ɔ:ɲɪʈ̃:

A SURVEY OF COMPILER OPTIMIZATION TECHNIQUES

Paul B. Schneck

Goddard Institute ~~of~~ for Space Studies

This survey describes the major optimization techniques of compilers and groups them into three categories: machine dependent, architecture dependent, and architecture independent. Machine-dependent optimizations tend to be local and are performed upon short spans of generated code by using particular properties of an instruction set to reduce the time or space required by a program. Architecture-dependent optimizations are global and are performed while generating code. These optimizations consider the structure of a computer, but not its detailed instruction set. Architecture-independent optimizations are also global but are based on analysis of the program flow graph and the dependencies among statements of source program. The paper also presents a conceptual review of a universal optimizer that performs architecture-independent optimizations at source-code level.

KEYWORDS AND PHRASES: compiler, optimization, flow graph, connectivity, interval analysis, common subexpression, architecture independent, global flow analysis, code motion, constant propagation, dead variable elimination, tree height minimization, parallel processor, vector processor.

CR CATEGORIES: 4.12, 4.22

INTRODUCTION

Most computer systems support a multiplicity of programming languages; for a particular language, the translators or compilers often exist in three versions. The first version is a small, fast compiler, which is for program development and has extensive diagnostics and debugging aids. The second version is a reentrant conversational compiler, which is used for online development of programs and has comprehensive editing facilities. The third compiler is the optimizing compiler, which is used for translating production programs into efficient object code and is larger and slower than the others. In this paper, we examine the techniques employed in optimizing compilers and make some quantitative comparisons between the programs of the optimizing compilers and other compilers. A large number of the examples and references in the paper are FORTRAN-related, because FORTRAN is the most widely used production programming language.

The history of optimizing compilers dates back at least as far as FORTRAN I (1). At that time, most programming was done in machine language, and a compiler that offered convenience at the expense of machine time would not have been acceptable. The following quotation from an International Business Machines Corporation (IBM) specification reveals that the convenience of the new language was believed insufficient to cause its widespread acceptance.

> ...FORTRAN may apply complex, lengthy techniques in coding a problem which the human coder would have neither the time nor inclination to derive or apply. Thus, in many cases, FORTRAN may actually produce a better program than the normal human coder would be apt to produce. (1)

Even in that very first FORTRAN compiler, 25 percent of the instructions were for optimization.

OPTIMIZATION TECHNIQUES

The following three sections describe various optimization techniques that have been used in compilers or have been suggested for compilers. Very little has been done to classify optimizations; they are grouped here by function.

Compiler optimization techniques operate on three levels: machine dependent, architecture dependent, and architecture independent. *Machine dependent* is used to describe the instruction-level sensitivities of a compiler. *Architecture dependent* denotes those parts of a program that relate to the general hardware implementation, but not to a specific machine. *Architecture independent* (used in lieu of the more familiar phrase—machine independent) indicates those aspects of program formulation that do not depend on a particular computer system or even on a type of implementation (e.g. pipeline processing). Optimizations originating in the academic and scientific community tend to be global, while, until recently, manufacturers have concentrated on local and machine-dependent techniques.

Machine-Dependent Optimization

One of the earliest references on compilation techniques concerns the Project for the Advancement of Coding Techniques (PACT), an experimental compiler. The target machine was the IBM 701, and the PACT compiler, described by Miller and Oldfield (2), produced code sensitive to the register-placement curiosities of that machine. No formal techniques were employed; rather, a set of rules was coded in tabular form to control code generation. To a large extent, this same technique is applicable today for machine-dependent code optimization. The FORTRAN I compiler contained a sophisticated arithmetic translator by Sheridan (3) that performed association and commutation to take advantage of the AC/MQ relationship on the IBM 704. For example, a string of multiplications and divisions was reordered to minimize the number of register transfers (*exchanges*) that had to be performed.

McKeeman (4) proposes a postprocessing technique for optimization, which can be considered as a window traversing the sequence of generated (unoptimized) code. If the instructions visible in the window match one of a number of patterns, the code is transformed. In this manner redundant stores, multiplications by two, and register transfers can easily be optimized. Bagwell (5) describes a set of clever coding tricks (special cases) that may be implemented for almost any machine. Although performed during code generation, this is essentially McKeeman's approach. These machine-dependent optimizations are the most descriptive of available techniques.

Architecture-Dependent Optimization

Three optimization techniques are classified here as architecture dependent. These techniques are used for machines

having one or more of the following general characteristics:

1. The computer has *n* accumulators.
2. The computer can execute several independent instructions in parallel.
3. The computer executes arithmetic and logical instructions upon multiple data streams.

The evolution of computer architecture has followed a path from the single-accumulator IBM 704 to the multiple-accumulator CDC 6600, which is capable of executing several instructions in parallel, to the ILLIAC IV, which operates on up to 64 data items simultaneously. Optimization techniques have had a parallel evolution.

The n Accumulator Computer

Straightforward code generation of expressions involving noncommutative operations poses a special difficulty for a one-accumulator computer. In an expression such as $(a+b)/(c-d)$, the denominator should be computed first to be available for division when the numerator is computed and is in the accumulator. Anderson (6) discusses a technique that implements this procedure and eliminates the need to store and recover the values of many subexpressions. Anderson's technique for a one-accumulator computer looks ahead and delays code generation for the left-side expression of a noncommutative operator until code generation for the right side occurs. One a multiple-accumulator machine, the technique is also valuable because it decreases the number of registers required to evaluate an expression.

