A CATALOGUE OF OPTIMIZING TRANSFORMATIONS

Frances E. Allen
John Cocke
IBM Thomas J. Watson Research Center
Yorktown Heights

A result of the recent work in optimization has been to systematize the potpourri of optimizing transformations that a compiler can make to a program. This paper catalogues many of these transformations.

INTRODUCTION

A catalogue of optimizing transformations which a compiler can make on a program is presented in this paper. The catalogue does not pretend to include all possible transformations; what is presented is a categorization of most of the transformations which are currently reasonably well understood.

The basic purpose of the transformations presented is to improve the execution time of compiled programs. Some attention is given to execution space, but no attention is given to optimizing compile time or to the more general questions of total job or system optimization.

The term *optimization* is a misnomer in that it

is not generally clear that a particular, so called, optimizing transformation even results in an improvement to the program. A more correct term would be "amelioration." Aho, Sethi and Ullman [1] have been able to prove the optimality of a set of transformations on straight-line code. The criterion they used for measuring optimality was the length of machine code generated. We do not have, however, any general means of establishing optimal bounds, much less any means of determining that a general set of transformations produces code satisfying given bounds.

Another aspect of optimization which needs further study is the question of the correctness of transformations. We would like to be able to prove that applying a set of transformations to a program produces an equivalent program. Some investigations into this question are reported by Ullman [15]. In developing the transformations presented here, care has been taken not only to assure that the obvious data values are correct and available when required but that "side effects" are maintained and language rules honored. These assurances are built into the algorithms implementing these transformations and are not reported here. They are generally a rather ad hoc collection of considerations.

The survey of optimizing transformations which follows is organized so that the interprocedural optimizations are presented first, transformations best performed on a program form close to the source language are presented next, followed by the so-called machine independent and machine dependent optimizations. The classification of transformations into machine independent optimizations and machine dependent optimizations is used to distinguish the transformations which are applicable to most register-oriented computers from those which require knowledge of the specific characteristics of a given computer. For example, eliminating redundant instructions is a machine independent transformation, whereas register allocation is machine dependent. Very little is said about the language level best suited for the transformations. Clearly, however, the machine

dependent optimizations create and/or transform instructions close to the machine level. Most of the machine independent optimizations are best performed on subexpressions; the language level of these subexpressions can be high, that is, close to source language level, or low, close to the machine language level.

Each transformation included in the catalogue is identified and described briefly (but is not formally defined). An example is given, followed by some discussion of the transformation. The first transformation effects an interprocedural optimization.

PROCEDURE INTEGRATION

In order to smooth subprogram linkages and to provide more global program units for optimization, subprograms and/or their prologues may be merged into their calling subprograms. If subprogram C calls subprogram S (C → S), then one of the following linkages may be established.

1. *CLOSED*. This is a standard linkage and no intersubprogram optimization is performed. S, the called routine, is known to the system loader. No information about S is available during the compilation of C. This kind of linkage is inefficient because

a. registers must be saved and restored on entry and exit.

b. specific registers must be allocated to specific functions. This constrains register assignment, and may cause "register busy interlock" delays at execution time on close sequential calls in pipelined CPU's.

c. the parameters and global variables (e,g., variables in COMMON) set and/or used in S are not known during the compilation of C. The compilation

of C must assume that all such variables can be set and used in S. At the time of the CALL memory must contain the current values of such variables; first uses after the CALL must access storage.

d. in most cases, all information must be passed in storage.

e. the called program cannot take advantage of particular argument values, recurrences or relationships. For example, the called routine cannot take advantage of a constant valued argument

2. *OPEN*. An open linkage is not a linkage at all. The called routine, S, replaces the CALL statement in the caller, C, after the arguments on the CALL have replaced their corresponding parameters in S. The program S is completely integrated in C and is not known to the system.

There are many obvious advantages to this kind of linkage; two are:

a. There is no linkage overhead whatsoever.

b. C and S are optimized together. Advantage can be taken of particular argument values and relationships: constant arguments can be folded into the code, invariant instructions in S can be moved to infrequently executed areas of C, etc.

The major disadvantages and constraints on opening up subprograms are:

a. program sizes may become too large.

b. irreducible subprograms require special handling. (An *irreducible subprogram* is one which maintains a history, performs I/O operations, can return different function values for identical argument values, or does not return through a standard return point.)

c. a high-level version (not necessarily

source) of the program being called must be available to the compiler at the time the caller is compiled.

d. a change in the called routine obsoletes object modules into which it has been merged.

3. *SEMI-OPEN*. A semi-open linkage is a non-standard linkage in which the called routine, S, is compiled with its caller, C. S and C become part of the same object module; S is not known to the system The parameters become real locations which are used to hold argument values or locations. During optimization it may be recognized that actual parameter locations are not needed, and S and C become indistinguishable. All CALL's to S in C become branches to a location internal to the module.

On a computer with a paged memory or base-offset addressing, both the open and the semi-open forms increase the possibility of both routines being in the same page or in the same addressing area. The semi-open form of linkage has some of the same disadvantages and constraints as the open form. The primary advantages arise from the effects of optimizing the two routines as a unit. The object-time-association is, of course, more direct and faster than a standard linkage.

4. *SEMI-CLOSED*. The semi-closed linkage is a non-standard linkage in which the two routines are compiled as separate modules and are both known to the system. The called routine, S, is compiled first. The compilation of S determines the linkage registers and the parameter passing conventions to be used when S is called. This information, along with other information about S, is used during the compilation of C to establish the calling linkage. The other information about S which is available to C is

a. which global variables and parameters are and/or used by S.

b. which registers are used by S.

The advantages of this type of linkage are:

a. the compilation of S is not constrained by fixed register assignment at entry and exit points.

b. the compilation of C can take advantage of knowing how global variables are used in S to avoid memory references across CALL points and to carry information in registers.

c. registers unused in S need not be saved and restored at linkage time. The unused registers can be active in C across the linkage.

