),T14               /store the value into LARGE(I)
        BR      B3
B3:     iLDC    1           => T7       /increment by 1
        iADD    T6,T7       => T29      /J + 1
        i2i     T29         => T6       /update value of J
        iSLD    (T2)        => T8       /fetch N
        iCMPGT  T6,T8       => T24      /is J > N
        iBCOND  T24,B4,B2
B4:     iLDC    1           => T7       /constant 1 to increment I
        iADD    T5,T7       => T30      /I + 1
        i2i     T30         => T5       /first store into fetch location
        iSLD    (T2)        => T8       /get value of N
        iCMPGT  T5,T8       => T9       /is I > N
        iBCOND  T9,B5,B1                /more iterations to perform
B5:     EPILOG                          /return from subroutine
```

Figure 2.8 Flow Graph after Simplifications

- *Constant propagation* is the replacement of uses of variables that have been assigned a constant value by the constant itself. If a constant is used to determine a conditional branch in the program, the alternative branch is not considered.
- As with local optimization, algebraic identities, peephole optimizations, and constant folding will also be performed as the other optimizations are applied.

The following global optimizations are intentionally *not* applied because they make the task of dependence analysis more difficult later in the compiler.

- *Strength reduction* is not applied. Strength reduction is the transformation of multiplication by constants (or loop invariant expressions) into repeated additions. More precisely, if one has an expression *I* * 3 in a loop and *I* is incremented by 1 each time through the loop, then the computation of *I* * 3 can be replaced by a temporary variable *T* that is incremented by 3 each time through the loop.
- *Code motion* is not applied. A computation *X* * *Y* can be moved from within a loop to before the loop when it can be shown that the computation is executed each time through the loop and that the operands do not change value within the loop. This transformation inhibits loop interchange, which is performed to improve the use of the data caches, so it is delayed until the later global optimization phase.

Now inspect the flow graph, running your finger along several possible paths through the flow graph from the start block B0 to the exit block B5. The constant 1 is computed repeatedly on each path. More expensive computations are also repeated. Look at blocks B2 and B6. Many of the expressions computed in B6 are also computed in B2. Since B2 occurs on each path leading to B6, the computations in B6 are unnecessary.

What kind of technology can cheaply eliminate these computations? B2 is the dominator of B6 (this will be defined more precisely shortly), meaning that B2 occurs on each path leading from B0 to B6. There is a set of algorithms applied to the Static Single Assignment Form (to be defined shortly) of the flow graph that can eliminate repeated computations of constants and expressions when they already occur in the dominator. Some Static Single Assignment Form algorithms will be in the compiler anyway, so we will use this form to eliminate redundant computations where a copy of the computation already occurs in the dominator. This is an inexpensive generalization of local optimizations used during the construction of the flow graph, giving the results in Figure 2.9.

Repeat the exercise of tracing paths through the flow graph. Now there are few obvious redundant expressions. There are still some, however. Computations performed each time through the loop have not been moved out of the loop. Although they do not occur in this example, there are usually other redundant expressions that are not made redundant by this transformation.

Where are most of the instructions? They are in block B2, computing the addresses used to load array elements. This address expression changes each time through the loop, so it cannot be moved out of the loop that starts block B2. It changes in a regular fashion, increasing by 8 each time through the loop, so the later global optimization phase will apply strength reduction to eliminate most of these instructions.