Nakata (7) extends this procedure to handle *n* accumulators. The procedure is enhanced by the fact that some heuristic observations are included to make the output similar to ordinary coding practices. The programming problem of using a minimum number of accumulators is equivalent to a graph-theoretic tree transformation proposed by Redziejowski (8). He proposes an algorithm for performing the tree transformation and proves it equivalent to that of Nakata. A study by Schneider (9) of the properties of tree-structure representations of arithmetic expressions yields the number of required registers: For k nested parenthetical subexpressions with *n* operator precedence levels $(k+1)n+1$ registers are required.

Finkelstein (10) describes a technique, *deferred store*, which eliminates much of the unnecessary storing and loading of partial results within loops on multiple-register machines. (*Register* is used here to indicate either an accumulator or index register.) When an assignment statement is executed, the accumulator is not actually stored in the result variable. Instead, other registers replace those containing data for deferred stores. If a result variable is to be modified before the deferred store has been performed, the value of the variable is in place and need not be fetched. The following example indicates a common situation where the deferred store saves a significant amount of time:

$$\sum_{i=1}^{n} a_i$$

```
      DO  1  I = 1,N
    1 SUM = SUM + A(I)
```

A special case of the *n* accumulator machine is one where the accumulator is the top element of a pushdown stack. The Burroughs 5000 and English Electric KDF9 are examples of such a machine. Randell and Russell (11) describe a one-pass procedure for translation of arithmetic expressions into a Reverse-Polish form suitable for a stack machine. An interesting point is their architecture-independent optimization that calculates a constant during compilation when both operands are constants. Generalizations of this technique are discussed in the next section.

Parallel-Instruction Execution

The CDC 6600 computer was the first commercially available machine to overlap the execution of several instructions. This class of computer, capable of parallel-instruction execution, has independent functional units that operate simultaneously. The programmer (or compiler) need not be explicitly aware of the parallel-execution capability; it will be used when possible. However, if the instructions are ordered to maximize parallel execution, a performance advantage of up to a factor of three can be obtained. Allard, Wolf, and Zemlin (12) describe the parallel capabilities of the CDC 6600 and briefly mention the speed advantage gained by reordering instructions. Thorlin (13) describes the technique used in CDC FORTRAN, which is based on a PERT-like analysis of dependency and timing for ordering CDC 6600 instructions. The machine's independent-functional units are kept busy by placing unrelated instructions together and sequencing the longest activities first. A similar instruction-scheduling technique was implemented by Blum, et al. (14) in the IBM FORTRAN H compiler. It constructs a dependency array that defines the area within which each instruction may be moved. A weight is assigned to each instruction by adding a base weight (a function of the instruction time) and the weight of every instruction which is dependent on it; instructions are then ordered by decreasing weight.

However, the improvement attainable by instruction reordering is limited by the parallelism inherent in the original instruction sequence. Stone (15) summarizes techniques that may be used to translate arithmetic expressions to achieve a high degree of inherent parallelism. His process corresponds to a tree structure with minimal height. This is the opposite result from that of the *n* accumulator, where minimizing the number of registers increased the tree height. Figure 1 shows two different trees for evaluating an expression. The first tree employs only one register for evaluating the expression; the second tree results in minimum evaluation time on a machine with at least four simultaneous multipliers. Because of the effects of data store/load instructions, the second tree may not result in minimum time on machines with fewer multipliers. Figure 2 shows two different sets of instructions for evaluating the expression corresponding to the two trees, which result in serial and parallel execution.

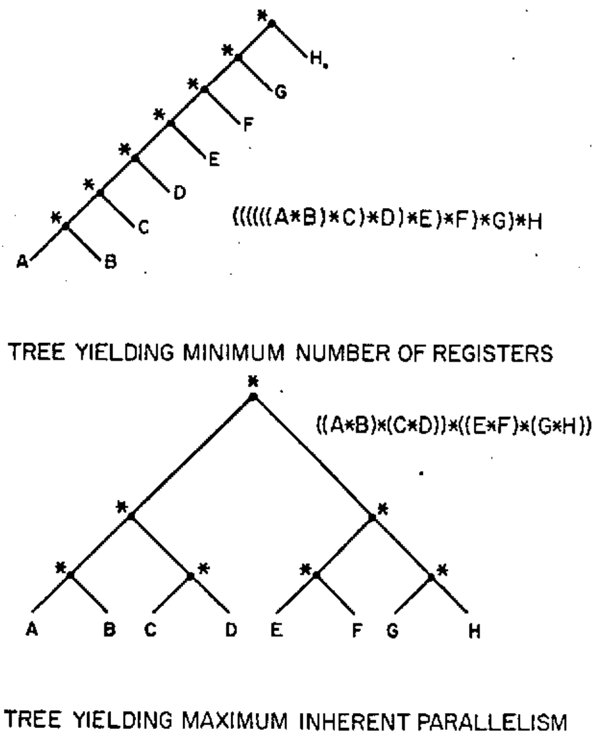

((((((A*B)*C)*D)*E)*F)*G)*H
A
B
C
D
E
F
G
H
a) TREE YIELDING MINIMUM NUMBER OF REGISTERS
((A*B)*(C*D))*((E*F)*(G*H))
A B C D E F G H
b) TREE YIELDING MAXIMUM INHERENT PARALLELISM

Figure 1. Tree Structure for Serial and Parallel Computation of an Expression.

