Global Value Numbers and Redundant Computations

Barry K. Rosen
Mark N. Wegman
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
F. Kenneth Zadeck
Department of Computer Science
Brown University
Providence, RI 02912

1 Introduction

词法算法

Most previous redundancy elimination algorithms have been of two kinds. The *lexical* algorithms deal with the entire program, but they can only detect redundancy among computations of lexically identical expressions, where expressions are *lexically identical* if they apply exactly the same operator to exactly the same operands. The *value numbering* algorithms, on the other hand, can recognize redundancy among expressions that are lexically different but that are certain to compute the same value. This is accomplished by assigning special symbolic names called *value numbers* to expressions. If the value numbers of the operands of two expressions are identical, and if the operators applied by the expressions are identical, then the expressions receive the same value number and are certain to have the same values. Sameness of value numbers permits more extensive optimization than lexical identity, but value numbering algorithms have usually been restricted in the past to basic blocks (sequences of computations with no branching) or extended basic blocks (sequences of computations with no joins).

We propose a redundancy elimination algorithm that is *global* (in that it deals with the entire program), yet able to recognize redundancy among expressions that are lexically different. The algorithm also takes advantage of second order effects: transformations based on the discovery that two computations compute the same value may create opportunities to discover that *other* computations are equivalent.

The algorithm applies to programs expressed as reducible [1] [9] control flow graphs. As the examples in section 7 illustrate, our algorithm optimizes reducible programs much more extensively than previous algorithms. In the special case of a program without loops, the code generated by our algorithm is provably "optimal" in the technical sense explained in section 8. This degree of optimization is achieved while improving the worst-case time bound, when compared with previous algorithms that perform extensive optimization.

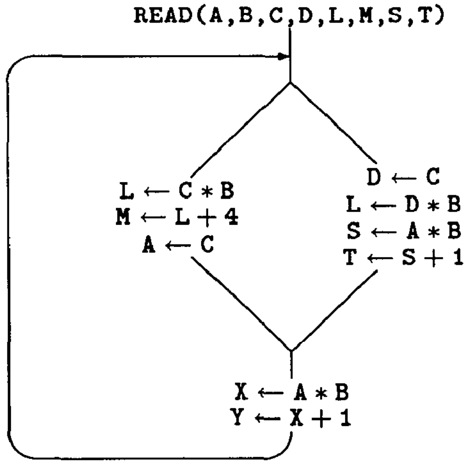


Figure 1: Original Program

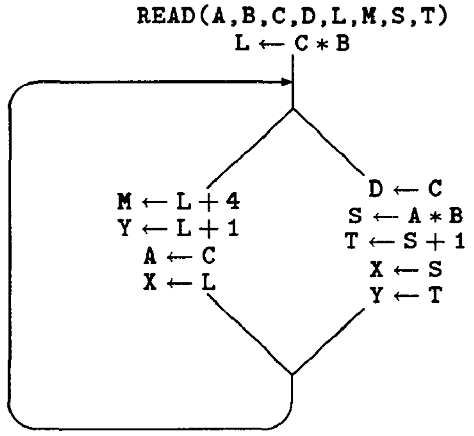


Figure 2: Improved Program

Figures 1 and 2 illustrate the optimization achieved by our algorithm. It removes a redundancy between computations of A*B, identifies a common use of C*B and removes it from the loop, and finally removes a partial redundancy [13] between computations of X+1.

Suppose the program has already been translated into

Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, California (January 1988)

a standard intermediate form that uses temporary names for the values of the subexpressions of complex expressions. In an intermediate text expression, the operands are always variable or constants. The worst-case time bound of our algorithm can be stated, somewhat pessimistically, as $O(C * N * E)$, where C is the number of computations in the program's intermediate text, N is the number of nodes in the control flow graph, and E is the number of edges in this graph. The worst pessimism comes from N. This factor does *not* represent anything done for every node in the graph; rather, it represents the possible increase in the number of computations present at intermediate stages in our algorithm. In the worst case that can be constructed, the expansion factor is N. Practical expansion factors will be much smaller. There is also some pessimism in the factor E because the algorithm includes searches that might explore most of the graph but are likely in practice to explore small fractions of it. Section 7 includes a comparison of bounds.

2 Example

This section sketches our algorithm by working through figure 1 in detail. This example illustrates several of the new techniques used by the algorithm, but necessarily does not discuss how to handle all cases. Sections 3-6 contain a complete description of the algorithm, and section 7 relates it to previous work.

Our algorithm will first change this program into a simpler representation. There is only one assignment for each variable in the new program. This transformation introduces many new names for each separate variable in the original program, at least one name for every assignment statement. In this particular example, the new names introduced for the variable V are of the form V_i for some integer i. In general, this subtask may be accomplished in various ways, as discussed in section 4.3. The phrase *single assignment* is already in use for programs that assign to each variable only once when running. Dynamically, a program with loops may assign to the same variable many times, even if only one assignment appears in the program text. Our transformation puts the program into *static* single assignment form, which we will abbreviate to *SSA form.*

To attain SSA form, we introduce a new type of assignment statement at some of the join nodes of the program, where a *join* node is any node that has two or more inedges. We consider the case of two inedges and call them "left" and "right" for ease of visualization. Then the new assignment will have the form $V_i \leftarrow \phi(V_j, V_k)$. If control reaches the join node along the left branch, then V_i is assigned the value of V_j; if control reaches along the right, then V_i is assigned the value of V_k. This transformation is illustrated in figure 3. This representation allows our algorithm to effectively manipulate value numbers when it manipulates lexical names. Some of our transformations preserve SSA form, and those that do not are immediately followed by restoration of SSA form.

Eliminating redundancies can have second order effects. Eliminating one computation can provide an opportunity to eliminate others. This motivates the notion of *rank.* Ranks are like heights of expression trees. Most of the computations that produce operands for an expression are assigned a rank lower than the rank of the expression. (The exceptions involve program loops.) We can therefore process the program in order of increasing ranks and be sure of getting most of the second order effects. The ranks are shown in parentheses to the left of each computation in figure 3. The subalgorithm to compute the ranks is in section 4.4.

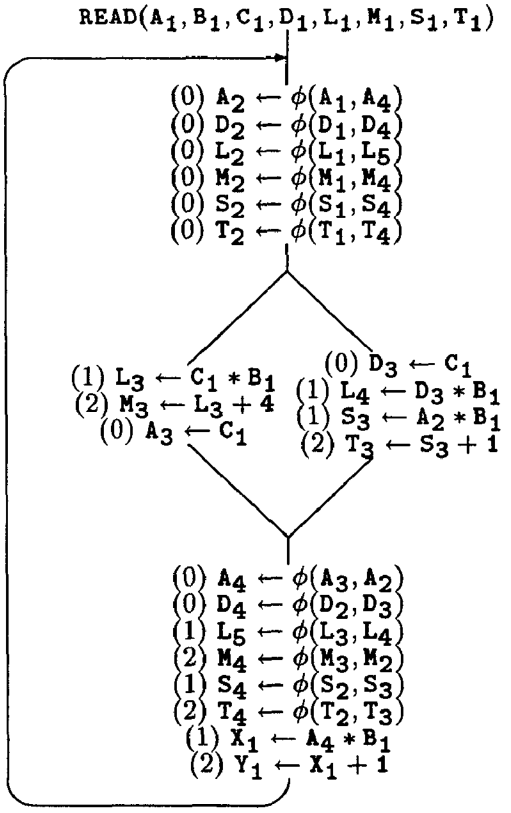

$READ(A_1,B_1,C_1,D_1,L_1,M_1,S_1,T_1)$
(0) $A_2 \leftarrow \phi(A_1,A_4)$
(0) $D_2 \leftarrow \phi(D_1,D_4)$
(0) $L_2 \leftarrow \phi(L_1,L_5)$
(0) $M_2 \leftarrow \phi(M_1,M_4)$
(0) $S_2 \leftarrow \phi(S_1,S_4)$
(0) $T_2 \leftarrow \phi(T_1,T_4)$
(1) $L_3 \leftarrow C_1 * B_1$
(2) $M_3 \leftarrow L_3 + 4$
(0) $A_3 \leftarrow C_1$
(0) $D_3 \leftarrow C_1$
(1) $L_4 \leftarrow D_3 * B_1$
(1) $S_3 \leftarrow A_2 * B_1$
(2) $T_3 \leftarrow S_3 + 1$
(0) $A_4 \leftarrow \phi(A_3,A_2)$
(0) $D_4 \leftarrow \phi(D_2,D_3)$
(1) $L_5 \leftarrow \phi(L_3,L_4)$
(2) $M_4 \leftarrow \phi(M_3,M_2)$
(1) $S_4 \leftarrow \phi(S_2,S_3)$
(2) $T_4 \leftarrow \phi(T_2,T_3)$
(1) $X_1 \leftarrow A_4 * B_1$
(2) $Y_1 \leftarrow X_1 + 1$

Figure 3: Program in SSA Form, with Ranks Assigned

In a program in SSA form, trivial assignments (i.e., one variable gets the value of another) can be removed simply by changing all uses of the target of the assignment to uses of the variable on the right-hand side of the assignment. In figure 4, the assignments to A_3 and D_3 have been removed. The ϕ functions that used these variables as operands have been changed along with the other expressions.

Phase 2 of our algorithm eliminates redundant computations by looping over ranks. For each rank, the nodes in the program graph are processed in an order established by the graph analysis explained in section 4.1. In the present example, the only edge drawn with an arrowhead in the figures is also the only *backedge* in a loop. With backedges ignored, the graph becomes a DAG and may be topologically sorted (i.e., the nodes may be listed in such a way that the source of each edge comes before the destination of that edge). This is *topsort* order, and we will process nodes in the reverse of topsort order. In the figures, we work up from the bottom.

