Comput. Lang. Vol. 8, No. 2, pp. 69–76, 1983
Printed in Great Britain. All rights reserved

0096-0551/83 $3.00 + 0.00
Copyright © 1983 Pergamon Press Ltd

CHARACTERIZATION OF PROGRAM LOOPS IN CODE OPTIMIZATION

D. M. DHAMDHERE and J. S. KEITH
Computer Centre, I.I.T. Bombay, India

(*Received* 20 *January* 1982; *revision received* 5 *April* 1983)

Abstract—Recent work in code optimization has led to development of new unified optimizing transformations[6,7]. Application of these transformations requires solution of bi-directional data flow problems over program flow graphs using iterative solution techniques. Appropriate characterization of program loops in the flow graph is necessary so as (i) not to hinder code movement, etc., and (ii) restrict optimization overheads to low levels. This paper reviews alternate loop characterizations and proposes a characterization which leads to minimum overheads and has certain nice properties from a practical viewpoint.

1. INTRODUCTION

Program loops are a fertile source of optimization possibilities. Program optimizers therefore spend a considerable amount of time and effort on the optimization of program loops. Common program transformations applied for loop optimization are:

(i) movement of loop invariant computations to places outside the loops;
(ii) strength reduction, i.e. replacement of operations consuming large amounts of processor time by operations which are performed faster;
(iii) assignment of registers to hold values frequently accessed within a loop.

Conventional approach to loop optimization entails applying individual optimizing transformations to program loops one after another. Thus, after identifying a program loop, the transformation of moving loop invariant computations out of a loop could be applied followed by the strength reduction optimization for computations remaining within the loop[1,2,3]. This approach requires identification of program loops through analysis of the control flow within the program (the so called *proper loops*[4]). Since this involves considerable effort on the part of the optimizer, it results in high optimization costs. Certain optimizers like the BLISS optimizing compiler[5] attempted to reduce optimization cost by restricting optimization to program loops implemented through iteration control constructs like *while* ... *do*, *repeat* ... *until*, etc. of the source language. This eliminates the need to identify program loops though at the cost of failure to optimize loops implemented through *if* ... *then goto* ... constructs.

Recent research in code optimization has attempted to reduce optimization costs through unification of certain conventional transformations. One such unification which might be termed *generalized code movement* unifies common subexpression elimination, code hoisting and loop invariant movement within one framework[6,7]. This unified framework can apply code movement to arbitrary program topologies without the need to identify program regions or program loops, thus bringing about a significant reduction in the optimization costs. Another important unification is achieved by integrating the optimization of strength reduction with generalized code movement[8,9], which on one hand enhances profitability of program optimization and on the other hand leads to significant savings in the optimization effort. Both the Morel–Renvoise[7] and Joshi–Dhamdhere[9] algorithms use program data flow analysis techniques to collect information regarding the definition and use of program variables preparatory to the optimization. A program is represented in the form of a *program flow graph* in order to apply the data flow analysis equations. Program loops implemented through *repeat* ... *until* ... or similar HLL constructs are easily represented in the flow graph, however representation of *while* ... *do* loops in the program flow graph poses certain interesting problems. Variant loop characterizations are possible, each with different attendant optimization costs. This paper presents a characterization for *while* ... *do* loops which leads to minimum optimization cost, and also enjoys certain other advantages from a practical viewpoint.

This paper is organized into 4 sections. Section 2 introduces the notation and certain concepts regarding program flow analysis. The Morel–Renvoise and Joshi–Dhamdhere unifications are then reviewed. Section 3 develops a practical characterization for *while*... *do* loops and illustrates its properties and advantages. For reasons of simplicity and brevity, illustrative examples used throughout this discussion are based on the Morel–Renvoise algorithm; however the issues being discussed are equally applicable to the Morel–Renvoise and Joshi–Dhamdhere algorithms. The last section summarizes the utility of our characterization from a practical viewpoint.

2. THE UNIFIED OPTIMIZING TRANSFORMATIONS

2.1 *Notation and definitions*

Definition 2.1. A *basic block* of a program is a sequence of instructions $\pounds = (i_1, i_2, \ldots, i_m)$ such that only i_1 can be the destination of a branch instruction and only i_m can be a branch instruction.