The major disadvantage is that a recompilation of S may obsolete all programs which have a semi-closed linkage to S. A sophisticated loader may avoid this difficulty.

Although most of the above description has been given in terms of subprogram linkages, two of the linkages can be generated in a single COBOL program. A COBOL PERFORM block which is always entered at its beginning (i.e., there are no branches into the middle of it) can be treated as a called subprogram to which internal linkages are generated or which is opened up. In the latter case, a PERFORM of the block is replaced by the block of code. If the PERFORM has a VARYING clause, this becomes analogous to the FORTRAN DO.

Many compilers automatically open certain functions. Beyond this the optimization described here does not exist in any commercially available compiler. In order to perform a procedure integration, the compiler has to have access to both procedures at essentially the same time. Therefore, a change has to be made to the standard compiler organization in which each procedure or subroutine is compiled to binary code independently of all others submitted with it. In addition to a different compiler design some criteria for determining

which type of linkage to make are needed. This may be best established by the programmer based on what he knows about the future usages of procedures, but it can be made by the compiler itself in certain cases. For example, it might be fairly easy to determine that a routine could be opened up if it appears just once in the programs that are associated with a job or if it is very small. The criteria for automatically integrating routines need further investigation and probably cannot be established without knowledge of the program environment.

LOOP TRANSFORMATIONS

Several transformations can be made on program loops to improve the execution time of the program. Although it is not necessary that these transformations be made on a language level which is close to the source language, they seem to be more easily identified and performed at that level. In the examples which follow, DOs are used to identify the loops being transformed; however, other encodings of loops could trigger the transformations. Three types of loop transformations are considered in this paper.

1. *Loop Unrolling*. A loop can be unrolled completely so that the successive computations implied by the loop appear sequentially or it can be partially unrolled as in the following example:

```
DO I = 1 TO 100 BY 1;
A(I) = A(I) + B(I);
END;
```

becomes when unrolled by 2:

```
DO I = 1 TO 100 BY 2;
A(I) = A(I) + B(I);
A(I+1) = A(I+1) + B(I+1);
END;
```

The advantages of loop unrolling are that

a. the number of instructions executed is reduced. In the preceeding example the number of increments and tests for loop control is halved.

b. more instructions are exposed for parallel execution. The two statements in the unrolled form of the preceding example can be executed at the same time since they are independent.

The major disadvantage of loop unrolling is the increased instruction space required. For this reason the criteria for unrolling loops should include the size of the loop and the relative frequency of executing the loop. Other factors are the severity of the object space constraints and the form of the loop itself. A loop with variable control parameters (e.g., DO I = J to K by L;) can be unrolled but requires code to test for end conditions. Nests of loops can be unrolled; an example of this will be given in the next section.

2. *Jamming or Loop Fusion.* The second loop-related transformation is called jamming or loop fusion. In this transformation two loops are put together and expressed by one loop as in the following example:

```
DO I = 1 TO 100;
A(I) = 0.;
END;
DO I = 1 TO 100;
B(I) = X(I) + Y;
END;
```

becomes

```
DO I = 1 TO 100;
A(I) = 0.;
B(I) = X(I) + Y;
END;
```

The advantages of this transformation are that

a. loop overhead is reduced;
b. code space is reduced;
c. more instructions are exposed for parallel execution and for local optimization.

The cases which can be found and transformed can be fairly elaborate or relatively simple. The simplest case involves two loops which together satisfy the following criteria:

a. If one loop is executed the other one is also; that is, they have the same execution conditionality.

b. The computations in either loop do not depend upon the computations in the other. This criterion can easily be relaxed to more particular situations.

c. The loops are executed the same number of times. By generating code for the end conditions, this criterion can be deleted.

The need for this transformation may occur during the compilation of languages which have array or vector operations. If statements involving array or vector operations are translated to the more basic element-by-element operations, fusible loops and nests of loops may appear quite frequently in the code. This optimization has been done by Ershov in his ALPHA compiler [9] and was described in a thesis by Wagner [16].

An example follows in which a matrix multiply program is unrolled by 2 and jammed to eventually yield a program which multiplies 2 x 2 matrices. The steps are:

a. original program:

```
DO I = 1 TO 100;
DO J = 1 TO 100;
A(I,J) = 0.;
DO K = 1 TO 100;
A(I,J) = B(I,K) * C(K,J  + A(I,J);
END; END; END;
```

b. the outer loop is unrolled by 2:

```
DO I = 1 TO 100 BY 2;
DO J = 1 TO 100;
A(I,J) = 0.;
DO K = 1 TO 100;
A(I,J) = B(I,K) * C(K,J) + A(I,J);
END; END;
DO J = 1 TO 100;
A(I+1,J) = 0.;
DO K = 1 TO 100;
A(I+1,J) = B(I+1,K) * C(K,J) + A(I+1,J);
END; END;
END;
```

c. the two inner sets of loops are jammed:

```
DO I = 1 TO 100 BY 2;
DO J = 1 TO 100;
A(I,J) = 0.;
A(I+1,J) = 0.;
DO K = 1 TO 100;
A(I,J) = B(I,K) * C(K,J) + A(I,J);
A(I+1,J) = B(I+1,K) * C(K,J) + A(I+1,J);
END; END; END;
```

