Code Motion of Control Structures in High-Level Languages

Ron Cytron
Andy Lowry[1]
Kenneth Zadeck

IBM T. J. Watson Research Center
Yorktown Heights, New York

1.0 Overview and Motivation

One trend among programmers is the increased use of abstractions. Through encapsulation techniques, abstractions extend the repertory of data structures and their concomitant operations that are processed directly by a compiler. For example, a compiler might not offer sets or set operations in its base language, but abstractions allow a programmer to define sets in terms of constructs already recognized by the compiler. In particular, abstractions can allow new constructs to be defined in terms of other abstractions. Although significant power is gained through the use of layered abstractions, object code quality suffers as increasingly less of a program's data structures and operations are exposed to the optimization phase of a compiler. Multiple references to abstractions are also inefficient, since the interaction between abstractions is often complex yet hidden from a compiler. Abstractions are most flexible when they are cast in general terms; a specific invocation is then tailored by the abstraction to obtain the appropriate code. A sequence of references to such abstractions can be inefficient due to functional redundancy that cannot be detected at compile-time. By integrating the references, the offending segments of code can be moved to a more advantageous position. Although procedure integration materializes abstracted constructs, the abstractions can still be ineligible for optimization using current techniques; in particular, abstractions often involve loops and conditional branches that can obscure code that would otherwise be eligible for code motion.

To make abstractions viable as an efficient programming tool, optimizations such as code motion must overcome the obstacles presented by abstractions. The problem of code motion has been addressed by Lowry and Medlock [Lowry69], Wulf [Wulf69], Schwartz [Schwartz73], Aho and Ullman [Aho78], Morel and Renvoise [Morel79] Reif and Lewis [Reif77] [Reif82], and Ferrante and Ottenstein [Ferrante83]. These approaches fall short of handling abstractions because they fail to consider the following issues in a unified manner:

1. Some code cannot be moved unless accompanied by its surrounding control structures. For example, two definitions of a variable may reach a use, with surrounding control structures determining which definition actually reaches the use. Unless the control structure accompanies the motion of the definitions, neither the definitions nor the use can be moved.
2. The motion of stores should be considered as well as the motion of expressions.
3. It can be profitable to move a computation from an area where it might never be executed to an area where it is always executed.
4. Second order effects are significant to thorough code motion. The motion of one piece of code may be dependent on the motion of some other piece of code.

This paper presents two code motion algorithms that account for the above issues and are particularly appropriate for abstractions. One algorithm is conservative with respect to the third issue; program performance can only be improved by this algorithm. Another algorithm is more aggressive; the resulting code *should* execute faster given a widely accepted model of branch behavior. A common subexpression algorithm is presented that accommodates control structures and enhances the effectiveness of the code motion algorithms.

2.0 General Approach

Since abstractions are typically implemented by procedure calls, a form of procedure integration is useful for incorporating the code due to abstractions. The technique proposed by Wegman and Zadeck [Wegman85] couples procedure integration with a powerful constant propagation algorithm that avoids some of the intermediate space problems normally associated with this technique.

In order to accommodate complex control structures, a program is regarded as a collection of *intervals* in the style of Tarjan [Tarjan74], Graham and Wegman [Graham76], and Schwartz and Sharir [Schwartz79]. Note that these differ from the maximal intervals originally suggested by Allen and Cocke [Allen70] in that maximal intervals contain some nodes that are not in the

[1] Author's current address is Department of Computer Science, Columbia University, New York, New York

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM-0-89791-175-X-1/86-0070 $00.75

strongly connected component dominated by the interval header. Wherever we refer to interval, we mean the former not the later.

Subject to criteria presented in ensuing sections, the statements and control structures that are moved from within the interval are moved before that interval. Each interval is accordingly augmented with a *landing pad*, just before entry, to provide a repository for moved code. The landing pads ordinarily contain prologue code and so are guarded by the same conditions that guard the interval; thus, code moved to a landing pad is never executed unless it would have been executed inside the interval. After moving as much code as possible out of an interval, that interval is summarized in terms of its data flow properties. The interval can then be considered as an atomic entity, eligible for motion out of its surrounding intervals.

3.0 Preliminaries

3.1 Strictness and Profitability

The goal of any optimization technique is to decrease the execution time of a program while maintaining the observed behavior of that program; accordingly, a particular transformation can be characterized with respect to it effectiveness by its *strictness* and *profitability*.

Strictness indicates whether a transformation is conservative with respect to decreasing the execution time of a program. Under a strict transformation, code can be relocated only if the execution of that code in its new location occurs no more frequently than in its original location. By relaxing strictness, code can be relocated to positions where it probably would be executed less frequently. This paper will pursue both strict and nonstrict transformations.

Profitability describes the degree that nonstrict transformations affect the performance of a given program. The profitability problem is generally unsolvable at compile-time, since program branching can depend on run-time values; however, compile-time predictions, based on the structure of a program as viewed in terms of its control flow graph, allow code to be relocated to areas of probable profitability. This paper will make two assumptions about the frequency of execution of statements within a program:

1. The frequency of executing a statement grows, possibly exponentially, as the number of intervals that surround that statement increases.
2. Every statement within an interval has a high probability of being executed whenever the interval is executed.

3.2 Correctness and Safety

An optimization technique must be applied in a context in which its effects on the observed behavior of a program are well understood. Accordingly, a particular transformation can be characterized with respect to program output by its *correctness* and *safety*.

The transformations presented in this paper are correct in the sense that the data dependences of a program are respected. The values used by expressions are always produced by the same computations as in the original program. The transformations may move the expressions to locations where the frequency of execution is reduced (strict transformations) or where the frequency of execution is expected to be reduced (nonstrict transformations).

Although nonstrict transformations may cause execution of a statement that would not have been executed in the original program, this need not violate correctness. The nonstrict transformations in this paper can only cause such spurious executions when it can be guaranteed that the calculated values will not be used incorrectly. Any statement left in the loop or reachable from the loop will always get the correct value.

Safety describes the degree to which a transformation preserves the observed behavior of a program. Safety is thus related to the detail at which the output is scrutinized. For example, if a program is observed as an instruction trace, then any motion whatsoever causes a change in the output. If a program is taken as a sequence of results and exceptional conditions, then code can be moved so long as the sequence remains unaltered. The issue of safety does not influence the actual mechanism of a code motion algorithm; rather, a given definition of safety dictates which operations are eligible for motion. The compiler writer can therefore allow any statement to be moved by the algorithms given here as long as that statement does not directly produce output. The decisions made by the compiler writer should reflect whatever view of safety is desired.

Many optimizing compilers allow interrupt producing statements to be moved under strict code motion. When this occurs, the frequency and location of interrupts may change from the original program, but a program produces interrupts after strict code motion if and only if it produced interrupts before strict code motion. Note that it is possible and efficient on some machine architectures to separate the parts of an operation that produce the calculation from the part of an operation that may produce an interrupt. When this is the case, the first part of the operation may be moved.

3.3 Landing Pads

Code can be moved out of intervals in either of two directions: *backward*, so that its execution precedes the interval, or *forward*, so that its execution follows the interval. In either case, code moved out of an interval should not be executed unless the interval would have been entered in the original program. This level of profitability is easily maintained by introducing *landing pads* into the interval control structure as shown in Figure 1.

As illustrated in Figure 1, an interval is equipped with one entry pad and possibly several exit pads. In order to restrict the entry pad from executing unless the interval is entered, the entry pad is guarded by the same conditions that guard its associated interval. In the example of Figure 1, the test T is duplicated to accommodate the landing pad. Although such duplication causes in a slight increase in program size, the execution time of the program is not increased, since the test is executed only once per iteration of the interval.[2] A different exit pad is associated with each interval exit; different code can then be placed in each exit pad, and such code is executed if its associated exit is taken. By the definition of an interval used in this paper, all landing pads are outside the interval with which they are associated. For each exit from an interval, control is transferred to the landing pad associated with that exit. After executing the code in the landing pad, control is then transferred to the original target of the branch.

[2] This transformation can be thought of as turning a WHILE construct into a REPEAT-UNTIL construct embedded inside inside an IF-THEN structure.