Definition 2.2. A *program flow graph* for a program P is a triple (N, E, n_0) where N is the set of nodes of the flow graph such that each node $n_i \in N$ is a basic block of program P. E is the set of edges (n_i, n_j), $n_i, n_j \in N$, such that control can pass from the basic block n_i to the basic block n_j in P. n_0 corresponds to the unique entry node of program P.

Definition 2.3. In a program flow graph (N, E, n_0), a node n_i is a *predecessor* of n_j iff there exists an edge (n_i, n_j) in E. A node n_k is an ancestor of n_j iff there exists a path from n_k to n_j in the program flow graph. The terms *successor* and *descendant* are analogously defined.

Pred (n_i), *ancestor* (n_i) represent the set of all predecessors and ancestors of node n_i in the program flow graph respectively. Similarly *succ* (n_i) and *descendant* (n_i) represent the set of all successors and descendants of n_i.

Definition 2.4. An expression e_i is said to be *available* at a program point in a flow graph if every path from the initial node to that program point contains an evaluation of e_i not followed by a definition of any operands of e_i.

2.2 *The Morel–Renvoise and Joshi–Dhamdhere algorithms*

The Morel–Renvoise algorithm[7] performs program optimization by suppressing partial redundancies in the program. The occurrence of an expression e_i in block b is *partially redundant* if (i) the expression is locally anticipable, i.e. evaluation of the expression is not preceded in the block by a definition of any of its operands, and (ii) the expression is *partially available* at b, i.e. there exists a block $b^* \in$ pred (b) such that e_i is available at the exit of b^*. A partially redundant computation in block b can be made totally redundant by inserting computations of e_i in some blocks $\{b'\} \subset$ ancestor (b) such that e_i becomes available at entry of block b. The set $\{b'\}$ should be so selected that:

(i) there does not exist a definition of any operand of e_i along the path from a block $\{b'\}$ to block b;
(ii) on every path from the initial node n_0 to block b which does not pass through a b′, there exists a node b_k such that e_i is evaluated in b_k and there does not exist a definition of any operand of e_i along the path from b_k to block b;
(iii) selection of the set $\{b'\}$ is optimally profitable from the viewpoint of execution time of the optimized program.

The concept of partial redundancy subsumes total redundancy, hence common subexpressions become a special case of partially redundant expressions with $\{b'\} = \phi$. Partial redundancy also extends to loop invariant expressions within a loop, since an invariant expression is available along the back edge implementing the loop.

The Morel–Renvoise algorithm performs code movement using the data flow equations shown in Table 1. $AVIN_b/AVOUT_b$ are properties associated with a block reflecting the availability of an expression at its entry/exit. This is a standard forward-flow problem in flow analysis. Anticipability of an expression at entry/exit $(ANTIN_b/ANTOUT_b)$ is computed using the property of local anticipability and the absence of a definition of an operand of e_i. $PAVIN_b/PAVOUT_b$

Table 1. The Morel–Renvoise algorithm

$$AVOUT_b = COMP_b + AVIN_b . TRANSP_b$$

$$AVIN_b = \prod_{p \in pred(b)} AVOUT_p$$

$$ANTIN_b = ANTLOC_b + TRANSP_b . ANTOUT_b$$

$$ANTOUT_b = \prod_{s \in succ(b)} ANTIN_s \qquad (2.1)$$

$$PAVOUT_b = COMP_b + TRANSP_b . PAVIN_b$$

$$PAVIN_b = \sum_{p \in pred(b)} PAVOUT_p$$

$$PPIN_b = CONST_b . (ANTLOC_b + TRANSP_b . PPOUT_b) . [\prod_{p \in pred(b)} (PPOUT_p + AVOUT_p)] \qquad (2.2)$$

$$PPOUT_b = \prod_{s \in succ(b)} PPIN_s$$

$$INSERT_b = PPOUT_b . \neg AVOUT_b . (\neg PPIN_b + \neg TRANSP_b) \qquad (2.3)$$

Where:

$COMP_b$ = true iff block b contains a computation of the expression not followed by a definition of any of its operands;
$TRANSP_b$ = true iff block does not contain a definition of any operand of the expression;
$CONST_b = ANTIN_b . (PAVIN_b + \neg ANTLOC_b . TRANSP_b)$. (2.4)

reflect partial availability of an expression at block entry and exit. This is used as a criterion to block mere proliferation of code without any attendant gain. $PPIN_b/PPOUT_b$ indicate whether it is possible to place e_i at entry/exit of a block according to the criteria of feasibility and safety of code placement. From the blocks for which $PPIN_b/PPOUT_b$ are true, equation (2.3) selects the blocks which should constitute the set $\{b'\}$ where computations of e_i are to be inserted.

