SEL 79-026

SYMBOLIC DEBUGGING OF OPTIMIZED CODE

by

John L. Hennessy

Technical Report 175

July 1979

This research was supported by the
National Science Foundation under Grant number ENG78-578 l.

SEL 79-026

SYMBOLIC DEBUGGING OF OPTIMIZED CODE*

by

John L. Hennessy

Technical Report 175

July 1979

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California

* This research was supported by the National Science Foundation under Grant number ENG78--5781.

SYMBOLIC DEBUGGING

OF

OPTIMIZED CODE *

John Hennessy
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

July 1979

ABSTRACT

The long standing conflict between the optimization of code and the ability to symbolically debug the code is examined. The effects of local and global optimizations on the variables of a program are categorized and models for representing the effect of optimizations are given. These models are used by algorithms which determine the subset of variables whose values do not correspond to those in the original program Algorithms for restoring these variables to their correct values are also developed. Empirical results from the application of these algorithms to local optimization are presented.

Key Words and Phrases: symbolic debugging, code optimization, directed acyclic graphs, flow graphs

CR Categories: 4.12, 4.42, 4.22

* This research was supported by the National Science Foundation under Grant number ENG78--5781.

1. INTRODUCTION

There is a classic conflict between the application of optimization techniques and the ability to employ symbolic debuggers. While program optimizations must ensure that the optimized code is functionally equivalent to the unoptimized code, an optimization normally alters the intermediate results. Thus if an optimized program aborts or is stopped by a debugger, the resulting states of the variables, may not reflect the values in the unoptimized code. We will refer to these variables as incorrect variables. The user is then faced with the difficult task of attempting to unravel the optimized code and determine what values the variables should have. This is especially difficult if the program contains errors, so that some variable values may actually be wrong. The importance of the ability to symbolically debug code has been discussed extensively[5,6,7,8,9].

Simple solutions to this conflict, for example omitting the optimization of undebugged code, are not acceptable for several reasons. First, many compilers apply optimization techniques as a normal part of the compilation process. This is especially true for local optimization, where redundant stores are eliminated or code reordering is done. Secondly, the ability to symbolically and reliably debug code is an important asset which should not be relinquished. In large optimized programs, e.g., an operating system, the error state may not be duplicative. Lastly, applying a symbolic debugger to optimized code is an important method for discovering errors in the optimizer.

In this paper the problem is considered for both standard local optimizations and a set of important global optimizations. The emphasis of this approach is:

1. To detect incorrect variables in an optimized program using information available to the compiler as it compiles and optimizes the program
2. To recover the correct values of these variables.
3. To add no extra overhead to the executing program whenever possible.
4. To consider only approaches whose payoff is reasonable and cost relatively low.

First, the effects of local and global optimization are demonstrated. Then the problem of local optimization is examined: a model to represent the code optimization process, incorrect variable detection, and variable value recovery are discussed. Results of an empirical study of these algorithms are given. The same approach is utilized for golbal optimi-ation. The conclusion discusses the possiblities for incorrect variable detection and value recovery in standard compilers.

2. THE EFFECTS OF OPTIMIZATION ON SYMBOLIC DEBUGGING

Both global and local optimization affect the ability to do symbolic debugging since they eliminate and reorder stores to variables. This effect occurs for different reasons. Local optimization may eliminate stores to variables which are stored to later in the same basic block. Altering of the order of code execution in a basic block may also be done in the local optimization process. The effects of these two types of local optimization are shown in the following code segment:

Initial Code	Optimized and Reordered Code
1. A := B + C	1'. G, B := D + E
2. B := D + E	2'. C := F + H
3. c := F + H	3'. A := 2 * B
4. G := D + E	
5. A := 2 * B	

Error Occurs at	Reported at	Variables with the wrong value
1'	2	A
2'	3	A, G
3'	5	A

Global optimization can also affect the ability to do symbolic debugging. A number of common global optimizations do not affect symbolic debugging; these include constant folding, copy propagation, global common subexpression elimination, code hoisting (provided only expressions are hoisted), and loop unrolling. These optimizations do not alter stores to variables in the program either by changing their position or eliminating them Therefore, the values of variables in a program using only these optimizations will not differ from the values in the optimized program on a statement by statement basis.

Three important global optimizations impact the ability to do correct symbolic debugging: code motion, induction variable elimination, and dead variable elimination. All of these global optimizations cause difficulties for the debugger becuase they either remove or reorder assignements.

Code motion causes difficulty because operations Within loops are moved. If these operations include assignments to variables, the resulting program will not have variable values corresponding to the original program The following example illustrates this problem

<u>unoptimized code</u>	<u>optimized code</u>
<u>repeat</u>	i := j;
S_1;	<u>repeat</u>
i := j;	S_1;
S_2	S_2
<u>until</u> p;	<u>until</u> p

where j is a loop constant.

Now there are three possible stopping points in the optimized loop:

Execution Stops before	Reoorted at	Is I correct?
i := j	i := j inside loop	Yes
S1	S1	No, if first time through loop
S2	S2	Yes

A second cause of difficulty may be induction variable elimination. Since this optimization may completely eliminate assignments to variables, the variable may obviously be incorrect. The following example shows this:

```
unoptimized code                  optimized code

for i := 1 to n do begin          for Tj := 2 to 2*n do begin
    S1;                               S1;
    j := i * 2                        S2 - replace all references
                                           to j by Tj
    S2                            end
end     (i is not used in the loop body, j not used in S1, Tj
         is a new temporary)
```

At all possible break points in the loop body i and j are incorrect. If i and j are live after the loop they will need to be fixed up at the loop end by the optimized code.