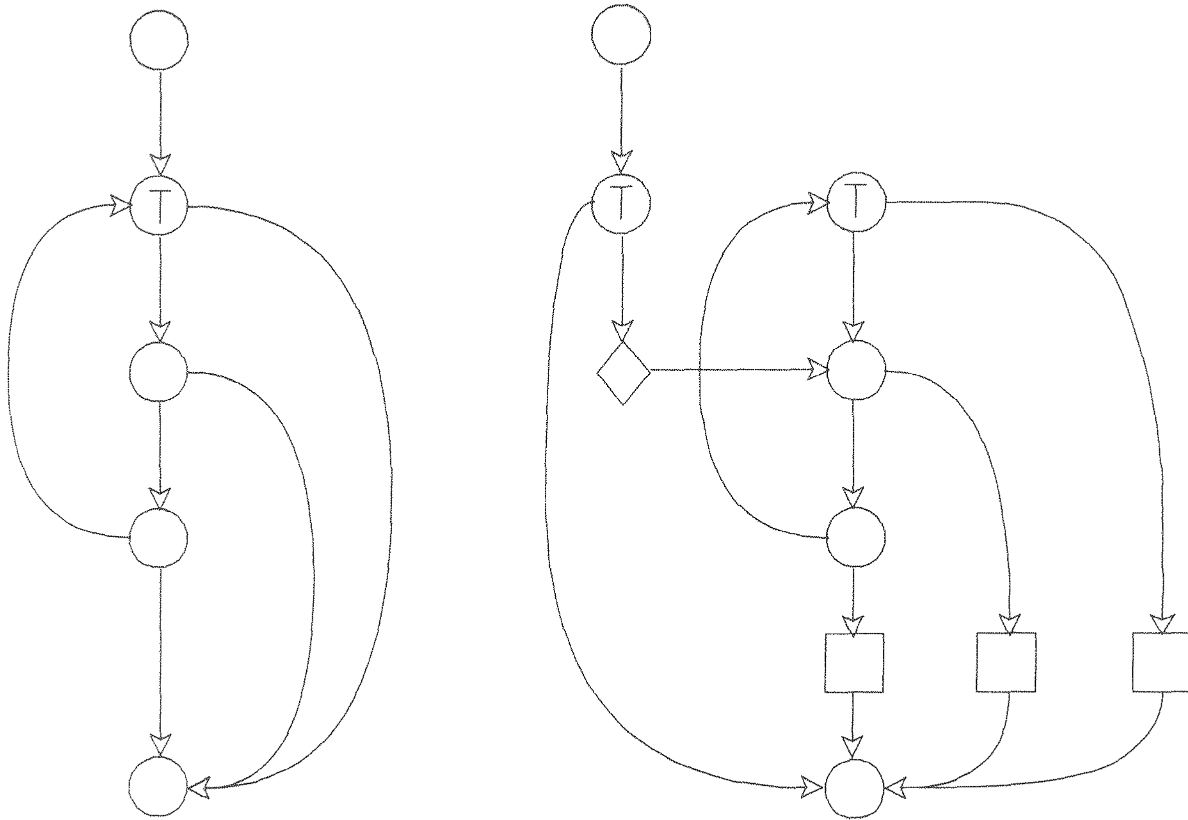

Figure 1. Landing Pads: Landing pads are inserted into a loop as shown here. The entry pad is shown as a diamond and exit pads appear as squares.

3.4 Ordering the Nodes of an Interval

The algorithms presented in this paper consider the nodes of an interval in their topological order with respect to the control flow graph DAG of the interval. This DAG is comprised of all control flow arcs except those targeted for the header of the interval (the back-edges). A topological order is only a partial order over the nodes, yet a deterministic algorithm considers the nodes with respect to some total order. Although there is considerable freedom in choosing that total order, the algorithms presented in this paper use the following *dominated topological order*, for reasons that accompany the presentation of the algorithms.

A dominated topological order is any topological ordering of the control flow graph with the following constraint: Of all the nodes that follow a particular node N, those nodes that N dominates precede all other nodes. This ordering of nodes is more restrictive than the interval orders defined by Tarjan or Graham and Wegman.

The actual construction of the dominated topological order is simplified through the following observation: Every edge of the control flow DAG is either in the dominator tree or is targeted on the sibling of an ancestor (in the dominator tree) of the source of the edge. The left column of Figure 10 shows each node of a DAG labelled with the order as visited by a dominated topological traversal of the DAG.

4.0 Code Motion Algorithms

The ensuing sections consider five transformations that can improve program performance. The first algorithm, *RENAME*, accepts a standard intermediate program representation comprised of operations on source variable names and produces an equivalent program expressed in terms of compiler-managed temporary names. Where possible, references (uses and definitions) to the source names are replaced by references to these temporaries. The purpose of this transformation is to remove spurious dependences that arise from a single variable name that holds multiple expressions. As a result, many more temporary names can be generated than source names that are replaced. Although the RENAME algorithm performs a limited amount of code motion, its true purpose is to afford the other algorithms more latitude by eliminating unnecessary dependences.

There are two code motion algorithms: *STRICT* and *NONSTRICT*. Although the NONSTRICT algorithm is more aggressive with respect to the conditions that permit code motion, both algorithms have the same general structure. The intervals of a program are examined, from their innermost to outermost nesting. Upon visiting an interval, code is moved to the entry or exit landing pads associated with that interval. Upon leaving an interval, the interval is summarized in terms of its dataflow information. Once summarized, the interval behaves as any other statement; the term *statement* will therefore refer to simple statements as well as summarized intervals. The landing pads associated with a summarized interval are contained in the surrounding interval.

Both STRICT and NONSTRICT account for the control structure inside an interval. By visiting the statements of an

interval in the proper order, control structures can be copied to landing pads, thus allowing the subsequent motion of statements guarded by the control structures. Although a copy of the control structure must remain inside the interval unless all guarded statements are moved, the guards inside the interval can be replaced by simple bit tests that are computed in the landing pad. If all statements are moved, then the control structure can be deleted from the interval. With the exception of control structures, code appears in either a landing pad or in the interval.

The STRICT algorithm performs strict code motion. Unlike traditional code motion algorithms that only analyze control flow information to determine profitability, this algorithm utilizes such information to create a repository for statements moved to the landing pad. Upon visiting an interval, branches that can be decided in the landing pad are copied there. Statements are moved to the landing pad, so as to guard them by the same conditions that determine their execution in the interval. Thus, a statement moved to the landing pad is executed only if it would have been executed in the interval.

To maintain profitability, the STRICT algorithm cannot move code that is guarded by branches that cannot be decided in the landing pad. However, if some code is common across *all* paths from an immovable branch, then that code can be moved above the branch by an advanced form of *common subexpression elimination* that accommodates control structures. The *COMMON* algorithm uses a restricted form of pattern matching over the control flow graph to identify areas that are common across, yet independent of, all paths from a branch. Like the STRICT algorithm, control structures are copied above the branch. Instances of statements that are common across all paths are replaced by a single copy above the branch, positioned appropriately within the copied control structure. By removing such statements from immovable control structures, the statements become eligible for code motion by the STRICT algorithm. Although the COMMON algorithm should be performed before the STRICT algorithm, the subjects are covered in the reverse order because the deficiencies of the STRICT algorithm motivate the COMMON algorithm.

In the NONSTRICT algorithm, the strictness constraint is relaxed; statements guarded by immovable branches are moved to the landing pad, even though such motion causes a spurious execution of the moved statements. Like the STRICT algorithm, all loop-invariant statements are moved to positions in the landing pad where they are guarded by the same movable tests that guarded their execution in the interval. Unlike the STRICT algorithm, such code may be guarded by immovable branches in the original program. Thus, the code moved by the NONSTRICT algorithm is a superset of the code moved by the STRICT algorithm.

Note that the COMMON algorithm also improves the performance of the NONSTRICT algorithm by making more code available for more profitable movement. For code that is common across some (but not all) paths from a branch, the NONSTRICT algorithm has the property of moving all copies of such code above the branch. The *COMBINE* algorithm maintains a dictionary of the expressions that are moved to the landing pad. Any expression for which an entry exists in the dictionary is removed from the interval but not duplicated in the landing pad.

4.1 Renaming of Variables

The RENAME algorithm consists of a collection of transformations that are applied in succession prior to code motion, in order to increase the effectiveness of the code motion. Essentially these transformations will have the effect of expressing the summary of the dataflow information by utilizing a large number of program temporary names.[3] The STRICT, COMMON, NONSTRICT and COMBINE algorithms then use this information rather than the source names, because source names do not have the inherent dataflow information. The major advantage of this approach is that the subsequent algorithms perform transformations that preserve the consistency of this representation. Therefore, there is no need to recompute the dataflow information to get second order effects.

The final result of the transformations is related to the *Global Value Graph* described by Reif and Tarjan [Reif81], in that we make explicit the *birthpoints* for all variables, along with the uses that they reach. The concept of birthpoint is very closely related to the dataflow concept of a definition. Whereas a single use of a variable may be reached by several definitions of that variable, a use is reached by exactly one birthpoint. The birthpoint is located in the control flow graph so as to intercept and represent collectively all those reaching definitions.