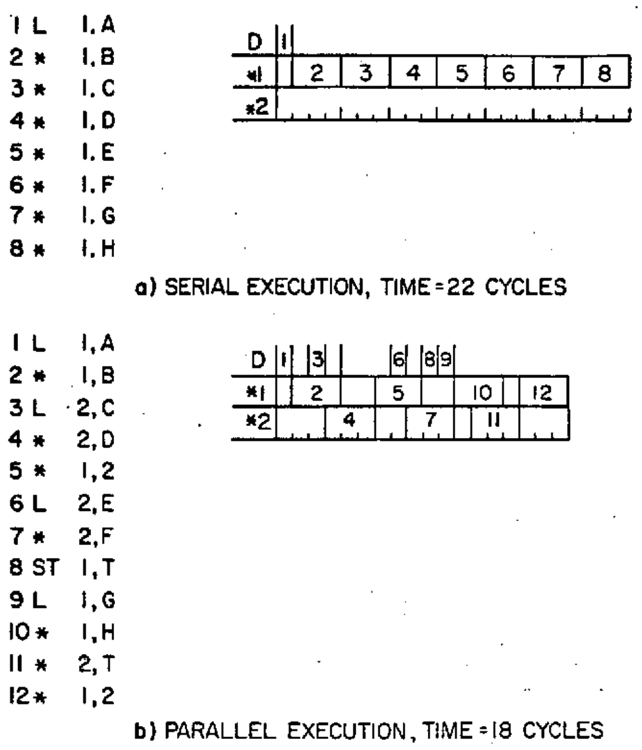


Figure 2. Resonances Used in Serial and Parallel Computation of an Expression

Han (16) examines the general problem of minimizing the tree height of a set of expressions. His procedure for determining an expression's minimum tree height can be used by a compiler to measure the degree of parallelism obtainable in a program. Ramamoorthy and Gonzalez (17) describe a method for attaining the maximum amount of parallel execution on a machine with a fixed number of processing units. Their method orders subexpressions so that some expressions can be delayed if insufficient processing units are available to perform all computations in parallel.

Multiple Data Streams

The preceding section discussed some techniques relevant to computers whose architecture permitted parallel execution of instructions while maintaining the standard instruction set. This section discusses the optimization techniques applicable to computers where the instruction set reflects the computer's capacity to perform a single instruction on many data items. Two computers are in this category: the CDC STAR and the Burroughs ILLIAC IV. The STAR, a pipeline computer, processes operands sequentially, but with a high degree of overlap. The ILLIAC, a parallel computer, processes 64 operands simultaneously. The instruction sets of the two machines are remarkably similar, and high-level language programs must be designed from the same viewpoint for both machines.

For programs written in a procedural language, Burkhardt (18) describes some occurrences of inherent parallelism. He points out that parallelism may occur from the arithmetic-expression level, through independent iterations of a loop, to parallel-task execution within an operating system environment. Millstein (19), reporting on the design of a FORTRAN compiler for the ILLIAC IV, discusses a compiler that will detect parallelism in the use of subscripted variables in DO loops. This three-step procedure first determines data dependencies and then, if there are more dependencies between loop iterations, examines flow within the loop and determines an expression ordering. The first two steps are analyzed by graph-theoretic techniques; the last by ad hoc methods. A later report by Lamport and Presberg (20) gives a detailed description of the algorithms and techniques used to permit parallel execution of DO loops.

Schneck (21) developed a simplified algorithm for the detection of parallelism in standard FORTRAN programs and defined the concept of *feedback* which prevents parallel execution. Testing for feedback involves a flow analysis of the source program and a search for subscript forms that cause feedback. In the absence of feedback, statements are rewritten to indicate parallel execution. Additionally, scalar variables that might bar parallel execution are expanded to vectors. Thus, the following statement may be performed entirely in parallel.

```
      DO    1   I = 1,20
            A = (B(I) + B(I+1)) * .5
            C(I) = C(I) + A
1           D(I) = D(I) - A
```

Kuck, Muraoka, and Chen (22) performed an analysis similar to Schneck's. Their orientation was to define a machine architecture to process *ordinary* programs. They conclude that, even for simple programs, a multiple-processor organization, consisting of 16 processors, is of value.

Architecture-Independent Optimization

Architecture-independent optimization techniques are global in nature; they perform a flow analysis on the source program to obtain necessary information. This section summarizes major architecture-independent optimizations, which are illustrated in Figure 3. The most widely applied optimization is *common subexpression elimination*. When a calculation is performed, a search is made to determine if the calculation was performed previously and need not be repeated; if so, the prior result replaces the calculation. *Dead variable elimination* removes statements that assign values to unused variables in the program. These unused variables most frequently result from program modifications, but may also be due to common subexpression elimination. *Code motion* refers to the rearrangement of expressions permitting a calculation to occur in a low-frequency program segment and to be available for use in a high-frequency segment. Finally, *constant propagation* removes calculations containing only known constants from the program and performs them in the compiler. This is certainly code motion to a low-frequency segment.

Frequency Analysis

The original FORTRAN compiler (23) contained an optimizer that gathered information on the source program's structure. The source program was analyzed and broken down into a set of basic blocks,[1] and a table listing the predecessors of each basic block was created. This table was then used in a Monte Carlo simulation to find the relative frequency execution of each basic block. A random number generator, augmented by programmer estimates supplied in FREQUENCY statements,[2] was used to traverse paths in the program flow graphs, and a count was kept for each basic block. Figure 4 shows a program flow graph, which is used as an example throughout this section, and Table 1 shows the relative frequencies obtained by simulation.

Next, the source program blocks were optimized in order from highest to lowest frequency. The target computer, an IBM 704, had only three index registers, and much of the compiler's optimization centered on assigning them efficiently in the most frequently executed blocks of the program. Code generation was performed for a machine assumed to have as many

[1] A basic block is the fundamental program flow unit; it is a segment of code with only one entry and one exit point.