Dead variable elimination can also cause problems, because a store can be omitted whenever there are no uses of that value. This occurs if a variable is stored into, before the last value stored to the variable is used, or if the variable is never referenced again. In the first case, if execution is stopped between the two stores, the value will be incorrect. In the second case if execution is stopped anywhere after the dead store the value is incorrect. For example, if i and j are not live after the loop in the previous example, their values will be incorrect in the optimized code.

2. A MODEL OF LOCAL OPTIMIZATION AND CODE GENERATION

In order to describe algorithms which assist a debugger in locating incorrect variable values it is necessary to have a model for code generation. Labelled DAGs, a variant of the DAG structure defined by [1,2,4], can be used to represent the code optimization process, while retaining information necessary to determine the form of the original code.

An augmented computation DAG is a Directed Acrylic Graph with the following labels on nodes:

1. Leaves are labelled by unique identifiers, either variables or constants.
2. Interior nodes are labelled by an operator symbol.
3. Nodes may optionally have an extra set of labels which are variable names. There are two type of variable name labels:
   a. Current type label-indicates that the node represents the current value of the labelling variable. A third field of the label indicates another DAG node, whose purpose is to indicate the original code ordering.
   b. Old type label-indicates that the value of this node represents the value of the label variable at an intermediate point in the computation. The third field is also present. These labels represent old values of a variable, they are removed during the DAG construction algorithms [2].
4. All nodes contain a code start label which indicates the start of the machine instruction sequence for this node.

The purpose of an augmented computation DAG, hereafter called DAG, is to represent both the locally optimized reordered code and the original unoptimized code. This is done by constructing the DAG, so that a post order traversal will result in an execution pattern which matches the original code sequence. This is accomplished by the semantics associated with the DAG labels. The node pointer field of a current or old type label indicates the original position of the variable store. Thus in traversing the DAG, a store to a variable in a label is delayed (the node value must be retained) until the node indicated by the second field is visited. The node field pointer may not point to a node earlier in the DAG, that is a node which is built first.

Also the DAG must indicate the order of statements within the original basic block. Each statement has an unique root, and the roots are implicity ordered by the order of the statements. Each rooted DAG is then traversed in post order, ordering the DAGs by the implicit statement order. We will assume the nodes in the DAGs are ordered, $d < d'$, if d is visited first in a postorder traversal of the DAGs.

An algorithm for constructing these DAGs is not given here, simple modifications to the algorithms found in [2], suffice. It is useful to examine an example of these DAGs.

Unoptimized Code

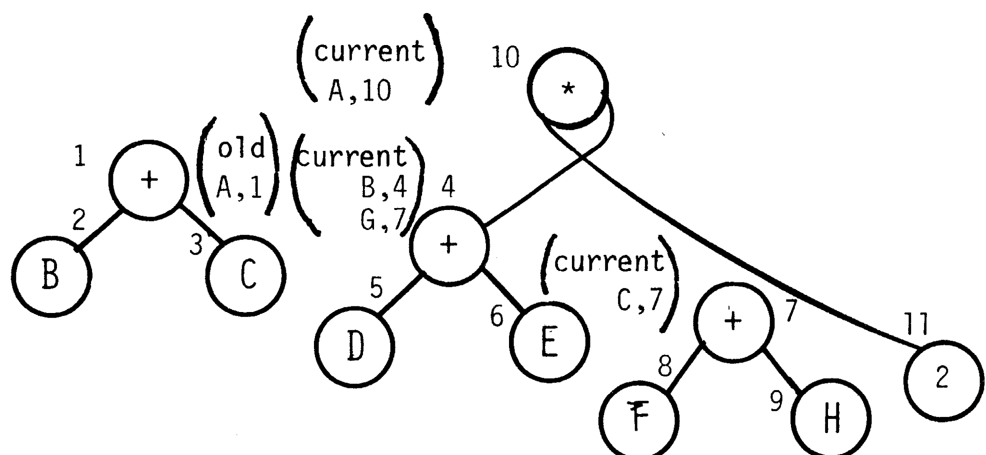
A := B + C

B := D + E

C := F + H

G := D + E

A := 2 * B



DAG REPRESENTATION

The nodes have been numbered for ease in interpreting the DAG, The order of the roots is: 1,4,7,10. Note that the pointer field for the G label on node 4, is node 7, since the store to G, should not occur until after node 7. Code start labels should be associated with each node.

In general, code may be generated for the nodes of the DAG in any order. For simplicity two restrictions are imposed. First, the set of instructions generated for a node in the DAG must be consecutive. Secondly, all stores of a value corresponding to a node d, must occur at the end of the instructions for d. Thus, if an error occurs at d, it must occur before any stores of the value computed at d have occured. The purpose of this is to ensure that the code for a node in the DAG either stores to all current labels on the DAG or has stored to none of them These restrictions can be removed at the expense of enlarging the size of the DAG, and keeping one label per node. No restrictions on reordering are imposed so code for nodes may be generated in any order.

Before discussing the algorithms for finding variables with incorrect values some discussion of the symbolic debugger's action is needed. The symbolic debugger can not stop at a point where some of the variables in current type labels on a node have been stored to. This is enforced by the fact that all stores occur at the end of the node, contiguously.