The first computation we consider is the assignment to X_1 in figure 4; this is the lowest ranked computation within the last node in topsort order. Intuitively, we move this

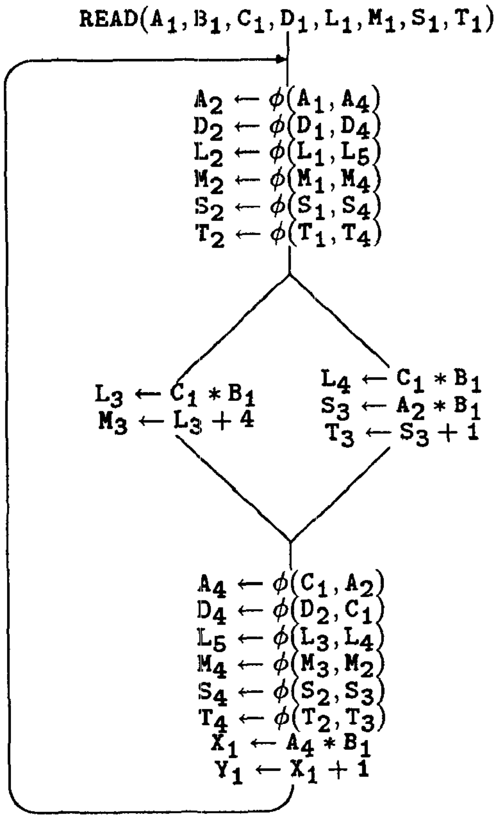

READ(A_1,B_1,C_1,D_1,L_1,M_1,S_1,T_1)
$A_2 \leftarrow \phi(A_1,A_4)$
$D_2 \leftarrow \phi(D_1,D_4)$
$L_2 \leftarrow \phi(L_1,L_5)$
$M_2 \leftarrow \phi(M_1,M_4)$
$S_2 \leftarrow \phi(S_1,S_4)$
$T_2 \leftarrow \phi(T_1,T_4)$
$L_3 \leftarrow C_1 * B_1$
$M_3 \leftarrow L_3 + 4$
$L_4 \leftarrow C_1 * B_1$
$S_3 \leftarrow A_2 * B_1$
$T_3 \leftarrow S_3 + 1$
$A_4 \leftarrow \phi(C_1,A_2)$
$D_4 \leftarrow \phi(D_2,C_1)$
$L_5 \leftarrow \phi(L_3,L_4)$
$M_4 \leftarrow \phi(M_3,M_2)$
$S_4 \leftarrow \phi(S_2,S_3)$
$T_4 \leftarrow \phi(T_2,T_3)$
$X_1 \leftarrow A_4 * B_1$
$Y_1 \leftarrow X_1 + 1$

Figure 4: A_3, D_3 Removed

computation Q to nodes earlier in topsort order (upward in the figures). We move Q as far as possible without changing the semantics of the program. Because Q is at a node with two inedges, we split it into copies Q_1 and Q_2 as it moves upward. Each copy will have its own chance to move upward later, and one or both may eventually be found to be redundant. The original computation Q will be redundant as soon as both copies have been placed in predecessor nodes. Because we will eliminate global redundancies of the current rank after moving computations of the current rank, it will do no harm to leave Q at its original node after placing the copies. Figure 5 shows Q_1 and Q_2 at the inedges of the join node. Each copy Q_i assigns to a new temporary variable Z_i. Ordinarily, a computation cannot move past an assignment to one of its operands, and A_4 is assigned to at the join node. However, the assignment to A_4 is from a ϕ function. We can move a computation past this particular type of assignment by renaming the operands to reflect the values from which each operand was formed. The general technique is described in section 5.1.2.

We can recognize that there are two computations in the same node with identical operands and operators. Thanks to SSA form, this identity implies that they must compute the same value, and one can be eliminated simply by replacing it with a trivial assignment from the output of the other. The replacement by the trivial assignments is shown in figure 6.

We remove the new trivial assignments as soon as they are introduced. In figure 7, those trivial assignments have been removed. Also shown in that figure is the result of copying the calculation of $C_1 * B_1$ into the branch node.

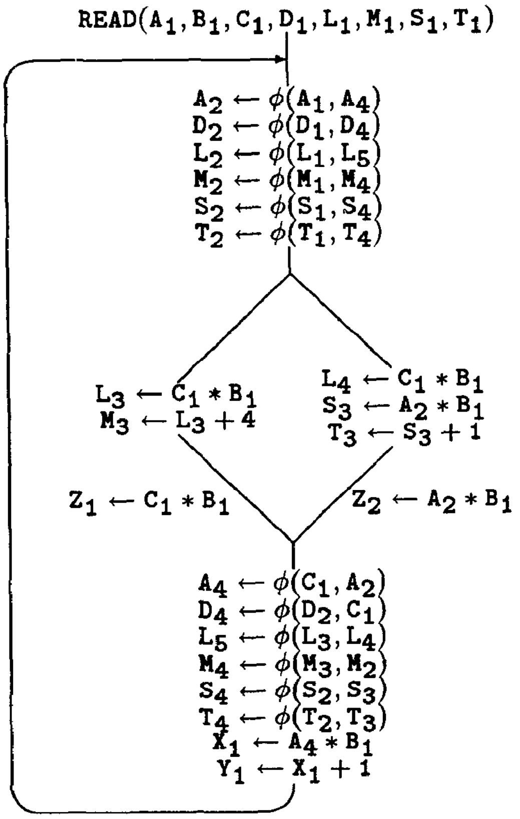

READ(A_1,B_1,C_1,D_1,L_1,M_1,S_1,T_1)
$A_2 \leftarrow \phi(A_1,A_4)$
$D_2 \leftarrow \phi(D_1,D_4)$
$L_2 \leftarrow \phi(L_1,L_5)$
$M_2 \leftarrow \phi(M_1,M_4)$
$S_2 \leftarrow \phi(S_1,S_4)$
$T_2 \leftarrow \phi(T_1,T_4)$
$L_3 \leftarrow C_1 * B_1$
$M_3 \leftarrow L_3 + 4$
$L_4 \leftarrow C_1 * B_1$
$S_3 \leftarrow A_2 * B_1$
$T_3 \leftarrow S_3 + 1$
$Z_1 \leftarrow C_1 * B_1$
$Z_2 \leftarrow A_2 * B_1$
$A_4 \leftarrow \phi(C_1,A_2)$
$D_4 \leftarrow \phi(D_2,C_1)$
$L_5 \leftarrow \phi(L_3,L_4)$
$M_4 \leftarrow \phi(M_3,M_2)$
$S_4 \leftarrow \phi(S_2,S_3)$
$T_4 \leftarrow \phi(T_2,T_3)$
$X_1 \leftarrow A_4 * B_1$
$Y_1 \leftarrow X_1 + 1$

Figure 5: Splitting of $X_1 \leftarrow A_4 * B_1$

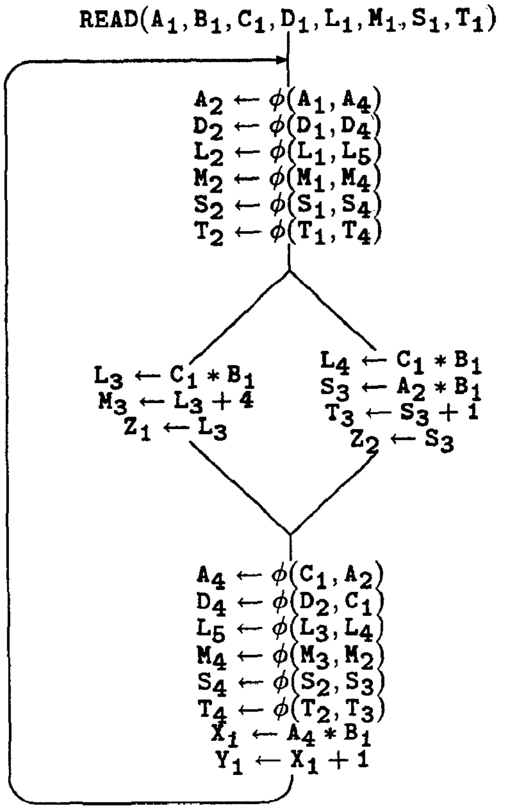

READ(A_1,B_1,C_1,D_1,L_1,M_1,S_1,T_1)
$A_2 \leftarrow \phi(A_1,A_4)$
$D_2 \leftarrow \phi(D_1,D_4)$
$L_2 \leftarrow \phi(L_1,L_5)$
$M_2 \leftarrow \phi(M_1,M_4)$
$S_2 \leftarrow \phi(S_1,S_4)$
$T_2 \leftarrow \phi(T_1,T_4)$
$L_3 \leftarrow C_1 * B_1$
$M_3 \leftarrow L_3 + 4$
$Z_1 \leftarrow L_3$
$L_4 \leftarrow C_1 * B_1$
$S_3 \leftarrow A_2 * B_1$
$T_3 \leftarrow S_3 + 1$
$Z_2 \leftarrow S_3$
$A_4 \leftarrow \phi(C_1,A_2)$
$D_4 \leftarrow \phi(D_2,C_1)$
$L_5 \leftarrow \phi(L_3,L_4)$
$M_4 \leftarrow \phi(M_3,M_2)$
$S_4 \leftarrow \phi(S_2,S_3)$
$T_4 \leftarrow \phi(T_2,T_3)$
$X_1 \leftarrow A_4 * B_1$
$Y_1 \leftarrow X_1 + 1$

Figure 6: Local Redundancies Removed

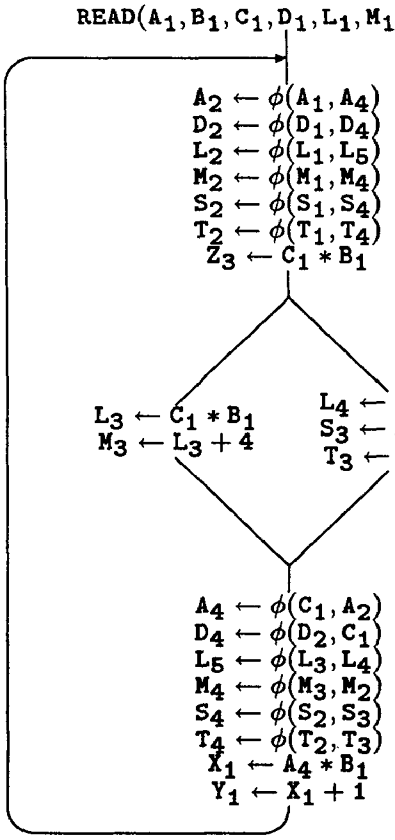

$READ(A_1,B_1,C_1,D_1,L_1,M_1,S_1,T_1)$
$A_2 \leftarrow \phi(A_1,A_4)$
$D_2 \leftarrow \phi(D_1,D_4)$
$L_2 \leftarrow \phi(L_1,L_5)$
$M_2 \leftarrow \phi(M_1,M_4)$
$S_2 \leftarrow \phi(S_1,S_4)$
$T_2 \leftarrow \phi(T_1,T_4)$
$Z_3 \leftarrow C_1 * B_1$
$L_3 \leftarrow C_1 * B_1$
$M_3 \leftarrow L_3 + 4$
$L_4 \leftarrow C_1 * B_1$
$S_3 \leftarrow A_2 * B_1$
$T_3 \leftarrow S_3 + 1$
$A_4 \leftarrow \phi(C_1,A_2)$
$D_4 \leftarrow \phi(D_2,C_1)$
$L_5 \leftarrow \phi(L_3,L_4)$
$M_4 \leftarrow \phi(M_3,M_2)$
$S_4 \leftarrow \phi(S_2,S_3)$
$T_4 \leftarrow \phi(T_2,T_3)$
$X_1 \leftarrow A_4 * B_1$
$Y_1 \leftarrow X_1 + 1$

Figure 7: $C_1 * B_1$ Copied to Branch Node

Copying can occur here because the same value is calculated in *both* successors of the branch node. The copy assigns to a new temporary Z_3.