```
B0:   PROLOG
      iLDC      1           => T7     /constant 1 always in T7
      i2i       T7          => T5     /I = 1, first copy into value
      iSLD      (T2)        => T8     /fetch value of N
      iCMPGT    T7,T8       => T31    /is I > N
      iBCOND    T31,B5,B1             /no iterations to execute
B1:   iSUB      T5,T7       => T10    /I - 1
      iLDC      4           => T11    /constant 4
      iMUL      T11,T10     => T12    /4*(i-1)
      iADD      T3,T12      => T13    /address(LARGE(I))
      i2i       T7          => T14    /put in register for LARGE(I)
      iSST      (T13),T14             /store the value into LARGE(I)
      iLDC      8           => T15    /constant 8
      iMUL      T15,T10     => T16    /8*(I-1)
      iADD      T4,T16      => T17    /address(VALUE(I))
      iMUL      T8,T10      => T19    /N*(I-1)
      iMUL      T15,T19     => T32    /8*N*(I-1)
      iADD      T1,T32      => T33    /address(A(1,I))
      dSLD      (T33)       => SF2    /value A(1,I)
      dABS      SF2         => SF3    /DABS(A(1,I))
      d2d       SF3         => SF1    /copy into value of VALUE(I)
      dSST      (T17),SF1             /store value in VALUE(I)
      iLDC      2           => T23    /constant 2 - Initialize index
      i2i       T23         => T6     /setup value of J
      iCMPGT    T23,T8      => T34    /is 2 > N
      iBCOND    T34,B4,B2             /yes - no iterations
B2:   iSUB      T6,T7       => T25    /J - 1
      iADD      T19,T25     => T26    /N*(I-1) + (J-1)
      iMUL      T15,T26     => T27    /8*(N*(I-1) + (J-1))
      iADD      T1,T27      => T28    /address(A(J,I))
      dSLD      (T28)       => SF4    /value A(J,I)
      dABS      SF4         => SF5    /DABS(A(J,I))
      dSLD      (T17)       => SF1    /value of VALUE(I)
      dCMPLE    SF5,SF1     => SF6    /comparison
      dBCOND    SF6,B3,B6             /skip update of variables
B6:   d2d       SF5         => SF1    /update value of VALUE(I)
      dSST      (T17),SF1             /store it back in memory
      i2i       T6          => T14    /put in register for LARGE(I)
      iSST      (T13),T14             /store the value into LARGE(I)
      BR        B3
B3:   iADD      T6,T7       => T29    /J + 1
      i2i       T29         => T6     /update value of J
      iCMPGT    T6,T8       => T24    /is J > N
      iBCOND    T24,B4,B2
B4:   iADD      T5,T7       => T30    /I + 1
      i2i       T30         => T5     /first store into fetch location
      iCMPGT    T5,T8       => T9     /is I > N
      iBCOND    T9,B5,B1              /more iterations to perform
B5:   EPILOG                          /return from subroutine
```

Figure 2.9 After Dominator Value Numbering

2.5 Interprocedural Analysis

All other phases of the compiler handle the program flow graph for one procedure at a time. Each phase accepts as input the program flow graph (or abstract syntax tree) and generates the program flow graph as a result. The interprocedural analysis phase accumulates the program flow graphs for each of the procedures. It analyzes all of them, feeding the program flow graphs for each procedure, one at a time, to the rest of the phases of the compiler. The procedures are not provided in their original order. In the absence of recursion, a procedure is provided to the rest of the compiler before the procedures that call it. Hence more information can be gathered as the compilation process proceeds.

The interprocedural analysis phase computes information about procedure calls for other phases of the compiler. In the local and global optimization phases of the compiler, assumptions must be made about the effects of procedure calls. If the effects of the procedure call are not known, then the optimization phase must assume that all values that are known to that procedure and all procedures that it might call can be changed or referenced by the procedure call. This is an inconvenient assumption in modern languages, which encourage procedures (or member functions) to structure the program.

To avoid these conservative assumptions about procedure calls, this phase computes the following information for each procedure call:

MOD
: The set of variables that might be modified by this procedure call.

REF
: The set of variables that might be referenced by this procedure call.

Interprocedural analysis also computes information about the relationships and values of the formal parameters of a procedure, including the following information:

Alias
: With call-by-reference parameters, one computes which parameters possibly reference the same memory location as another parameter or global variable.

Constant
: The parameters that always take the same constant value at all calls of the procedure. This information can be used to improve on the constant propagation that has already occurred.

When array references are involved, the interprocedural analysis phase attempts to determine which part of the array has been modified or referenced. Approximations must be made in storing this information because only certain shapes of storage reference patterns will be stored. When the actual shape does not fit one of the usual reference patterns, a conservative choice will be made to expand the shape to one of the chosen forms.

2.6 Dependence Optimization

The purpose of dependence optimization for a RISC processor is to decrease the number of references to memory and improve the pattern of memory references that do occur.

This goal can be achieved by restructuring loops so that fewer references to memory are made on each iteration. The program is transformed to eliminate references to memory, as in Figure 2.10, in which a transformation called *scalar replacement* is used to hold the value of *A*(*I*), which is used on the next iteration of the loop as the value *A*(*I*-1). Classic optimization techniques cannot identify this possibility, but the techniques of dependence optimization can. A more complex transformation called *unroll and jam* can be used to eliminate more references to memory for nested loops.

When the references to memory cannot be eliminated completely, dependence-based optimization can be used to improve the likelihood that the values referenced are in the cache, thus providing faster reference to memory. The speed of modern processors exceeds the speed of their memory systems. To compensate, one or more cache memory systems have been added to retain the values of recently referenced memory locations. Since recently referenced memory is likely to be referenced again, the hardware can return the value saved in the cache more quickly than if it had to reference the memory location again.

```
DO I = 2, N                      IF (N > 1) THEN
   A(I) = A(I-1) + A(I)             T = A(1)
ENDDO                               DO I = 2, N
                                       T = T + A(I)
                                       A(I) = T
                                    ENDDO
                                 ENDIF
```

Figure 2.10 Example of Scalar Replacement

```
DO I = 1, N                      DO J = 1, N
   DO J = 1, N                      DO I = 1, N
      B(I,J) = A(I,J)*2.0              B(I,J) = A(I,J)*2.0
   ENDDO                            ENDDO
ENDDO                            ENDDO
```

Figure 2.11 Striding Down the Columns

Consider the Fortran fragment in Figure 2.11 for copying array *A* into *B* twice. In Fortran, the elements of a column are stored in sequential locations in memory. The hardware will reference a particular element. The whole cache line for the element will be read into the cache (typically 32 bytes to 128 bytes), but the next element will not come from the cache line; instead, the next element is the next element in the row, which may be very far away in memory. By the time the inner loop is completed and

the next iteration of the outer loop is executing, the current elements in the cache will likely have been removed.

The dependence-based optimizations will transform Figure 2.11 into the right-hand column. The same computations are performed, but the elements are referenced in a different order. Now the next element from A is the next element in the column, thus using the cache effectively.

The phase will also unroll loops to improve later instruction scheduling, as shown in Figure 2.12. The left column is the original loop; the right column is the unrolled loop. In the original loop, the succeeding phases of the compiler would generate instructions that would require that each store to *B* be executed before each subsequent load from *A*. With the loop unrolled, the loads from *A* may be interwoven with the store operations, hiding the time it takes to reference memory. Another optimization called *software pipelining* is performed later, which increases the amount of interweaving even more.