Another issue is where the symbolic debugger reports that it has stopped, or in the case of a user set breakpoint at what locations such breakpoints are allowed. Suppose the debugger is invoked by an error in the source program, which corresponds to a node d in the DAG. Since d may be contained in several subDAGs, the debugger must choose where to report the error.

The location must be in the first statement corresponding to a subtree containing d. If the error is not reported in the first statement containing d, that node must have been evaluated without error in the original source program (clearly not possible).

A second question is where to allow breakpoints to be inserted. This problem arises since the user can insert a breakpoint at a statement which corresponds to a node d, contained in multiple subDAGs, i.e., common to several statements. In the optimized program, execution would then stop at the node d, reporting that execution halted at the statement corresponding to the first instance of d. To prevent this possibility breakpoints can only be inserted at statements which either have a unique starting node, i.e., the first node of the statement, is not contained in any other subDAG; or at a statement, S, where the starting node of S, is first reached in the subDAG for S.

3. DETECTION OF INCORRECT VARIABLES FOR LOCAL OPTIMIZATION

Given that a program is halted at machine instruction d, what subset of the variables contain incorrect values? To address this question it is necessary to classify the ways in which a variable could be incorrect. There are two distinct situations':

1. A variable could have been assigned a value prematurely, due to either common subexpression elimination or code reordering. In our earlier example variable G, would have this property. Call this set of variables the Roll Back Variables (RBV).
2. A variable could have not yet been assigned a value, due to either an eliminated store to that variable or reordering of the code. Call this set of variables the Roll Forward Variables (RFV). Variable A has this property in the example.

These sets, RBV and RFV, are computed by following algorithm

Algorithm Detection of incorrect variable values.

Input : DAG, D, and an node, d, which is the error or breakpoint location.

output : The sets RFV and RBV.

Algorithm

```
    RFV  := φ
    RBV  := φ
    (* First pass- find all identifiers which should have been
       stored to *)
    for all nodes n < d do
         FIXED := {v | (v is a current label on n) and (actual node
                        pointer (v) < d)}
    for all nodes n ε D do begin
```

```
            if (n > d) and (code start (n) > code start (d)) then
            (* n should have been executed but was not *)
            RFV := RFV ∪ {v | v is a current variable label on n}
        (* Compute RFV set arising from eliminated stores *)
        if n < d then RFV := RFV U ( { v | v is an old type variable label
                                      on n } - FIXED);
        (* Compute RBV set *)
        if code start (d) < code start (n) then
            if n < d then RBF := RBV U { v | v is a current type variable
                                        label on n) and (actual node pointer for
                                        v ≥ d)}
            else if n > d then RBV := RBV U { v | v is a current variable
                                          label on n }
end
```

This algorithm finds the set of all variables whose values are not correct at the node, d, in the DAG, according to the original program There are only two ways in which a variable can have the wrong value: it is stored into prematurely (RBV set), or it is not stored into when the oriqinal proqram stored into it (RFV set). Consider the RBV set, $v \in RBV$ set if the following conditions hold:

1. v is stored into in node n
2. n should not have been executed, i.e. $n > d$; or the store should not have been executed, i.e. actual node pointer for $v > d$.
3. n was executed, i.e. code range $(n) <$ code range (d).

Clearly, the algorithm computes RBV exactly.

Now consider RFV; a variable $v \in RFV$, if it was not stored into because:

1. The store was eliminated, and it was not stored into (i.e., not in FIXED), before d.

or

2. The store should have occured before d (i.e., $n < d$) and did not (i.e., code range (n) $>$ coderange (d)).

Hence RFV is computed correctly.

The running time of this algorithm depends on the execution time on the loop; if the number of nodes in D is $|D|$, then the running time is $O(|D|\alpha)$ where a is the execution time of one iteration.

If we utilize bit vectors with constant operations to represent the label sets we can improve the running speed of each iteration. By just using bit vectors, each step of the loop body is constant, except for:

RBV := RBV $\cup$ {v | (v is an old type label on n) and (actual node pointer for v $\geq$ d)}

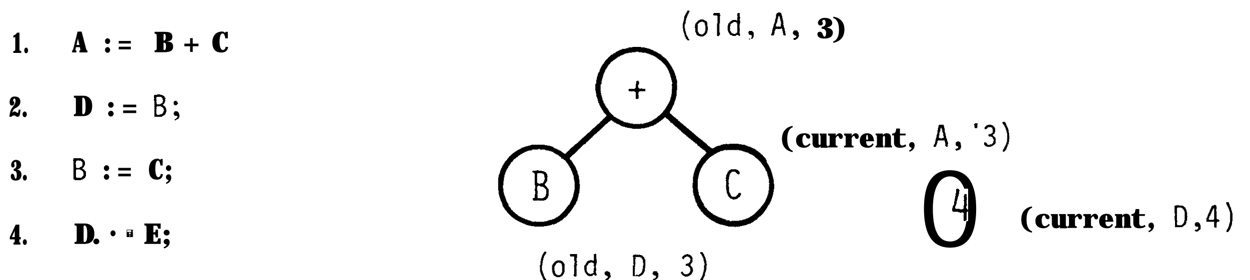
Thus running time of the algorithm is 0 (number of nodes * number of eliminated stores). In practice the number of variables referenced in a basic block is small and the number of eliminated stores much smaller, hence bit vectors are a reasonable representation and the algorithm runs in close to 0 (number of nodes) for almost all real examples.

4. RECOVERY OF INCORRECT VARIABLES FOR LOCAL OPTIMIZATION

Given an augmented computation DAG and a stopping point in the DAG, a symbolic debugger could report which variable values were reliabily known; the debugger could even print the optimized code. The algorithms in this section attempt a preferred solution: to find the variable values in the unoptimized code and report those values. Using this method, the effect of the optimization on program semantics is eliminated.

