TR 80-003

ON OPERATOR STRENGTH REDUCTION

John H. Crawford
Mehdi Jazayeri

ON OPERATOR STRENGTH REDUCTION

John H. Crawford
Intel Corporation
3065 Bowers Avenue
Santa Clara, California

Mehdi Jazayeri
Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina

ABSTRACT

Strength reduction is a program optimization technique which attempts to replace expensive operations in a program by a series of equivalet, cheaper operations. We discuss the global application of this technique and present an approach to its implementation. This approach is based on well-known optimization techniques of live variable analysis and redundant computation elimination. A discussion of the safety and profitability of the technique is also included.

Key Words and Phrases: Program optimization; strength reduction; safety and profitability of optimization; time vs. space tradeoffs.

1. Introduction

Strength reduction is the name given to a class of program improving techniques that replace expensive operations by a series of equivalent, cheaper operations, if such a replacement results in an overall performance gain. For example, if two additions cost less than one multiply, then 3*A can be replaced by A+A+A. Similarly, X**2 can be replaced by X*X. This form of strength reduction is highly machine dependent; it is also local in that it encompasses only one statement. This kind of improvement is best left to the code generator.

The kind of strength reduction considered here is the kind traditionally associated with loops. It replaces multiplies of the form I*LC where I is an inductive variable and LC is a loop constant, by:

a. initializing a temporary T to I*LC prior to entrance to the loop, and

b. adding statements T=T+ (LC*LC') after statements of the form I=I+LC'.

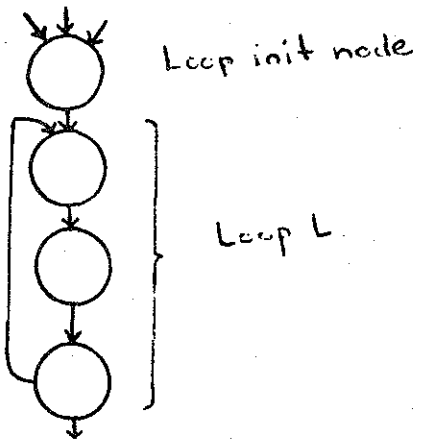
The value of LC*LC' can be computed outside the loop. The temporary T will then hold the current value of I*LC throughout the loop. The major targets of strength reduction are the multiplications used to calculate the addresses of array elements.

In this paper we present a slight generalization of strength reduction, formulate a strategy for its implementation, and discuss the safety and profitability of the operation.

Schaefer (1) discusses the problem of strength reduction. Algorithms for strength reduction appear in (2), (3) and (4). The major characteristic of the approach presented here is that it is entirely based on other, simpler optimization techniques. We assume the reader is familiar with (very) live variable analysis and redundant computation elimination.

2. Terminology

A loop initialization node or loop init node is the only immediate predecessor of the loop entry node that is not in the loop. If no such node exists for a loop, one may be created by adding a dummy node whose successor is the loop entry node, and whose predecessors are the predecessors of the loop entry node not in the loop. The loop init node dominates every node inside the loop.



With respect to a particular loop, L:

A variable, v, is a loop constant if no assignments are made to it inside the loop;

An expression, e, is a loop constant if its constituent operands are loop constants;

A variable i, is an inductive variable if every statement assigning a value to i is of the form i:=j±LC where LC is a loop constant and j is either i or an inductive variable. Control variables of iterative DO loops are almost always inductive variables.

3. Generalization and Applications

Strength reduction need not be limited to loops, inductive variables, and loop constants. We will show that no special consideration of inductive variables is required. We will present a method that attempts strength reduction of multiplies of all variables. This method can easily be extended to other operations, as well as to smaller or larger regions of the program. This generalization only requires that the variables corresponding to loop constants be constant throughout the scope of the strength reduction. This generalized form of strength reduction may be implemented using the well-known redundant computation elimination and live analysis techniques.

As shown in the introduction, strength reduction inserts multiplies of the form (LC*LC') into the program. Unless these multiplies can be folded or removed from the loop, the technique may actually degrade program performance. Even if they can be moved out of the loop, these multiplies will increase the program's size. It is expected that mose of the "loop constants" encountered by strength reduction will be true constants.

One consequence of the application of strength reduction is the removal of references to the iterative variable. If all references to the iterative variable can be removed, then the assignments to the iterative variable can be eliminated as dead. Test replacement is used to replace more uses of the iterative variable. Since the temporaries involved in strength reduction are just multiples of the iterative variable, tests of the form "I relation Ct" can be replaced by a test of the form "Ti relation Ct*Ci". Test replacement has no beneficial effect on the program by itself, but it improves the effect of applying strength reduction by helping to allow the elimination of assignments to the iterative variable.

4. Examples

```
        I:=1;
LOOP:   A:=I*C1;
        B:=I*C2;
        I:=I+C3;
        IF I<C4 THEN GOTO LOOP;
```

This loop can be improved by the application of strength reduction and test replacement by:

a. initializing T1 in the loop init node to I*C1=1*C1=C1, and initializing T2 to C2.

b. inserting the statements

T1:= T1 + (C3*C1);

T2:= T2 + (C3*C2);

after the statement I:=I+C3;

c. replacing the test I<C4 by T1<C1*C4.