Any computation of the current rank that reaches a loop header (i.e., a node that is the destination of a backedge) may be copied to the node immediately preceding the loop, provided that the computation is redundant along each path through the loop body and back to the header. In this case, the computation of $C_1 * B_1$ is copied out of the loop, as shown in figure 8. The general subalgorithm is in section 5.2.2.

Now that rank 1 computations have been copied to their earliest possible locations, it is time eliminate every computation Q of rank 1 that is *redundant* because there are equivalent computations $E_1, \ldots, E_n$ such that every path from the start of the program to Q performs one of the E_i before reaching Q. The redundant computation is replaced by a use of a new temporary. At the location of each E_i, an assignment to the new temporary from the output of E_i is added. (If $n = 1$, then the redundant computation may be simply replaced by a use of the output of E_1, without a new temporary.) The computations of the current rank are checked in any convenient order, and each redundant computation is eliminated before proceeding. The details of this subalgorithm are presented in section 5.4.

Identifying computations by the variables they assign to, we might find the redundant computations in figure 8 in the order X_1, L_3, L_4, Z_3. The results of finding them in this order are essentially as in figure 9, where there are now two assignments to the variable Z_5. We restore SSA form

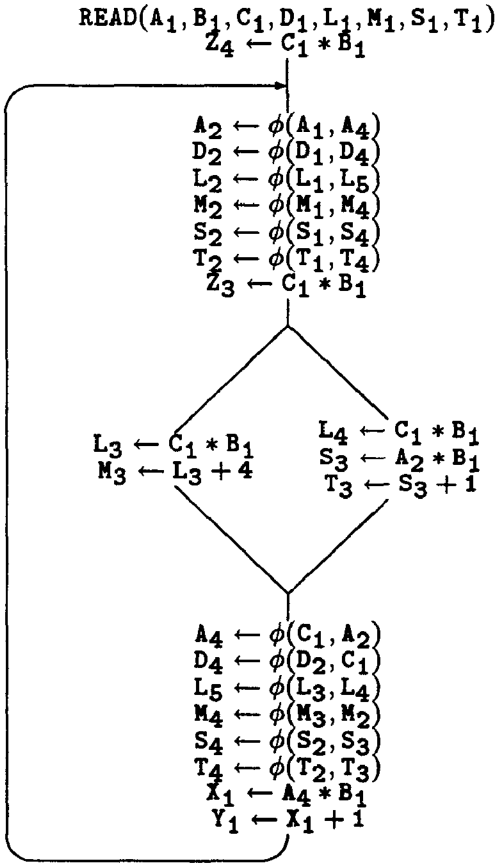

$READ(A_1,B_1,C_1,D_1,L_1,M_1,S_1,T_1)$
$Z_4 \leftarrow C_1 * B_1$
$A_2 \leftarrow \phi(A_1,A_4)$
$D_2 \leftarrow \phi(D_1,D_4)$
$L_2 \leftarrow \phi(L_1,L_5)$
$M_2 \leftarrow \phi(M_1,M_4)$
$S_2 \leftarrow \phi(S_1,S_4)$
$T_2 \leftarrow \phi(T_1,T_4)$
$Z_3 \leftarrow C_1 * B_1$
$L_3 \leftarrow C_1 * B_1$
$M_3 \leftarrow L_3 + 4$
$L_4 \leftarrow C_1 * B_1$
$S_3 \leftarrow A_2 * B_1$
$T_3 \leftarrow S_3 + 1$
$A_4 \leftarrow \phi(C_1,A_2)$
$D_4 \leftarrow \phi(D_2,C_1)$
$L_5 \leftarrow \phi(L_3,L_4)$
$M_4 \leftarrow \phi(M_3,M_2)$
$S_4 \leftarrow \phi(S_2,S_3)$
$T_4 \leftarrow \phi(T_2,T_3)$
$X_1 \leftarrow A_4 * B_1$
$Y_1 \leftarrow X_1 + 1$

Figure 8: $C_1 * B_1$ Copied Above Loop

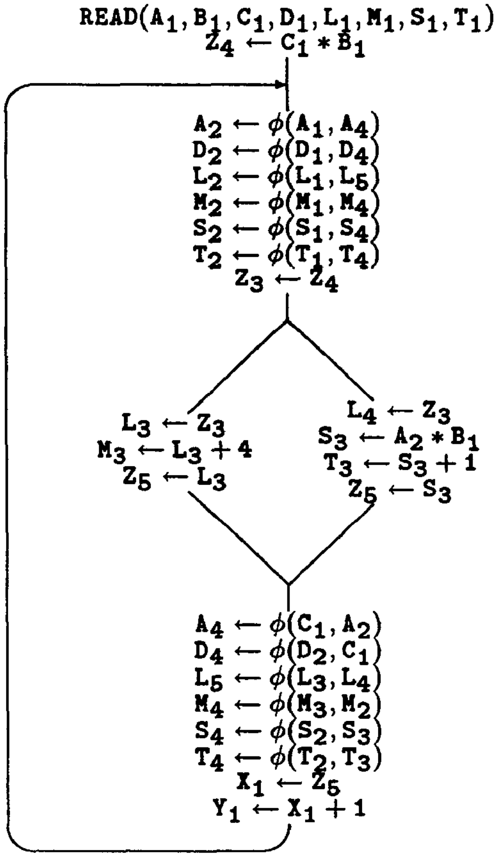

$READ(A_1,B_1,C_1,D_1,L_1,M_1,S_1,T_1)$
$Z_4 \leftarrow C_1 * B_1$
$A_2 \leftarrow \phi(A_1,A_4)$
$D_2 \leftarrow \phi(D_1,D_4)$
$L_2 \leftarrow \phi(L_1,L_5)$
$M_2 \leftarrow \phi(M_1,M_4)$
$S_2 \leftarrow \phi(S_1,S_4)$
$T_2 \leftarrow \phi(T_1,T_4)$
$Z_3 \leftarrow Z_4$
$L_3 \leftarrow Z_3$
$M_3 \leftarrow L_3 + 4$
$Z_5 \leftarrow L_3$
$L_4 \leftarrow Z_3$
$S_3 \leftarrow A_2 * B_1$
$T_3 \leftarrow S_3 + 1$
$Z_5 \leftarrow S_3$
$A_4 \leftarrow \phi(C_1,A_2)$
$D_4 \leftarrow \phi(D_2,C_1)$
$L_5 \leftarrow \phi(L_3,L_4)$
$M_4 \leftarrow \phi(M_3,M_2)$
$S_4 \leftarrow \phi(S_2,S_3)$
$T_4 \leftarrow \phi(T_2,T_3)$
$X_1 \leftarrow Z_5$
$Y_1 \leftarrow X_1 + 1$

Figure 9: Rank 1 Redundancies Removed

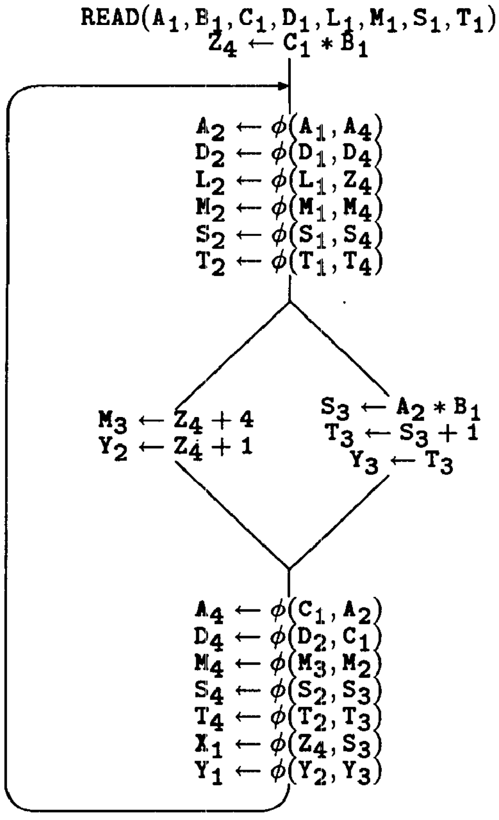

$READ(A_1,B_1,C_1,D_1,L_1,M_1,S_1,T_1)$
$Z_4 \leftarrow C_1 * B_1$
$A_2 \leftarrow \phi(A_1,A_4)$
$D_2 \leftarrow \phi(D_1,D_4)$
$L_2 \leftarrow \phi(L_1,Z_4)$
$M_2 \leftarrow \phi(M_1,M_4)$
$S_2 \leftarrow \phi(S_1,S_4)$
$T_2 \leftarrow \phi(T_1,T_4)$
$M_3 \leftarrow Z_4 + 4$
$Y_2 \leftarrow Z_4 + 1$
$S_3 \leftarrow A_2 * B_1$
$T_3 \leftarrow S_3 + 1$
$Y_3 \leftarrow T_3$
$A_4 \leftarrow \phi(C_1,A_2)$
$D_4 \leftarrow \phi(D_2,C_1)$
$M_4 \leftarrow \phi(M_3,M_2)$
$S_4 \leftarrow \phi(S_2,S_3)$
$T_4 \leftarrow \phi(T_2,T_3)$
$X_1 \leftarrow \phi(Z_4,S_3)$
$Y_1 \leftarrow \phi(Y_2,Y_3)$

Figure 10: Movement of $Y_1 \leftarrow X_1 + 1$

and remove trivial assignments before proceeding.