d. the next inner loop, the J loop, is unrolled by 2 and the two resulting K loops are jammed; then the innermost loop, the K loop, is unrolled by 2. The result of these transformations is:

```
DO I = 1 TO 100 BY 2;
DO J = 1 TO 100 BY 2;
A(I,J) = 0.;
A(I,J+1) = 0.;
A(I+1,J) = 0.;
A(I+1,J+1) = 0.;
DO K = 1 TO 100 BY 2;
A(I,J) = B(I,K) * C(K,J) + A(I,J);
A(I,J) = B(I,K+1) * C(K+1,J) + A(I,J);
A(I,J+1) = B(I,K) * C(K,J+1) + A(I,J+1);
A(I,J+1) = B(I,K+1) * C(K+1,J+1) + A(I,J+1);
A(I+1,J) = B(I+1,K) * C(K,J) + A(I+1,J);
A(I+1,J) = B(I+1,K+1) * C(K+1,J) + A(I+1,J);
A(I+1,J+1) = B(I+1,K)*C(K,J+1)+A(I+1,J+1);
A(I+1,J+1)=B(I+1,K+1)*C(K+1,J+1)+A(I+1,J+1);
END; END; END;
```

e. the eight expressions in the inner loop can be simplified to 4 by a transformation alluded to later in this paper.

```
DO I = 1 TO 100 BY 2;
DO J = 1 TO 100 BY 2;
A(I,J) = 0.;
A(I,J+1) = 0.;
A(I+1,J) = 0.;
A(I+1,J+1) = 0.;
DO K = 1 TO 100 BY 2;
A(I,J) = B(I,K+1) * C(K+1,J) + B(I,K)
          * C(K,J) + A(I,J);
A(I+1,J+1) = B(I+1,K+1) * C(K+1,J+1)
          + B(I+1,K) * C(K,J+1)
          + A(I+1,J+1);
END; END; END;
```

3. *Unswitching*. The third transformation related to loops is the opposite of jamming. It is called unswitching and involves breaking a loop containing a loop-independent test into two loops with the test selecting which of the two loops to execute. The following is the quintessential example of this transformation.

```
DO I = 1 TO 100;
IF (T) THEN X(I) = A(I) + B(I);
       ELSE X(I) = A(I) - B(I);
END;
```

becomes

```
IF (T) THEN
       DO I = 1 TO 100;
          X(I) = A(I) + B(I);
       END;
             ELSE
       DO I = 1 TO 100;
          X(I) = A(I) - B(I)
       END;
```

This optimization has the advantage of reducing the number of instructions executed. More instruction space is required, however.

It is not clear how frequently unswitchable loops appear in programs and, therefore, how important this transformation is.

The machine independent optimizations which transform subexpressions and subexpression relationships are now considered. In this transformations a distinction is made between those involving subexpressions in the same "basic block" and those involving subexpressions in different basic blocks. A basic block is a linear sequence of instructions with one entry point, the first instruction in the block, and one exit point, the last instruction in the block. Since the internal data dependency relationships within a basic block can be found by relatively simple analysis techniques [2,8], the optimizations performed on basic blocks can, in many cases, be more elaborate than those performed on the program as a whole. For example, an algorithm exists [12] for finding the best assignment of n registers to the given sequence of instructions in the block. Such an algorithm does not exist in the presence of flow. An arithmetic statement by itself constitutes a basic block — usually a non-maximal

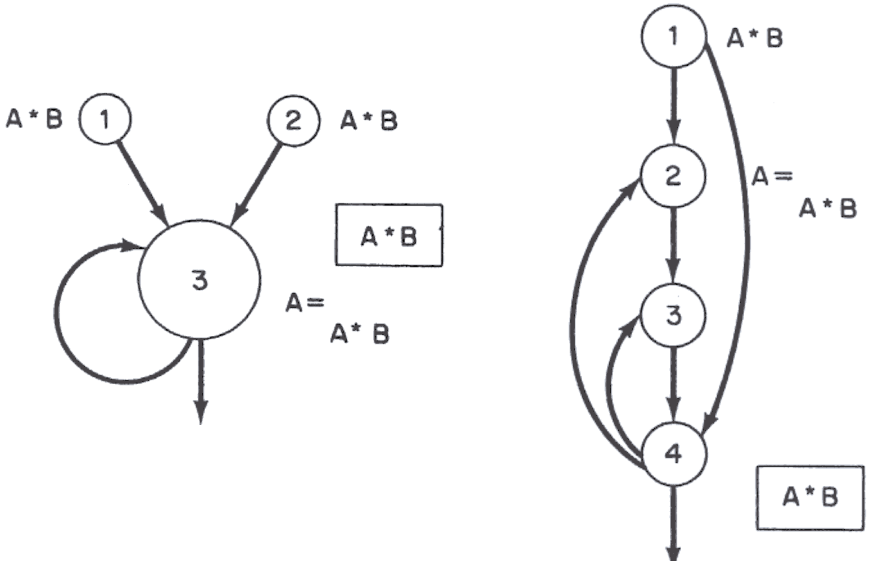
basic block — and the literature [3,5,6,10,11] describes many optimizing transformations for such statements.

The primary emphasis in the collection of transformations presented here is on global optimization; that is, it is on transforming the program based upon the dependency relationships which are found to exist between expressions in different basic blocks as well as in the same block.

In the examples that follow, a directed graph is used to express the control flow relationships between the basic blocks in a program. The nodes of the graph represent basic blocks and the edges control flow paths.

REDUNDANT SUBEXPRESSION ELIMINATION

This optimization, which is also called *common subexpression elimination*, involves finding and eliminating those computations which calculate values already available. Consider the following examples in which the redundant subexpression is identified by being enclosed in a box.



The subexpression A*B has, of course, a subsequent use: its value is used in another expression, is assigned to a variable or used in a test. In both examples, correct though not necessarily identical values are available on every path to the subexpression and hence to its use. The subexpression is therefore redundant.

In the two examples the redundant subexpression was formally identical with the subexpressions producing usable values. One form [7] of global analysis for this optimization depends upon the existence of formal identities. Another form of analysis, based upon the value number algorithm [8], does not depend upon explicit formal identities for the identification of a redundant calculation. Consider the following example of straight-line code:

```
      = A * B
C     = A
      = C * B
```

C*B is not formally identical with A*B but computes the same value so is a redundant calculation. The two methods for detecting redundant subexpressions find different cases as well as some of the same cases. The value number method as it is currently formulated would not find either of the cases shown in the first two examples in this section; the method requiring formal identities would not find C*B in the last example.