```
DO I = 1, N
   B(I) = A(I)
ENDDO
```

```
DO I = 1, N, 4
   B(I) = A(I)
   B(I+1) = A(I+1)
   B(I+2) = A(I+2)
   B(I+3) = A(I+3)
ENDDO
DO I = I, N
   B(I) = A(I)
ENDDO
```

Figure 2.12 Original (left) and Unrolled (right) Loop

This book will not address the concepts of parallelization and vectorization, although those ideas are directly related to the work here. These concepts are covered in books by Wolfe (1996) and Allen and Kennedy.

2.7 Global Optimization

The global optimization phase cleans up the flow graph transformed by the earlier phases. At this point all global transformations that need source-level information have been applied or the information has been stored with the program flow graph in an encoded form. Before the general algorithm is performed, several transformations need to be performed to simplify the flow graph. These initial transformations are all based on a dominator-based tree walk and the static single assignment method. The optimizations include the original dominator optimizations together with the following.

- *Lowering*: The instructions are lowered so that each operation in the flow graph represents a single instruction in the target machine. Complex instructions, such as subscripted array references, are replaced by the equivalent sequence of elementary machine instructions. Alternatively, multiple instructions may be folded into a single instruction when constants, rather

than temporaries holding the constant value, can occur in instructions.

- *Reshaping*: Before the global optimization techniques are applied, the program is transformed to take into account the looping structure of the program. Consider the expression *I* * *J* * *K* occurring inside a loop, with *I* being the index for the innermost loop, *J* the index for the next loop, and *K* the loop invariant. The normal associativity of the program language would evaluate this as (*I* * *J*) **K* when it would be preferable to compute it as *I* * (*J* * *K*) because the computation of *J* * *K* is invariant inside the innermost loop and so can be moved out of the loop. At the same time we perform strength reduction, local redundant expression elimination, and algebraic identities.
- *Strength Reduction*: Consider computations that change by a regular pattern during consecutive iterations of a loop. The major example is multiplication by a value that does not change in the loop, such as *I* * *J* where *J* does not change and *I* increases by 1. The multiplication can be replaced by a temporary that is increased by *J* each time through the loop.
- *Elimination*: To assist strength reduction and reshaping, the redundant expression elimination algorithm in the dominator optimization phase is repeated.

The techniques proposed here for code motion are based on a technique called "elimination of partial redundancies" devised by Etienne Morel (Morel and Renvoise, 1979). Abstractly, this technique attempts to insert copies of an expression on some paths through the flow graph to increase the number of redundant expressions. One example of where it works is with loops. Elimination of partial redundancies will insert copies of loop invariant expressions before the loop making the original copies in the loop redundant. Surprisingly, this technique works without knowledge of loops. We combine three other techniques with code motion:

1. A form of strength reduction is included in code motion. The technique is inexpensive to implement and has the advantage that it will apply strength reduction in situations where there are no loops.
2. Load motion is combined with code motion. Moving load operations can be handled as a code motion problem by pretending that any store operation is actually a store operation followed by the corresponding load operation. So a store operation can be viewed as having the same effect on the availability of an expression as a load operation. As will be seen in this example, this will increase the number of load operations that can be moved.
3. Store operations can also be moved by looking at the flow graph backward and applying the same algorithms to the reverse graph that we apply for expressions to the normal flow graph. We only look at the reverse graph for store operations.

In this particular example, code motion only removes the redundant loads of the constants 4 and 8. The load of *VALUE*(*I*) is moved out of the inner loop. It is not a loop-invariant expression since there is a store into *VALUE*(*I*) in the loop. However, the observation that a store may be viewed as a store followed by a load into the same register means that there is a load of *VALUE*(*I*) on each path to the use of *VALUE*(*I*), making the load within the loop redundant. This gives the code in Figure 2.16.

```
B0:  PROLOG
     iLDC    1           => T7   /constant 1 always in T7
     i2i     T7          => T5   /I = 1, first copy into value
     iSLD    (T2)        => T8   /fetch value of N
     iCMPGT  T7,T8       => T31  /is I > N
     iBCOND  T31,B5,B01          /no iterations to execute

B01: i2i     T3          => T13  /address(LARGE(I))
     iLDC    4           => T11  /increment for address(LARGE(I))
     i2i     T4          => T17  /address(VALUE(I))
     iLDC    8           => T15  /increment for address(VALUE(I))
     i2i     T1          => T33  /address(A(1,I))
     iMUL    T15,T8      => T38  /increment for address(A(1,I))
     iLDC    2           => T23  /constant 2 - Initialize index
     BR      B1

B1:  i2i     T7          => T14  /put in register for LARGE(I)
     iSST    (T13),T14           /store the value into LARGE(I)
     dSLD    (T33)       => SF2  /value A(1,I)
     dABS    SF2         => SF3  /DABS(A(1,I))
     d2d     SF3         => SF1  /copy into value of VALUE(I)
```

```
B1:  i2i      T7          => T14   /put in register for LARGE(I)
     iSST     (T13),T14            /store the value into LARGE(I)
     dSLD     (T33)       => SF2   /value A(1,I)
     dABS     SF2         => SF3   /DABS(A(1,I))
     d2d      SF3         => SF1   /copy into value of VALUE(I)
     dSST     (T17),SF1            /store value into VALUE(I)
     i2i      T23         => T6    /setup value of J
     iCMPGT   T23,T8      => T34   /is 2 > N
     iBCOND   T34,B4,B12           /yes - no iterations