Removal of trivial assignments has a second order effect on the ϕ function that computes L_5 in figure 9. The removal process renames both operands of that function to Z_4, so this ϕ fuction can be replaced by a new trivial assignment $L_5 \leftarrow Z_4$ (which will then be removed). This is one example of the many ways that subalgorithms of our algorithm provide opportunities for each other.

It is now time to process the program for the next rank. The computation of interest is the assignment to Y_1. (The operand X_1 is of lower rank and has already been moved out of the block.) In figure 10, we split the computation and move it up each side. Since it matches the computation of $S_3 + 1$ on the right branch, it is replaced with a trivial assignment on the right branch.

The trivial assignment to Y_3 must now be removed (not shown). Phase 3 then eliminates ϕ's and returns the program to conventional form, with no limit on the number of assignments to any variable. The output is as was shown in figure 2.

3 Overview of the Algorithm

Figure 11 displays the algorithm in pseudocode. The section numbers in parentheses on some lines indicate where to find detailed explanations. For the moment, the previous section's intuitive explanations should suffice for concepts like backedges, topsort order, and loop header nodes. The pseudocode also mentions landing pads of loops. These are nodes added to the program control flow graph to provide convenient places to put code that is moved out of loops.

```
/* Phase 1 */                                        (4)

Perform depth-first search, noting
  backedges and topsort order.                       (4.1)
Insert some nodes into
  control flow graph.                                (4.2)
Translate to SSA form.                               (4.3)
Assign ranks.                                        (4.4)
Remove trivial assignments.                          (4.5)

/* Phase 2 */                                        (5)

for each rank R = 0,1,...  do
  for each node n
    (in reverse topsort order) do
    select on the node type of n
      case normal:                                   (5.1)
        Move any movable computations of
        rank R from successors into n.
        Identify any computations of
        rank R that may be movable
        from n into predecessors.
      case loop header:                              (5.2)
        Proceed much as in normal case,
        but move certain computations of
        rank R out of the loop.
      case landing pad:                              (5.3)
        Proceed much as in normal case,
        but move certain computations of
        rank R from loop exits to n.
    end
  end
  Eliminate globally redundant
    computations of rank R.                          (5.4)
end

/* Phase 3 */                                        (6)

Translate from SSA form.
Eliminate empty nodes.
```

Figure 11: Overview of the Algorithm

4 Phase 1: Preprocess the Program

First we do some preliminary analysis. Then we insert some empty nodes into the graph at various convenient places. These nodes will become places to which code can be moved. Finally, we perform special transformations and analyses that will make the program easy to manipulate in Phase 2.

It is customary to describe optimizing transformations under various simplifying assumptions about the program text. For example, it is assumed that assigning to one variable does not affect the value of any other variable. It is also assumed that program statements are either simple assignments

$$\langle\texttt{variable}\rangle \leftarrow \langle\texttt{expression}\rangle$$

or simple tests that branch on Boolean variables. Complications like procedure calls or the READ statement in figure 1 pose some subtle problems [16] that have given rise to a substantial literature on the analysis of aliasing, side effects, and so on. Applying our algorithm to programs with realistic complications is no more difficult than applying previous algorithms, so we make the customary assumptions freely. In section 2, READ was treated as a set of assignments from arbitrary distinct constants. This was correct in context. In general, READ is like a procedure call that both uses and modifies a file parameter while modifying the parameters representing variables read in.

4.1 Analyze the Graph

As is usual, we assume that the program text has been grouped into basic blocks and that the control flow graph has been built with a node for each basic block and an edge for each transfer of control. We assume that all nodes are reachable from the node representing program entry, and that each node has at most two outedges. These last assumptions are not crucial, but they are convenient in several places.

A *backedge* of the program control flow graph is any edge whose destination is an ancestor of its source in the tree defined by a depth-first search [18] rooted at the program entry node. (In a reducible graph, the set of backedges does not depend on the arbitrary choices made during the depth-first search [9].)

With backedges ignored, the graph becomes a DAG and may be topologically sorted (i.e., the nodes may be listed in such a way that the source of each edge comes before the destination of that edge). The sorting can be done during the depth-first search by noting the order in which subsearches terminate. Hecht and Ullman [10] show that this order (sometimes called "endorder" or "postorder") is the reverse of a topological order. (Though stated only for reducible graphs, Lemma 4 of [10] actually applies to all graphs.) Throughout this paper, *topsort* order will be the reverse of the order in which subsearches terminate.

A *loop header* is any node that is a destination of a backedge. Given a loop header $\mathtt{h}$ reached by backedges from nodes $\mathtt{s}_1, \ldots, \mathtt{s}_k$, the corresponding *loop body* consists of all nodes $\mathtt{u}$ such that there is a path of the form $\mathtt{h} \xrightarrow{*} \mathtt{u} \xrightarrow{*} \mathtt{s}_i$ that traverses no backedges. An edge from a node in a loop body to a node not in the loop body is an *exit edge* of the loop. The destination of an exit edge is an *exit node.*

At each loop header, we wish to keep a list of loop entrance and exit edges. There are several ways to compute the loop body and loop exit edges for each loop without tracing paths. A simple way that depends upon the program being reducible is as follows. (In a reducible graph, the edges that enter a loop are just the inedges of the header that are not backedges.)

We first determine the nodes within the loop with header h by searching the graph, starting at the sources of the backedges to h. The search follows edges backwards (from destination to source), and ignores backedges. Each branch of the search that reaches h (or a previously visited node) terminates. Each node visited is marked as being in the loop body for h. Exit edges can be determined by examining nodes in the loop body for any outedges that do not go to other nodes in the loop body.

The worst-case time for this technique is within the overall time of our algorithm. More efficient, but more complicated, ways to compute loop bodies and loop exit edges can be obtained by adaptations of known analytic techniques [19]. This is done by traversing the loops innermost to outermost (by visiting the headers in reverse topsort order) and by merging the set of nodes in an inner loop into the next outer loop (by means of an efficent union-find algorithm).

4.2 Modify the Graph

The modifications made in this section can be performed in linear time, and they add a linearly bounded number of nodes and edges to the graph. The modifications are so slight that the results of previous analysis can easily be updated to allow for them. We do the analysis first because we need to identify loops.

We give each loop a *landing pad* representing entry to the loop from outside, as distinct from looping back. Landing pad insertion is illustrated in figures 12 and 13 for a while loop (i.e., a loop whose header has an outedge that is an exit from the loop), which is the only case that is not completely straightforward. Following [8], we duplicate the old header (forming two TEST nodes) and make the loop look like an until loop guarded by a TEST. Then the start of the loop body becomes the header and receives a landing pad. (Making while look like until permits more extensive optimizations.) Formally, each loop header is given a new predecessor that will be its only predecessor outside the loop. The old edges entering the loop at the header become the inedges of this new predecessor. Because landing pads are new nodes added for every loop header, no node will be both a landing pad and a loop header.

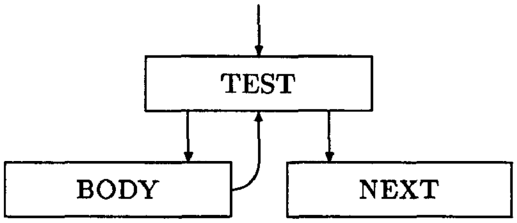


Figure 12: While Loop Without Landing Pad

Any edge that goes directly from a node with more than one outedge (*a branch node*) to a node with more than one inedge (*a join node*) is split into a pair of edges, one from the branch node to a new node and another from the new node to the join node.

Splitting edges will allow the algorithm to move a computation from a join node to each of its predecessors without running the risk of exposing that computation to control paths that go through the predecessor and not the original node. In figure 14, the insertion of a node between (c) and (b) will allow the computation at (b) to be moved without inserting it into the (c) to (d) control

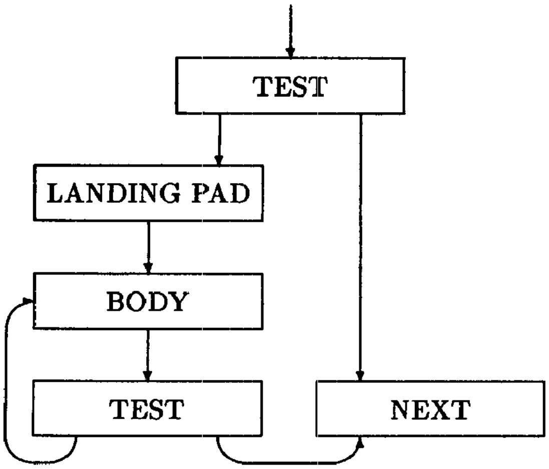


Figure 13: Addition of a Landing Pad to a While Loop

path. The computation is redundant along the (a) to (b) control path. If it is moved to a node along the (c) to (b) control path then it will be available at (b) regardless of which path enters (b). That movement will improve the (a) to (b) control path.

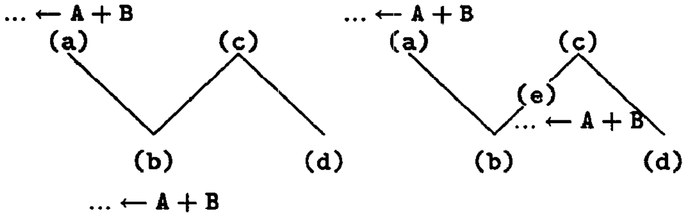


Figure 14: Insertion of a Node to Split an Edge

A *virtual edge* is added to the landing pad for each exit from the loop that that landing pad is associated with. The virtual edge goes from the landing pad to the exit node. (The list of exit edges was calculated in section 4.1 and is associated with the header node of the loop.) Virtual edges provide a mechanism for moving code past a loop without moving it into the loop, but most of the work we do will be along the *real* edges that were in the graph initially.

4.3 Translate to SSA Form

We rename variables throughout the program to put it into *static single assignment (SSA)* form: each variable is assigned to exactly once in the program text. A new type of assignment statement is added at join nodes, to indicate that a variable is assigned the value of one variable (if control enters along one inedge) or another variable (if control enters along another inedge).

