CODE OPTIMIZATION

- Local optimization
- Loop optimization
- Data flow analysis – Transformation of useful information from all part of a program to a place where it is required and can be used

COMMON TECHNIQUES

- Common sub expression elimination
- Strength reduction – high strength evaluation to low strength evaluation

```
do I = 1 to 21 step 2
 ----
k = I *5
----
end
```

→

```
Itemp = 5
   do I = 1 to 21 step 2
    ----
    k = Itemp
   Itemp = Itemp + 10
   ----
   end
```

- Loop test replacement – Replacing a loop test phrase in terms of one variable by an equivalent loop test phrase in terms of another variable

```
Itemp = 5
do I = 1 to 21 step 2
 ----
 k = Itemp;
→
 Itemp = Itemp + 10;
 ----
 end
```

→

```
      I = 1; Itemp = 5
L:    If  I >21 then goto Next
      ----
       k = Itemp;
       Itemp = Itemp + 10;
       ----
       I = I + 2;
      goto L;
Next :  ….
```

- Dead code elimination – removing a code not used in a block

```
A =2;
If x < y then do
A = B + C;
----
End;
else do
----
End;
```

→

```
If x < y then   do
  A = B +  C;
  -----
End;
else do
  A = 2
  ----
End;
```

Code movement(frequency reduction) -

Seeks to improve execution time characteristics of a program by moving the evaluation of an expression to other part of the program

```
If M > 26 then do
 Y = X**2 – Z**2;
----
End;
Else do;
---
End;
A = (X**2 – Z**2)/2;
```

→

```
If M > 26 then do
T = X**2 – Z**2;
Y=T;
 ----
  End;
 Else do;
  T = X**2 – Z**2
   ---
 End;
A = T/2;
```

- Loop invariant code movement
- induction variable – Variables whose values form an arithmetic progression at the loop header
- Code hoisting – Moving computations outside the loop as much as possible

```
do I = 1 to 100                    t = x * Pi;
  A[I] =  x * Pi * I;      →       do I = 1 to 100
end                                A[I] =  t * I
                                   end
```

- Loop unrolling – Avoid a test at every iteration by recognizing that the number of iterations is constant and replicating the body of the loop

```
begin
   I =1;
   While I <= 100 do
      begin
         A[I] = 0;
          I = I + 1
     end
end
```

→

```
begin
   I =1;
   While I <= 100 do
    begin
          A[I] = 0;
           I = I + 1;
          A[I] = 0;
           I = I + 1
    end
end
```

- Loop Jamming - Merge bodies of two loops provided that each loop executes the same no of times

```
begin
For I := 1 to n do
For  J := 1 to n do
   A[I, J] := 0;
For I := 1 to  n do
A[I, J] := 1
end
```

→

```
For I := 1 to n do
   begin
          For  J := 1 to n do
           A[I, J] := 0;
          A[I, J] := 1
    end
```

Basic Blocks

sequence of consecutive statements executed sequentially

Construction –

finding the leader

- First statement is a leader
- Any st. which is the target of conditional or unconditional goto is a leader
- Any st. which immediately follows a conditional goto is a leader

a basic block starts with a leader and includes all statements prior to the next leader

Flow graph – Directed graph constructed from basic blocks which act as nodes in the graphs

An Edge B1 → B2 If

- B2 could immediately follow B1
- If there is conditional or unconditional jump from the last st. of B1 to the first st. of B2

```
Begin
    Prod = 0
    I  = 1
    Do
     begin
          Prod = prod + A[I] * B[I]
          I = I +1
     end
     While I <= 20
End
```

1. T1 = 4*I
2. T2 = addr(A) - 4
3. T3 = T2[T1]
4. T4 =4 * I
5. T5 = addr(B) – 4
6. T6 = T5[T4]
7. T7 = T3 * T6
8. T8 = Prod +T7
9. Prod = T8
10. T9 = I +1
11. I = T9
12. If I <=20 goto 1

DAG REPRESENTATION OF BASIC BLOCK

A directed graph with no cycle where leave nodes are labeled by identifiers or constants and interior nodes are labeled by operator symbol

Uses

- Common sub expression elimination in a block
- Determines which names are used in a block and evaluated out side the block
- Which name/statement of a block could have their values used outside the block

DAG construction algo.

NODE(ID) – A function that creates a node for identifier ID

1. If NODE(B) is undefined, create it and label it B. If NODE(C) is undefined create it and label it C
2. In case of A = B op C, if the nodes B and C don't' have a parent node labeled *op* , create one whose left child is B.
   In case of A = op B, if the node B does not have a parent labeled *op*, create it
   In case of A = B, the node created B
3. Append A to the list of attached identifier for nodes found in 2.