We will not elaborate on the basis of the equations here. Interested readers are referred to the original paper[7] for relevant details or to[9] for a slight improvement of the original algorithm. Here we will only introduce the concept of *safety* of code movement and see how it is incorporated in the Morel–Renvoise algorithm. Use of the anticipability term in equations (2.2) implies that a computation is inserted in a block only if the computation occurred along all paths through that block in the original program. In other words, no new computations would be inserted along any path. Thus, no exceptional situations like arithmetic overflows/underflows would occur in the optimized program which would not have occurred in the original program. This is the criterion of safety of code movement[10]. Figure 1 illustrates the results of applying the Morel–Renvoise algorithm to a program. Evaluation of a*b in block 2 is totally redundant, hence it is eliminated. a*b of block 4 is partially redundant because of its loop invariance. It is therefore hoisted out of the loop and inserted in block 3.

Data flow equations for the basic Joshi–Dhamdhere algorithm are shown in Table 2. This algorithm replaces a high strength expression v_i*constant by a reference to temporary location t_i. A Q-unit definition of t_i is placed ($QINSERT_b$ = true) in a path following a Q-unit definition

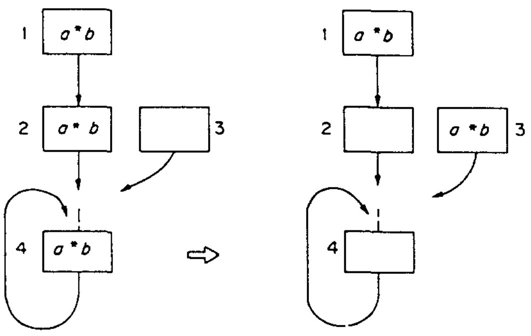

Fig. 1

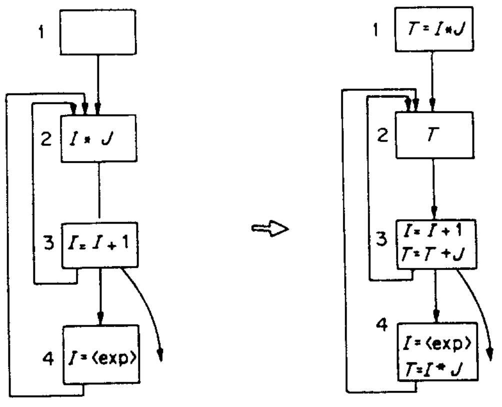

Fig. 2

Table 2. The Joshi–Dhamdhere algorithm

$$\mathrm{XPPIN}_b = (\mathrm{ANTLOC}_b + \mathrm{XTRANSP}_b . \mathrm{XPPOUT}_b) . [\prod_{p \in pred(b)} (\mathrm{XPPOUT}_p + \mathrm{AVOUT}_p)]$$

$$\mathrm{XPPOUT}_b = \prod_{s \in succ(b)} \mathrm{XXPIN}_s$$

$$\mathrm{QINSERT}_b = \mathrm{XPPOUT}_b . \neg\, \mathrm{AVOUT}_b . (\neg\, \mathrm{XPPIN}_b + \neg\, \mathrm{XTRANSP}_b)$$

$$\mathrm{OINSERT}_b = \mathrm{XPPOUT}_b . \mathrm{OUNIT}_b . \neg\, \mathrm{QINSERT}_b$$

Where:

OUNIT_b = true iff block b: (i) does not contain a Q-unit definition of variable v_i, (ii) contains a 0-unit definition of v_i not followed by an expression involving v_i;

$\mathrm{XTRANSP}_b = \mathrm{TRANSP}_b + \mathrm{OUNIT}_b$;

ANTLOC_b.AVOUT_b, TRANSP_b as in Morel–Renvoise algorithm.

v_i = ⟨expression⟩, while a 0-unit definition of t_i is placed ($\mathrm{OINSERT}_b$ = true) in a path following a 0-unit definition $v_i = v_i +$ constant. An illustration of strength reduction is contained in Fig. 2. QINSERT = true for blocks 1 and 4, while OINSERT = true for blocks 3. Space restrictions preclude any detailed discussion of this algorithm. Interested readers are referred to Ref. [9] for relevant details and certain refinements of this algorithm.