The major advantages of this optimization are that

a. fewer instructions are executed, and

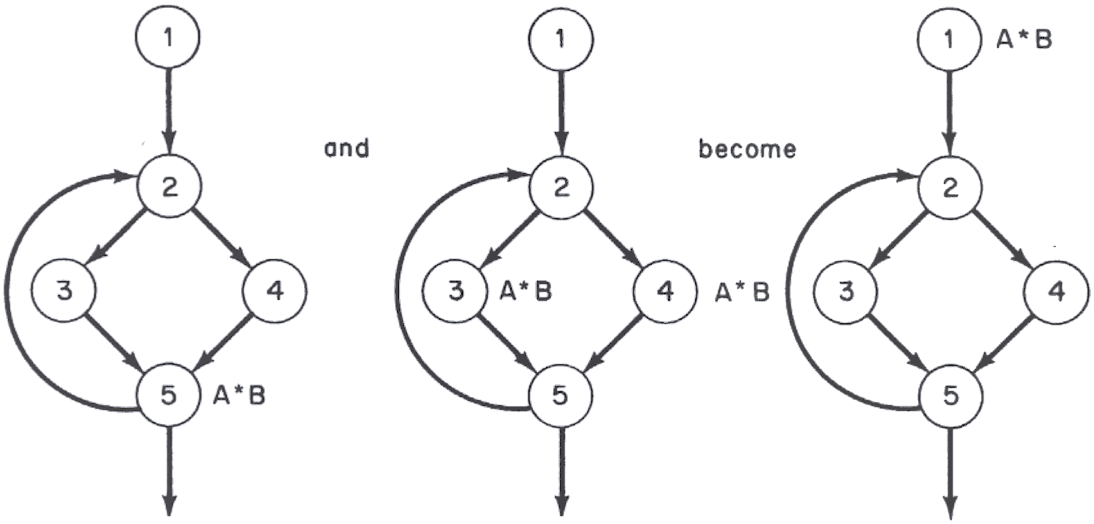
b. instruction space is saved.

The disadvantage is that register usage is extended.

In the next section it will be seen that code motion and redundant subexpression elimination are intimately related. By inserting instructions at propitious places in the code, more subexpressions can be eliminated than could be in the original program form.

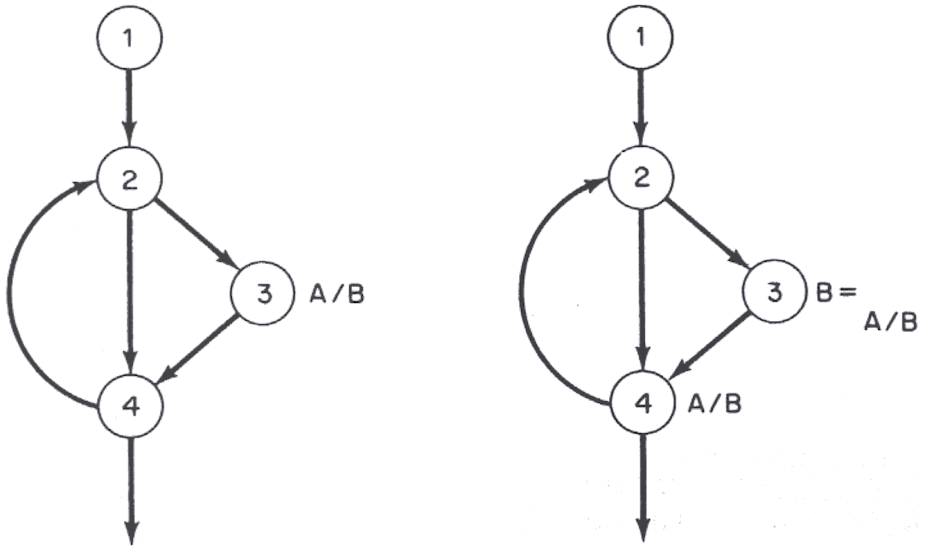
CODE MOTION

A subexpression can be moved if the value available to its uses is not changed by this move and if the move is "safe." The safety criterion is somewhat vague, but it essentially means that a moved subexpression will not cause side effects to occur which would not have occurred if the subexpression had remained in its original position. Two examples of movable subexpressions are:



In each case it was assumed that neither A nor B were changed by the other instructions in nodes 2, 3, 4 and 5, that is A*B is invariant with respect to the loop. Even if it is assumed that A*B might cause an overflow condition it is clear that the occurrence of this side effect is not being altered by moving the subexpression into node 1.

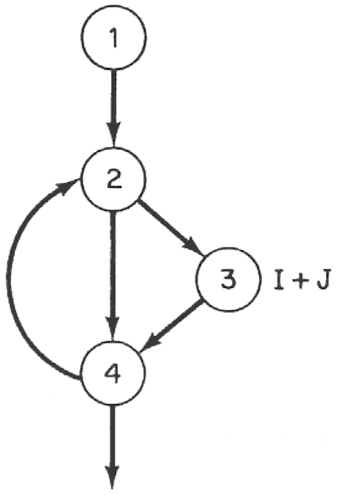
It will be noted that the number of times the overflow may occur is probably being altered. The true correctness of moving any subexpression which can cause side effects is probably questionable and certainly dependent on language rules. If a standard system action is taken such as stopping the program or flagging the result, then maintaining the number of occurrences of the side effect is not important, but maintaining the conditionality for its occurrence is important. Consider the following two examples:



Assuming A/B is considered an unsafe subexpression, i.e., one which might cause a side effect, then, clearly, A/B in node 3 in the first example cannot be moved to node 1. However, the second example has an A/B subexpression in node 4 which may, at first glance, appear to be movable to node 1. It is not, however, because a divide check error may result which would not occur in the program as given.