The proper placement of birthpoints for a variable depends only on the control flow graph and the pattern of definitions for that variable; definitions for other variables are irrelevant. Although the transformations presented in this section compute birthpoints for all variables simultaneously, the concept is most easily understood in terms of an analysis that considers one variable at a time.

Computing birthpoints for a given variable x consists of partitioning the control flow graph into a number of disjoint collections of nodes, called *components*, of the graph. Each component must meet all of the following constraints:

1. Every node in a component must be reached by exactly the same definitions for x as all the other nodes in that component. Thus, all the nodes in a single component share a common definition set with respect to x.
2. The component must correspond to a single-entry region of the control flow graph. That is, with the exception of exactly one node in the component, no node may have a control flow predecessor that is not in the component.
3. A component is the maximal collection of nodes that meets the above constraints.

The single entry node for each component is a birthpoint for the variable x. Because of the single-entry nature of the components, only one birthpoint can reach any node. Each birthpoint represents the common set of definitions that reach the nodes in the component.

Birthpoints arise for two distinct reasons. First, any definition for a variable is also a birthpoint for that variable, as it is the only definition that meets any use that occurs before the next join point. Second, birthpoints appear wherever multiple birthpoints for a variable reach a node along different incoming edges. The first type of birthpoint is called a *definition birthpoint*, whereas the second is called a *join birthpoint*.

The transformations described in this section locate the birthpoints for all variables in an interval, and replace all source variables with temporary variables that denote the birthpoints.

[3] This information summarizes the DEF and USE information but requires a worst case assumption with respect to MAY. This assumption is that a MAY site both USEs and DEFs the variable.

At a definition birthpoint, the birthpoint name becomes the target of the assignment. Assignments are added to transmit values among the temporary variables as required at join birthpoints.

For any scalar name in the source program, the transformations can introduce a number of temporary names for that variable. The analysis described by Banerjee [Banerjee79], Wolfe [Wolfe82], and Allen [Allen83] determines independence of subscripted variables. With the appropriate extensions, these techniques can be applied to treat elements of arrays as scalars and to recognize that two subscripted references are always distinct. Intuitively, expressions that are not bound to user-accessible names have greater freedom with respect to code motion. Each temporary name constrains the ordering of the only those definition sites that reach its birthpoint. A reference to a source name constrains the ordering of all the definition sites for that variable.

Consider the example shown in Figure 2. Only the definitions at (2) and (3) can reach the use at (4). If definitions (2) and (3) can be removed from the interval in a manner that preserves their relat.ve order of assignment, then the use at (4) can be moved as well. Note that this can be done irrespective of what happens to the definition at (1) as long as the results of definitions (2) and (3) are available after the definition to (1). Renaming captures the multiple values that were assigned to a single name into distinct names, making them available throughout a computation. Provided that the dependences for each temporary names are respected, code may be moved by the STRICT and NONSTRICT algorithms.

```
while ()
(1) a ← not movable

    if movabletest1
      then do
(2)       a ← movable

        if movabletest2

(3)         then a ← movable


          else ;

(4)       ? ← a
      end
(5) ? ← a

end
```

Source Program

```
if not () goto exit
loop:
  a.1 ← not movable
  a.5 ← a.1
  if movabletest1
    then do
      a.2 ← movable
      a.4 ← a.2
      if movabletest2
        then do
          a.3 ← movable
          a.4 ← a.3
        end
        else ;
      a.5 ← a.4
      ? ← a.4
    end
  ? ← a.5
if () goto loop
exit:
```

Renamed Program

```
if not() goto exit
  if movabletest1
    then do
      a.2 ← movable
      a.4 ← a.2
      if movabletest2
        then do
          a.3 ← movable
          a.4 ← a.3
        end
        else ;
        ? ← a.4
    end
loop:
  a.1 ← not movable
  a.5 ← a.1
  if movabletest1
    then do

      a.5 ← a.4

    end
  ? ← a.5
if () goto loop
exit:
```

After Code Motion

Figure 2. Renaming to Break Order Dependencies

4.1.1 Renaming Transformations

The transformations are defined in the rest of this section. Figure 3 and Figure 4 represent the transformation process step by step. Each transformation preserves the correctness of the program. These transformations are restricted to scalars where the pattern of aliasing is understood. No reordering can be done on variables that are potentially aliased, because the real patterns of loads and stores for those variables are unknown. See Myers [Myers81], Cooper [Cooper83] or Burke [Burke84] for a discussion of aliasing analysis.

In Figure 3, the left column contains a program fragment that is expanded to four basic blocks, as shown in the right column. Note that in the intermediate code, the interval has been equipped with an entry pad before the *loop* label and an exit landing pad after the *exitpad* label. The test that guards the interval has been duplicated above the entry pad.

```
while (i < n)
  if n > 3
    then do
      k ← n * 6
      m ← k * i
    end
    else do
      k ← n * 8
    end
  j ← k + 3
  i ← i + 1

end
```

Program Fragment

```
    if (i >= n) goto exit
    loop:
1:    if n > 3
2:      then do
          k ← n * 6
          m ← k * i
        end
3:      else do
          k ← n * 8
        end
4:    j ← k + 3
      i ← i + 1
      if (i < n) goto loop
    exit_pad:
    exit:
```

Intermediate Code

Figure 3. A Program is Composed of Basic Blocks

The first transformation identifies the definition birthpoints in the interval and assigns each a unique temporary name.[4] Because this processing is performed interval by interval, the birthpoints result not just from the definition sites within the interval, but also from those definitions that enter the interval. In order to identify the variables that may reach the interval, a special form of definition is placed in the entry landing pad for every variable either used or defined within the interval. This special definition is called an identity assignment and has the form $x \leftarrow x$ for variable x. Each identity assignment forces the creation of a definition birthpoint that acts as an interface between this interval and the rest of the program.

The definition birthpoints are identified by splitting every assignment statement into two statements. The first statement computes the right-hand side of the original statement and stores the result into a unique temporary name. The second statement performs an assignment from the temporary name into the original program variable. In this way, the use and definition components of each statement are separated; thus, the part of a statement that computes values can be moved independently of the definitions produced by the statement. Note that the identity assignments added to the entry pad are included in this splitting process. The first transformation as applied to the above example is shown in the left column of Figure 4.

The second transformation identifies the join birthpoints in the interval DAG. Reif and Tarjan [Reif81] describe algorithms for locating join birthpoints in arbitrary flow graphs. However, since interval bodies are single-entry DAGs when they are processed, a simpler solution suffices for such graphs. The process is described for a single variable x.

The approach begins by adding a birthpoint at the top of the interval for x if x is defined within the interval. This is the birthpoint for the values carried along the back-edges of the interval that join with the values entering the interval through the header.

Next, the nodes in the DAG are visited in topological order. When a node is visited that is not already a birthpoint for x, its DAG predecessors are examined. If all the predecessors are covered by the same birthpoint for x, then this node is marked as covered by that birthpoint. Otherwise, a new join birthpoint for x is established at the node.

In the second column of Figure 4, join birthpoints have been added for all defined variables at the top of the interval. Within the interval, join birthpoints have been added for $k.5$ and $m.4$. Two assignments are created for each birthpoint added. The first is of the form $x.n \leftarrow x$. The second is of the form $x \leftarrow x.n$.

The third transformation takes the names created at the definition and join birthpoints and forward-substitutes these names into the uses that are covered[5] by these definitions. The result of this transformation is shown in the third column of Figure 4.

The fourth transformation removes all remaining assignments to source variables from the interval. Definitions that are dead in the interval body may be removed immediately. Any other definition must have a subsequent birthpoint as its only use. By back-substituting the birthpoint variable for the left-hand side of the assignment, the source variable assignment is removed. After this has been accomplished for all source variable definitions, the assignments from source to birthpoint variables that appear at the join birthpoints themselves must also be removed. A *birthpoint marker* is created for use by the subsequent algorithms.

Additionally, this transformation transmits values defined within the interval to the appropriate external use sites. The targets of those definitions are birthpoint variables, whereas the uses have not yet been renamed. The interface is accomplished by adding assignments to source variables at each exit pad. Each assignment assigns whichever birthpoint variable is currently active for the source variable being defined.

Note that all assignments involving source variables that were added to landing pads by either transformation one or transformation four will undergo renaming when the next outermost interval is processed, since the landing pads will be considered as members of that interval. When the entire program has been processed in this fashion, no references to source variables will remain.[6]

In order for the code motion algorithms to perform correctly, it is important for the introduction of birthpoints to preserve the semantics of uninitialized variables. This may be accomplished by introducing a definition for each variable at the top to the program. This definition would assign the special value *uninitialized* to each variable. This will assure that the temporary variables will not rearrange the order of assignment for uses where an uninitialized variable may reach. Of course it is not necessary to actually generate any code for these definitions.