An important observation is that it may not always be possible to recover a variable value, especially if the value must be rolled back. In the earlier example, the value of G would not be recoverable if execution stopped at 2'.

In attempting to recover the incorrect variables two assumptions are made. First, the semantics of operations on nodes are not utilized, except to apply the operation at a node. For example, consider the following code segment and its augmented DAG.



Suppose the symbolic debugger is invoked with execution halted at 4, the last value of D is B, but the value of B was destroyed by the execution of statement 3 (B := C). If we use information about the operation +, at node 3, we see that B = A - C, since the values of A and C are both

available we could restore B. For simplicity the above approach is not considered by ignoring semantics of operations.

In the following algorithms, we concentrate on determining whether a variable is restorable. The restoration process is derived in the algorithm and variable recovery could easily be accommodated. The algorithms do not examine the current availability of partial results, e.g., the value of a node may be represented by a register or a temporary. If the current values of temporaries and registers are to be utilized, a Simple map between nodes and these values is required. Marking the nodes as available in the algorithms which follow will then have the correct result.

The problem of recovering incorrect variable values can be subdivided into correcting the subsets RBV and RFV computed in the last section. To recover a variable $v \in RBV$, two conditions must hold: v must be an old type label on some node n < stopping-node point d, and the value computed at n must be available or computable. The variables in RFV can be recovered by one of two strategies, which correspond to why the variable is in RFV.

<u>Algorithm</u> Finding the subset of recoverable variables from a set of variables whose values are in error.

Input: A DAG, D, with a stopping point d, and the two sets RFV and RBV.

output: The set $Recoverable \subseteq (RFV \cup RBV)$, whose members are variables with recoverable values.

The algorithm uses the function Available.

Available (n, d) - finds if the value at node n is available if execution stops at node d.

Available (n, d) = if $\exists v$(v is a current label on n <u>and</u> code range (n) < code range (d)) <u>then</u> true

<u>else</u> <u>if</u> n is a parent node <u>then</u>

```
          Available (right child, d) and
          Available (left child,d)
     else (* leaf node *)
          let v be the variable name of the leaf;
          ∃p(p contains v as the current label and code range (p) < code
           range (d))
```

Alaorithm for Recovery Detection

```
Recoverable := φ ;
(* First we recover RBV variables *)
for all v ε RBV do begin
     n.●  =max(mJm contains an old label for v } ;
     if ( n <> null) and
          Available (n, d) then Recoverable := Recoverable U {v};
end;
(* Recover RFV set *)
for all v ε RFV do begin
     if ∃n(v is current label on n and n < d) then
          Recoverable := Recoverable U v
     else begin
          n := max {m | m contains an old label for v} ;
          if n < >null and Available (n, d)
               then Recoverable := Recoverable U { v } .
     end
```

This algorithm computes the set Recoverable, s.t.

Recoverable $\subseteq$ (RBV U RFV) and

v ε Recoverable $\rightarrow$ value of v, at point d in the original code can be computed.

v ε ((RFV U RBV) - Recoverable) $\rightarrow$ value of v at point d is not recoverable.

In order to show that this algorithm correctly computes Recoverable, it must be shown that all variables in RFV and RBV are correctly recovered, and that the sets RBV and RFV are independant.

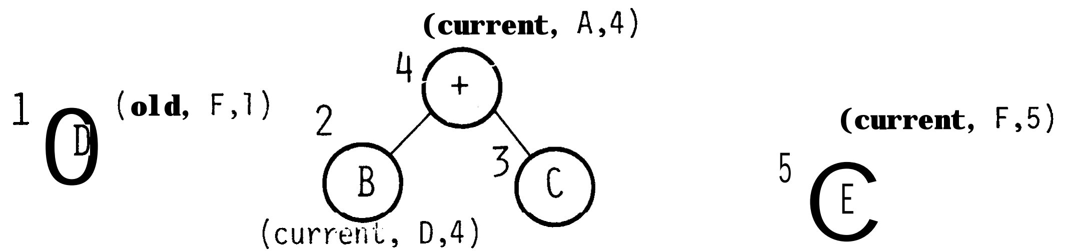
To recover v ε RBV, it is necessary to have an earlier value for v, since the store which must be rolled back will have destroyed the value of v. Hence to recover v, an old label for v, on a node n $<$ d (the stopping point), must be found. The value corresponding to n must be available if v is recoverable. To recover v ε RFV, we must consider how v could be in RFV. There are two cases, reordered code with a store not yet executed, and an eliminated store. The first case occurs only if a store exists on a node n $<$ d, and code range (n) $>$ code range (d); these cases are recovered by executing all nodes less than d. In the second case we must find the value v should have at d; this value is the value at the node n, where n contains an old label for v, and no node greater than n and less than d, has such a label. Therefore, v is recoverable if the value of the node is available.

To find if the value of a node is available, there are several cases. First, if the node has any current type labels, then the value is recoverable, since the value is stored in the variable corresponding to the label. If the node represents a parent node without current labels, then the value is recoverable if both children are recoverable. Lastly, the node may be a leaf.