Each mention of a variable V will be replaced by a mention of one of the new names for V. The various new names will be denoted V_i where i is an integer. After renaming, every point in the program will be *reached* (in the sense explained below) by exactly one of the names for V. Intuitively, the name that reaches a point represents whatever value V has when control reaches that point. New names are to be generated and assigned so as to satisfy the following *SSA rules*, which are stated for join nodes that have just two inedges (called "left" and "right") to minimize notation:

1. Each variable V at the start of the program is assigned the name V_0 there. This name reaches the start of the program and any other point p such that every path from the start to p is free of assignments to names of V.

2. Each assignment to V is replaced with an assignment to V_i for some unique positive i. The name V_i is the one that reaches the point p immediately after the assignment, and any other point q such that every path from p to q is free of assignments to names of V.

3. The name of V that reaches an edge in the graph is the one that reaches the end of the code associated with the source of the edge.

4. At any node in the graph where the same name for V reaches all inedges of the node, that name is the one that reaches the entry point p of the node, and any other point q such that every path from p to q is free of assignments to names of V. (In particular, the entry point of any node with one predecessor is reached by the name that reaches the end of the predecessor.)

5. At any join node in the graph where two different names for V reach the inedges of the node, a new assignment is inserted. The new assignment has the form $V_k \leftarrow \phi(V_i, V_j)$, where V_i and V_j are the two names of V that reach the left and right inedges of the node and V_k is a unique new name. The name V_k is the one that reaches the entry point p of the node, and any other point q such that every path from p to q is free of assignments to names of V.

   The meaning of $V_k \leftarrow \phi(V_i, V_j)$ at a node u is that if control reaches u by the left inedge, then $V_k \leftarrow V_i$. If control reaches by the right inedge, then $V_k \leftarrow V_j$.

6. If V_i is the new name of V that reaches a point in the transformed program, then the value of V_i is always the same as the value of V at the same point in the original program.

   The results of renaming are illustrated in figure 15.

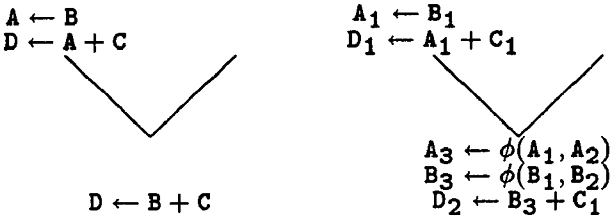


Figure 15: Example of Renaming

The foregoing specifications for SSA form can be implemented in several ways. The next subsection explains a simple way to use the rules to achieve SSA form, followed by a subsection explaining more complex ways that lead to more extensive optimization.

4.3.1 Simple SSA Form

The following algorithm is essentially from [8], where the formulation looks different because explicit ϕ functions are not used.

Visit the nodes in topsort order, performing the following steps for each node:

1. If the node is a loop header, then insert a ϕ function for each variable V. The target of the assignment is a new name for V. The first operand of the ϕ function is the name of V that reaches the node from the landing pad. There is another operand for each backedge and these will be filled in later.
2. If the node is a join node that is not a loop header, then apply rule 5 of the SSA rules.
3. For each assignment in the node, apply rule 2 of the SSA rules.
4. If the node is the source of a backedge, then the names that reach the bottom of the node are used to fill in operands of ϕ functions at the destination of the backedge.

These rules define a renaming to SSA form in which distinct variables have distinct names. This *simple* SSA form may be used, but the time bounds of our overall algorithm leave room for more sophisticated renaming explained in the next subsection.

4.3.2 Reduced SSA Form

The algorithm for simple SSA form sometimes assigns more names than are needed, and this can cause opportunities for optimization to be missed when programs have loops. Consider figure 16. The simple algorithm will give separate sets of names to P and Q, and the redundancy in this program will not be eliminated. A more ambitious algorithm might notice that P and Q always have the same value and can therefore share storage.

```
P ← 1
Q ← 1
while(...) do
    P ← P + 1
    Q ← Q + 1
end
```

Figure 16: Example of More Names than Necessary

Section 7.2 will briefly discuss some previous work on recognizing equivalences among program variables. Finding all equivalences is an undecidable problem, but various decidable subproblems are known. In effect, these algorithms translate a program to simple SSA form and then compute a set of pairs of variables, such that the value assigned to one variable in a pair is always the same as the value assigned to the other. If variables A and B are paired, and if the computation assigning to B *dominates* the computation assigning to A (by coming before it on all paths through the program), then it is safe to replace the original computation for A by the statement A ← B. For any given means of recognizing equivalences, a program is in *reduced* SSA form if it is in SSA form and if all such replacements have been done. The optimization in section 4.5 will then be applicable.

In figure 16, any equivalence algorithm that recognizes the equivalences between the various SSA names of P and Q will lead to a reduced SSA form in which each name Q_i of Q is defined by assigning from the corresponding name P_i of P. The optimization in section 4.5 will effectively merge these names.

4.4 Assign a Rank to Each Computation

Moving forward over the program, without traversing any backedges of loops, we will assign a *rank* to any variable or expression appearing at any point in the program. When computing the rank of an expression in a loop header, we may need to use a rank value for an operand variable that reaches the loop header along a backedge. The rank of such a variable will not have been computed yet, but we can recognize this situation and can use the value 0 instead of the rank. In the following rules, the *available* rank is either the rank (if already computed) or 0 (otherwise). Because the program is in SSA form, the rank of any variable or expression is unambiguously defined as:

- 0, if the variable is an entry point name V_0.
- the rank of the expression assigned to the variable, if it is assigned to within the program.
- 0, if the expression is a constant.
- the maximum of the available ranks of the operands, if the expression is a variable or a ϕ function.
- 1 + the maximum of the available ranks of the operands, if the expression is not a constant, a variable, or a ϕ function.

The *rank of a computation* is the rank of the variable i' assigns to. The steps in phase 2 will be performed once foı each rank assigned in the program (starting with rank 0). This ensures that before any computation is processed, all of the computations that produce operands for that computation have already been processed.

The assignment of ranks has an interesting property that enhances the performance of our algorithm. There is no need to maintain the order of computations within a basic block. All that is required is to remember the rank of each computation. Code can be generated later by sorting the computations by rank, earliest first. Code for computations of the same rank can be generated in any order. This is useful because we will frequently add, delete, or look for computations in a block. All of these operations are easily implemented with a hash table. The *local computation table (LCT)* is maintained for each node in the program flow graph. The LCT contains the set of computations that occur at that node. It is primarily accessed in three ways, as follows:

- We can loop over all the computations.
- We can loop over all the computations of a given rank
- Given an expression, we can loop over all the computations with right-hand sides identical to the expressicn.

In all cases, the time required to find n computations i. $O(n)$.

4.5 Remove Trivial Assignments

Assignment statements that have trivial right-hand sides (only a single variable) have a special meaning for programs in SSA form. These statements can be thought of as assertions that the two variable names (the one mentioned on the left side and the one mentioned on the right side) represent the same value. The initial list of trivial assignments to be removed includes any that were originally in the program as well as any that were added by recognizing equivalences. This *worklist* may be maintained in any convenient way, and removal of the next item on the list may cause other items to be added.

Given a trivial assignment $A \leftarrow B$, we replace each mention of one variable (including those in ϕ functions) by a mention of the other. To facilitate this replacement, we will maintain a list of uses of each variable in the program. Whichever of A and B has the shorter list of uses will be replaced by the other. Every time a variable is replaced it must be replaced by a variable which will be used at least twice as often. Thus, the number of times that a variable can be the survivor in removal of trivial assignments is bounded by the logarithm of the total number of uses.

Renaming of operands of a computation may make the right-hand side become identical with the right-hand side of another computation in the same node. There may also be several computations in a node that happen to have identical right-hand sides initially. However they arise, such matchings are *local redundancies.* Thanks to SSA form, the matching computations really are equivalent. We maintain a worklist that begins with the initial local redundancies in the SSA form of the program. This worklist may gain entries during removal of trivial assignments. Eliminating a local redundancy, on the other hand, creates a trivial assignment. Removal of trivial assignments and elimination of local redundancies feed each other's worklists until both processes quiesce with empty worklists. When both worklists are nonempty, it does not matter which list is serviced first.

Later steps in our algorithm will sometimes create trivial assignments or local redundancies. The later steps will always reactivate this step and then wait until this step has emptied its worklists.

The details of eliminating local redundancies are as follows. If the LCT of a node has n computations of the form $V_i \leftarrow E$ where E is a common expression and i runs from 1 to n, then the process of merging those computations is as follows:

1. Retain one of the computations, say $V_n \leftarrow E$. The retained computation is chosen to be one with maximum rank, among those being merged.
2. Replace each of the remaining computations $(i < n)$ by a trivial assignment of the form $V_i \leftarrow V_n$.
3. Put each trivial assignment on the worklist for removal.

In short, merging n computations of the form $V_i \leftarrow E$ results in one computation of the form $V_n \leftarrow E$ and $n - 1$ trivial assignments that will be removed.

Renaming of operands of ϕ functions may have other second order effects, beyond creating local redundancies:

- $V_i \leftarrow \phi(V_j, V_j)$ is replaced by $V_i \leftarrow V_j$.
- $V_i \leftarrow \phi(V_i, V_j)$ is replaced by $V_i \leftarrow V_j$.

In each case, the new assignment is added to the worklist of trivial assignments.

5 Phase 2: Eliminate Redundancies

As was explained in section 3, we loop over the ranks in increasing order. Within this loop, we first loop over the nodes in reverse topsort order, moving code and eliminating any local redundancies created by this motion. We then eliminate the global redundancies of the current rank.

The basic processing step for each node is to move any available computations from any of the successors into the current node, followed by making computations available in the current node to any of the predessors. Code motion is actually broken into two steps: copying a computation C into the LCT of the node into which C is being moved and then deleting C from the LCT of the node from which C is being moved. When we "move" a computation during the pass over the nodes in reverse topsort order, we only do the copying. The original computation is temporarily left in the original LCT, where it is now redundant. Along with other redundancies, this original computation will be eliminated by the global redundancy subphase. Several technical simplifications in record keeping are made possible by postponing deletion when code is moved.

The *movable computation table (MCT)* is maintained for each edge in the program flow graph that is not a backedge. The MCT contains all computations that are currently available for movement from the node that is the destination of the edge. Each entry in the MCT at an edge contains a computation C and a pointer to the corresponding computation B in the LCT of the destination of the edge. Some general points about this table should be borne in mind:

1. The MCT holds *candidates* for movement that may or not move. While in the MCT, these computations can be accessed in the same way as LCT computations. They also count as uses of their operands in removal of trivial assignments.
2. The MCT persists, like the LCT associated with each node. It may gain or lose individual computations, but it is never reset during the course of phase 2.
3. Visiting the nodes in reverse topsort order assures that the MCTs of all outedges of a node are filled in with movable computations of the proper ranks by the time that node is to be processed.