The basic intent of code motion is to move instructions from frequently executed areas of the program to less frequently executed areas. Since it is not always apparent what the relative execution frequencies of various areas are, an improvement may not always result.

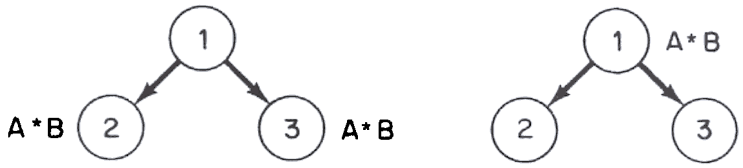
Consider the following example:



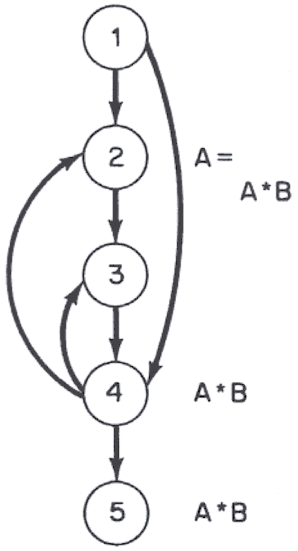
Assuming that I+J is movable and is a safe subexpression, i.e., it cannot cause any side effects, then it may still not be acceptable to move I+J to node 1: node 3 may never be executed. In the absence of any information to the contrary, it is generally assumed that the branch back to 2 from 4 is more frequently taken than the exit from 4, and that the branch from 2 to 3 is taken 50% of the time.

The primary advantage of the code motion transformation, however, is to reduce the number of instructions executed. It has the same disadvantage as redundant subexpression elimination: register usage is extended.

A variant on code motion which does not reduce the number of instructions executed but does save instruction space is called *hoisting*. This transformation is exemplified by the following:



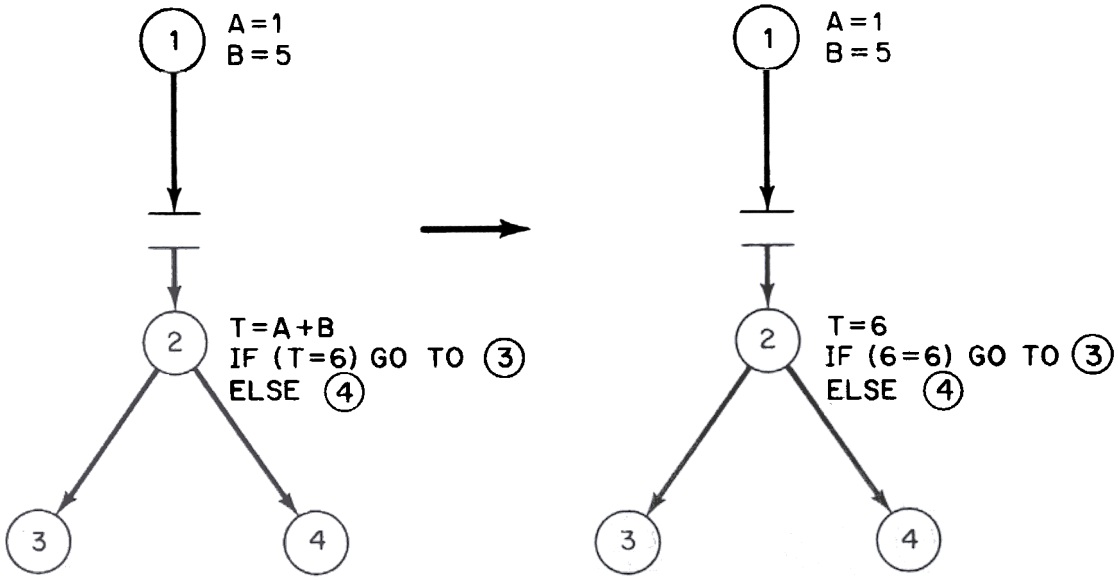
So far in this section, pure code motion has been discussed. Techniques currently exist which essentially combine code motion with redundant subexpression elimination, and indeed subexpressions are not moved at all but eliminated against inserted subexpressions. The following example shows how this can happen.



The A*B in 4 cannot be safely moved and the A*B in 5 cannot be profitably moved. However, if an A*B were placed in node 1, then both of them could be eliminated. An analysis procedure exists which accomplishes this.

CONSTANT FOLDING

Constant folding, also called *subsumption* and *constant propagation*, replaces uses of variables that have been set to a constant by the constant itself. Consider the following example:



The constant assignments to A and B in node 1 replace their uses in node 2. This finds T = 6 which is then propagated to the use of T in the IF statement. As can be seen by this example, constant folding not only involves replacing uses of variables by constants but also involves performing operations whose operands are constants and propagating the result.

A variant on constant propagation involves collecting constants into addresses. In the following example, which is expressed in a very mixed language level, the resulting A+5 represents an address rather than an execution time add.