[2] According to John Cocke, the FREQUENCY statement was removed from the language after it was discovered to have been incorrectly implemented (frequencies were being computed inversely) without having encountered any user reaction.

```
     A =B+C
     Y =Y+1.
     Z =C
     Q =(Z+B)*SIN (.7854)
     DO 1 I=1,100
1    P(I)= P(I)*(A+B)
```

sample program

```
     A =B+C
           ←———a
           ←———b
     Q =A*.7071  ←——c,d.
     T00001 =A+B  ←——e
     DO 1 I=1,100
1    P(I)=P(I)*T00001
```

optimized program

a. ELIMINATION OF DEAD VARIABLE

b. ELIMINATION OF DEAD VARIABLE, CAUSED BY c

c. COMMON SUBEXPRESSION ELIMINATION

d. CONSTANT PROPAGATION

e. CODE MOTION

Figure 3. Architecture-Independent Optimization

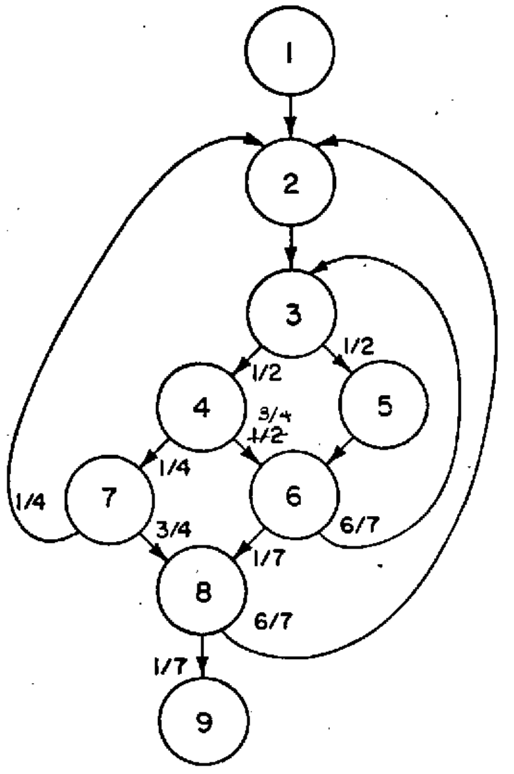


Figure 4. Program Flow Graph with Branch Probabilities

index registers as required. Then, the optimizer efficiently assigned the 704's three index registers within the highest frequency block of the program, while interpolating instructions to save and restore index register values when necessary. As other blocks were processed, index register assignments were made to match those of *adjacent* (immediate predecessor or successor) higher frequency blocks. When no adjacent blocks had already been processed, no matching of index registers was necessary. When just one adjacent block had already been processed, it was necessary only to choose a matching permutation of the index registers. If two or more adjacent blocks had already been processed, the possibility of matching all index registers was uncertain. Therefore, it became necessary to add instructions for loading index registers with the values required by adjacent blocks. The progression of processing from high-frequency to low-frequency blocks caused the added instructions to be located in the later low-frequency blocks.

Table 1. Relative Frequencies Obtained by Simulation

BLOCK	RELATIVE FREQUENCY
1	1.0
2	8.2
3	33.0
4	16.5
5	16.5
6	28.9
7	4.1
8	7.1
9	1.0

In another study, Horwitz (24) describes a graph-theoretic procedure for index-register allocation: An optimal index-register allocation may be obtained for straight-line (loop-free) programs. Horwitz's algorithm is a practical procedure for carrying out the highly combinatorial assignment process similar to that employed by FORTRAN I. Also, Luccio (25) provides a further reduction of the enumeration required for an optimal allocation. The program graph is partitioned, and the allocation problem may be solved separately for each subgraph and then combined.

Day (26) discusses an alternate linear-programming approach for assignment of registers. He demonstrates an optimal algorithm and gives two others which provide good approximations, but are fast enough for use in a compiler.

Matrix Analysis

Prosser (27) describes a Boolean-matrix approach to flowgraph analysis that avoids the lengthy Monte Carlo techniques used in FORTRAN I. The predecessor information obtained by analysis of the program is used to construct a *connection* matrix (Table 2). The connection matrix C has a *1* at C_{ij} if, and only if, program block j is a direct successor of program block i. By repeated matrix multiplication, the connection matrix may be used to determine the sets of blocks participating in loops. Prosser also introduces the *dominance* relation to indicate that a particular block (dominator) must be traversed before another block (dominee) can be reached. This construct is extremely valuable. When a calculation is moved out of a block, it must be moved to a dominator block to assure that it will be performed. These matrix operations yield valuable, if lengthy, methods for obtaining program-flow information. Warshall (28) describes a simplification of the multiplication of $n \times n$ Boolean matrices that reduces the time required from $O(n^3)$ to $O(n^2)$, which makes matrix techniques practical.

Table 2. Boolean Connection Matrix C

		SUCCESSOR BLOCK								
		1	2	3	4	5	6	7	8	9
PREDECESSOR BLOCK	1	0	1	0	0	0	0	0	0	0
	2	0	0	1	0	0	0	0	0	0
	3	0	0	0	1	1	0	0	0	0
	4	0	0	0	0	0	1	1	0	0
	5	0	0	0	0	0	1	0	0	0
	6	0	0	1	0	0	0	0	1	0
	7	1	0	0	0	0	0	0	1	0
	8	1	0	0	0	0	0	0	0	1
	9	0	0	0	0	0	0	0	0	0

For some general results on what may be obtained from the connectivity matrix of a program flow graph, Ramamoorthy (29) presents algorithms that identify unessential nodes, enumerate the maximum *strongly connected regions* (i.e. loops), and partition the flow graph into disjoint subgraphs. These matrix manipulations are basic to obtain the information required for program optimization. Unessential nodes may be discarded from a program because they will never be executed. The identification of the maximum strongly connected regions permits locating relative-constant expressions and moving them to lower frequency regions, as well as indicating on which blocks the optimization process should concentrate. Partitioning the flow graph into disjoint subgraphs permits working with smaller units at a time, which results in a significant decrease in the combinatorial efforts expended in optimization.