The actual processing for each node depends on the type of the node. There are three cases:

- Loop headers.
- Landing pads.
- All other nodes (*normal* nodes).

Recall from section 4.2 that no node can be both a loop header and a landing pad. We will consider the case of the normal nodes first and then consider the algorithms for the other types as special cases.

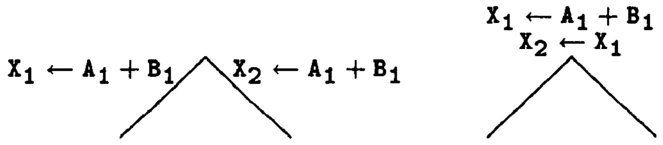
5.1 Processing of Normal Nodes

The work performed while visiting a normal node has two parts, corresponding to the following two subsections. The first part examines the movable computation tables for all the outedges of the node currently being visited and determines which computations may be moved into the node. The second part determines which computations may be made available for movement to a predecessor.

5.1.1 Move Computations from Successors

If the current node has only one outedge, then everything in the MCT of the outedge will be moved into the current node. If the current node has more than one outedge, then it has exactly two outedges. Let M_1 and M_2 be the MCTs. Consider each computation C_1 of the current rank in M_1. If the right-hand side of C_1 matches the right-hand side of a computation C_2 in M_2 (regardless of rank), then both computations will be moved into the current node. Similarly, the computations of the current rank in M_2 are checked against the computations in M_1 (regardless of rank).[1] The computations in one MCT that do not share right-hand sides with computations in the other MCT remain where they are for the present.

Moving computations into the current node may create local redundancies. An incoming computation may have the same right-hand side as one that is already there. If the current node has two outedges, then each incoming computation from one edge has the same right-hand side as an incoming computation from the other edge. However they arise, the new local redundancies are put on the worklist for removal. In figure 17, for example, the two computations of $A_1 + B_1$ have been moved into the node. The computation for X_2 has been changed to a trivial assignment from X_1 by the local redundancy removal algorithm in section 4.5.



$X_1 \leftarrow A_1 + B_1$
$X_2 \leftarrow A_1 + B_1$
$X_1 \leftarrow A_1 + B_1$
$X_2 \leftarrow X_1$

Figure 17: Moving Computations to a Node

The MCT entry for each moved computation is inspected to determine what action (if any) is required for the original LCT computation B that the MCT entry points to. If B has the current rank R, then no action is required. If B has a lower rank, then it is *promoted* to rank R so as to ensure that it will be eliminated during the global redundancy subphase. Once any required action has been taken, the MCT entry is deleted.

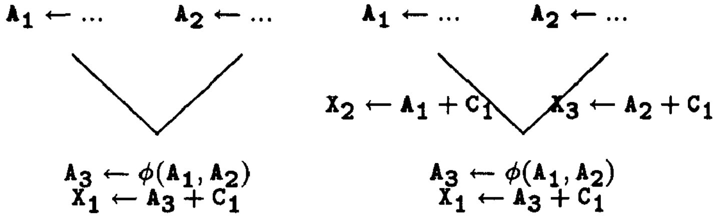
5.1.2 Identify Movable Computations

Computations in the LCT of the current node are identified as movable if they satisfy all of the following conditions:

- The computation has the current rank.
- The computation is not itself a ϕ function.
- No operand of the expression appears as the output of some other non-ϕ computation in the current node.

Any computation identified as movable is added to all of the MCTs of the inedges. This process may require that the computation C be modified in order to be moved. This modification is performed according the following technique, where E is the right-hand side of C:

1. If the current node has just one real inedge, then we generate a unique new variable U for each real or virtual inedge and put the new computation $U \leftarrow E$ into the MCT of that inedge.
2. If the current node is a join node, then all inedges are real and are treated essentially as above, but E may need to be modified in each new MCT entry. Any operand of E that is the output of a ϕ function in the current node must be replaced by the appropriate operand of that ϕ function for each inedge. We call this process *ϕ renaming*. In figure 18, for example, A_3 is defined by a ϕ function in a node and is used in the computation $A_3 + C_1$. Therefore A_3 is replaced by A_1 in the expression used in the left-edge MCT entry and replaced by A_2 in the expression used in the right-edge MCT entry.



$A_1 \leftarrow \ldots$
$A_2 \leftarrow \ldots$
$A_3 \leftarrow \phi(A_1, A_2)$
$X_1 \leftarrow A_3 + C_1$
$A_1 \leftarrow \ldots$
$A_2 \leftarrow \ldots$
$X_2 \leftarrow A_1 + C_1$
$X_3 \leftarrow A_2 + C_1$
$A_3 \leftarrow \phi(A_1, A_2)$
$X_1 \leftarrow A_3 + C_1$

Figure 18: Moving a Computation Up Each Edge

5.2 Processing for Loop Headers

The processing of a loop header is similar to that of a normal node. Computations are moved into a loop header exactly as described in section 5.1.1, but a different technique is used to identify computations that can move out of a loop header.

In the case of the normal node, computations are moved out of a node based only on local conditions. In the case of a loop header, we wish to move computations out of the loop (and into the landing pad). In order to accomplish this safely, we must verify that there exist computations in the loop that will make the result available

[1] There is a subtle reason for looping over all computations of the current rank in M_1 and *then* over all computations of the current rank in M_2. We need to find all pairs of computations (C_1 in M_1, C_2 in M_2) with matching right-hand sides, such that one computation C_i is of the current rank. (The other will be of at most the current rank.) Considering only pairs with both computations of the current rank would miss opportunities. To meet our time bound, on the other hand, these pairs must be found in time linear in the number of pairs found and in the number of computations of the current rank in either table.

whenever control enters the header from around the loop. To do this, we use a technique called *question propagation* to determine if there is some computation within the loop that will compute the same value as the computation we wish to remove.

5.2.1 Question Propagation

Question propagation searches as much of the graph as necessary, to determine whether a given computation Q is redundant. The search is like the movement of computations in section 5.1, but the rules for propagating a question about a computation are somewhat different from the rules for moving the computation itself. Virtual edges are ignored, and other differences will be explained shortly. A question about Q is a renaming of the variables in the right-hand side of Q. The renaming uses the variable names appropriate to wherever the question is at the moment; it is just the ϕ renaming of the expression explained in section 5.1.2.

The given computation Q is tentatively marked as *redundant* before question propagation begins. In the course of propagation, it may be marked as *not redundant.* Once Q has been marked as not redundant it can never be made redundant, so the search may terminate at that point. If Q is still marked as redundant when propagation terminates, then Q is indeed redundant and may be eliminated.

If any operand of Q is defined by a non-ϕ computation in the LCT of the node n containing Q, then the search is terminated and Q is marked not redundant. Otherwise, questions are propagated to predecessors of n. A question not answered within a node will be propagated to predecessors (with ϕ renaming as appropriate), unless it is stopped by one of the rules stated below.

The stopping rules for question propagation are stated below for two cases. During the current section's processing of a loop header, the *local* search is confined to the relevant loop body, and all questions originate at the loop header. During the subsequent elimination of redundancies, the *global* search will examine as much of the entire control flow graph as necessary, and all computations of the current rank are sources of questions. The rules for terminating the search are slightly different in these two cases. Another difference is the fact that the global search maintains a list (initially empty) of computations that might make Q redundant.

1. In local search, if a question is about to be propagated to predecessors of the loop header, then the search is terminated and Q is marked not redundant.

2. In global search, if a question is about to be propagated to predecessors of the program entry node, then the search is terminated and Q is marked not redundant.

3. In both searches, if questions are propagated to a branch node along more than one of its outedges and the questions are not the same on both outedges, then the search is terminated and Q is marked not redundant.

4. In both searches, if questions are propagated to a branch node along more than one of its outedges and the questions are the same on both outedges, then only the first question to arrive is propagated further.

5. In both searches, if a question is propagated to a node that contains a computation C with a right-hand side matching the question, then the question is not propagated further in this direction. (The marking of Q does not change.) In global search in this case, if C is not Q, then the list of pointers to computations that might make Q redundant is augmented by a pointer to C.

6. In both searches, if a question is propagated to a node and a non-ϕ computation in the LCT of that node defines one of the operands of the question, then the search is terminated and Q is marked not redundant.

   This case is only tested if no match is found in the previous case. This test and the previous test are made regardless of rank. These tests are simplified by having the program in SSA form. If an expression and an assignment that creates an operand for that expression occur in the same node, the expression that uses the operand will necessarily have to be after the computation that creates the operand.

5.2.2 Move Computations out of a Loop

Any computation in the loop header that is identified as redundant by the local search in the previous step can be moved out of the loop. We do this in two steps. Intuitively, we copy the computation to the landing pad (which makes the old copy in the header redundant) and then eliminate the old copy when various other redundancies are eliminated. Leaving the old copy in the header temporarily is a technical convenience. Let $V_h \leftarrow E$ be the computation C to be moved out, and let V_p be a newly generated name. Let E_p be the result of ϕ renaming the operands of E (as in section 5.1.2) for the inedge from the landing pad. Add $V_p \leftarrow E_p$ to the MCT for the edge from the landing pad to the loop header. This MCT entry points back to C. As usual, "moving" is just copying for the moment. The computation C persists in the loop header, but it is now redundant and will be eliminated by the global redundancy subphase.

5.3 Processing of Landing Pads

Processing of a landing pad is similar to processing of a normal node. We begin by moving computations into the landing pad from its successor (the loop header) exactly as in section 5.1.1. Then we try to move computations from the loop exits directly to the landing pad, without ever putting them inside the loop. Computations can be moved into the landing pad from the exits of the loop if the following conditions are satisfied:

1. The computation to be moved is in the MCTs of all the virtual outedges of the landing pad of the loop. If there is more than one virtual outedge, then we consider each in turn. For each virtual outedge, we loop over MCT entries of the current rank and try to match them with MCT entries (regardless of rank) for the other outedges. This is like the processing of normal nodes with more than one real outedge in section 5.1.1.