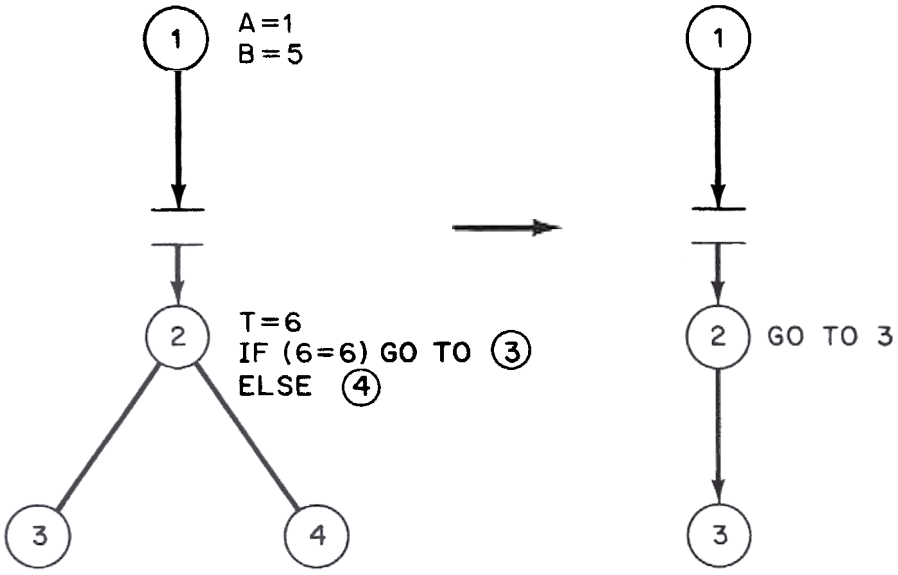
```
K = 5                              K = 5

s = (I*10)+K        becomes        s' = I*10

Load A  (s')                       Load A+5 (s')
```

Constant propagation has many obvious advantages and no disadvantages. It is a particularly important optimization on code in which subprograms have been opened. Arguments to subprograms are frequently

constants and, when constants replace the parameters of the subprogram, many transformations may be made to the subprogram.

DEAD CODE ELIMINATION

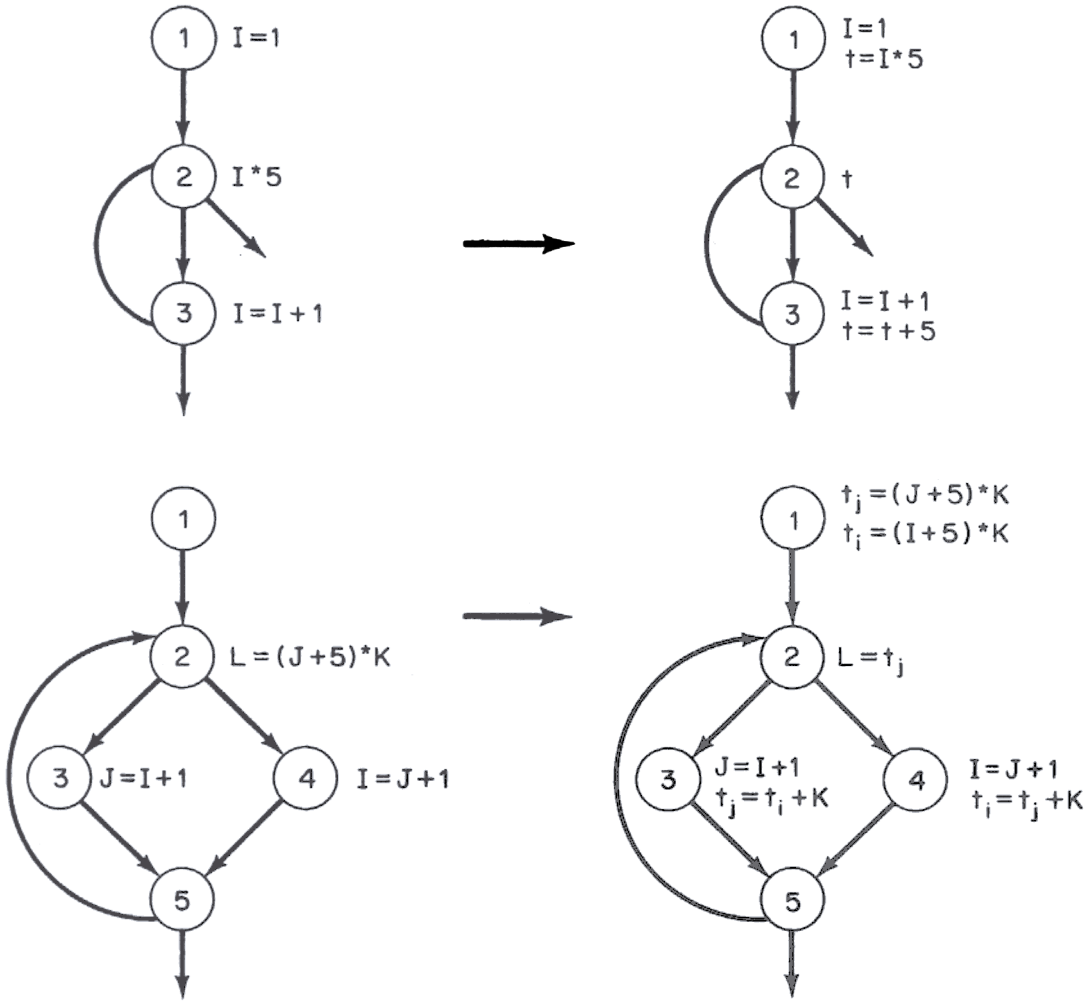
Primarily as a result of constant folding, instructions become "dead." Instructions are considered dead when they cannot be executed because they are in an area of the program which cannot be reached, or when their results are never used. After constant folding, the first example in the previous section had an unreachable block, and, assuming there were no other uses for the definitions of A and B in block 1 and of T in block 2, those instructions are dead.