[4] In the examples, the temporary names for variable x have the form $x.n$.

[5] A definition is said to cover a use if it is the only definition reaches that use.

[6] This is not true for external variables. Each nonintegrated subroutine call must be treated as a birthpoint for all external variables. Stores for these variables are still required before any external subroutine call. After the subroutine call, a reference must be made that resets the appropriate temporary variable. This result can be sharpened in the presence of interprocedural analysis as mentioned above.

```
if (i >= n) goto exit
i ← i.1 ← i
j ← j.1 ← j
k ← k.1 ← k
m ← m.1 ← m
n ← n.1 ← n
loop:
1:  if n > 3
2:    then do
        k.3 ← n * 6
        k ← k.3
        m.3 ← k * i
        m ← m.3
      end
3:    else do
        k.4 ← n * 8
        k ← k.4
      end
4:  j.3 ← k + 3
    j ← j.3
    i.3 ← i + 1
    i ← i.3
    if (i < n) goto loop
  exit_pad:
  exit:
```

After Separation

```
   if (i >= n) goto exit
   i ← i.1 ← i
   j ← j.1 ← j
   k ← k.1 ← k
   m ← m.1 ← m
   n ← n.1 ← n
   loop:
1:   i ← i.2 ← i
     j ← j.2 ← j
     k ← k.2 ← k
     m ← m.2 ← m
     if n.1 > 3
2:     then do
         k.3 ← n.1 * 6
         k ← k.3
         m.3 ← k * i
         m ← m.3
       end
3:     else do
         k.4 ← n * 8
         k ← k.4
       end
4:   k.5 ← k
     k ← k.5
     m.4 ← m
     m ← m.4
     j.3 ← k + 3
     j ← j.3
     i.3 ← i + 1
     i ← i.3
     if (i < n) goto loop
   exit_pad:
   exit:
```

After Addition of Birthpoints

```
   if (i >= n) goto exit
   i ← i.1 ← i
   j ← j.1 ← j
   k ← k.1 ← k
   m ← m.1 ← m
   n ← n.1 ← n
   loop:
1:   i ← i.2 ← i
     j ← j.2 ← j
     k ← k.2 ← k
     m ← m.2 ← m
     if n.1 > 3
2:     then do
         k.3 ← n.1 * 6
         k ← k.3
         m.3 ← k.3 * i.2
         m ← m.3
       end
3:     else do
         k.4 ← n.1 * 8
         k ← k.4
       end
4:   k.5 ← k
     k ← k.5
     m.4 ← m
     m ← m.4
     j.3 ← k.5 + 3
     j ← j.3
     i.3 ← i.2 + j.3
     i ← i.3
     if (i.3 < n.1) goto loop
   exit_pad:
   exit:
```

After Forward Substitution

```
   if (i >= n) goto exit
   i.2 ← i.1 ← i
   j.2 ← j.1 ← j
   k.2 ← k.1 ← k
   m.2 ← m.1 ← m
   n.1 ← n
   loop:
1:   birthpoint i.2
     birthpoint j.2
     birthpoint k.2
     birthpoint m.2
     m.4 ← m.2
     if n.1 > 3
2:     then do
         k.3 ← n.1 * 6
         k.5 ← k.3
         m.3 ← k.3 * i.2
         m.4 ← m.3
       end
3:     else do
         k.4 ← n.1 * 8
         k.5 ← k.4
       end
4:   birthpoint k.5
     k.2 ← k.5
     birthpoint m.4
     m.2 ← m.4
     j.3 ← k.5 + 3
     j.2 ← j.3
     i.3 ← i.2 + j.3
     i.2 ← i.3
     if (i.3 < n.1) goto loop
   exit_pad:
     i ← i.3
     j ← j.3
     k ← k.5
     m ← m.4
   exit:
```

After Back-Substitution

Figure 4. The Transformations Involved in Renaming

Note that many temporary variables may be created whose only other references are to set other temporaries. After the last transformation described above, it is advisable to perform a pass of dead code elimination and remove all of the temporary names that are never otherwise referenced.

After all of code has been moved it is possible to coalesce many of these temporaries into a much smaller number of compiler temporaries by a process known as coloring. While this process is NP complete for an exact solution, Chaitin [Chaitin81] [Chaitin82] has developed a heuristic approach that has very good performance.

4.2 The Strict Algorithm

In this section, a strict algorithm is presented for moving statements from inside an interval to the landing pad guarding the entrance of that interval. To accomplish strict execution, the algorithm first copies the control flow that surrounds movable statements. Consider the example shown in Figure 5.

```
i ← 1
while (i < n)
  if j = 3
    then k ← 6
    else k ← 8
  i ← i + 1
end
```

Figure 5. A Simple Example

Neither assignment to *k* can be moved without violating both correctness and strictness, unless the test and branch accompany the motion.

The STRICT algorithm visits the statements of an interval to determine which statements are eligible for movement to the entry pad. The order of the traversal is the dominated topological order that was defined in Section 3.4. This ordering guarantees that when a statement is considered for movement, any other

statement whose movement might free this statement for movement will already have been visited. Thus a single pass over the interval is sufficient to determine the movability of the code.

As the traversal progresses, movable tests are copied into the entry pad, and replaced in the interval body by simple bit tests. Other code that is found to be movable is moved to the landing pad without leaving any code behind in the interval. After the processing is complete, the interval body and the landing pad may both be subjected to a form of dead code elimination that locates and discards tests that no longer guard any code.

The tests for movability involve the concept of *control dependence*, which is developed by Ferrante, Ottenstein, and Warren [Ferrante84]. Informally, a statement is control dependent on those tests that may cause the statement to be executed. More precisely, a statement Y is control dependent on a statement X if

1. There exists a path from X to Y such that all nodes in the path, except X and Y, are post-dominated[7] by Y, and
2. X is not post-dominated by Y.

Consider the program fragment shown in Figure 6. By the above definitions, each assignment to j is control dependent on the test of i, but the the test of k and assignment to m are not. A larger example of a program flow graph and its associated control dependence graph can be found in Figure 10.

```
while (i < n)
  if i < 3
    then j ← 5
    else j ← 7

  if k = 0
    then m ← 6

  i ← i + 1
end
```

Figure 6. Variant and Invariant Branching

The conditions that allow a statement or a test to be movable are:

1. The statement must not be control dependent on any immovable test.

   This condition insures that the conditions for executing the statement are invariant of the interval and can be duplicated in the entry pad. It accounts for control dependences that govern the correctness and strictness of relocating a statement. The NONSTRICT algorithm relaxes this constraint.

   In Figure 6, this condition precludes movement of either assignment to j because they are control dependent on the loop-variant test on i. However, the test on k and the assignment to m that it guards are unaffected by the test and branch on i.

2. Definitions that reach uses within the statement must come from statements outside the interval. A birthpoint marker is treated as a use of the variable it marks.

   This condition accounts for one of two data dependence constraints that must be observed to insure correctness. A violation of this constraint can reorder statement execution in such a way that a variable is used prior to its having been assigned the proper value.

   Note that if all the definitions for a particular name can be moved from the interval to the landing pad, then statements reached by those definitions may become eligible for movement.

   Since this condition concerns correctness rather than strictness, it also appears in the nonstrict algorithms.

3. A definition for a variable may not be moved if the birthpoint for the variable has already been processed, or if an earlier definition for the same variable was found to be immovable.

   This data dependence constraint is required to insure the correctness of the transformed code. The birthpoint constraint depends on the observation that the birthpoint for a variable either dominates all definitions for the variable in the interval, or it post-dominates them. In the former case, this condition will preclude movement of any of the definitions. This is necessary since the birthpoint must result from the merging of a loop-carried definition and a definition from outside the interval. Moving any definition for such a variable would result in an incorrect value for the variable during the first iteration of the interval.

   The second part of this condition is motivated by similar considerations. Consider the type of join birthpoint that post-dominates the definitions that reach it. These definitions all reference the same variable name and must therefore be executed in topological order. Since the landing pad is executed before the interval, definitions that reach this birthpoint can be placed in the landing pad, provided that the ordering of definitions is respected. Thus, in a topological traversal, no definition can be placed in the landing pad unless all previously encountered definitions are also moved.

The actual construction of an appropriate control structure begins by copying a skeleton of the control structure from the interval into the landing pad. In the landing pad, loop exits are redirected to the bottom of the landing pad. If such a branch is taken in the landing pad, then the loop is guaranteed to exit from its first iteration at the same exit in the interval. The statements and the conditional tests are then processed in dominated topological order. Any code that is found to be movable is simply moved from its position in the interval body to the corresponding position in the landing pad skeleton.