In a FORTRAN optimizer, Allen (30) uses matrix methods for the analysis of a program's flow graph. The connection matrix is used to obtain a set of strongly connected regions which the optimizer processes from the inside out. Within a basic block, redundant expressions are evaluated only once and then eliminated. Constant propagation is performed, and expressions are replaced by their computed values. Within a loop, invariant instructions are moved out, strength reduction is performed, and tests are simplified. Unused definitions and computations are eliminated where the flow information indicates this is possible. The procedures to effect these optimizations take advantage of the bit-parallel operations found in most machines and perform logical operations, a word at a time. Allen's article is extremely comprehensive and shows the details involved in applying each of the optimizing techniques.

Following Allen's optimization techniques, Kleir and Ramamoorthy (31) describe optimization procedures for microprograms, which may be viewed simply as another source language requiring translation to machine language. The connectivity matrix is used to find strongly connected regions which are processed innermost to outermost, and code motion is performed to decrease instruction execution frequency. Within a basic block, common-subexpression elimination and dead-variable elimination is performed (referred to by the authors as redundant actions and negated actions, respectively).

In his book, Gries (32) devotes an entire chapter to a discussion of code optimization techniques. He views optimization at three levels: within a basic block, within a loop, and globally. The global optimization techniques are patterned after Allen.

In FORTRAN I, subscript calculations were performed at any definition of a variable used in the subscript, which led to inefficient codes when many definitions occurred with few uses. Most other compilers simply recomputed a subscript for each use or kept track of local multiple uses of a subscript (e.g., within an assignment statement). In an article, Ryan (33) considers the problem of determining where a common subscript (or any common expression) may be computed with minimum frequency to be available when required. Ryan's algorithm may be used with a multipass compiler and permits locating computations within a lower frequency region at a distance from the point of use.

In the 1960s, the advent of new hardware brought a new class of compilers and optimization techniques. In an IBM technical report, Medlock and Lowry (34) describe the optimization techniques that are the foundation of the IBM FORTRAN H compiler. These techniques extend the dominance relationship introduced by Prosser. In addition, a new defining relationship makes it possible to replace the dominance array with four vectors and reduce the space required. Frequency information is obtained by inverting the probability connection matrix p, where p_{ij} indicates the probability that program block j will succeed block i. Table 3 shows the probability connection matrix for the example flowchart and its inverse. In Table 4, row one indicates frequency relative to block one.

Table 3. Probability Connection Matrix P

P		SUCCESSOR BLOCK								
		1	2	3	4	5	6	7	8	9
PREDECESSOR BLOCK	1	0	1	0	0	0	0	0	0	0
	2	0	0	1	0	0	0	0	0	0
	3	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0
	4	0	0	0	0	0	$\frac{3}{4}$	$\frac{1}{4}$	0	0
	5	0	0	0	0	0	1	0	0	0
	6	0	0	$\frac{6}{7}$	0	0	0	0	$\frac{1}{7}$	0
	7	$\frac{1}{4}$	0	0	0	0	0	0	$\frac{3}{4}$	0
	8	$\frac{6}{7}$	0	0	0	0	0	0	0	$\frac{1}{7}$
	9	0	0	0	0	0	0	0	0	0

Table 4. Frequency Matrix

$(I-P)^{-1}$		SUCCESSOR BLOCK								
		1	2	3	4	5	6	7	8	9
PREDECESSOR BLOCK	1	0.00	8.00	32.00	16.00	16.00	28.00	4.00	7.00	1.00
	2	0.00	7.00	32.00	16.00	16.00	28.00	4.00	7.00	1.00
	3	0.00	7.00	31.00	16.00	16.00	28.00	4.00	7.00	1.00
	4	0.00	7.02	30.65	15.33	15.33	27.57	4.08	7.00	1.00
	5	0.00	6.98	31.35	15.67	15.67	28.43	3.92	7.00	1.00
	6	0.00	6.98	31.35	15.67	15.57	27.43	3.92	7.00	1.00
	7	0.00	7.14	28.57	14.29	14.29	25.00	3.57	7.00	1.00
	8	0.00	6.86	27.43	13.71	13.71	24.00	3.43	6.00	1.00
	9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Determination of relative frequencies again permits ordering the processing of blocks from highest to lowest frequency. Within that order, common subexpressions may be eliminated and expressions may be moved from high- to low-frequency blocks. The dominator relationships are used to ensure availability of an expression when needed. When profitable, strength reduction will be performed with initialization instructions placed in a dominator block. Subsumption, or substituting one variable for another if they are equal at each reference, minimizes the number of simple replacement operations in a program. In another report, Lowry and Medlock (35) describe the FORTRAN H production compiler, discuss the compiler's implementation, and suggest several additional optimizations. An interesting point about the compiler is that it is written primarily in FORTRAN. First run on the IBM 7094, the compiler was used to create a new version of itself for the IBM 360. When the optimizer had been tested, it was used to translate the compiler, which resulted in a 25 percent decrease in size and a 35 percent decrease in compilation time. To achieve a reasonable processing speed, the compiler uses bit-vectors which can be processed by the bit-parallel logical instructions available on the IBM 360. Lowry and Medlock also discuss register assignment and code generation techniques.