B12: iADD     T33,T15     => T28   /address(A(2,I)) = address(A(1,I)) + 8
     BR       B2

B2:  dSLD     (T28)        => SF4   /value A(J,I)
     dABS     SF4          => SF5   /DABS(A(J,I))
     dCMPLE   SF5,SF1      => SF6   /comparison
     dBCOND   SF6,B3,B6             /skip update of variables

B6:  d2d      SF5          => SF1   /update value of VALUE(I)
     dSST     (T17),SF1            /store it back in memory
     i2i      T6           => T14   /put in register for LARGE(I)
     iSST     (T13),T14            /store the value into LARGE(I)
     BR       B3

B3:  iADD     T6,T7        => T29   /J+1
     i2i      T29          => T6    /update value of J
     iADD     T28,T15      => T35   /update value of address(A(J,I))
     i2i      T35          => T28
     iCMPGT   T6,T8        => T24   /is J>N
     iBCOND   T24,B4,B2

B4:  iADD     T5,T7        => T30   /I+1
     i2i      T30          => T5    /first store into fetch location
     iADD     T13,T11      => T36   /increment address(LARGE(I))
     i2i      T36          => T13   /put pointer back in place
     iADD     T17,T15      => T37   /increment address(VALUE(I))
     i2i      T37          => T17   /put pointer back in place
     iADD     T33,T38      => T39   /increment address(A(1,I))
     i2i      T39          => T33   /put pointer back in place
     iCMPGT   T5,T8        => T9    /is I > N
     iBCOND   T9,B5,B1              /more iterations to perform

B5:  EPILOG                        /return from subroutine
```

Figure 2.16 After Code Motion

Now, we can move the store operations forward using partial redundancy on the reverse program flow graph, as shown in Figure 2.17. The stores into *VALUE*(*I*) and *LARGE*(*I*) occurring in the loop can be moved to block B4. Although we think of this as a motion out of the loop, the analysis has nothing to do with the loop. It depends on the occurrence of these store operations on each path to B4 and the repetitive stores that do occur in the loop. Together with dead-code elimination this gives us the final result of the optimization phases.

```
B0:   PROLOG
      iLDC    1           => T7    /constant 1 always in T7
      i2i     T7          => T5    /I = 1, first copy into value
      iSLD    (T2)        => T8    /fetch value of N
      iCMPGT  T7,T8       => T31   /is I > N
      iBCOND  T31,B5,B01           /no iterations to execute

B01:  i2i     T3          => T13   /address(LARGE(I))
      iLDC    4           => T11   /increment for address(LARGE(I))
      i2i     T4          => T17   /address(VALUE(I))
      iLDC    8           => T15   /increment for address(VALUE(I))
      i2i     T1          => T33   /address(A(1,I))
      iMUL    T15,T8      => T38   /increment for address(A(1,I))
      iLDC    2           => T23   /constant 2 - Initialize index
      BR      B1

B1:   i2i     T7          => T14   /put in register for LARGE(I)
      dSLD    (T33)       => SF2   /value A(1,I)
      dABS    SF2         => SF3   /DABS(A(1,I))
      d2d     SF3         => SF1   /copy into value of VALUE(I)
      i2i     T23         => T6    /setup value of J
      iCMPGT  T23,T8      => T34   /is 2 > N
      iBCOND  T34,B4,B12           /yes - no iterations

B12:  iADD    T33,T15     => T28   /address(A(2,I)) = address(A(1,I))+8
      BR      B2

B2:   dSLD    (T28)       => SF4   /value A(J,I)
      dABS    SF4         => SF5   /DABS(A(J,I))
      dCMPLE  SF5,SF1     => SF6   /comparison
      dBCOND  SF6,B3,B6            /skip update of variables

B6:   d2d     SF5         => SF1   /update value of VALUE(I)
      i2i     T6          => T14   /put in register for LARGE(I)
      BR      B3

B3:   iADD    T6,T7       => T29   /J + 1
      i2i     T29         => T6    /update value of J
      iADD    T28,T15     => T35   /update value of address(A(J,I))
      i2i     T35         => T28
      iCMPGT  T6,T8       => T24   /is J > N
      iBCOND  T24,B4,B2

B4:   iSST    (T13),T14              /store the value into LARGE(I)
      dSST    (T17),SF1              /store the value into VALUE(I)
      iADD    T5,T7       => T30   /I + 1
      i2i     T30         => T5    /first store into fetch location
      iADD    T13,T11     => T36   /increment address(LARGE(I))
      i2i     T36         => T13   /put pointer back in place
      iADD    T17,T15     => T37   /increment address(VALUE(I))
      i2i     T37         => T17   /put pointer back in place
      iADD    T33,T38     => T39   /increment address(A(1,I))
      i2i     T39         => T33   /put pointer back in place
      iCMPGT  T5,T8       => T9    /is I > N
      iBCOND  T9,B5,B1             /more iterations to perform
```

```
        iADD    T33,T38     => T39   /increment address(A(1,I))
        i2i     T39         => T33   /put pointer back in place
        iCMPGT  T5,T8       => T9    /is I > N
        iBCOND  T9,B5,B1             /more iterations to perform