A complication arises when an immovable test is encountered. The test that was naively copied to the skeleton must be removed, along with any additional control structure that is directly or indirectly control dependent on the test. Since an immovable test cannot be decided in the landing pad, control is transferred to its immediate post-dominator in the landing pad. Code that is subsequently found to be control dependent on the test remains in the interval due to the first condition.

When this traversal is finished, unreachable portions of the skeleton can be removed from the landing pad. Such nodes must have been situated between an immovable test and its post-dominator, and thus its sole means of entry was eliminated when the immovable branch was deleted from the landing pad.

[7] A node W is post-dominated by a node V if all paths from W to the program exit include V.

```
while (i < n)



  if n > 3
    then do
      k ← n * 6

      m ← k * i

    end
    else do
      k ← n * 8

    end




  j ← k + 3

  i ← i + 1








end
```

Original Loop

```
   if (i >= n) goto exit
   i.2 ← i.1 ← i
   j.2 ← j.1 ← j
   k.2 ← k.1 ← k
   m.2 ← m.1 ← m
   n.1 ← n
   loop:
1:   birthpoint i.2
     birthpoint j.2
     birthpoint k.2
     birthpoint m.2
     m.4 ← m.2
     if n.1 > 3
2:     then do
         k.3 ← n.1 * 6
         k.5 ← k.3
         m.3 ← k.3 * i.2
         m.4 ← m.3
       end
3:     else do
         k.4 ← n.1 * 8
         k.5 ← k.4
       end
4:   birthpoint k.5
     k.2 ← k.5
     birthpoint m.4
     m.2 ← m.4
     j.3 ← k.5 + 3
     j.2 ← j.3
     i.3 ← i.2 + j.3
     i.2 ← i.3
     if (i.3 < n.1) goto loop
   exit__pad:
     i ← i.3
     j ← j.3
     k ← k.5
     m ← m.4
   exit:
```

Intermediate Form

```
   if (i >= n) goto exit
   i.2 ← i.1 ← i
   j.2 ← j.1 ← j
   k.2 ← k.1 ← k
   m.2 ← m.1 ← m
   n.1 ← n
1:   T1 ← (n.1 > 3)
     if T1
2:     then do
         k.3 ← n.1 * 6
         k.5 ← k.3
       end
3:     else do
         k.4 ← n.1 * 8
         k.5 ← k.4
       end
4: birthpoint k.5
   j.3 ← k.5 + 3
   loop:
1:   birthpoint i.2
     birthpoint j.2
     birthpoint k.2
     birthpoint m.2
     m.4 ← m.2
     if T1
       then do


2:       m.3 ← k.3 * i.2
         m.4 ← m.3
       end







     k.2 ← k.5
4:   birthpoint m.4
     m.2 ← m.4

     j.2 ← j.3
     i.3 ← i.2 + j.3
     i.2 ← i.3
     if (i.3 < n.1) goto loop
   exit__pad:
     i ← i.3
     j ← j.3
     k ← k.5
     m ← m.4
   exit:
```

After Strict Code Motion

Figure 7. Full Example of Strict Code Motion

In Figure 7, the example shows three forms of a program fragment. Note that this is the same example used in Section 4.1 The left column is the high-level source. The center column is the intermediate code after renaming. The right column is the code after being processed by this algorithm. Processing proceeds according to the dominated topological ordering of the control flow graph: the block before the **if**, the **then** block, the **else** and finally the block after the test.

1: The birthpoint of *i.2* (where the value of *i* carried around the loop joins with the value entering from above) is not moved out of the loop, because it is treated as a use for *i.2* that is reached by a definition that has not yet been moved out of the loop. The same is true for the birthpoints of *j.2*, *k.2*, and *m.2*. In fact, none of the join birthpoints that are inserted at the top of a loop will ever be moved out of the loop.

The assignment to *m.2* cannot be moved because its use of m.2 is reached by a definition remaining in the loop.

The computation of $n.1 > 3$ can be moved outside the loop since neither *n.1* nor 3 are redefined within the loop. The temporary *T1* is assigned the value of the test and the branch within the loop tests this temporary rather than recalculating the real condition.

2: The calculation of *k.3* may be moved from the loop since the entire right side is outside the loop or constant.

The assignment to *k.5* may be removed since the birthpoint for *k.5* has not been reached yet.

The assignment to *m.3* is not moved since its use of *i.2* is reached by a definition that is in the loop. This subsequently holds the assignment to *m.4* inside the loop.

3: All of the code in the **else** block is movable in the same manner that the code was movable in the **then** block.

4: The birthpoint *k.5* is movable since both definitions of *k.5* have been moved. However, since a definition for *m.4* was left in the loop, the birthpoint for *m.4* must be left as well.

The assignments to *k.2, m.2* and *j.2* must remain in the loop because their birthpoints have been processed.

The assignment to *j.3* is movable since all definitions for *k.5* have been removed from the loop, and there is no birthpoint for *j.3*.

The definition for *i.3* is not movable because its use of *i.2* is reached by a definition of *i.2* That very definition is therefore held in the loop as well, as is the calculation of the test condition $i.3 < n.1$.

4.3 Structural Common Subexpression Elimination

The STRICT algorithm cannot move code out of an interval unless the control structure that governs the execution of that code is also moved. The goal of the COMMON algorithm, described in this section, is to move code out from under the influence of immovable tests, thereby increasing its likelihood of being eligible for movement. The algorithms described here may be performed on a renamed interval, prior to the application of the STRICT algorithm.

The STRICT and COMMON algorithms are similarly motivated. Algorithms should accommodate secondary effects in a natural way. The motion of simple statements should be accompanied by the motion of their relevant control structures, since since common control structures occur quite often in the context of high-level data abstractions. Consider the example in Figure 8.

```
while i
  if LV
    then b ← f(d)
    else b ← f(e)
end
```

Figure 8.

If procedure integration has incorporated the references to *f*, then a reasonable amount of code should be common to both branches of loop-variant test *LV*. Even if the procedure integration has extensively tailored the call sites, as proposed by Wegman and Zadeck [Wegman85], portions of the incorporated code, due perhaps to addressing and parameter checking, should be common to both successors. Consider the following example:

```
                          if LI
                            then x ← a
                            else x ← b
if LV                     if LV
  then if LI                then if LI
    then do
      x ← a
      y ← c + 1               then y ← c + 1
    end
    else do
      x ← b
      y ← c + 2               else y ← c + 2
    end
  else if LI                else if LI
    then do
      x ← a
      y ← c + 3               then y ← c + 3
    end
    else do
      x ← b
      y ← c + 4               else y ← c + 4
    end
Before Movement           After Movement
```

Figure 9. A Movable Test That is Control Dependent on an Immovable Test

The loop-variant test *LV* ordinarily prevents movement of any code dependent on that test. In this case, the circumstances that determine which value is assigned to *x* are governed only by the loop-invariant test *LI*, because it is common to both sides of the *LV* branch. The assignments to *x* can thus be removed from control of *LV* as seen in the right column of Figure 9. As in the STRICT algorithm, statements such as the assignments to *x* must be accompanied by their associated control structure when moved.

The COMMON algorithm will initially be presented that can move assignment statements above a test, if the assignment statement occurs along all paths from that test. This algorithm will be enhanced to allow commoning of certain classes of control structures as well.

The algorithm is composed of two alternating phases that iterate over all the tests that occur in an interval. Given a test *T*, the first phase constructs collections of statements to be analyzed for possible movement above *T*. Each collection represents statements controlled by one of the possible branches from *T*. In the second phase, the collections are compared to discover which statements are common along all paths from *T*, and each such statement is moved to a single instance above *T*.

The movement is performed so that a particular statement is never moved past more than one of its guards at a time. Thus, in order to insure that a statement will be moved past as many guarding tests as possible, the tests that control the execution of a statement must be considered successively, in an inner-to-outer fashion with respect to conditional nesting. In the two-phase analysis, a test must be considered only after all the tests that it directly or indirectly controls have been considered. This is accomplished by visiting the tests according to a reverse

topological traversal of the control dependence DAG that is constructed for the interval.[8]

Whenever two or more common assignment statements are merged into a single statement and moved above a common guard, the variables on the left sides of those statements must be aliased throughout the remainder of the compilation. Because the alias relationship need never be undone, a disjoint union-find structure can be used to maintain the necessary global dictionary. Each set maintained by the dictionary corresponds to a collection of aliased names. The name by which a set is known, called the *representative name*, is returned by the *find* operation on any member of its associated set. If two variables are aliased, they will always have the same representative name. Therefore, all references to variables will be made via the aliasing dictionary.