Leaf values are available if the value they represent has not been stored into. This leads to a potential dependence of the sets RBV and RFV. Consider the following code sequences:

Unoptimized Code	Reordered Optimized Code
1. F:=D	1. D:=B
2. A := B + C	2. A := B + C
3. D := B	3. F := E
4. F := E	

Now suppose instruction 2 of the optimized code fails. The augmented computation DAG Eooks like:



with code range (2) < code range (4) < code range (5), 1 < 4, and 4 < 5. Then RFV = {F} and RBV = {D}. F is recoverable if the value of node 1 is recoverable, but node 1 is recoverable if D is recoverable. In this case our algorithm reports D is not recoverable; it is not necessary to recover either set first.

This DAG demonstrates that a variable, v, in RFV can be recoverable only if $v' \in RBV$ is recoverable. But we claim these two sets are still independent, since this situation can only arise when v' is a leaf value. If v' is a leaf value and the variable v' has already been stored into, then the value of node is never recoverable. Therefore, the sets retain the independence and the Recovery Algorithm obtains the correct result.

This algorithm for variable recovery is reasonably efficient. The algorithm for computing the function Available runs in time less than

$O(|D|)$ where $|D|$ is the number of nodes in the DAG D. Recovering each variable $v \in RBV$ is a two step process; find the last old store to v, and then determine if that node is available. Both of these operations are bounded by $O(|D|)$. Likewise recovering each member of the RFV set has runtime complexity bounded by $O(|D|)$. Thus the running time of the recovery algorithm is bounded by : $O((|RFV| + |RBV|) * |D|)$.

What happens when the debugger encounters an error when trying to restore some variables? Although this problem sounds complex, the solution is quite simple. The symbolic debugger can only encounter an error at node n, if $n < d$. Therefore, if the code was not optimized the error would have occurred at n. The debugger now reports that the error has occurred at n, rather than d, and attempts to recover the variable values in effect at node n. The major effect of this type of error is that the number of unrestorable variables may grow, since more stores between n and d, may have been executed. This process of moving back the error point must halt since only a finite number of statements are contained in the basic block, and the error point is only moved in one direction.

6. EMPIRICAL RESULTS FOR LOCAL OPTIMIZATION

A trial implementation of these algorithms for a subset of Pascal has been done in Pascal. A compiler for the Pascal subset was constructed, the compiler produces a stack based intermediate form, very similar to P-Code. The subset currently does not include pointers or sets.

The implementation of these algorithms consisted of two seperate steps. First, a program which builds DAGS from the stack based intermediate form is used. The DAG construction algorithm of [2] is employed. Code generation is then done using the hueristic algorithms for DAGs [2]. The code generation algorithm allows an arbitrary choice of nodes at one point. This choice was randomized to provide the maximum variety of code orderings.

The result of the DAG building and code generation is an augmented computation DAG which includes the extended label fields and the code start value for each node. The algorithms for erroneous variable detection and recovery then utilize the DAG as input together with some stopping point in the code.

The most significant results of the study for a group of programs consisting of over 5000 lines are shown below (BB stands for a basic block):

Average size of a BB in P-Code instructions	9.0
Average number of variable references/BB	1.8
Average number of variables assigned/BB	.5
Average percentage of BB with common subexpressions	67%
Average percentage of BB with nontrivial common subexpressions (i.e. more than one term)	30%
Average number of terms/common subexpression	1.8

Average number of common subexpression/BB	1.5
Average number of common subexpression/ BB with at least one common subexpression	2.4
Average precentage of BB with common subexpressions that have incorrect variables	20%
Average percentage of incorrect variables that are recoverable	58%

There are several important observations to be made based on this data. First, the frequency of variable references, and common sub-expressions agrees with data collected by others. The high number (2.4) of common subexpression within blocks with at least one common subexpression, increases the potential gain for local optimization. This is especially true if such blocks are within inner loops.

The percentage of blocks with common subexpressions that have incorrect variables is relatively high (20%); this indicates a real possibility that a symbolic debugger may encounter a variable with an incorrect value. Employing recovery strategies is fairly sucessful, allowing recovery of about 58% of the incorrect variables. This would seem to indicate that recovery attempts are worthwhile.

7. A GRAPH MODEL FOR GLOBAL OPTIMIZATION

In order to analyze programs which have been globally optimized a representation for the unoptimized and optimized program is needed. The model will consist of two parts: a global flow graph model, where each basic block is a node, and augmented DAGs for each node in the flow graph. The model will also distinguish certain flow graph nodes and utilize DAG node pointers between different flow graph nodes. We consider global optimzations as defined in [3].

Code motion is defined as the movement of loop invariant code from inside the loop to a special node, called the preheader, which precedes the loop. For simplicity only code motion to the front of the loop is considered. The standard requirements on the moved code are imposed. Code motion is represented by a preheader node which contains an augmented DAG. The DAG differs from a local optimization DAG, in that the actual node pointer on a label, points to a DAG node within the loop body. This node pointer indicates the original placement of an assignment.

Induction variable elimination causes a similar difficulty, i.e. assignment to variables have been deleted and moved in position. Representing induction variable elimination requires knowledge of where the original assignments occurred in the loop body, and the functions relating the induction variables. For simplicity, we assume that all induction variables, which are functions of a single basic induction variable, are replaced by a single new induction variable. This new induction variable is a loop temporary not appearing in the program This condition can be removed at the cost of slightly increasing the complexity of the model for induction variable elimination.

I is an induction variable if and only if:

1. I is a basic induction variable, i.e., I is assigned once in the loop, with an assignment of the form $I := I \pm C$, where C is a loop constant.

or

2. $I \in$ Family (J), where J is a basic induction variable and there exist m and n loop constants, s.t. I is assigned once in the loop and the value assigned to I is $m*J \pm n$.