B5:     EPILOG                       /return from subroutine
```

Figure 2.17 After Store Motion

2.8 Limiting Resources

The program flow graph for the procedure has now been transformed into a form suitable for generating instructions for the target machine. There is a one-to-one correspondence between the operations in the program flow graph and instructions for the target machine. There are still three things to determine about the resulting program.

- *Peephole optimization*: Multiple instructions must be combined into single instructions that have the same effect. This includes the classic peephole optimizations together with simplifications involving folding constants into instructions that can use constants.
- *Instruction scheduling*: The order of the instructions must be found. By reordering the instructions, the delays inherent in instructions that take more than one machine cycle can be hidden by the execution of other instructions.
- *Register allocation*: The temporaries used for values in the program flow graph must be replaced by the use of physical registers.

Unfortunately, instruction scheduling and register allocation are interdependent. If the compiler reorders the instructions to decrease execution time, it will increase the number of physical registers needed to hold values. On the other hand, if one allocates the temporaries to physical registers before instruction scheduling, then the amount of instruction reordering is limited. This is known as a *phase-ordering problem*. There is no natural order for performing instruction scheduling and register allocation.

The LIMIT phase performs the first of these three tasks and prepares the code for instruction scheduling and register allocation. It attempts to resolve this problem by performing parts of the register allocation problem before instruction scheduling, then allowing instruction scheduling to occur. Register allocation then follows, plus a possible second round of instruction scheduling if the register allocator generated any instructions itself (spill code).

Before preparing for instruction scheduling and register allocation, the compiler lowers the program representation to the most efficient set of instructions. This is the last of the code-lowering phases.

We begin by modifying the flow graph so that each operation corresponds to an operation in the target machine. Since the instruction description was chosen to be close to a RISC processor, most instructions already correspond to target machine instructions. This step is usually called code generation; however, our view of code generation is more diffuse. We began code generation when we built the flow graph, we progressed further into code generation with each lowering of the flow graph, and we complete it now by

guaranteeing the correspondence between instructions in the flow graph and instructions in the target machine.

2.9 Instruction Scheduling

A modern RISC processor is implemented using what is called a *pipeline architecture*. This means that each operation is divided into multiple stages, with each stage taking one machine cycle to complete. Because each stage takes one cycle, a new instruction may start on each cycle, but it may not complete for some number of cycles after its initiation. Unfortunately, most techniques for code generation attempt to use a value as soon after its calculation is initiated as possible. This was the preferred technique on earlier machines because it limited the number of registers that were needed. However, this order slows down the execution on a RISC processor, since the value is not immediately available. The instruction scheduler reorders the instructions to initiate instructions earlier than their use so that the processor will not be delayed.

Recent RISC processors can start the initiation of several instructions simultaneously. These instructions must be independent and use different function units within the processor. The scheduler must form these groups of instructions, called *packets*. All instructions in a packet can be issued simultaneously.

The original instruction schedulers scheduled instructions within a single block, possibly taking into account the instructions that ended the preceding blocks. They did this by creating a data structure called the *instruction dependence graph*, which contained the operations as nodes and directed edges between two nodes if the first operation must be executed before the second operation. The edges were labeled with the number of machine cycles that must occur between the execution of the two instructions. The scheduler then performed a topological sort of the instruction dependence graph specialized to minimize the total number of cycles that the ordering of instructions required.

Scheduling limited to blocks does not use the multiple instruction-issue character of RISC processors effectively. Blocks are usually small, and each instruction within them depends on some other instructions in the block. Consider the problem of instruction scheduling as filling in a matrix, with the number of columns being the number of instructions that can be issued simultaneously and the number of rows being the number of machine cycles it takes to execute the block. Block scheduling will fill in this matrix sparsely: There will be many empty slots, indicating that the multiple-issue character of the machine is not being used. This is particularly a problem for load, store, multiply, divide, or floating point operations which take many cycles to execute. RISC processors usually implement other integer operations in one cycle. There are several techniques incorporated in the compiler for ameliorating this problem:

- *Unroll*: Earlier phases of the compiler have performed loop unrolling, which increases the size of blocks, giving the block scheduler more chance to schedule the instructions together.
- *Superblock*: When there is a point in a loop where two paths join, it is difficult to move instructions from after the join point to before it. When the succeeding block in the loop is short, the compiler has earlier made a copy of the block so that the joined path is replaced by two

blocks, joined only at the head of the loop. This transformation is applied at the same time that loop unrolling is performed.

- *Move*: The normal optimization techniques used for code motion attempt to keep temporaries live for as short a sequence of instructions as is possible. When scheduling, we will schedule each block separately. For blocks that are executed frequently, we will repeat the code motion algorithm, but allow the motion of instructions from one block to another even when there is no decrease in execution of the instruction.
- *Trace*: Consider the most frequently executed block, *B*, determined either by heuristics or profile information. Find the maximal path including *B* that involves the most frequently executed predecessors and successors of each block on the path. Now consider this path as if it were a block, with some modifications to the dependence graphs to ensure proper actions at condition branches. See if there are any instructions on this path that can be moved to earlier (or later) blocks.
- *Software, pipelining*: In the special case of a loop that is a single block, software pipelining can give a good schedule. Software pipelining uses dependence information provided by the dependence graph (not the instruction dependence graph) to overlap the schedules for one iteration of the loop with the following iterations. This does not decrease the length of time that each iteration takes (it may increase it), but allows the iterations to start more quickly, thereby decreasing the execution time of the whole loop. Blocks and loops that can be software pipelined are identified before other scheduling occurs and are handled separately.

During instruction scheduling, some peephole optimization occurs. It can happen during scheduling that instructions that were not adjacent have become adjacent, creating situations such as a store followed by an immediate load from the same location. It is therefore effective to apply some of the peephole optimizations again.

When instruction scheduling is completed, the order of instructions is fixed and cannot be changed without executing the instruction scheduler again. In that case, it may only be necessary to rerun the block scheduler.

We have shrunk the register requirements so the values in registers can fit in the physical registers at each point in the flow graph. Now we will reorder the instructions to satisfy the instruction-scheduling constraints of the target processor. We will assume a processor such as the Alpha 21164, which can issue four instructions on each clock cycle. Many of the integer instructions take one cycle to complete. Most floating point operations take four cycles to complete. In any given cycle one can issue one or two load instructions or a single store instruction. A store instruction cannot be issued in the same cycle as a load instruction. We will assume that the other integer operations can be filled in as necessary. Instructions such as integer multiply or floating point divide take a large number of cycles.

The problem is to group the instructions into one to four instruction packets such that all the instructions in a packet can be issued simultaneously. The compiler also reorders the instructions in an attempt not to use an operand until a number of cycles following the issue of the instruction that computes it to ensure that the value is available.

The load and store operations take a variable amount of time, depending on the load on the memory bus and whether the values are in caches. In the Alpha 21164, there are two caches on the processor chip, and most systems have a further large cache on the processor board. A load instruction takes two cycles for the cache nearest the processor, eight cycles in the next cache, twenty cycles in the board cache, and a long time if data is in memory. Furthermore, the processor contains hardware to optimize the loading of consecutive memory locations. If two load operations are each issued on two consecutive cycles to consecutive memory locations, the processor will optimize the use of the memory bus.

It is important that useless branches are at least not counted when determining scheduling. This is marked with an asterisk (*) in the cycle location.

There are hardware bypasses so that a compare instruction and a branch instruction can be issued in the same cycle. Note that the assignment to SI9 (in B1) can be moved forward eliminating an extra slot. Also note that B12 is only reached from the preceding block, so NOPs do not need to be inserted.

Now note that the inner loop starting with block B2 consists of three blocks. The first block is the conditional test and the third block updates the iterations. All but one of the computations from the third block can be moved to the first block (hoisting), while the remaining instructions can be scheduled more effectively by making a copy of the iteration block (super block scheduling).

Note that NOPS were inserted in the middle of the code. The machine picks up four instructions at a time, aligned on 16-byte boundaries. It must initiate all instructions in this packet of four instructions

before going on to the next packet. To execute the instructions in the smallest amount of time, we must maximize the number of independent instructions in each packet. The resulting scheduled instructions are shown in Figure 2.23.