4.3.1 Phase One: Identifying Candidates for Movement

Once a test *T* has been chosen as a potential target for commoning, the control dependence graph is used to construct a number of collections of statements that are considered for movement above *T*. Each collection corresponds to one of the conditional branches emanating from *T*, and contains any statement whose only control dependence is on *T* by that branch.[9] It is easy to show that each included statement is also dominated by *T*.

Any assignment that appears in all the collections constructed for *T* must occur on all paths emanating from *T*, and so must be executed regardless of the outcome of *T*. The second phase extracts those common statements and moves them above *T*.

Unless a statement is dominated by *T*, some path that excludes *T* could lead to executing that statement in the original program. Only statements dominated by *T* are considered for commoning.

4.3.2 Phase Two: Selecting and Moving Common Code

Assignments are considered common if their right-hand sides are identical. Since the COMMON algorithm is designed to work on renamed programs, the left-hand sides of any two assignments are necessarily distinct temporary variables in the case of definition birthpoints. When common assignments are promoted to a single instance above a test, the temporary variables that were the targets of the original assignments are unioned into the aliasing dictionary. The representative name for the variables remains as the target of the new assignments. The aliasing dictionary is searched for names occurring in right-hand side expressions to make subsequent expressions eligible for commoning.

Suppose a target *T* is considered with statement collections *C1* and *C2*. The second phase proceeds as follows:

1. All right-hand sides of *C2* are hashed into a dictionary so that any given expression can be quickly checked for inclusion in *C2*.
2. The statements in *C1* are examined in topological order according to the control flow graph. If an assignment is found that is eligible for movement, its right-hand side is looked up in the hash table for *C2*.
3. Whenever a match is found in step two, each of the assignments is moved from its current position, and a new assignment is constructed immediately before the test *T*. The two variables are entered into the aliasing dictionary, and the resulting representative name is used as the target of the new assignment. A Birthpoint can also be moved if those definitions are the only ones that reach the birthpoint.

In order to avoid the rehashing the statements in *C2* after moving an assignment, none of the representative names of variables used in *C2* must change as a result of the new aliasing. This can be accomplished easily when performing the *union* operation by choosing the *C2* name as the result name.[10]

Note that to be eligible for movement, the right-hand side of an assignment must not use any variables for which definitions exist that are dominated by the test *T*. This is trivially satisfied by the above algorithm due to the renaming. If a definition did exist, it would have caused a new name, and the statements' right-hand sides would not have matched.

4.3.3 Commoning of Control Structures

This section presents an extension of the above algorithm, that includes the motion of entire regions of control structures as well as assignment statements. Candidate common structures are identified by the fact that they have common tests and are control dependent and dominated by the same test. This is analogous to the matching of assignments in the previous section, except that conditional expressions rather than right-hand sides are compared.

In general, a moved control structure is not copied exactly. Some of the code contained in the structure may not be common to all instances, or some portions may be immovable due to data dependence considerations. For this reason, the regions to be moved must be editable, in that they can be copied selectively by parts without incurring undue costs. In particular, due to the difficulty of backing out of aliasing and other decisions, situations in which a piece of code is moved and then later pulled back are avoided.

This policy can be enforced by restricting movement to structures in which all required editing is in terms of single-entry single-exit regions of the flow graph. In order to avoid a costly pre-examination of the structure to determine which nodes head such regions, a simpler test is performed to decide whether or not the region under consideration is a series-parallel (S-P) graph--the type that normally occurs in structured programs.[11] In such a graph, there is a one-to-one correspondence between split nodes and join nodes, such that each split node dominates its corresponding join node, and conversely, each join node post-dominates its corresponding split node. Thus, each split node (arising from a conditional branch) necessarily heads a single-entry single-exit region, and thus may be edited from the overall structure without disturbing the other parts of the graph.

---

[8] The control dependence DAG was introduced in Section 4.2. An example of such a DAG is presented in Figure 10.

[9] That is, following that branch of *T* guarantees execution of the statement, whereas some other branch of *T* may avoid executing the statement.

[10] Although this works if only two-way branching is encountered, it can be extended to multi-way branching as well. In this case, collections *C2* through *Cn* will be hashed, and collection *C1* will be traversed as above. In this case, however, it is not possible to keep an aliasing operation from changing representatives for all of the collections *C2* through *Cn*. Instead, update of the global alias dictionary must be deferred, and a simple table linking names from *C1* with its aliases in the other collections is maintained. This suffices throughout processing of test *T* since every variable in *C1* will have at most one alias in each of the other collections. After processing *T*, the information in this simple alias table is merged with the global dictionary.

[11] The decision that only S-P control structures are eligible for movement somewhat limits the power of the COMMON algorithm. However, the S-P restriction is a reasonable compromise between power and cost. Moreover, the S-P restriction does not limit the algorithm when applied to structured programs.

A single pass over the control flow DAG suffices to determine whether a node heads a S-P region.[12]

In phase two of the COMMON algorithm, when common tests heading S-P regions are identified along all branches from test node T, a new copy of the common test is created above T, and the common portions of the S-P regions are copied to form a new region under the promoted test. Any code in the original regions that is not common must remain in those regions, along with any required control structures.

Discovery of common code here is roughly the same as in the simple COMMON algorithm. The only difference is that conditions guarding a statement in the S-P region must be hashed along with the statement itself. In this way only statements in the same place relative to the entrance of the S-P region will be considered common.

As in the simple COMMON algorithm, any assignments that are commoned as a result of structure commoning must cause updates to the aliasing dictionary in order to allow the commoning of later assignments and tests that make use of the aliased variables.

In order to ease the process of copying code from common control structures, traversal of the structure in the CI collection is performed according to the dominated topological order defined in Section 3.4. The domination aspect of this ordering allows the S-P diamonds to be copied easily without having to close up dangling control flow paths.

4.4 A Nonstrict Algorithm

The STRICT algorithm disallows movement of any code that is control dependent on immovable tests. In this algorithm, strictness is relaxed, thus increasing the opportunities for movement substantially, although the profitability of some moves becomes questionable. This algorithm is based on the assumption that it is generally more profitable to execute every statement in the interval once if the interval is to be executed, rather than possibly execute some statement many times by leaving it in the interval. As in the STRICT algorithm, all motion is from the body of the interval to the landing pad.

Like the STRICT and COMMON algorithms, the motion of a control structure precedes the motion of its guarded statements. If that control structure is itself dependent on some immovable test, then motion of the control structure improves profitability of statements that it guards, but such statements are not executed strictly. Further, the motion of a control structure can free other code for movement, but at the cost of strictness. Strictness can only be guaranteed for statements that are control dependent only on movable tests.

As in the STRICT algorithm, the intervals of a program are traversed from their innermost to outermost nesting. Upon visiting an interval, the DAG that corresponds to the body of the interval is traversed by the dominated topological order described in Section 3.4. Upon visiting a node that is control dependent only on movable tests, that node can be immediately placed in the landing pad, since its control dependent predecessors have already been visited by the dominated topological traversal. Thus far, the motion is strict. However, a node that is control dependent on some immovable test cannot be guarded by that test in the landing pad. For nonstrict execution, statements of such nodes can still reside in the landing pad, and the location chosen for those statements determines the profitability of the nonstrict motion. In particular, the node may be control dependent on some movable test, which, by nature of the traversal, already appears in the landing pad.

More specifically, the motion of control structures must account for the types of branches upon which a node N is control dependent.[13]

1. If N is control dependent on an immovable branch, then its execution cannot be strict with respect to that branch. Suppose N is control dependent on branch b of node P. The nodes along the frontier determined by a traversal from P along its b branch are executed under conditions that require execution of node N. Reaching any node of that frontier must guarantee the execution of node N, and so the branches of all nodes on that frontier must be targeted to node N. Similarly, suppose node U unconditionally transfers control to node N. By the above mechanism, the branch from U may have been redirected to some node other than N. The same mechanism must be applied to the frontier determined by U, which may consist of only U if no redirection was performed on U.[14]
2. If N is control dependent only on movable branches, then N can be placed as in the original control flow DAG, provided that the relevant branches are still unresolved. If any branch relevant to the placement of N is already resolved, then N is placed as if it were under control of some immovable branch.

This nonstrict motion results in three levels of profitability, each associated with a set of criteria that determines the correctness of moving statements.

1. Nodes that are control dependent only upon movable branches can be executed strictly. Thus, the conditions described in the STRICT algorithm apply to the motion of statements from such nodes.
2. Nodes that are control dependent on immovable branches are unconditionally executed with respect to the immovable branches. The conditions that permit motion in this case are described in Section 4.4.2.
3. Due to the placement strategy, some nodes may be control dependent only on movable branches, yet be placed as if they were under control of some immovable branch. Such nodes are executed profitably. The motion is in general more profitable when there are many movable branches between an immovable branch and N. The conditions that permit motion in this case are described in Section 4.4.3.