Consider each Family, F of induction variables: for each family there must be a temporary variable in the loop, call this T_F. Since $T_F = m_F*J + n_F$, for each $i \in F$, we can find m'_i, n'_i s.t. the value assigned to i in the loop is equal to $m'_i*T_F + n'_i$. These m'_i and n'_i are given by:

$$m'_i = \frac{m_i}{m_F} \qquad\qquad n'_i = n_i - \frac{m_i n_F}{m_F}$$

where m_i and n_i are the original constants, s.t. $i \in$ Family (J) $\rightarrow$ i is assigned the value $m_i J + n_i$ in the loop.

In addition to knowing the loop constants m'_i and n'_i, and the new base induction variable T_F, the original location of the assignment for each induction variable is needed. All this information can be stored in the loop preheader node.

Dead variable elimination can be represented with the augmented DAGs, by marking the stores to the dead variable as old.

The proposed model represents both the original and the optimized code. To reconstruct the effect of the original code the following rules are followed:

1. For a preheader node

   a. Evaluate the DAG but postpone all assignments according to the node pointers associated with the assignment label. These assignments need only be executed on the first execution of the loop body.

   b. For each induction variable, i, in the preheader do the following: on each loop execution compute $m'_i * TF + n'_i$, store this value into i, when the location of the original assignment is reached.

2. All old label stores are also treated as new stores and executed when their node pointer points to the node just executed. This recovers dead variable stores.

A small example is given below:

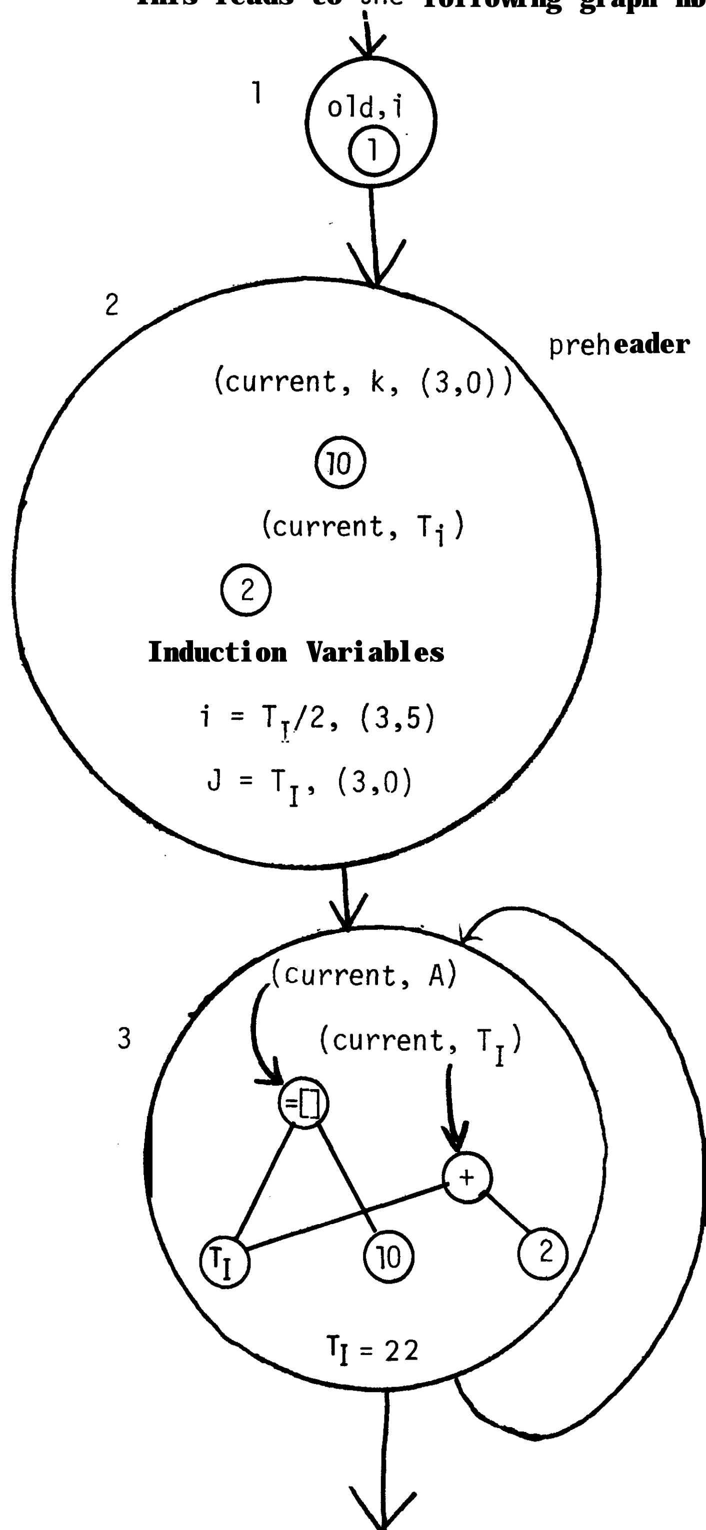
```
i := 1;
repeat
    j := i * 2
    k := 10;
    A[j] := k;
    i: = i + 1
until i = 11
```

After global optimization:

```
T_I := 2;
k : = 10;
repeat
    A[T_I] : 10;
    T_I := T_I + 2
until T_I = 22
```

(* Assuming i and j are not live after the loop *)

This leads to the following graph model:



Note: A label pointer is represented by (m,n) where m is a flowgraph node and n is a DAG node. The label pointer (m,zero) indicates a store immediately upon entrance to the node. Self referencing label pointers are omitted.