```
Number     Instruction                    Comment
0    B0:   PROLOG
1          iLDC     1          => T5      /(0) I = 1
2          iSLD     (T2)       => T8      /(0) fetch value of N
3          iBLE     T8,B5                 /(1) no iterations if N <= 0
4          NOP                            /(1)

5    B1:   iLDC     1          => T14     /(0) LARGE(I) = 1
6          dSLD     (T1)       => SF2     /(0) value A(1,I)
7          CPYS     SF2        => SF1     /(1) DABS(A(1,I))
8          iLDC     2          => T6      /(1)
9          iCMPLT   T8,#2      => T34     /(2) is 2 > N
10         iBCOND   T34,B4                /(2) yes - no iterations

11   B12:  iADD     T1,#8      => T28     /(0) address(A(2,I)) = address(A(1,I)) + 8
12         NOP                            /(0) to line up loop entry

13   B2:   dSLD     (T28)      => SF4     /(0) value A(J,I)
14         iADD     T28,#8     => T28     /(0) schedule address update early
15         iCMPLE   T6,T8      => T24     /(1) is J > N
16         iADD     T6,#1      => T6      /(1) J + 1
17         CPYS     SF4        => SF4     /(3) DABS(A(J,I))
18         dCMPGT   SF4,SF1    => SF6     /(7) comparison
19         dBCOND   SF6, B6               /(11) skip update of variables
20         iBCOND   T24, B2               /(12)

21   B4:   iSST     (T3),T14               /(0) store the value into LARGE(I)
22         iADD     T3,#4      => T3      /(0) increment address(LARGE(I))
23         dSST     (T4),SF1               /(1) store the value into VALUE(I)
24         iADD     T4,#8      => T4      /(1) increment address(VALUE(I))
25         iADD     T5,#1      => T5      /(2) I + 1
26         S8ADDQ   T8,T1      => T1      /(2) increment address(A(1,N))
27         iCMPLE   T5,T8      => T9      /(3) compare fetch(I) ? fetch(N)
28         iBCOND   T9,B1                 /(3) more iterations to perform