In both the algorithms, information flows from a block to its predecessors as also to its successors. This bi-directional data flow precludes the possibility of using faster techniques of solving data flow equations. The solutions are therefore obtained by iterating over the equations until the properties settle to their final values. It is reported that 4 to 6 iterations are required for the solution of the data flow problems[7,9]. Joshi[11] proves that the upper limit on the number of iterations required using depth-first numbering of blocks is $d' + 2$ where d' is the depth of an augmented program flow graph P*.

3. CHARACTERIZATION OF LOOPS

Figure 3 illustrates the representation of a loop as envisaged by conventional loop optimization techniques. A synthetic block SYNTH is placed along all entry-edges of the loop. Loop invariant computations are moved from the loop body to SYNTH so that they are evaluated only once prior to loop entry instead of being evaluated for every iteration of the loop. Safety considerations restrict the scope of code movement to loop invariant expressions occurring in blocks which lie along all paths from loop entry to loop exits (the *articulation blocks* of the loop[4]). This characterization is primarily applicable to *repeat*... *until* kind of loops implemented either through HLL constructs or through *if*... *then*... *goto*... conditionals. An important property of these loops which facilitates their optimization is that the loop body is traversed at least once whenever control reaches the ENTRY blocks during program execution. Thus, moving a*b into SYNTH does not constitute insertion of a new computation along a program path if care is taken to see that a*b occurred in an articulation block of the loop.

This characterization cannot be extended naturally to *while*... *do* loops—whether indexed or non-indexed. Consider the illustration of Fig. 4 where a*b occurring within an articulation block of the loop is moved into SYNTH. Since the loop test is carried out at the start of loop, it is possible that during execution a loop may not be executed even once. Placement of a*b into SYNTH is

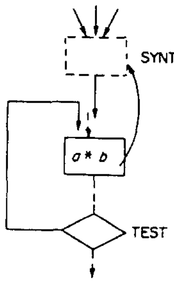

Fig. 3

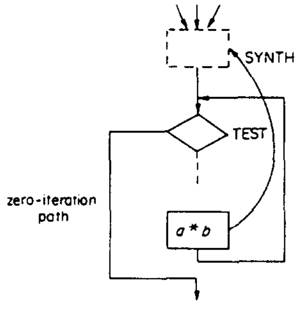

Fig. 4

thus unsafe unless the loop exit block or its successors also contained evaluations of a*b. Since the Morel–Renvoise algorithm incorporates the safety criteria, such code would not move out of the loop at all. Considering the preponderance of simple *while* ... *do* and indexed *while* ... *do* constructs in contemporary higher level programming languages (e.g. *for* statements of PL/I, Algol, Pascal, the new DO of Fortran-77 etc.), this limitation is very severe and needs to be eliminated for worthwhile program optimization.

Figure 5 shows a possible characterization of the *while* ... *do* loop (hereafter referred to as model A characterization) which eliminates above drawbacks. The synthetic block SYNTH is placed under the loop control condition housed in TEST1. TEST2 houses an identical condition and is part of the loop. If the loop concerned is an indexed *while* ... *do*, then control variable initialization would be housed in TEST1 or in SYNTH according to language semantics, while incrementation etc. could be housed in TEST2. Duplication of the loop test permits loop invariant computations to move out of the loop. In case of nested loops, movement of the invariant computations through successive levels of nesting would be governed by safety vis-a-vis the zero iteration path. If the computations are anticipated at entry to the DO-SUCC block, then they are automatically candidates for movement across the TEST1 block.