STRENGTH REDUCTION

The strength reduction optimization replaces certain computations using recursively defined variables by recursively defined computations. This transformation is a generalization of a relatively common optimization: the replacement of subscript calculations involving the DO loop induction variable

by index register increments. Two examples follow:



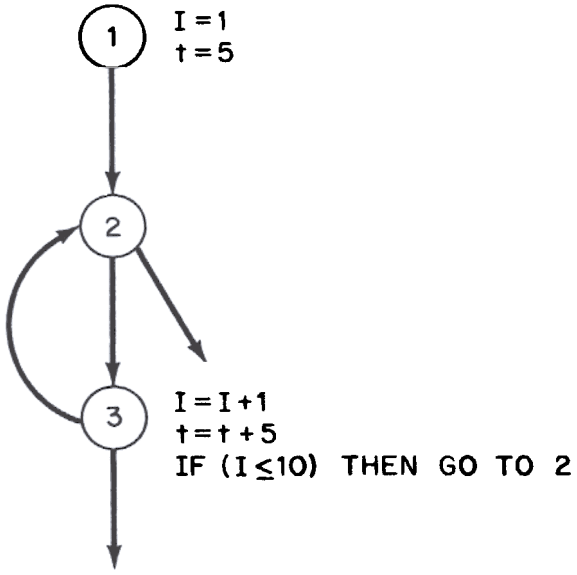
The first example is the traditional case if I*5 is part of a subscript calculation. It should be noted that I*5 in block 1 is a candidate for constant folding. The second example is a more general form of strength reduction in which the variables I and J are recursively defined in terms of each other.

The advantage of strength reduction is that faster computations are used. Although it cannot of course be guaranteed that the new recursively defined variables (the t's in the above examples)

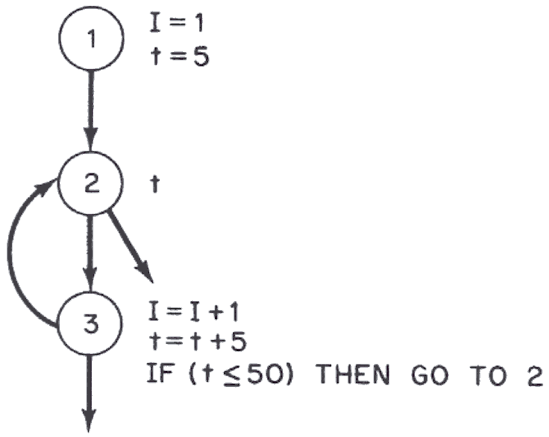
will end up in index registers and be updated by register increments, they will be prime contenders for such register instructions. The strength reduction optimization is not at all limited to instructions transformable to register increments. More complete descriptions of this optimization are contained in [2] and [8].

LINEAR FUNCTION TEST REPLACEMENT

After performing the strength-reduction optimization and introducing new recursively defined variables, it is frequently the case that the only use for the original recursively defined variable is in a test. The test can often be replaced by a test on an introduced variable, thereby making the instructions associated with initializing and incrementing the program variable dead. Suppose that the first example in the preceding section were derived from a DO loop in which the DO statement was DO I = 1 TO 10; then after strength reduction and folding this becomes



The test can be replaced by a test on t:



I = 1 and I = I + 1 are now dead and can be eliminated.

CARRY'S

The carry optimization is a somewhat specialized optimization which recognizes when a subscript calculation for referencing sequential elements of an array does not need to be re-initialized when the reference changes to the next row or column. The following example shows the result of applying the carry, linear function test replacement, strength reduction, folding and dead expression elimination optimizations.

```
DCL A(10,10);     becomes

                                  t = 11
DO I = 1 TO 10;             L1: A - 11 (t) = 0
DO J = 1 TO 10;                 t = t + 1
A(I,J) = 0.;                    IF (t<111) THEN
END;                              GO TO L1
END;
```

(A - 11 is an address)

We now consider the machine dependent transformations.

INSTRUCTION SCHEDULING

In this optimization, sequences of instructions are ordered to minimize the execution time of the sequence. This optimization is used when the target computer has pipelined units.

Suppose a computer has the following characteristics:

a. an instruction can be started every cycle provided there are no register or data interlocks

b. the instructions are taken in order

c. the divider takes 12 cycles

d. the three-cycle adder is pipelined and can start a new add on every cycle

e. stores take 1 cycle

$T_1 = A + B$	may	$T_4 = X/Y$
$T_2 = T_1 + C$	become	$T_1 = A + B$
$T_3 = T_2 + D$		$T_2 = T_1 + C$
$E = T_3$		$T_3 = T_2 + D$
$T_4 = X/Y$		$E = T_3$
$Z = T_4$		$Z = T_4$

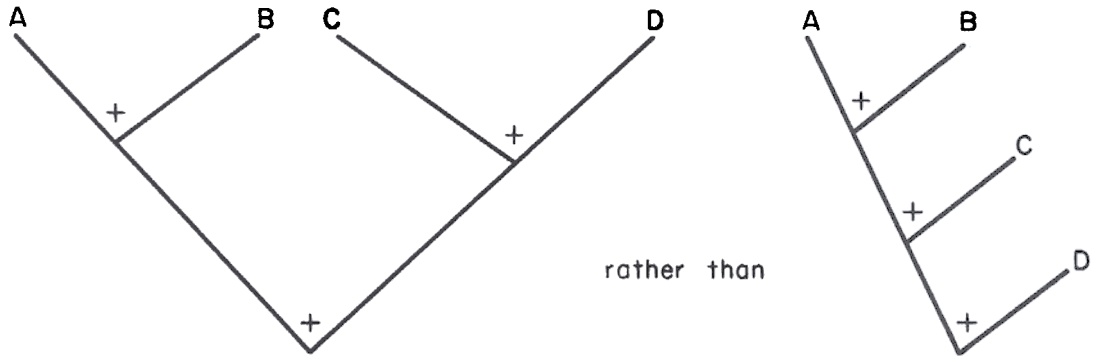
The original sequence, the one on the left, takes 23 cycles; the reordered sequence takes 13 cycles. In effect the adds and the store into E are free.

The advantage is clear: execution time is decreased. The disadvantage is that register usage is extended.

PARSING METHODS

Several variants on the usual left-to-right associative parse can produce better code. An interesting one is the minimum depth parse.