29   B5:   EPILOG                         /(0) return from subroutine

30   B6:   d2d      SF4        => SF1     /(0) Infrequently executed code moved
31         iSUB     T6,#1      => T14     /(0) put in register for LARGE(I)
32         iBCOND   T24,B2                /(1)
33         BR       B4                    /(1)
```

Figure 2.23 Scheduled Instructions

2.10 Register Allocation

The register allocation phase modifies the program flow graph by replacing temporaries with physical registers. There are categories of techniques for performing register allocation on the complete

procedure. One is based on graph-coloring algorithms. A graph is formed with each temporary being a node. An undirected edge exists between two nodes if they cannot occupy the same physical register. Register allocation reduces to coloring this graph, where each color represents a different physical register.

The alternative method for register allocation is based on bin packing, where there is a bin for each physical register. Two temporaries can be allocated to the same bin if there is no point in the program where both need to have a value.

Each of these techniques has advantages and disadvantages. The graph-coloring technique is superior when considering conditional branching. Since the bin-packing algorithms typically approximate the set of points where a temporary holds a value by some data structure where it is easy to take intersections of the sets, bin packing does not perform as well as graph coloring with branching.

Bin packing performs better than graph coloring when straight-line code is considered. Since bin packing can traverse the blocks as it performs assignment, it can determine when the same register can be reused immediately. It can also use information about the operations in the program and their order to decide which temporaries to store to memory when too many registers are needed (this can happen even though the LIMIT phase has been executed). Graph coloring has no concept of locality of reference.

This compiler's register allocator combines the two techniques. Because LIMIT has been run, little register spilling will occur. Graph coloring is therefore used to assign registers to temporaries that hold values at the beginning of some block, in other words, in those situations in which graph coloring performs best. A modification of bin packing suggested by Hendron (1993) will be used to schedule temporaries within each block.

Previous attempts at splitting the temporaries that are live at the beginning of blocks (global allocation) from those that are live within a block (local allocation) have encountered difficulties because performing either global or local allocation before the other could affect the quality of register allocation. This problem is resolved by the existence of the LIMIT phase, which has performed spilling of global temporaries before either allocation occurs.

Note that the presence of LIMIT has eliminated most register spilling during register allocation. It does not eliminate all of it. There can be secondary effects of conditional branching that can cause register spilling during either graph coloring or bin packing. This situation is unavoidable, since optimal register allocation is NP-complete. In the situations in which spilling occurs, the register allocator will insert the required store and load operations.

Now we apply register allocation to the example. First the compiler must recompute the points where temporaries are live, because instruction scheduling has changed these points (see Figure 2.24). Note that the scheduler has introduced a redefinition of a local register, so we need to either do superblock scheduling earlier (when we don't know that it will pay off) or redo right number of names, or locally redo right number of names when we create these problems. We only deal with the integer registers here; the floating point registers in this case are simple because they all interfere and so one assigns each to a different register.

After the lifetime information for temporaries has been computed, the compiler uses a graph-coloring algorithm to allocate the registers that are live at the beginning of some block, or registers which are directly assigned to a physical register. The ones assigned to a physical register are preallocated; however, they must be considered here to avoid any accidental assignments. The physical registers will be named using $0, $1, and so on. Note that the temporaries corresponding to formal parameters are assigned to physical registers specified by the calling standard for the target machine. The globally assigned registers are listed in Figure 2.25, together with the kind of register. In this case all of the registers needed are called *scratch registers*, which means that the value in the register need not be saved and restored if the register is used in the procedure.

```
T1    [1,28],[30,33]   Global
T2    [1,2)            Global
T3    [1,28],[30,33]   Global
T4    [1,28],[30,33]   Global
T5    (1,28],[30,33]   Global
T6    (8,20],[30,33]   Global
T8    (2,28],[30,33]   Global
T9    (27,28)          Local
T14   (5,21),(31,33]   Global
T24   (15,20),[30,32)  Global
T28   (11,20],[30,33]  Global
T34   (9,10)           Local
SF1   (7,23),(30,33]   Global
SF2   (6,7)            Local
SF4   (13,18),[30,30)  Global
SF6   (18,19)          Local
```

Figure 2.24 Live Ranges after Scheduling

```
T1              $16   Parameter
T2              $17   Parameter
T3              $18   Parameter
T4              $19   Parameter
T5              $22   Scratch Register
T6              $23   Scratch Register
T8              $24   Scratch Register
T14             $25   Scratch Register
T24             $20   Scratch Register
T28             $21   Scratch Register
SF1             $f10  Floating Scratch
SF4             $f11  Floating Scratch
Stack Pointer   $30   Stack Pointer
Return Address  $27   Return Address
```

Figure 2.25 Global Register Assignments

After that the registers that are live at the beginning of any block have been allocated, we can allocate the symbolic registers that are live only within a single block. In this small example there are only a few. In realistic programs, these registers greatly outnumber the globally live registers. These local registers are listed in Figure 2.26. A register is reused if at all possible because the compiler wants to minimize the number of registers used. This avoids the necessity of using a register that is not a scratch register and would thus require that a store operation be inserted at the beginning of the procedure to save its value and a load inserted at the exit to restore the value.

The resulting assembly code is shown in Figure 2.27. The temporaries have all been replaced by registers. There were no spill instructions inserted, so the instruction schedules have not changed.

```
T9              $23             Reuse of Register
T34             $21             New Register
SF2             $f10            Reuse of Register
SF6             $f12            New Register
```

Figure 2.26 Local Register Assignments

```
Number     Instruction                    Comment
0    B0:   PROLOG
1          iLDC     1          => $22    /(0) I = 1
2          iSLD     ($17)      => $17    /(0) fetch value of N
3          iBLE     $17,B5               /(1) no iterations if N <= 0
4          NOP                           /(1)

5    B1:   iLDC     1          => $24    /(0) LARGE(I) = 1
6          dSLD     ($16)      => $f10   /(0) value A(1,I)
7          CPYS     $f10       => $f10   /(1) DABS(A(1,I))
8          iLDC     2          => $23    /(1)
9          iCMPLT   $17,#2     => $21    /(2) is 2 > N
10         iBCOND   $21,B4               /(2) yes - no iterations