Graph-Theoretic Analysis

Busam (36) reviews the UNIVAC 1100 series, which employs a three-pass optimizing compiler. The first pass encodes all operations into a uniform tabular format. To achieve maximum recognition of common subexpressions, redundant information may be added to the table, while flow information is maintained in a list containing all statement numbers and references. The second pass scans the code in reverse order and performs common-subexpression elimination and movement of loop-invariant computations. The compute point of each expression is determined, and the expression evaluation is moved to that (lower frequency) point. In contrast, IBM FORTRAN H will successively move a computation out of each loop level until it can no longer be moved. The determination of an expression's compute point permits high-speed compilation because an expression need not be moved more than once. The second and third passes of the compiler are also concerned with register assignment and code generation.

In the USSR, the ALPHA automatic programming system (Yershov 37 and 38) for the M-20 computer produces object code that in some cases is nearly the equal of hand coding. The major architecture-independent optimizations include optimization of subscripts within FOR loops and the elimination of redundant subexpressions within a basic block. A feature of the ALPHA compiler, not found elsewhere, is the attempt to minimize the number of locations occupied by data within a program. While many optimizations result in a decrease in the size of a program because fewer instructions are generated, ALPHA specifically minimizes the storage requirements for data and variables by permitting several variables to share a storage location if their uses do not interfere with one another. This is a generalization of subsumption, which coalesces two variables into one if they contain the same value whenever used.

Much of the work in the past two years has centered around the use of the Cocke-Allen interval analysis technique, which was described first by Cocke and Schwartz (39). This technique is based on the *interval*, a partially ordered set of basic blocks with the following properties:

1. An interval is a set function of a distinguished block called the head.
2. All blocks in an interval, except the head, have all their immediate predecessor blocks in the interval.

Intervals are easily and rapidly constructed, and they readily identify inner loops and a possible processing order within each loop. The ordering induced by the interval construction is valuable because dominators always precede their dominees. Common subexpression elimination is simplified in this context because the redundancy of a computation is indicated by its presence in a dominator block. The interval construction process may be iterated (treating intervals as basic blocks), and higher order loops will then be identified. Most program graphs will be reduced to a single node by repetition of this procedure. Those few program graphs which are not reducible may be transformed into reducible graphs by the process of *node splitting* (Cocke and Miller (40)). Figure 5 shows the intervals obtained from the example flowchart and how they may be used to compute relative frequencies.

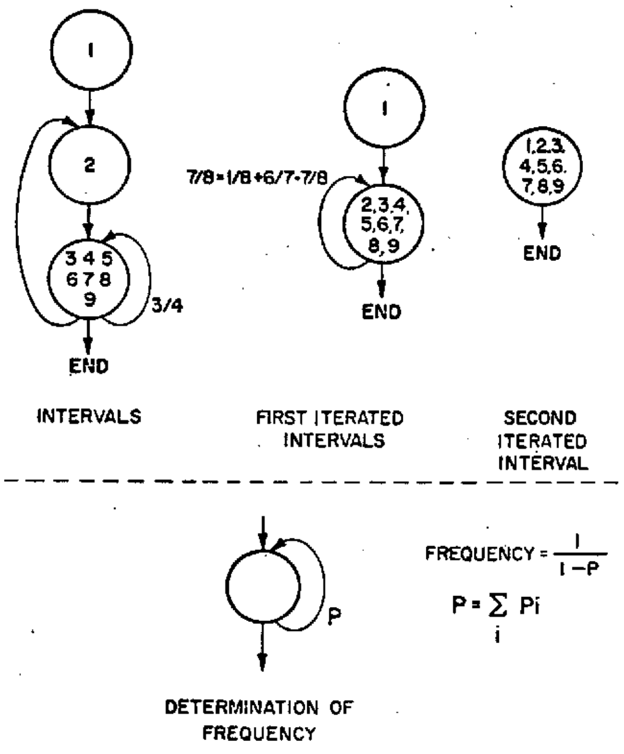


Figure 5. Interval Analysis and Frequency Determination—Relative Frequency of a Loop May Be Determined from the Probability of a Loop-Closing Branch

In the proceedings of the Association of Computing Machinery SIGPLAN'S Symposium on Compiler Optimization, Allen (41) indicates that over 90 percent of the program graphs subject to analysis were reducible. She gives algorithms that determine the back dominator of each node in an interval, the articulation blocks of an interval, and the maximum strongly connected region within an interval. The use of interval analysis for global optimization is shown with an example that demonstrates how information is relayed through successive iterations of the interval construction. In the same proceedings, Cocke (42) describes a method for common-subexpression elimination based upon interval analysis. The information required to determine whether a computation is redundant may be coded as a large system of Boolean equations. The interval technique permits solution of this system without having to perform a tedious Gauss elimination. Only two passes through the system of equations are required. A later paper by Allen and Cocke (43) summarizes techniques for identifying intervals and properties of blocks within intervals. The concept of node splitting to permit reduction of an arbitrary program flow graph is discussed in detail.