The cost of using the model A characterization in an optimizing compiler is quite considerable. Two synthetic blocks TEST1 and SYNTH have to be introduced in the program flow graph to facilitate movement of loop invariant code. Those blocks have to take part in the data flow analysis of Morel–Renvoise or Joshi–Dhamdhere algorithms. If we consider a program wherein 10% of the basic blocks are the TEST blocks due to *while* ... *do* constructs, this loop characterization would lead to insertion of 20% new blocks, hence 20% overheads. In the following we develop an alternative characterization aimed at reducing these optimization overheads.

An alternative characterization

Figure 6 illustrates an alternative characterization and an exploded view of the same which resembles the characterization model A. In this characterization no new blocks need to be introduced in the program graph to represent a *while* ... *do* loop. Under the assumption made above (i.e. assuming 10% of program blocks are *while* ... *do* blocks), there is no change in the number of program blocks as against a 20% increase in the same when model A was used. Figure 7 illustrates a modified view of this alternative characterization (we will call this the model B characterization) which we propose as the model for use in program optimization. We will first prove the equivalence of this model to the original model of Fig. 5 from the viewpoint of optimization, and then consider the practical aspects of its use.

Definition 3.1. An *empty* block is one which does not contain any computations whatsoever, i.e. does not contain any expression evaluations or any variable definitions.

Definitions 3.2. An *inert* block is one which, when inserted (or existing) along one or more edges

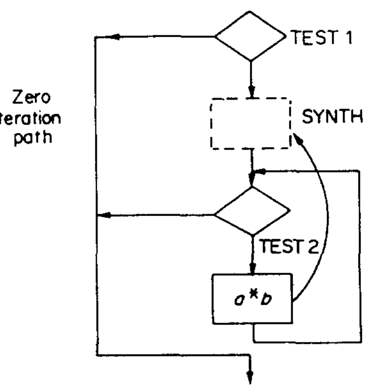

Fig. 5. Characterization model 'A'.

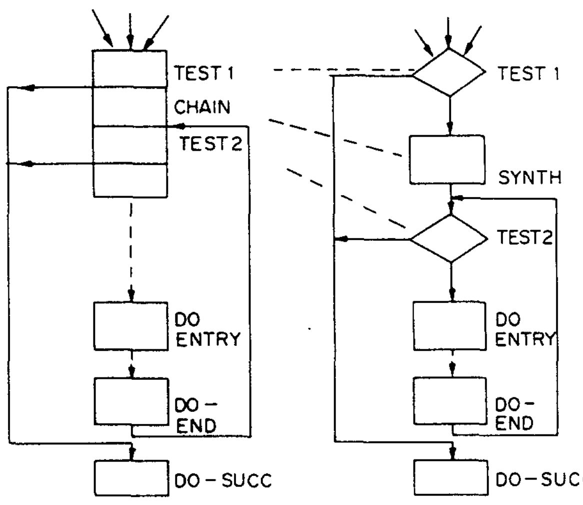

Fig. 6

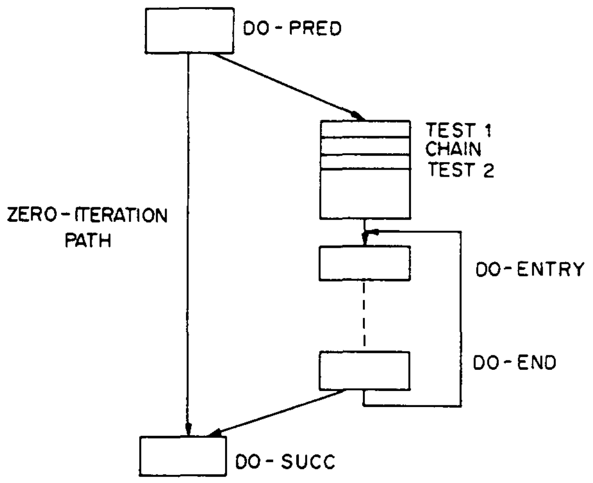

Fig. 7. Characterization model 'B'.