Using the method given for restoring the original code the result is:

```
         i := 1;
         TI := 2;
loop:    j := TI;
         k := 10:
         A [TI] := 10;
         TI := TI + 2;
         I := TI/2; (* integer divide *)
         if TI ≠ 22 go to loop
```

This code will correctly preserve variable values, independent of where code is stopped.

8. DETECTION OF INCORRECT VARIABLES FOR GLOBAL OPTIMIZATION

This section gives methods for detecting which variables have incorrect values if execution is stopped at some point in globally optimized code. This issue is addressed separately from local optimization for simplicity purposes. Several methods for detecting these incorrect values are presented; the next section discusses recovery issues.

Consider an assignment to a variable, v, which is moved to the preheader by the optimization of code motion. Such an assignment could only have been moved if the right hand side was loop invariant. Therefore, the variable v will have the correct value if the execution reaches the original location of the assignment at least once. Therefore v is incorrect iff: the loop body is not completely executed the first time, and the stopping point in the loop body is before the actual node pointer on the assignment in the preheader. The second condition holds if the stopping point dominates the node indicated by the actual node pointer. This is easily checked.

There are two methods for determining that the loop body was not executed once. The simplest method is to add a flag for each loop, which is set false in the loop preheader and true in the last statement of the loop body. Then the body is executed at least once if the flag is true when the program is stopped. The major disadvantage of this approach is the extra overhead of one assignment on each loop execution.

Related approaches may be possible for certain types of loops, such as for statements. For example, with the statement <u>for</u> i := m to n do S, the code is stopped in the first execution of S, if i = m If i ≠ m, then

the loop body must have completely executed at least once. This approach has several difficulties, e.g., the body may alter either i (some languages prohibit this) or m Then this test is no longer suitable.

An alternative approach which completely eliminates undetectable errors due to code motion, is to unroll the loop, so that the first execution of the loop body is straightline code. Under these conditions the loop preheader is not needed for code motion, since these variables can be assigned in the unrolled loop body. For example, the statement <u>for</u> i := 1 <u>to</u> n <u>do</u> S, becomes i := 1; S; for i :=2 <u>to</u> n <u>do</u> S. If execution halts in the first copy of S, no variables are incorrect because of global optimization, since no assignments have been moved. If the program halts on a later execution of S, code motion has no effect, since all assignments would have occurred at least once.

There are two disadvantages to this approach. The first is that the size of the code for the loop is increased, possibly substantially. A second minor disadvantage exists if the debugger wishes to halt execution at a point in the loop body, two calls to the debugger must be inserted. Likewise the debugger must know that both copies of S map to the same loop body. The major advantage is that additional algorithms for incorrect variable detection with code motion are not needed.

Detection of incorrect values for induction variables is reasonably straightforward. If I is the set of induction variables in a loop header for L, then $i \in I$ has the wrong value if: the loop has executed more than once, or the stopping point is after the point indicated by the actual node pointer for i. Note that this is, in some sense, the inverse of the condition for a variable assignment moved by code motion. Thus the same techniques

used for detecting loop execution in code motion work to determine if the loop has been executed at least once.

Elimination of stores to dead variables cause an incorrect variable to occur if the eliminated store node was executed and no other store to the variable has been reached. The problem this causes is that a variable v, to which a store was eliminated is incorrect at point p, if there exists a path from that eliminated store to p. But, this does not mean v is necessarily incorrect since there may be another path from the store to p, that assigns to v.

A store to a variable v in flowgraph node n is a dead store if v is an old label on some node in n and v is not a current label on any node in n. A variable v may be endangered at p, if there is a path from the start node which goes through a node n containing a dead store of v, and reaches p, without going through a node containing an actual store to v between n and p. This is clearly a data flow analysis problem A variable v can be in one of three possible states at p, if a dead store of v exists:

1. v is not endangered on any path to p and hence v has the right value at p.
2. v is endangered on all paths to p, then v has the wrong value.
3. v is endangered on a subset of the paths to p, then v is incorrect only if an endangered path was used to reach p most recently.

Thus it can be determined if v is possibly incorrect. To resolve case 3, some mechanism such as a flag is needed to determine which path was used to reach p. This method has the serious disadvantage that is probably more costly than the original dead store elimination. With the requirement of data flow and flags, it appears that incorrect variable determination for dead stores is not profitable.

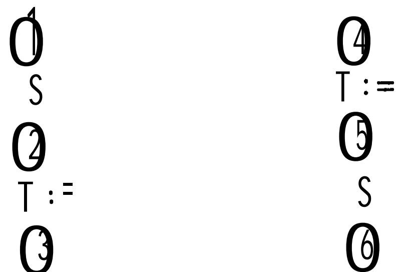
9. RECOVERY OF INCORRECT VARIABLES FOR GLOBAL OPTIMIZATION

The problem of recovering incorrect variables in the presence of global optimization is more difficult than local optimization, since the chances for success are generally less. Code motion and dead store recovery require the same techniques. If v is a variable with the wrong value at p, either because of code motion or elimination of a dead store to v, then the previous value of v must be recovered. In order to determine what the previous value is, data flow information is necessary. In the case of code motion, the set of definitions for v that reach the loop preheader are needed. For dead store elimination the set of definitions for v that reach p are needed. Call these definitions D. If $|D| > 1$ then v can not be recovered since it is impossible to tell which one of the definitions for v was the most recent. If $|D| = 0$ then v was undefined at p and recovery is not necessary. If $|D| = 1$, then some attempt at recovery can be made. Let D be a statement of the form v :=A op B, if there are no definitions for A or B, between D and p, then v still has the value A op B. Otherwise, to recover v, the values of A and B at D must be recovered. This process can continue until v is recovered or the number of definitions of some variable is greater than one, in which case the attempt fails.