4.4.1 An Example of Control Structure Copying

Consider the control flow DAG shown in the left column of Figure 10. The arcs are directed, with the destination of an arc lower in the figure than its source. Using the techniques of [Ferrante84], the control dependence graph is contructed as shown in the middle column of Figure 10.

[12] Consider a test node T. Visit the successors of T in topological order. Each node will be labeled with the conditions that guard its execution (relative to conditions on T). For the graph to be S-P, that labeling may be represented by a conjunctive clause, with negated terms where false branches are taken. At each join node, the conditions labeling each of the predecessors must be identical, except the last term; one should be the complement of the other. The list becomes empty at an exit node. If the labeling cannot be represented in the manner described here, then T does not head a S-P region, other than the trivial region consisting solely of T itself.

[13] For purposes of this discussion, a node that is independent of all tests is considered to be trivially dependent on a movable test that always causes execution of that node.

[14] This process can be accomplished efficiently by maintaining frontier pointers in the nodes and adjusting the pointers using path-compression techniques.

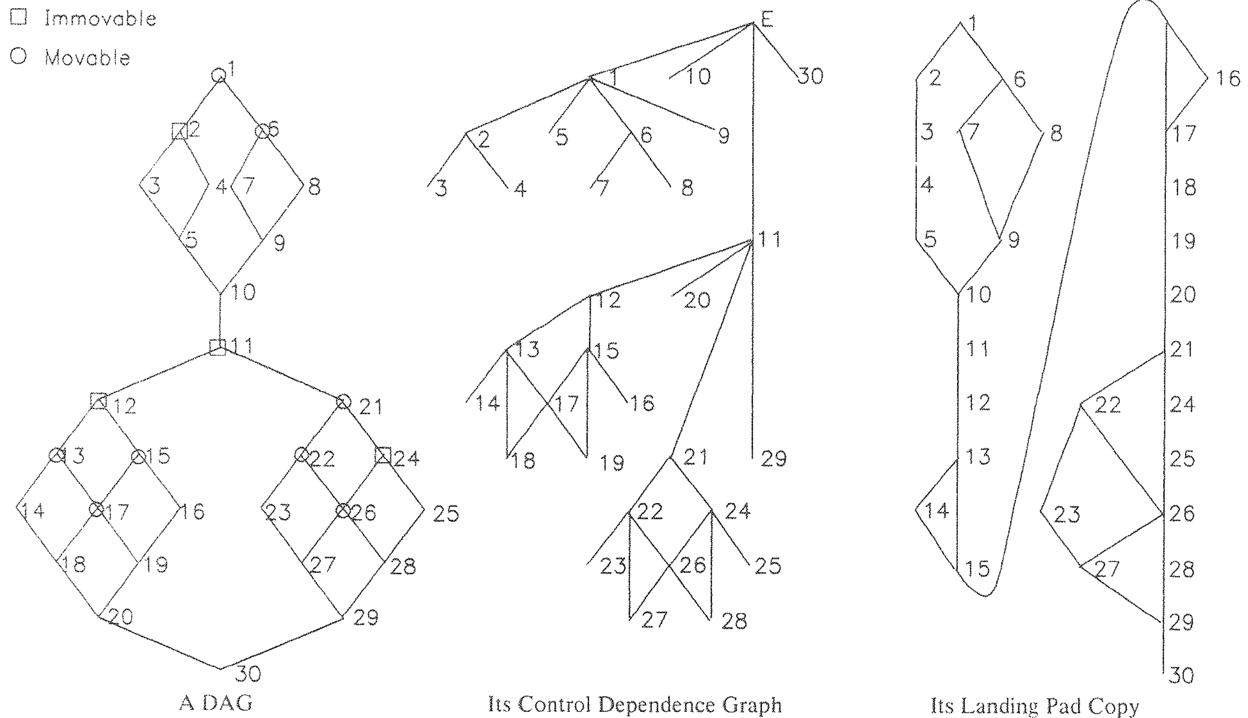

Figure 10.

The right column shows the results of applying the above techniques to the DAG and its control dependence graph. This section examines in detail how code can be copied into the landing pad. Thus far, no issues of safety are addressed. This section addresses only where code can be placed in the landing pad, given that the conditions that determine safety are satisfied. The algorithm traverses the DAG in the order shown by the node labels. Each node of the DAG corresponds to a straight-line sequence of code terminated by a branch. Node 1 is control dependent only on program entry.[15] The code of node 1, including the movable branch with which it exits, can be moved. Node 2 is control dependent on that movable branch, so its code can be moved up to its immovable branch. Nodes 3 and 4 are control dependent on the immmovable branch of node 2, so they are executed unconditionally with respect to node 2. The branch of node 3, previously targeted for node 5, is resolved prematurely to cause this unconditional execution. The branch from node 1, which is still open, can protect node 6, which is dependent only on the movable branch from node 1. Because the relevant branches are still open and only movable branches are the source of control dependences, nodes 7 and 8 are placed under control of node 6, node 9 is the successor of nodes 7 and 8, and nodes 10 and 11 follow the join of nodes 5 and 9. Nodes 12 and 13 are control dependent on immovable branches and follow node 11 in the landing pad; there are no relevant branches to close. Node 14 is control dependent on the movable branch of node 13, and is appropriately guarded in the landing pad.

Node 15 deserves special attention. It is control dependent on the immovable branch of node 12. All unresolved branches between nodes 12 and 15 must be closed and targeted to node 15. Node 13 contains such a branch. The branch has two successors; one successor is already closed and targeted to node 14. The other branch, originally targeted for node 17, must now be targeted for node 15. Thus, node 15 is executed less profitably than could be expected. In particular, arriving at node 14 implies that node 15 is not executed. However, node 15 must be placed such that if node 12 executes (the node upon which it is control dependent), then node 15 executes. The conditions that determined the placement of nodes 14 and 15 also apply to the placement of nodes 16 and 17.

Consider nodes 18, 19, and 20. They are dependent only upon movable branches, but these branches have already been resolved. They are therefore placed as if they depended on immovable branches.

Node 21 is dependent on the immovable branch of node 11, and all branches for nodes 11 through 21 are already resolved. Node 21 concludes with a movable branch, which is copied into the landing pad. Node 22 is then placed under control of that branch. Similarly, nodes 23 and 24 are placed under control of the movable branches of node 22 and 21, respectively. Nodes 25 is dependent on the immovable branch of node 24 so its execution follows node 24. Node 26 is control dependent on nodes 24 and 22, thus forcing all unresolved branches in nodes 22 through 26 to be targeted to node 26. The branches from nodes 22 and 25 are thus targeted for node 26. Node 27 can be placed as in the original DAG since the relevant branches are still open. Node 28 is control dependent on the immovable branch at node 24. All branches from 24 through 28 are targeted for node 28. Nodes 29 and 30 are similarly placed

4.4.2 Unconditionally Executed Statements

Statements executed unconditionally are more restricted with respect to code motion. These restrictions are required to assure

[15] Like unconditional branches, program entry is a movable branch.

that the semantics of the program are preserved in the absence of normal control structures that guard the statements.[16]

1. Definitions that reach uses within the statement must come from statements outside the landing pad or other unconditionally executed statements already moved to the landing pad.

   This restriction does not allow these unconditionally executed statements to use values generated by sections of code are either strictly or profitably executed since these values may not be available.

2. A variable defined by this statement must cover all of its uses, i.e. it must be the only definition of that name to reach any use of that name.

   Since none of the control structure is moved with this code, multiple definitions for the same variable cannot be distinguished. This restriction will only affect assignments between temporary variables since the renaming guarantees that all expressions with non-trivial right-hand sides generate unique names.

4.4.3 Profitably Executed Statements

Statements of this type are control dependent on a chain of branches. The first member of that chain is an immovable branch and it is followed only by movable branches. Consider a *region* of nodes, defined as those nodes dominated by the first movable branch of that chain.[17] Statements of a region are eligible for motion, subject to the first criterion of the unconditionally executed statements and two additional criteria:

1. If a variable defined by the statement has a birthpoint, then that birthpoint must be in the region. This condition guarantees that the sequencing of the definitions that reach the birthpoint is preserved in the landing pad.
2. Definitions for variables may be moved out if the birthpoint of the variable has not been processed and no earlier definitions for that variable remain within the interval.