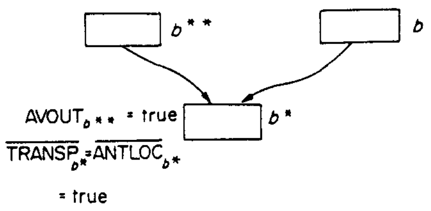

Fig. 8

incident on a program block or emanating from it, does not alter the data flow properties of the program in a way which would affect the movement of code over the program. A block b is *inert* if it satisfies one of the following conditions:

(i) block b is (a) empty, or (b) devoid of any definitions;
(ii) if there exists in b a definition of one of the operands of an expression e_i, then the following conditions are satisfied:

(a) $\forall\, b^* \in \text{pred}(b)$, $\text{AVOUT}_{b^*} = \text{false}$,

or

$\forall\, b^* \in \text{succ}(b)$, $b^{**} \in \text{pred}(b^*) \ni \text{AVOUT}_{b^{**}} = \text{true}$,

$\overline{\text{TRANSP}_{b^*}}\,.\,\overline{\text{ANTLOC}_{b^*}} = \text{true also}$,

(b) $\forall\, b^* \in \text{succ}(b)$, $\text{ANTIN}_{b^*} = \text{false}$,

or

$\forall\, b^* \in \text{pred}(b)$, $b^{**} \in \text{succ}(b^*) \ni \text{ANTIN}_{b^{**}} = \text{true}$,

$\overline{\text{TRANSP}_{b^*}}\,.\,\overline{\text{COMP}_{b^*}} = \text{true also}$.

The first alternatives of (ii) a,b imply that there is no forward or backward flow property which is altered by insertion of block b, while the second alternatives imply that even if certain properties are altered, their alteration does not affect code movement in the program. For example, consider the flow graph of Fig. 8. $\text{AVOUT}_{b^{**}} = \text{true}$ raises the possibility that some evaluation of e_i in some descendant(s) of block b** might have become redundant if block b had not been inserted. However, $\overline{\text{TRANSP}_{b^*}}\,.\,\overline{\text{ANTLOC}_{b^*}} = \text{true}$ implies that e_i is not anticipated at entry of b*, i.e. there is no evaluation of e_i in any descendant of b* which was rendered partially redundant because of $\text{AVOUT}_{b^{**}} = \text{true}$. Thus, no code movement has been affected by inserting b.

Equivalence of the characterization models A,B for the purpose of code optimization can be established as follows: The TEST1, SYNTH and TEST2 blocks of model A are equivalent to the TEST1, CHAIN and TEST2 blocks of model B. If language semantics require control variable initialization of an indexed *do* to be placed in SYNTH block of model A, then the same would have to be placed in the CHAIN sub-block of model B. Model B is obtained from the characterization of Fig. 6(a) by the following transformations:

(i) the edge(TEST1, DO-SUCC) has been replaced by the edge(DO-PRED, DO-SUCC). We can consider this to be movement of the head of this edge across the TEST1 sub-block to the exit of its predecessor block DO-PRED;
(ii) the edges (DO-END,TEST2) and (TEST2, DO-SUCC) have been replaced by the edge(DO-END,DO-SUCC);
(iii) the edge (DO-END,TEST2) has been replaced by the edge(DO-END,DO-ENTRY)

That these transformations do not alter the effective data flow properties in a manner which affect code movement in the program can be seen from the following lemmas.

Lemma 3.1. In a program flow graph, an edge (b, b′) can be replaced by the edges $\{(b^*, b')\}\ \forall\, b^* \in \text{pred}\,(b)$ without affecting code movement optimization provided block b is *inert*.

Lemma 3.2. In a program flow graph, an edge (b*, b′) can be replaced by an edge (b*, b) without affecting code movement optimization provided (i) $b \in \text{ancestor}\ (b')$, (ii) b′ lies along all paths starting on block b, and (iii) all blocks on the path from b to b′ (including b but excluding b′) are *inert*.

Proofs of lemmas 3.1, 3.2 are obvious from the definition of an inert block.