This process has three significant disadvantages. First is a very costly process since several data flow problems must be solved. Secondly, it would seem to be very ineffective because of the stringent requirements which are needed to recover a variable value. This is especially true of the requirements that $|D| \leq 1$ and that no definitions for the operands exist between D and p. Another problem is that the operands may represent temporaries rather than variables. In this case, to recover v, the temporary

values must be recovered. The same techniques as recovering v apply. Thus in general it will not be cost effective to attempt to recover incorrect variables due to either code motion or dead store elimination.

A more fruitful recovery problem is that of induction variable elimination. Let V be the set of induction variables replaced by T and let T be updated in the loop by a statement, S, $T := C_1 * T + C_2$. The stopping point p can be in one of six locations 1-6 as shown below:



This creates four distinct classes; the correct value of v is given by the class of p:

p	
1,3,4,6	v is a function of current value of T, $v = m'_v * T + n'_v$.
2	v is a function of the next value of T, $v = m'_v * next(T) + n'_v$.
5	v is a function of the previous value of T, $v = m'_v * previous(T) + n'_v$.

The values of T are given by

$$T_i = C_1 * T_{i-1} + C_2.$$

Therefore, the following algorithm recovers V:

```
for all v ε V do begin
    n := actual node pointer for v (from preheader);
    case position of s, n, p of
  1,3,4,6 : v := m'_v*T + n'_v;
    2 : v := m'_v*(C_1*T + C_2) + n'_v
    5 : v := m'_v*(T/C_1 - C_2) + n'_v
    end
end
```

This algorithm recovers all variables which are incorrect due to induction variable elimination. The approximate running time is constant for each variable. Thus a fairly efficient means for recovery of eliminated induction variables is possible.

10. CONCLUSIONS

This paper has argued the importance of symbolic debugging capability in the presence of optimization. The effects of optimization on this capability are examined. Algorithms for detecting variables whose values do not reflect the unoptimized program are given. The problem of recovering the correct values for these variables is also examined.

Several important results come from this work. First, detection of incorrect variables with local optimization is quite easily done. The necessary information could easily be produced by a compiler and placed with the symbol table which is needed for a symbolic debugger to operate. Variable recovery is fairly successful for local optimization and requires no additional information.

Global optimization causes more difficulty than local optimization due to its dynamic nature as opposed to the static behavior of basic block code. Detection of incorrect variables requires some additional overhead in the case of code motion and induction variable elimination. This overhead may be acceptable, depending on the importance of retaining symbolic debugging capabilities. Recovery of correct variable values is difficult for code motion. Since variables are troublesome in the presence of code motion, on only the first execution, recovery can be reasonably omitted for code motion.

Recovery for induction variable elimination is important, since the variables will be incorrect for the duration of the loop. Without recovery of induction variables the user may not be able to determine the number of executions of a loop, when an error occurs. Fortunately, recovery of induction variables is very successful, since all variables can be recovered. Use of a program variable to replace a family of induction variables is

also possible, but causes some additional complexity.

Elimination of dead stores results in insurmountable difficulties both for detection of incorrect variables and their recovery. If symbolic debuggers are to be applied to code, this optimization should probably not be employed.

If the compiler is designed with options that control optimization and debugging, the following modes seem reasonable:

DEBUGGING OPTION	Optimization: Local (could be default)	Optimization: Local and Global
Normal (default)	Detection and recovery of incorrect variables is supported.	Detection and recovery for local optimization. Detection for code motion. Detection and recovery for induction variables. No global dead store elimination.
OFF		Detection and recovery for local optimization only.

ACKNOWLEDGEMENTS

J. Barth provided the initial motivation to study this problem

S. Graham suggested loop unrolling as a means of detecting the first execution of the loop body.

REFERENCES

1. Aho, A.V., and Ullman, J.D. Optimization of straightline code. *Siam Journal on Computing 1:1*, (1972), 1-19.

2. Aho, A.V., and Ullman, J.D. The *Theory of Parsing, Translation, and Compiling*, Vol II: *Compiling*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

3. Allen, F.E., and Coche, J. A catalogue of optimizing transformations, in Rustin, R. (ed.), *Design and Optimization of Compilers*. Prentice-Hall Englewood Cliffs, New Jersey, 1972.

4. Cockle, J., and Schwartz, J.T. Programming languages and their compilers. Courant Institute, New York University, New York, 1970.

5. Evans, T.G., and Darley, D.L. On-line debugging techniques: A survey. Proc. *AFIPS FJCC* 29, 1966, 37-50.

6. Mark, B.L. Design of a checkout compiler. *IBM Systems* J 3, 1973, 315-327.

7. Satterthwaite, E.H. Debugging tools for high level languages. *Software-Practice and Experience* 2, 1972, 197-217.

8. Satterthwaite, E.H. Source language debugging tools. Ph.D. Th., STAN-CS-75-494, Stanford U., Stanford Calif., 1975.

9. Wolman, B.L. Debugging PL/I programs in the MULTICS environment. Proc. *AFIPS* FJCC 47, 1972, 507-514.