Kennedy (44) has built upon Allen and Cocke's work. He has divided questions concerning data flow into two classes:

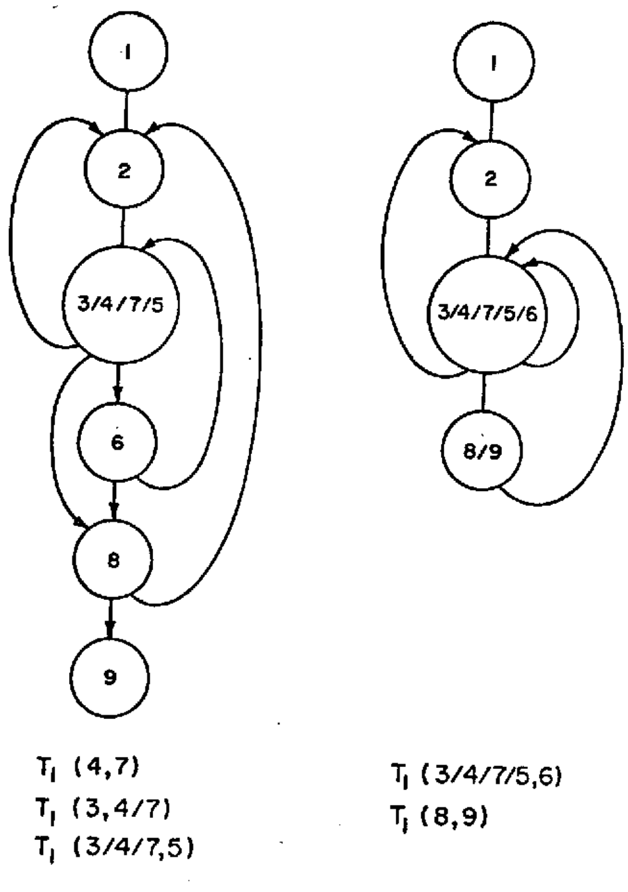
1. Those referring to the status of variables on entry to a block
2. Those referring to the effect of computations within a block on later computations

Since the first class of problems has been solved, Kennedy gives an approach to the second. An algorithm for the identification of dead variables is shown, which, like Cocke's common subexpression elimination algorithm, requires only two passes. The two passes perform logical operations on bit-vectors, which may be performed in parallel on most machines, resulting in a very high-speed algorithm.

The *value number* technique described by Cocke and Schwartz assigns a unique identifier to each calculation in a program. Whenever a calculation is to be performed, a table lookup determines whether it is currently available. Because a numeric identifier is associated with each calculation, formal identity is not required to find a common subexpression (Figure 3). Program-flow properties that render a match impossible are reflected by assigning new identifiers at statement labels.

Schneck and Angel (45) have combined the interval analysis and value number techniques in an optimizer which accepts and produces FORTRAN programs. All of the techniques referred to at the beginning of this section are implemented, and new optimizations are introduced. Strict ordering of nodes within an interval permits the value number technique to be applied globally and eliminates virtually all common subexpressions. A second pass over the program, in the manner of Cocke and Kennedy, permits global constant propagation to be performed. The efficacy of optimization at the level of a programming language is also discussed.

A recent paper by Hecht and Ullman (46) introduces a pair of transformations which may be used in flow-graph analysis. Transformation T_1 removes an edge which begins and ends at the same node. Transformation T_2 condenses node *a* into its unique immediate predecessor *b* resulting in *a/b*. A flow graph is called *collapsible* if, and only if, repeated application of T_1 and T_2 results in a single node. Figure 6 indicates the collapsibility of the example flow graph. Collapsibility is shown to be equivalent to interval reducibility. The time to determine collapsibility is $O(n \log n)$, while the time to determine interval reducibility may be $O(n^2)$. Information obtained by interval analysis may also be obtained by application of T_1 and T_2.



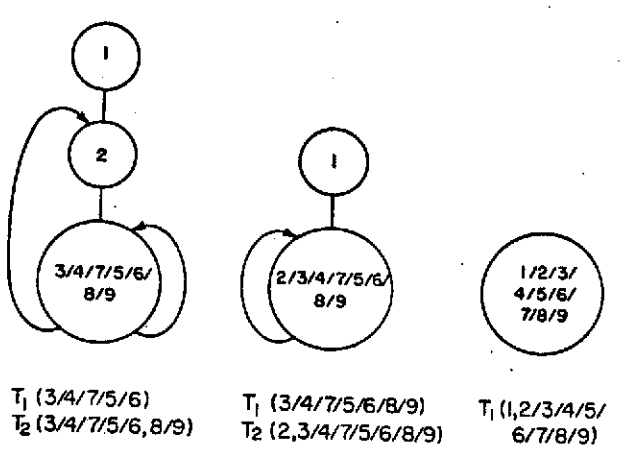


Figure 6. Collapsibility—Repeated Applications of Two Transformations Yield a Single Node in *O(n log n)* Steps

CONCLUSION

The powerful architecture-independent optimizations are responsible for most of the increased speed obtained by an optimizing compiler. Schneck and Angel (45) have shown that these optimizations may be applied before compilation and achieve almost all that a compiler can. With IBM's FORTRAN H, architecture-independent optimization accounts for 80 percent of the speed increase. With the CDC compiler for the 6600, external optimization produces code faster than the compiler can. In summary, an external architecture-independent compiler, supplemented by a machine-oriented compiler, is the most cost-effective technique. This is true for the manufacturer, who may devote less time to the compiler, as well as for the programmer, who will find debugging easier in this environment because he can see what changes have been effected.