Transformations (i), (ii) to obtain model B fall under purview of lemma 3.1. Transformation (iii) is based on lemma 3.2. Equivalence of models A and B is thus proved. Hence the code movement resulting from the use of model B would be the same as that resulting from the use of model A. However, there is one difference in the optimization steps. Using model A, invariant code moved from the loop would be placed in SYNTH by considering the property $\text{INSERT}_{\text{SYNTH}}$ of the Morel–Renvoise algorithm (equations 2.3). Using model B, only the code which can be placed into DO-PRED or any of its predecessors would move out of the loop. Loop invariant code not satisfying the zero-iteration safety constraint would not move out of the loop since SYNTH has no independent existence. Such code needs to be identified separately after application of the Morel–Renvoise algorithm. This code in its intermediate form of triples/quadruples would have to be chained to the CHAIN sub-block of the DO block, so that the code generation pass can properly explode the DO block into the blocks TEST1, SYNTH and TEST2. This code will be the set of those expressions which satisfy the boolean conditional

$$\text{AVOUT}_{\text{DO-END}} \cdot \text{ANTIN}_{\text{DO}} \cdot \overline{\text{PPIN}}_{\text{DO}} = \text{true} \tag{3.1}$$

This requires only two boolean operations over bit vectors.

Theorem 3.1. It is feasible, safe and sufficient to introduce expressions identified by $\text{AVOUT}_{\text{DO-END}} \cdot \text{ANTIN}_{\text{DO}} \cdot \overline{\text{PPIN}}_{\text{DO}} = \text{true}$ into the SYNTH block of a DO loop.

Proof. (i) Let $\text{ANTIN}_{\text{DO}} = \text{false}$ for an expression e_i. Then the primary condition for hoisting e_i out of the loop is violated. Any evaluations of e_i existing in the loop are non invariant and hence cannot be moved out.

(ii) let $\text{PPIN}_{\text{DO}} = \text{true}$, then either $\text{PPOUT}_b = \text{true}$ or $\text{AVOUT}_b = \text{true}\ \forall\, b \in \text{pred(DO)}$ (equation 2.2). $\forall\, b \in \text{pred(DO)} \ni \text{PPOUT}_b = \text{true}$, e_i would be hoisted into block b or into a $b^* \in \text{ancestor (b)}$. If $\nexists b \in \text{pred(DO)} \ni \text{PPOUT}_b = \text{true}$, then $\text{AVOUT}_{b'} = \text{true}\ \ \forall\, b' \in \text{pred(DO)}$. e_i is then simply redundant in the loop and can be eliminated. In either case, e_i need not be inserted into SYNTH.

(iii) $\text{AVOUT}_{\text{DO-END}} = \text{true}$ implies $\text{PAVIN}_{\text{DO}} = \text{true}$, which in turn implies $\text{CONST}_{\text{DO}} = \text{true}$ (equation 2.4). $\text{PPIN}_{\text{DO}} = \text{false}$ implies $\exists\, b \in \text{pred(DO)} \ni \text{PPOUT}_b = \text{false}$ (equation 2.2). This must be due to $\text{PPIN}_{\text{DO-SUCC}} = \text{false}$, i.e. it must be due to the safety of loop invariant movement vis-a-vis the zero-iteration path. Hence e_i cannot be placed in $b \in \text{pred(DO)}$. At the same time $\text{AVOUT}_{\text{DO-END}} = \text{true}$ implies that it can be hoisted out of the loop. Hence e_i can be placed in block SYNTH.

4. CONCLUDING REMARKS

Model B characterization for the *while* ... *do* constructs requires introduction of a single DO block per *do* construct. In this respect the optimization-time overheads are the same as for the representation of *repeat* ... *until* constructs (Fig. 3), except for the application of equation 3.1 after the data flow properties settle to their final values. Compared to model A, two less program blocks are required to represent a loop, which can lead to savings of the order of 20% as seen in section 3.

Model B is also particularly convenient from another practical viewpoint—namely, the profitability of optimizing a program. A major hindrance to high gains of optimizing *while* ... *do* loops is the safety constraint imposed by the presence of the zero-iteration path. This restricts the movement of loop invariant code only to the SYNTH blocks of model A. In case of nested loops, code would move out of at most one loop even if it is invariant of the entire nested loop structure. Considering that traversal of zero-iteration paths occurs rarely in practice, and in any case a programmer can be expected to have sufficient knowledge of the possibility of such traversal, it is justifiable to relax safety constraints vis-a-vis the zero-iteration path under an explicit user option. Other safety constraints would remain in force thus providing a good compromise between higher gains of optimization and complete safety. Even though provision of such an option is likely to be a controversial issue, its practical gains cannot be ignored easily. Model B facilitates the implementation of this option very easily. A loop would be characterized as in model B when complete safety is required. When zero-iteration safety constraint is to be suppressed, the edge connecting the DO-PRED and DO-SUCC blocks could be ignored while solving the data flow equations.