2. Each of the operands of the expressions must be available in the landing pad. Because the program is in SSA form, this condition is equivalent to having no assignments to the operands inside of the loop. An easy way to test this condition is to check the topsort number of the node that contains the definition of the operand. If it is earlier than or equal to the number of the landing pad node, the condition is satisfied.

Once the computations that can move into the landing pad have been identified, the process of moving them is exactly the same as in section 5.1.1.

The process of identifying computations that can be moved out from the landing pad is exactly the same as in section 5.1.2.

5.4 Eliminate Global Redundancies

This subphase is performed after the pass over nodes in reverse topsort order for the current rank. We loop over the computations of the current rank in any convenient order. (For example, we could visit the nodes in topsort order and loop over the computations of the current rank in each node's LCT.)

For each computation Q, we first test whether Q has ever been promoted in rank. If so, then it may now have higher rank than some computations that use the result of Q. The overall structure of our algorithm is such that this can only matter if the using computations are in the same node as Q, so we loop over the local uses of the result of Q. Each local use is promoted, if necessary, so has to have rank R+1 or greater. By the time Phase 2 is complete, the proper relation between ranks of computations and ranks of their operands will have been restored within each node.

The next step in processing the computation Q is to check for redundancy by applying the subalgorithm in section 5.2.1 with the global rules. If Q is found to be redundant, then it is eliminated by the following technique:

1. Create a new variable V to remember the value that Q will (redundantly) compute.
2. For each of the computations C that was put on the list of computations that might make Q redundant, an assignment of the form V ← (output of C) is inserted.
3. The expression part of Q is replaced with a use of V.

If more than one computation C was in step 2 above, then the program will no longer be in SSA form. It can be restored to SSA form by applying the rules of section 4.3 to the new variable V. The simple rules of section 4.3.1 seem best here, although there are some contrived situations where new equivalences could be recognized. Once SSA form has been restored, the trivial assignments can be removed by putting them on the worklist for the algorithm of section 4.5.

6 Phase 3: Normalization

To put the program into a more normal form, we order the code in each nonempty node, eliminate the purely formal ϕ functions, and delete empty nodes.

In SSA form we keep all computations in a node as a set, without order. Ordering is implied by ranks: something of rank 2 depends on a value of rank 1, and hence the rank 1 expression must be computed first. The sequencing information implicit in a variable's rank must be made explicit by putting all assignments of low rank before high rank assignments in each node. This clears the way for reversion to multiple assignments to the same variable.

Every computation of the form A ← ϕ(B, C) is replaced by an assignment A ← B on one of the entering branches, and by A ← C on the other. Each assignment is placed at the end of the code.

Any node with no code will have at most one successor. If it does have a successor, then it can be deleted after its inedges have been changed to be inedges of the successor.

The program is now much as it was originally, but there are more variables and fewer redundancies. Many of the variables can be merged together by graph coloring register allocation techniques [5] [6]. The *live range* of a variable consists of those nodes that lie on paths from an assignment to the variable to a use of the assigned value. If two variables have disjoint live ranges, then those variables can be merged into one variable. When we move a computation upwards we may shorten the live ranges of its operands, but we may lengthen the live range of its result. We have no statistical information on whether these changes in live ranges are generally helpful or harmful. A topic for future research is to find an algorithm which moves computations in order to aid register allocation by decreasing the live ranges.

7 Related Work

7.1 Redundancy Elimination

A computation C is *redundant* along a control flow path if it is preceded by an equivalent computation B, and so could be replaced by a use of the value computed at B. A computation is *fully* redundant if it is redundant along every path that reaches it (starting from the program entry point). A computation is *partially* redundant if it is redundant along some path that reaches it (starting from the program entry point). Elimination of many of the full redundancies has long been a major goal of optimizing compilers [2] [12]. Partial redundancies have received less attention. The major relevant work is that of Morel and Renvoise [13] (abbreviated *MR* hereafter) and the extensions to MR implemented in the PL.8 compiler [4].

Our algorithm is like MR in one respect: we eliminate partial redundancies by moving computations to places where some of the moved copies become fully redundant. Our integration of analysis and optimization is unlike the more traditional organization of MR, which puts analysis first and optimization second. Morel and Renvoise use an iterative data flow analysis, with a bit vector position for each lexically distinct expression. The system of simultaneous equations computes several bit vectors at each flow graph node. The vectors at each node depend on vectors at both predecessors and successors. The worst-case time required is $O((E + N) * N * C)$, where E is the number of edges in the graph, N is the number of nodes in the graph, and C is the number of computations.

The PL.8 compiler chooses temporary names systematically. If an expression like (A*B)+C appears twice in a program, then the same temporary variable is used for A*B in both places. This systematic naming allows the compiler to detect second order effects by applying MR repeatedly, until nothing changes. This may require as many iterations as there are ranks, plus one more iteration to detect stabilization.

The overall worst-case time of the PL.8 compiler's algorithm is $O((E + N) * N * C * R)$, where E, N, C are as above (for a single pass of MR) and R is the number of ranks. Both Morel and Renvoise and the PL.8 compiler group report that an application of MR typically requires only 3 to 5 iterations, rather than the worst-case number $N + 1$ of iterations.

The example in figure 1 illistrates the differences among MR, the PL.8 compiler, and the algorithm presented here. At the join node, what is now the value of A*B was computed under the name C*B along one path and so is unavailable to MR, the PL.8 compiler, or any other lexical method. If we remove the trivial assignment and replace all uses of C by uses of A in the original program, we get a program that is easier to optimize. A single pass of MR will eliminate the redundant computation of A * B for X. It will fail to do anything with the partially redundant computation of X + 1. This partial redundancy is a second order effect, and it will be eliminated by the PL.8 compiler's second pass of MR.

For programs with reducible control flow graphs, our algorithm gets everything that the PL.8 compiler gets. We identify many redundancies that the PL.8 compiler misses because of our use of global value numbering. The worst-case time bound for our algorithm is $O(C * N * E)$. If all the parameters in both bounds are replaced by a nominal parameter n, then the $O(n^3)$ of our algorithm improves upon the $O(n^4)$ of the PL.8 compiler's algorithm.

7.2 Value Numbering

Value numbering as originally conceived by Cocke and Schwartz [7] was the symbolic execution of a basic block, giving all variables entering that block distinct symbolic values. Common subexpression elimination on basic blocks is straightforward. If a symbolic value is computed twice, eliminate it the second time. Thus, in the code

```
C ← A;  D ← A * B;  E ← C * B;
```

both D and E have the symbolic value

```
( A @ entry ) * ( B @ entry )
```

and the second computation can be eliminated. Hashing of symbolic values allows the value numbering to proceed without having to manipulate large values. Several compilers (including the PL.8 compiler) have generalized this original value numbering from basic blocks to extended basic blocks.

Reif and Lewis [14] introduce a global approach to value numbering. Their approach implicitly includes one of many possible ways to recognize that two variables will always have the same value. There are other constructions [3] [15] that are similar in spirit to the work of Reif and Lewis, with various tradeoffs among the amount of information, the worst-case complexity, and the difficulty of implementation. Any of these constructions could be used to start our algorithm by putting the program into reduced SSA form.

Starting the process of redundancy elimination is only part of the task, as the code in figure 19 illustrates. The computation for T is redundant with the computation for R along one path and with the computation for S along the other path. The difference between E and A prevents any purely lexical approach from removing these redundancies. Value numbering is unaffected by this difference, but it still can answer only questions posed by the program it sees. The three variables R, S, T need to have three distinct value numbers. With the traditional organization that puts analysis first and optimization second, no optimization will be performed. Our algorithm will move the computation for T backwards along the inedges of the join node. One copy will be redundant with R while the other copy will be redundant with S. Our algorithm will eliminate both redundancies. It is only after the computation for T has been split into two copies that the relevant questions about equality of value numbers can be posed.

```
if P
     then do
          read(A)
          R ← B * A
     end
     else S ← B * A
E ← A
T ← B * E
```

Figure 19: Redundancy Eliminated by Our Algorithm's Integration of Analysis and Optimization

7.3 Other Issues

The p-graph construction [17] is a precursor of static single assignment form that has sometimes been used in optimization [11]. The explicit ϕ functions used in SSA form make it easier to work with.

Allen [2] assigns ranks locally (within basic blocks) and uses them to organize the elimination of redundancies within blocks and the movement of invariant code out of loops. Our global ranks are used similarly, to organize more extensive elimination and movement.

The name "landing pads" is recent [8], but similar ideas have been around for a long time [2] [12]. Morel and Renvoise suggest using landing pads to aid in the analysis, splitting a few edges if necessary. We split more edges and thereby perform some additional redundancy elimination, as figure 14 in section 4.2 illustrates. Morel and Renvoise recognize their loss here [13, p. 102], but they choose to allow no splitting beyond adding landing pads.

8 Optimality Criterion

In this section we will discuss what limits the best that can be done to eliminate redundant computations. We will describe what limits we have chosen to place on the algorithm. We will show that our algorithm is optimal (within these limits) on DAGs. Then we will examine some cases missed on programs with loops.

8.1 Programs that are DAGs

For reasons explained below, it is not possible to remove all redundancies from DAGs. We will enumerate three kinds of redundancy that it seems unreasonable to try to eliminate. An optimal algorithm would then be one that eliminates all *other* redundancies.

1. Even on DAGs it is undecidable whether two expressions will compute the same value. We will say that two values are *transparently* equivalent if they are constructed by the same sequence of operations on the same original operands. Thus in

   ```
   A ← B
   C ← E + (A * 3)
   D ← E + (B * 3)
   ```

   we will recognize that `C = D`, while in

   ```
   A ← B + 3
   C ← A + 2
   E ← B + 2
   D ← E + 3
   ```

   we will not recognize that `C = D`. For purposes of this subsection, transparent equivalence is the only kind we consider in defining reduced SSA form.

   We will not eliminate redundancies caused by computations that are equivalent but not transparently equivalent.

2. Eliminating all computations which transparently produce the same value may require either combinatorial explosions or unsafe transformations.

   Consider a DAG having a path along which a value is computed at a node u. Suppose the path proceeds through a node v, and later the same value is computed at a node w. Moreover, suppose that in the DAG there is a path which goes through v but does not compute the value. An example where this occurs is figure 20. Nodes u and w are the two computations of A*B. Node v is any node in between the two `if` statements.