The expressions I*C1 and I*C2 are now available throughout the loop, and the program can be optimized by redundant computation elimination. Note that the expressions C3*C1 and C3*C2 can be folded, since C1, C2, and C3 are all constants. Also, if I is not live on exit from the loop, the two assignments to I can be eliminated by dead assignment elimination, yielding:

```
        T1:=C1;
        T2:=C2;
LOOP:   A:=T1;
        B:=T2;
        T1:=T1+(C3*C1);
        T2:=T2+(C3*C2);
        IF T1<(C4*C1) THEN GOTO LOOP;
```

This has eliminated two multiplies and one add from the loop body, and inserted three adds. Two assignments were inserted in the loop init node, and one assignment was deleted. The loop will execute somewhat faster, but the program size has increased by one statement. If multiplies take considerably longer than adds to execute, and the loop is traversed many

times, this transformation will substantially reduce the execution time of the program.

An example of the application of strength reduction outside of loops is:

```
A:= I*C1;
I:= I + C2;
B:= I*C1;
```

which if I is dead after the last statement can be transformed to:

```
A:= I*C1;
B:= A + (C2*C1);
```

Note that the expression C2*C1 can be folded, since C1 and C2 are both constants.

5. Strategy

The approach is heavily based on other optimization techniques. We assume that the reader is familiar with the basic optimization techniques.

a. If expression I*Ci is available in Ti immediately before a statement of the form I:=I±Cj, make it also available after S by tentatively inserting Ti:=Ti±(Cj*Ci) after S.

b. Insert computations of the form I*Ci in the appropriate loop init nodes.

c. Apply (tentative) redundant computation elimination.

d. Retain all computations tentatively inserted in step (a) that are either very live after the computation, or (if this is a loop strength reduction) very live at entrance to the loop. Delete all other inserted computations.

e. Redo (this time permanent) redundant computation elimination to reflect the effects of the inserted computations.

6. Safety and Profitability

The standard code motion safety requirements will be met if the expression temporary of a strength reduced multiply is very live at all points where a computation of the corresponding expression was inserted.

The profitability of this transformation cannot be guaranteed unless the multiplies removed by strength reduction would have been executed at least as frequently as the inserted additions. As with the other code motion techniques, this can be assured if the expression temporary for the multiply is very live at each instruction inserted by the technique, both in the loop init node and after the increment statements.

This technique tends to trade space for time. When used outside of loops, the usual case is to break about even on space, since the usual case is that one add is inserted for each multiply removed. When used in loops, a space increase usually results, since two statements must usually be inserted (one in the loop, one in the init node) to remove one multiply. If the assignments to the iterative variable can be removed, some of this space loss can be recouped.

One special case where this technique is extremely profitable is where the inserted additions can be done with the auto-increment hardware found on machines like the PDP-11. Unfortunately, global optimizations such as this are usually too far from the machine representation of the object code to perform such an optimization effectively. However, it may be possible to set things up properly so that a later improvement phase could do the adds with the autoincrement hardware.

Because this technique tends to increase the size of the program, it seems unreasonable to apply it indiscriminately throughout the program. It can be profitably applied to the few critical inner loops of the program. In these loops with high execution frequencies, this technique would yield a large improvement in execution speed for a small increase in program size. In the less frequently executed parts of the program, however, the slight improvement in execution time that could be realized may not be worth the increase in program size incurred.

It seems reasonable to strength reduce wherever it does not require initializations, i.e., where it does not require inserting computations in the loop init node. Due to the space/time tradeoff involved with strength reduction in loops, some restrictions are needed on that form of the technique. It seems reasonable to apply that technique to only the innermost, most deeply nested loops in the program which permit the elimination of their iterative variable. With this restriction, most of the expected time improvement is obtained, with a minimal increase in program size.

Acknowledgements

The algorithm and analyses were developed for the PLCD cross-compiler at the University of North Carolina. Drs. Steve Weiss and Dave Parnas of UNC read and made valuable comments on the earlier versions of this paper.

References

(1) M. Schaefer, A Mathematical Theory of Global Program Optimization, Prentice Hall, Englewood Cliffs, N.J., 1973.

(2) F. E. Allen, "Program Optimization," Annual Review of Automatic Programming, Vol. 5, Pergamon, Elmsford, N.Y., pp. 239-307, 1969.

(3) J. Cocke and K. Kennedy, "An Algorithm for Reduction of Operator Strength, " Technical Report 476-093-2, Dept. of Mathematical Sciences, Rice University, Houston, Texas. 1974.

(4) F. E. Allen, J. Cocke and K. Kennedy, "Reduction of Operator Strength, " Technical Report 476-093-6, Dept. of Mathematical Sciences, Rice University, Houston, Texas, 1974.