SUMMARY

Recent research in code optimization has led to the development of unified optimizing transformations like the generalized code movement transformation of Dhamdhere–Isaac[6] and Morel–Renvoise[10], and the composite hoisting-and-strength reduction transformation of Dhamdhere–Isaac[8] and Joshi–Dhamdhere[9]. These transformations involve bi-directional data flow problems over the program flow graph. Solution of these data flow problems require use of iterative solution methods. Optimizers using these transformations incur high optimization costs due to the high time complexity of these iterative solution methods.

Appropriate characterization of program loops is very critical in this context so as (i) not to hinder code movement, and (ii) to restrict optimization overheads to low levels. For example, it is found that conventional characterization of loops can block movement of loop invariant code out of *While . . . do* loops due to the constraints of safety of code movement optimization. Simple expedients of overcoming this drawback require introduction of new blocks in the program flow graph. It is found that optimization-time overheads of this approach can be as high as 20%.

This paper proposes an alternative characterization of program loops which satisfies both the criteria mentioned above. Correctness of code optimization using this characterization is formally proved. This characterization guarantees safety of code movement and incurs no extra optimization time overheads. It also has the nice property that it enables profitability of optimization to be enhanced by suppressing the safety of optimization vis-a-vis the zero-iteration path of a *while . . . do* loop under a user option. This characterization has been successfully used in an optimizing extended Fortran-77 compiler.

REFERENCES

1. Aho A. V. and Ullman J. D., *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts (1977).
2. Cocke J. and Kennedy K., An algorithm for reduction of operator strength. *Commun. Ass. Comput. Mach.* **11**, 850–856 (1977).
3. Dhamdhere D. M., *Compiler Construction—Principles and Practice*. Macmillan, India (1983).
4. Schaefer M., *A Mathematical Theory of Global Program Optimization*. Prentice-Hall, Englewood Cliffs, New Jersey (1973).
5. Wulf W., *The Design of an Optimizing Compiler*. Elsevier, New York (1975).
6. Dhamdhere D. M. and Isaac J. R., Profitability of code movement optimization. Computer Centre Report, I.I.T. Bombay (1978).
7. Morel E. and Renvoise C., Global optimization by suppression of partial redundancies. *Commun. Ass. Comput. Mach.* **22**, 96–103 (1979).
8. Dhamdhere D. M. and Isaac J. R., A composite algorithm for strength reduction and code movement optimization. *Int. J. Comput. Information. Sci.* **9**, 243–273 (1980).
9. Joshi S. M. and Dhamdhere D. M., A composite hoisting-strength reduction transformation for global program optimization—Parts I & II. *Int. J. Comput. Math.* **11**, 21–41 and 111–126 (1983).
10. Kennedy K., Safety of code movement. *Int. J. Comput. Math.* **3**, 112–130 (1972).
11. Joshi S. M., Complexity of some unusual data flow problems. Computer Science Report, T.I.F.R. Bombay (1981).

About the Author—DHANANJAY MADHAV DHAMDHERE received his B. Tech. and M. Tech. degrees in Electrical Engineering from Indian Institute of Technology, Bombay in 1970 and 1972, and Ph.D. in Computer Science also from Indian Institute of Technology, Bombay in 1979.

Dr Dhamdhere joined the staff of Indian Institute of Technology, Bombay in 1972 as a Research Associate. In 1974 he became an Assistant Professor of Computer Science, a post which he continues to hold today. He held the additional charge of System Software group during 1974–81 for the Institute's EC-1030 computer system. His teaching and research interests are in the area of programming languages, compilers and operating systems. He is the author of many research papers and two books entitled *Compiler Construction* and *Introduction to System Software*.

About the Author—JATINDER SINGH KEITH received his M. Tech. degree in Computer Science from the Indian Institute of Technology, Bombay in 1981. His interests are in the area of programming languages.