   Redundancy could be eliminated by putting a copy of the code in v and then a copy of the Q conditional under each branch of the P conditional. This would allow statement w to be removed from the copy under the true branch of P. Such a transformation leads to an exponential blowup in the size of the program.

   ```
   if P
      then (...  X ← A * B ...)          (u)
      else (...)
   (...)                                 (v)
   if Q
      then (...  Y ← A * B ...)          (w)
      else (...)
   ```

   Figure 20: Redundancy with Crossed Paths

   A second way to eliminate the redundancy is to move a copy of the computation of A*B to before the test on P. This would then allow us to change statements (u) and (v) into trivial assignments. Such a transformation is unsafe because it introduces a calculation of A*B along the path executed if P and Q are both `false`. This path had no such calculation before.

   More formally, we will not eliminate redundancies of the following form: (A) there is a path from a node u at which a computation is performed, through a node v, to a node w at which an equivalent computation is performed; (B) there is another path from the root through v to w along which the computation at w is the first computation of the value; and (C) there is a path from v to an exit of the DAG that does not contain a computation equivalent to the redundant one at w.

3. There are programs in which two inequivalent computations are performed and (depending on the later control flow) either one or the other will make a later computation redundant. The redundant computation cannot be eliminated unless additional trivial assignments are inserted into the program to ensure that the correct value is stored into the location that the value will be picked up from. There are programs that require an arbitrary number of such assignments, and the cost of the loads and stores can exceed the cost of the computation.

   For example, consider figure 21. If P is true, then the computation of A*B at the end of the program fragment will be redundant with the computation of C*B. If P is false, then it will be redundant with the earlier computation of A*B.

   ```
   X ← A * B
   ...
   Y ← C * B
   if P
      then A ← C
   Z ← A * B
   ```

   Figure 21: Redundancy with Inequivalent Computations

   Formally, if there are two paths through nodes u, v, w, and on one path a computation at u is redundant with a computation C at w and on the other path a computation at v is redundant with C, and on neither path are the values computed at u and v transparently equivalent, then we will not eliminate the redundancy.

We have enumerated three kinds of redundancy that it seems unreasonable to try to eliminate, and our algorithm does not try to eliminate them. It does try to eliminate all the others. On a DAG, it succeeds.

Theorem: After our algorithm has terminated on a DAG, any remaining redundancy is of one of the three kinds that the algorithm does not try to eliminate.

Sketch of proof:

We will consider the program at the end of Phase 2, when it is still in reduced SSA form and has no trivial assignments. By induction on ranks and path lengths, it can be shown that expressions are transparently equivalent along a path iff they are lexically identical aside from ϕ renaming at join nodes along the path. More precisely, suppose control flows from a node u to a node v along a path (which might be the null path from u to itself) and that expressions E and F appear in nodes u and v. We say that these expressions are lexically identical *aside from ϕ renaming* if the result of moving F backwards to u along the path is lexically identical to E. At each inedge of a join node along the path, moving an expression backwards may involve ϕ renaming, as in section 5.1.2.

Define a *failure* to be any remaining redundancy that is not covered by the enumeration and so should have been removed. To show that there are no failures, we will assume there are failures and derive a contradiction. Thanks to the local redundancy elimination in section 4.5, any failure involves a computation B at a node u and a computation C at a node w, such that the nodes are different and there is a path from u to w. (The specific path is considered part of the failure.) Both B and C compute the same expression E, apart from the renaming at join nodes along the path, and E cannot be a ϕ function.

We can associate two numbers with any failure: the rank of E and the length of the path from u to w along which the failure occurs. If there are any failures, then we can choose one with maximum path length from among those with minimum rank. We will derive a contradiction by showing that this chosen failure must be of kind 2 or 3.

Because the operands of E are available (apart from applications of ϕ functions) for B in u, none of them are computed in w. Section 5.1.2 placed an entry for C in the MCT of the last edge e on the failure's path from u to w. Let v be the source of this edge. The MCT entry did not move in section 5.1.1, so v has another outedge f such that no computation equivalent to C was placed in the MCT of f. By maximality of the path length in the chosen failure, any path from v that starts along f is free of computations equivalent to C.

The nodes u, v, w have been shown to satisfy conditions (A) and (C) in the definition of the second kind of redundancy we do not claim to eliminate. For the chosen failure to *be* a failure, condition (B) must be false. Along every path from the root through v to w, a computation equivalent to C comes before C. Thanks to edge splitting, the branch node v is the only predecessor of w. Therefore, C is fully redundant. Global question propagation, however, did not eliminate C as redundant in section 5.4. This implies that propagation was stopped by rule 3 in the details of question propagation (section 5.2.1). The redundancy is therefore of kind 3, and the contradiction is obtained.

8.2 Programs with Loops

In this subsection, we consider two cases that are missed by the global algorithm when applied to programs with loops. There are other cases, but these two seem most important among the known cases.

1. Code that is not moved to a loop header may still be code that is on every path from the header to an exit of the loop. Local question propagation does not consider lifting such code to the landing pad. Some simple cases of this failing are easily recognized, and our algorithm could easily be extended to handle them. The general case is difficult, however. A loop may have many exits reached by many paths. There may be several equivalent computations that collectively act like a single computation that appears on all relevant paths. Ideally, the algorithm would be extended to recognize such collections as they arise.
2. The algorithm does not consider the possibility of moving a computation along a backedge. The program fragment in figure 22 could be improved by treating the loop header like an ordinary join node. If we move a copy of the first calculation of A+B along the backedge around the loop while moving another copy to the landing pad, then we can move the copy inside the loop just as if it had originally been placed after the conditional statement. Copies of the copy would appear in the then and the else branches, and the then branch copy would be merged with the assignment to Y. This would lower the number of calculations of A+B to exactly one per iteration (apart from the initial calculation in the landing pad).

```
while (...)  do
    X ← A + B
    if P
       then do
          A ← ...
          Y ← A + B
       end
       else
end
```

Figure 22: Missed Optimization

Though attractive in this example, movement along backedges poses some difficult organizational questions. By the time a computation reaches the loop header, movement of computations within the loop body for the current rank has already been completed. Moving along a backedge will not help unless the rank used for the copy that stays in the loop is forced to be larger than the current rank. This leads to the possibility of an infinite regress: a computation reaches the loop header with rank R, is moved along a backedge with rank $R + \delta$, reaches the loop header again but now with rank $R + \delta$, is moved along the backedge with rank

$R + \delta + \delta$, and so on. When loops are nested, it is difficult to see how to keep enough records to avoid infinite regress without also missing opportunities. One might be tempted to try iteration of our entire algorithm, with the understanding that a computation moved along a backedge will not move further until the next iteration. This too can lead to an infinite regress, as figure 23 illustrates.

```
while (...)  do
     C ← A + B
     A ← C + B
end
```

Figure 23: Danger of Infinite Regress

9 Conclusions

We have shown how to obtain unusually extensive optimization of reducible programs at moderate cost, thanks largely to synergism among three innovations: global ranking of expressions, static single assignment (SSA) intermediate form, and ϕ renaming of expressions.

Global ranks let us exploit second order effects rapidly, without reanalysis. Ranks also help characterize the number of iterations needed when other algorithms are called repeatedly to exploit second order effects.

The SSA form of a program enables us to remove trivial assignments easily. It also allows us to recognize and exploit equivalences among expressions that would not be lexically identical in the usual form. If a program is in SSA form, then lexically identical expressions always have the same value, no matter where they occur. If a program is in reduced SSA form, furthermore, expressions with the same value number will be lexically identical aside from ϕ renaming along paths.

With the help of a linearly bounded amount of edge splitting, ϕ renaming enables us to preserve SSA form while moving a computation from a join node to its predecessors. Preserving SSA form throughout the intermediate steps is important because it allows the code associated with each node to be represented by a table that can be accessed and updated efficiently.

We have also specified a reasonable optimality criterion. In the special case of a program without loops, the code generated by our algorithm is provably optimal in the technical sense explained in section 8.

Acknowledgments

We thank Fran Allen, Trina Avery, and Ron Cytron for their comments on drafts of this paper.

References

[1] F. E. Allen. Control flow analysis. *Sigplan Notices*, July 1970.

[2] F. E. Allen. Program optimization. *Ann. Rev. Automatic Programming*, 5:239–307, 1969.

[3] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, January 1988.

[4] M. Auslander and M. Hopkins. An overview of the PL.8 compiler. *Proc. SIGPLAN'82 Symp. on Compiler Construction*, 17(6):22–31, June 1982.

[5] G. J. Chaitin. Register allocation and spilling via graph coloring. *Proc. SIGPLAN'82 Symp. on Compiler Construction*, 98–105, June 1982.

[6] F. Chow and J. Hennessy. Register allocation by priority-based coloring. *Proc. SIGPLAN'84 Symp. on Compiler Construction*, 19(6):222–232, June 1984.

[7] J. Cocke and J. T. Schwartz. *Programming Languages and Their Compilers; Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, April 1970.

[8] R. Cytron, A. Lowry, and K. Zadeck. Code motion of control structures in high-level languages. *Conf. Rec. Thirteenth ACM Symp. on Principles of Programming Languages*, 70–85, January 1986.

[9] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, July 1974.

[10] M. S. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM J. Computing*, 4(4):519–532, December 1975.

[11] D. B. Loveman and R. A. Faneuf. Program optimization - theory and practice. *Proc. Conf. on Programming Languages and Compilers for Parallel and Vector Machines*, 10(3):97–102, March 1975.

[12] E. S. Lowry and C. W. Medlock. Object code optimization. *Comm. ACM*, 12(1):13–22, 1969.

[13] E. Morel and C. Renvoise. Global optimization by supression of partial redundancies. *Comm. ACM*, 22(2):96–103, February 1979.

[14] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. *Conf. Rec. Fourth ACM Symp. on Principles of Programming Languages*, 104–118, January 1977.

[15] J. H. Reif and R. E. Tarjan. Symbolic program analysis in almost linear time. *SIAM J. Computing*, 11(1):81–93, February 1982.

[16] B. K. Rosen. Data flow analysis for procedural languages. *J. ACM*, 26(2):322–344, April 1979.

[17] R. M. Shapiro and H. Saint. *The Representation of Algorithms*. Technical Report CA-7002-1432, Massachusetts Computer Associates, February 1970.

[18] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.

[19] R. E. Tarjan. Testing flow graph reducibility. *J. Computer and System Sciences*, 9:355–365, December 1974.