4.5 The Combine Algorithm

The COMMON algorithm given in Section 4.3 is strict in the sense that it does not move an expression above a branch unless the expression is common along all paths from that branch. If some path failed to include the expression, then moving the expression above the branch causes its unconditional execution with respect to the branch.

Consider a situation in which an expression is not *quite* common along all paths from a branch. For example, from a three-way branch, some expression could occur along two paths but be absent on the third path. The COMMON algorithm is unable to move the expression since it is not common along all paths from the branch. Now suppose that the NONSTRICT code motion algorithm is applied to this example, and the test was not movable. Both instances of the expression could be moved above the branch, causing redundant execution of the expression.

A solution to this problem is to maintain a dictionary of available expressions that are unconditionally executed in the landing pad. An expression contained in the dictionary need not be placed again in the landing pad. References to expressions that are deleted in this manner must be accommodated by aliasing the left-hand side variables, as discussed in the COMMON algorithm.

5.0 Concluding Remarks

Although the algorithms presented in this paper are improvements over existing code motion techniques, the algorithms have the following shortcomings:

1. The success of the STRICT and NONSTRICT algorithms is greatly affected by the COMMON algorithm, since the COMMON algorithm transforms nonstrict code into either strict or profitably movable code. The COMMON algorithm has a heuristic component that limits its effectiveness to obtain a polynomial time bound.
2. Statements moved by the algorithms must receive the *same* values on *all* iterations of the loop. Consider the program shown in Figure 11:

```
while ()
  t ← i + 5

  i ← nonmovable

  t ← i + 5
end
```

Figure 11. Variant but Removable code

   The value of *i* changes for each iteration. As a consequence, the techniques presented here fail to move either of the assignments to *t* from the loop. It is correct to move either the first before the loop or the second after the loop. This example can be accommodated by unrolling the loop once, copying all of the right-hand sides of statements from the loop to the entry landing pad where they are unconditionally executed. The statements of the loop that are redundant are then eliminated, as are the statements in the landing pad that never contribute to the code within the loop. This case is easily analyzed for code that is unconditionally executed in the landing pad. Since the goals of STRICT and NONSTRICT are to execute code as profitably as possible, control structures accompany the motion of statements.

3. The success of the algorithms often depends on the order in which nodes are considered. Since the dominated topological order is not a unique total order, the solutions can be order dependent. The dominated topological order can usually select any successor of a branch as the next node to visit. The result of this freedom is that some assignments to join birthpoint names may not be removed. In Figure 12, if the **then** block is evaluated first, neither assignment to *t.3* is removed. If the **else** block is evaluated first, the assignment to *t.3* in that block is removed. This type of order dependence is not really important since in neither case is the birthpoint for *t.3* movable. Without a movable birthpoint, no real uses of *t.3* can be moved.

[16] Of course the loop condition still guards the statement on entrance to the landing pad.
[17] The nodes that comprise this region are easily determined by the dominated topological traversal.

```
if movabletest
  then do
    t.1 ← notmovable
    t.3 ← t.1
  end
  else do
    t.2 ← movable
    t.3 ← t.2
  end
  birthpoint t.3
```

Figure 12. Order Dependence

4. In the NONSTRICT algorithm, nodes are placed in the order determined by the dominated topological traversal. Upon placing a node *N* that is control dependent on an immovable branch, certain branches that are unresolved in the landing pad are targeted on node *N*. Such branches may be capable of guarding a subsequently placed node, but the NONSTRICT algorithm precludes reopening resolved branches. The work of Ferrante and Mace [Ferrante85] can place nodes such that they execute more profitably. For programs that correspond to some correct sequential order, the control dependences suffice in placing nodes. Since the immovable tests cannot be decided in the landing pad, such tests imply the concurrent execution of all successors, and data dependences might exist that preclude a correct sequential order. The algorithm of Ferrante and Mace can place the nodes of such programs, but either the control flow order is violated or tests have to be duplicated to maintain that order. Since that order is important to the algorithms, this approach is not used.

Although the above arguments appear to raise doubts as to the effectiveness of the algorithms presented in this paper, recall that one goal of these algorithms is to effectively accommodate abstractions. Code moved from intervals retains its relevant surrounding control structure, which makes these algorithms attractive for the stylized code that results from the heavy use of abstractions or other integrated subroutine mechanisms.

In the worst case, the algorithms may swell the code size by 2^d, where d is the depth of interval nesting. In the worst case, all of the control structure is copied to the landing pad for each interval and none of that control structure is deleted from within the interval. This situation, albeit unlikely, can be controlled if algorithms are restricted to some fixed nesting of inner intervals. For example, Scarborough has observed that most of the execution time is spent in inner loops [Scarbourgh80] and the IBM Fortran H compiler therefore performs optimizations only over the deeply nested intervals.

6.0 Acknowledgements

The authors wish to thank their colleagues who provided assistance with the formulation and preparation of this paper. Alan Demers assisted with an early formulation of the NONSTRICT algorithm. Ashok Chandra assisted with some of the graph manipulation algorithms. Fran Allen, Barry Rosen, Peter Markstein, and Mark Wegman gave many valuable comments that were useful in the development of this paper. The author also thank Karl Ottenstein and Mary Mace for their helpful comments.

Bibliography

[Allen70] Allen, F. E., Control flow analysis. *SIGPLAN Notices*, July 1970.

[Allen83] Allen, J. R., Dependence Analysis for Subscripted Variables and Its Application to Program Transformations. Published by Computer Science Department, Rice University, Houston Texas, April 1983.

[Banerjee79] Banerjee, U., Speedup of Ordinary Programs. Published by University of Illinois at Urbana-Champaign, Oct. 1979, DCS Report No. UIUCDCS-R-79-989.

[Burke84] Burke, M., An interval approach toward interprocedural analysis. Published by IBM, July, 1984, RC 10640 #47724.

[Chaitin81] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., Markstein, P. W., Register allocation via coloring. *Computer Languages*, 1981, vol. 6, page 47-57.

[Chaitin82] Chaitin, G. J., Register allocation and spilling via graph coloring. *Conference Record of the SIGPLAN '82 Symposium on Compiler Construction*, June 1982, page 98-105.

[Cooper83] Cooper, K. D., Interprocedural data flow analysis in a programming environment. Published by Mathematical Sciences Department, Rice University, Houston Texas, 1983.

[Ferrante83] Ferrante, J., Ottenstein, K. J., A program form based on data dependency in predicate regions. *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1983.

[Ferrante84] Ferrante, J., Warren, J. D., A program dependence graph and its use in optimization. Published by Springer-Verlag*Lecture Notes in Computer Science*, 1984, page 125-132.

[Ferrante85] Ferrante, J., Mace, M., On Linearizing Parallel Code, *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1985.

[Graham76] Graham, S. L., Wegman, M., A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, Jan. 1976, vol. 23, no. 1, page 172-202.

[Lowry69] Lowry, E. S., Medlock, C. W., Object Code Optimization. *Communications of the ACM*, 1969, vol. 12, no. 1, page 13-22.

[Morel79] Morel, E., Renviose, C., Global optimization by supression of partial redundancies. *Communications of the ACM*, Feb. 1979, vol. 22, no. 2, page 96-103.

[Myers81] Myers, E. W., A precise interprocedural data flow algorithm. *Eighth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1981, page 219-230.

[Reif77] Reif, J. H., Lewis, H. R., Symbolic evaluation and the global value graph. *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1977, page 104-118.

[Reif81] Reif, J. H., Tarjan, R. E., Symbolic program analysis in almost linear time. *SIAM Journal of Computing*, Feb. 1981, vol. 11, no. 1, page 81-93.

[Reif82] Reif, J. H., Lewis, H. R., Efficent symbolic analysis of programs. Published by Harvard University, Aiken Computation Laboratory, 1982, no. TR-37-82.

[Schwartz73] Schwartz, J. T., On Programming, An Interim Report on the SETL Project, Installment II: The SETL Language and Examples of Its Use. Published by Computer Science Department, Courant Institute of Mathematical Sciences, New York University, Oct. 1973.

[Schwartz79] Schwartz, J. T., Sharir, M., A design for optimizations of the bitvectoring class. Published by Courant Institute of Mathemtatical Sciences, Computer Science Department, New York University,, Sept. 1979, no. 17.

[Tarjan74] Tarjan, R. E., Testing flow graph reducibility. *Journal of Computer and Systems Sciences*, Dec. 1974, vol. 9, page 355-365.

[Wegman85] Wegman, M., Zadeck, F. K., Constant propagation with conditional branches. *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1985, page 291-299.

[Wolfe82] Wolfe, M. J., Optimizing Supercompilers for Supercomputers. Published by Computer Science Department, University of Illinois at Urbana-Champaign, 1982.