REFERENCES

1. International Business Machines Corporation, *The IBM Mathematical Formula Translating System,* FORTRAN, 1954.
2. R. C. Miller and B. J. Oldfield, "Producing Computer Instructions for the PACT I Compiler," *Journal of the ACM,* 3, 1956.
3. P. B. Sheridan, "The Arithmetic Translator–Compiler of the IBM FORTRAN Automatic Coding System," *Communications of the ACM,* 2, 1959.
4. W. M. McKeeman, "Peephole Optimization," *Communications of the ACM,* 8, 1965.
5. J. T. Bagwell, Jr., "Local Optimizations," *ACM SIGPLAN Notices,* 5, No. 7, 1970.
6. J. P. Anderson, "A Note on Some Compiling Algorithms," *Communications of the ACM,* 7, 1964.
7. I. Nakata, "On Compiling Algorithms for Arithmetic Expressions," *Communications of the ACM,* 10, 1967.
8. R. R. Redziejowski, "On Arithmetic Expressions and Trees," *Communications of the ACM,* 12, 1969.
9. V. Schneider, "On the Number of Registers Needed To Evaluate Arithmetic Expressions," *BIT,* 11, 1971.
10. M. Finkelstein,"A Compiler Optimization Technique," *The Computer Journal,* 11, 1968.
11. B. Randell and L. J. Russell,"*Single-scan Techniques for the Translation of Arithmetic Expressions in ALGOL 60,*" Journal of the ACM, 11, 1964.
12. R. W. Allard, K. A. Wolf, and R. A. Zemlin, "Some Effects of the 6600 Computer on Language Structures," *Communications of the ACM,* 7, 1964.
13. J. F. Thorlin, "Code Generation for PIE (Parallel Instruction Execution) Computers," *Proceedings of the Spring Joint Computer Conference,* 1967.

14. International Business Machines Corporation, TR 00.2240, *Current Technologies in FORTRAN Object Code Optimization*. D. Blum, S. K. Brown, A. G. Calavano, H. O. Hempy, and J. Suez, 1971.
15. H. S. Stone, "One-Pass Compilation of Arithmetic Expressions for a Parallel Processor," *Communications of the ACM*, 1967.
16. J. C. Han, *Tree Height Reduction for Parallel Processing of Blocks of FORTRAN Assignment Statements*, National Technical Information Service, PB-207985, 1972.
17. C. V. Ramamoorthy and M. J. Gonzalez, "Subexpression Ordering in the Execution of Arithmetic Expressions," *Communications of the ACM*, 14, 1971.
18. W. H. Burkhardt, "Automation of Program Speed-Up on Parallel-Processor Computers," *Computing*, 3, 1968.
19. R. E. Milstein, *Compiler Design for the ILLIAC IV*, National Technical Information Service, AD 719417, 1971.
20. L. Lamport and D. Presberg, *Concurrent Compiling*, National Technical Information Service, AD 742279, 1972.
21. P. B. Schneck, "Automatic Recognition of Parallel and Vector Operations in a Higher Level Language," *Proceedings of the ACM National Conference*, 1972.
22. D. J. Kuck, C. Muraoka, and S. C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-Like Programs and Their Resulting Speedup," *IEEE Transactions on Computers*, C-21, 1972.
23. J. W. Backus, et al. "The FORTRAN Automatic Coding System," *Proceedings of the Western Joint Computer*, 1957.
24. L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd, "Index Register Allocation," *Journal of the ACM*, 13, 1966.
25. F. Luccio, "A Comment on Index Register Allocation," *Communications of the ACM*, 10, 1967.
26. W. H. E. Day, "Compiler Assignment of Data Items to Registers," *IBM Systems Journal*, 9, 1970.
27. R. T. Prosser, "Applications of Boolean Matrices to the Analysis of Flow Diagrams," *Proceedings of the Eastern Joint Computer Conference*, 1959.
28. S. Warshall, "A Theorem on Boolean Matrices," *Journal of the ACM*, 9, 1962.
29. C. V. Ramamoorthy, "Analysis of Graphs by Connectivity Considerations," *Journal of the ACM*, 11, 1964.
30. F. E. Allen, "Program Optimization," *Annual Review of Automatic Programming*, New York: Pergamon Press, 1969.
31. R. L. Kleir and C. V. Ramamoorthy, "Optimization Strategies for Microprograms," *IEEE Transactions on Computers*, C-20, 1971.
32. D. Gries, *Compiler Construction for Digital Computers*, New York: John Wiley and Sons, 1972.
33. J. T. Ryan, "A Direction-Independent Algorithm for Determining the Forward and Backward Compute Point for a Term or Subscript During Compilation," *The Computer Journal*, 9, 1966.
34. International Business Machines Corporation, TR 00.1330, *Global Program Optimization*, C. W. Medlock and E. S. Lowry, 1965.
35. E. S. Lowry and C. W. Medlock, "Object Code Optimization," *Communications of the ACM*, 12, 1969.
36. V. A. Busam and D. E. Englund, "Optimization of Expressions in FORTRAN," *Communications of the ACM*, 12, 1969.
37. A. P. Yershov, "ALPHA–An Automatic Programming System of High Efficiency," *Journal of the ACM*, 13, 1966.
38. A. P. Yershov, *The ALPHA Automatic Programming System*, New York: Academic Press, 1971.
39. J. Cocke and J. T. Schwartz, *Programming Languages and Their Compilers*, New York: New York University, 1969.
40. J. Cocke and R. E. Miller, "Some Analysis Techniques for Optimizing Computer Programs," *Proceedings of the Second Hawaii International Conference of System Sciences*, 1969.
41. F. E. Allen, "Control Flow Analysis," *ACM SIGPLAN Notices*, 5, 1970.
42. J. Cocke, "Global Common Subexpression Elimination," *ACM SIGPLAN Notices*, 5, 1970.
43. F. E. Allen and J. Cocke, "Graph Theoretic Constructs for Program Control Flow Analysis," Unpublished paper.
44. K. Kennedy, "A Global Flow Analysis Algorithm," *International Journal of Computer Mathematics*, 3, 1971.
45. P. B. Schneck and E. Angel, "A FORTRAN to FORTRAN Optimizing Compiler," *The Computer Journal*, to be published in 1973.
46. M. S. Hecht and J. D. Ullman, "Flow Graph Reducibility," *SIAM Journal of Computing*, 1, 1971.