Since pipelined units are capable of having several independent instructions in execution at the same time, the operations in a source expression can be associated to produce an instruction sequence which minimizes hardware delays due to data dependencies and to unit-busy interlocks. For example, the expression A+B+C+D can be parsed as if it had been written as (A+B) + (C+D) rather than by the usual left-to-right association (((A+B) + C) + D). Depicted in tree form, this is:



Such a parse is called a minimum depth parse because it minimizes the depth of the parse tree and hence the depth of the data dependency tree. For the computer described in the previous section, the instruction sequence for the minimum depth parse of the expression A+B+C+D takes 7 cycles instead of 9.

A secondary but very important effect of the

minimum depth parse is to expose subexpression independence. Many optimizations involve transformations to and permutations of subexpressions. These transformations and permutations are generally severely constrained by data dependency considerations. In the above example, the subexpression C+D is exposed as independent of A and B by the minimum depth parse and can be moved and transformed independently.

Thus the minimum depth parse optimizes in that it exposes more independent subexpressions for optimization and for execution. The disadvantage is that register usage is extended.

REGISTER ALLOCATION

Allocating registers is quite possibly the hardest optimization to perform.

Several subproblems can be isolated:

a. Allocation can be separated from assignment. Allocation involves determining how many registers are required and allocating registers symbolically. Assignment involves determining which of the actual hardware registers will be used for each allocated register. Allocation is cognizant of hardware constraints. One of the subproblems in assignment is boundary matching: the same entity in program areas which may be executed sequentially should be assigned to the same register if possible.

b. Load-store motion moves loads and stores out of loops in order to be able to retain intermediate results in registers and to avoid unnecessary storage references. Other instructions can sometimes also be moved to obtain a better allocation.

c. The scope of the allocation and assignment can be local or global. The straight-line code of basic blocks can be allocated by simpler

techniques than an allocation which must be aware of the control flow. An optimizing allocation normally consists of both local and global allocation. The problem and solutions are described in more detail in [4] and [13].

STORAGE MAPPING

There are two aspects of storage mapping which are optimizable:

a. Space. The total amount of active space required for variables and for temporaries can be decreased by reuse of dead storage. Ershov [9] treated storage mapping as a generalization of the register allocation and allocated memory cells by some of the same techniques used in register allocation.

b. Adjacency. For computers with paging, base-offset addressing or for which there is any sort of storage staging system it is important that information which is used at the same time should be mapped together.

Other than the work already cited, very little has been done in this area.

We now consider some miscellaneous optimizations.

SHADOW VARIABLES

In this optimization variables are not only kept in the form specified by the program but also in a form more appropriate for their use. For example, it is frequently the case in COBOL programs that all numbers are kept in the decimal radix. It may, however, be determined that certain numbers are frequently used in arithmetic operations and, assuming binary arithmetic is faster

than decimal arithmetic on the target computer, can be kept in binary.

ANCHOR POINTING

Anchor point analysis minimizes the number of logical tests performed in a Boolean expression before a branch or fallthrough occurs. For example, IF (A.AND.B.AND.C) GOTO 10; can branch to the next statement if A is false without testing B and C.

SPECIAL CASE CODE GENERATION

This very important optimization involves detecting special situations in order to generate better code. For example, A^2 may be implemented by A*A rather than using a general exponential routine. There are dozens of examples and there are many compilers which have detected special cases.

PEEPHOLE (or WINDOW) OPTIMIZATION

The final code from a compiler can frequently be improved simply by a local scan of the sequence of instructions [14]. A window of, for example, 10 or 12 instructions can be examined for possible transformations.

This completes our catalogue of optimizations. It is not encyclopedic in any sense, nor is it all definitive. It does, however, report on that part of the recent work in program optimization which has attempted to systematize and extend the collection of optimizing transformations that have long existed.

REFERENCES

[1] Aho, Alfred V., Sethi, Ravi, and Ullman, J.D., "A Formal Approach to Code Optimization," *Proc. of a Symposium on Compiler Optimization*, SIGPLAN Notices, July 1970.

[2] Allen, F.E., "Program Optimization," *Annual Review in Automatic Programming*, Vol. 5, Pergamon, New York, 1969.

[3] Bagwell, John T., Jr., "Local Optimizations," *Proc. of a Symposium on Compiler Optimization*, SIGPLAN Notices, July 1970.

[4] Beatty, J.C., "A Global Register Assignment Algorithm," *this volume*.

[5] Breuer, Melvin A., "Generation of Optimal Code for Expressions via Factorization," *CACM*, June, 1969.

[6] Busam, V.A., and Englund, D.E., "Optimization of Expressions in FORTRAN," *CACM*, December 1969.

[7] Cocke, John, "Global Common Subexpression Elimination," *Proc. of a Symposium on Compiler Optimization*, SIGPLAN Notices, July 1970.

[8] Cocke, J., and Schwartz, J.T., "Programming Languages and Their Compilers," Preliminary Notes, Courant Institute of Mathematical Sciences, New York University, N.Y., April 1970

[9] Ershov, A.P., "ALPHA - An Automatic Programming System of High Efficiency," *JACM*, January 1966.

[10] Floyd, R.W., "An Algorithm for Coding Efficient Arithmetic Operations," *CACM*, January 1961.

[11] Frailey, Dennis J., "Expression Optimization Using Unary Complement Operators," *Proc. of a Symposium on Compiler Optimization*, SIGPLAN

Notices, July 1970

[12] Horwitz, L.P., Karp, R.M., Miller, R.E., and Winograd, S., "Index Register Allocation," *JACM*, January 1966.

[13] Kennedy, Ken, "Index Register Allocation in Straight-Line Code and Simple Loops," *this volume*.

[14] McKeeman, W.M., "Peephole Optimization," *CACM*, July 1965

[15] Ullman, Jeffrey, and A.V. Aho, "Code Optimization and Finite Church-Rosser Systems," *this volume*.

[16] Wagner, R., "Some Techniques for Algorithm Optimization with Application to Matrix Arithmetic Expressions," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, June 1969.