11   B12:  iADD     $16,#8     => $20    /(0) address(A(2,I)) = address(A(1,I)) + 8
12         NOP                           /(0) to line up loop entry

13   B2:   dSLD     ($20)      => $f11   /(0) value A(J,I)
14         iADD     $20,#8     => $20    /(0) schedule address update early
15         iCMPLE   $23,$17    => $25    /(1) is J > N
16         iADD     $23,#1     => $23    /(1) J + 1
17         CPYS     $f11       => $f11   /(3) DABS(A(J,I))
18         dCMPGT   $f11,$f10  => $f12   /(7) comparison
19         dBCOND   $f12, B6             /(11) skip update of variables
20         iBCOND   $25, B2              /(12)

21   B4:   iSST     ($18),$24            /(0) store the value into LARGE(I)
22         iADD     $18,#4     => $18    /(0) increment address(LARGE(I))
23         dSST     ($19),$f10           /(1) store the value into VALUE(I)
24         iADD     $19,#8     => $19    /(1) increment address(VALUE(I))
25         iADD     $22,#1     => $22    /(2) I + 1
26         S8ADDQ   $17,$16    => $16    /(2) increment address(A(1,N))
27         iCMPLE   $22,$17    => $23    /(3) compare fetch(I) ? fetch(N)
28         iBCOND   $23,B1               /(3) more iterations to perform

29   B5:   EPILOG                        /(0) return from subroutine

30   B6:   d2d      $f11       => $f10   /(0) Infrequently executed code moved
31         iSUB     $23,#1     => $24    /(0) put in register for LARGE(I)
32         iBCOND   $25,B2               /(1)
33         BR       B4                   /(1)
```

Figure 2.27 Code after Register Allocation

2.11 Rescheduling

The next phase is a rescheduling phase, which is only executed if the register allocator has changed the set of instructions that are executed. This can happen due to either a peephole optimization or the introduction of spill code. Neither of these occurred in this case, so the rescheduling operation is ignored.

If the register allocator generated any instructions, that is, register spilling occurred, then the instruction scheduler is executed again, but in this case only on blocks where load or store operations have been inserted.

2.12 Forming the Object Module

At last, we near the completion of our task. The instructions have been chosen; the registers have been chosen. All that remains is the clerical task of translating this information and the information about globally allocated data into an object module. This task includes the insertion of debugging information for the debugger. Since our task has been long, I am making light of this last phase. It involves little intricate technology. However, it is complex because the structures of object modules are complex and undocumented. Every document that I have seen describing object module form has serious errors. So this project involves experimental computer science—trying to determine what the linker is expecting. This phase will also generate the assembly language listing for the listing file, if it is requested.

2.13 References

Allen, R., and K. Kennedy. “Advanced compilation for vector and parallel computers.” San Mateo, CA: Morgan Kaufmann.

Frazer, C. W., and D. R. Hanson. 1995. *A retargetable C compiler: Design and implementation*. Redwood City, CA: Benjamin/Cummings.

Hendron, L. J., G. R. Gao, E. Altman, and C. Mukerji. 1993. A register allocation framework based on hierarchical cyclic interval graphs. (Technical report.) McGill University.

Hendron, L. J., G. R. Gao, E. Altman, and C. Mukerji. 1993. Register allocation using cyclic interval graphs: A new approach to an old problem. (Technical report.) McGill University.

Morel, E., and C. Renvoise. 1979. Global optimization by suppression of partial redundancies. *Communications of the ACM* 22(2): 96-103.

Wolfe, M. 1996. *High performance compilers for parallel computing*. Reading, MA: Addison-Wesley.

Chapter 3
Graphs

A prime prerequisite for being a compiler writer is being a "data structure junkie." One must live, breathe, and love data structures, so we will not provide the usual complete list of all background mathematics that usually appears in a compiler book. We assume that you have access to any one of a number of data structure or introductory compiler writing books, such as Lorho (1984) or Fischer and LeBlanc (1988). This design assumes that you are familiar with the following topics, which are addressed by each of the data structure books referenced.

- *Equivalence relations and partitions*. The compiler frequently computes equivalence relations or partitions sets. An equivalence relation is frequently represented as a partition: All of the elements that are mutually equivalent are grouped together into a set of elements. Hence the whole set can be represented by a set of disjoint sets of elements. Partitions are frequently implemented as UNION/FIND data structures. This approach was pioneered by Tarjan (1975).
- *Partial ordering relations on sets*. A compiler contains a number of explicit and implicit partial orderings. Operands must be computed before the expression for which they are an operand, for example. The compiler must be able to represent these relations.

The topics that are addressed in this chapter concern graphs. A number of the data structures within a compiler—the flow graph and the call graph, for instance—are represented as directed graphs. Undirected graphs are used to represent the interference relationship for register allocation. Thus these topics are addressed here to the extent that the theory is used in implementing the compiler. The topics addressed are as follows:

Data structures for implementing directed and undirected graphs
Depth-first search and the classification of edges in a directed graph
Dominators, postdominators, and dominance frontiers
Computing loops in a graph
Representing sets

3.1 Directed Graphs

A directed graph consists of a set of nodes *N* and a set of edges *E*. Each edge has a node that is its tail, and a node that is its head. Some books define an edge to be an ordered pair of nodes—tail and head; however, this makes the description of the compiler more difficult. It is possible to have two edges with the same tail and head. In a flow graph containing a C **switch** statement or a Pascal **case** statement, two different alternatives that have the same statement bodies will create two edges having identical tails and heads.