DAG Interpretation

- If a node is redefined (whose previous value is a leaf) then it may be used outside the block

- The identifiers for which a leaf is created (in step 1) are the ones whose values are used in the block

- If a node has more than one attached identifiers on its attached list. only one of these may be used outside the block

DATA FLOW ANALYSIS

Used for gathering information like use of a variable and its definition in a program – hence called u-d- chain

Use : any occurrence of an identifier (say A) as an operand

Definition : either an assignment to A or reading of a value for A

Point – is a position before or after a statement in a program

For each basic block B

GEN(B) : Set of generated definitions within block b that reach the end of this block

KILL(B) : Set of definitions outside of B that define identifiers that also has definitions within B

DATA FLOW EQUATIONS

Computes two sets IN and OUT for each basic block B

IN[B] : set of all definitions reaching to a point just before the first statement of B

OUT[B] : Set of all definitions reraching to a point just after the last statement of B

(1) OUT[B] = IN[B] – KILL[B] U GEN[B]

(2) IN[B] = U OUT[P] , Each P is predecessor of B

"-" is computed as AND NOT , and U as OR

Order of computation {IN[B] AND (NOT KILL[B])} OR GEN[B]

REACHING DEFINITION ALGO.

iterative algo. using data flow equations to compute IN and OUT to construct u-d-chain

```
For each block B do
   Begin
        IN[B] =   FI;
        OUT[B] = GEN[B]
   End;
    change = TRUE;
     While change do
                    Begin change = FALSE;
                          For each block do
                          Begin    NEWIN = U OUT[P];
                                    If NEWIN  < >  IN[B] then change = TRUE;
                                    IN[B] = NEWIN;
                                   OUT[B] = IN[B] - KILL[B] U GEN[B]
                            End
                   end
```

LOOP INVARIANT COMPUTATION

Loop detection

Dominators

d DOM n : node d dominates node n

Iff every path path from initial node to node n goes through node d

Dominator properties

1. a DOM a => Every node dominates itself
2. a DOM b and b DOM a => a =b
3. a DOM b & b DOM c => a DOM c
4. Dominators of each node are linearly ordered

Immediate dominator of *n* is a dominator of *n* other than *n* itself which in turn is dominated by every dominator of *n* other

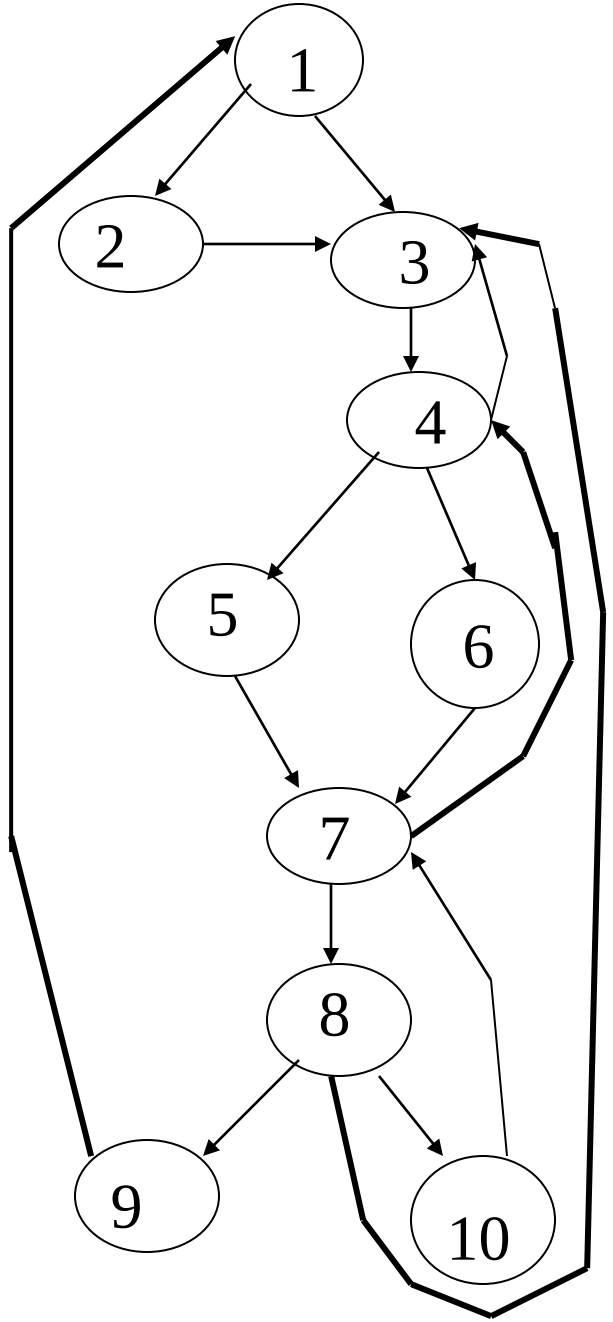
Back edge- an edge a---> b where b is head and a is tail, whose head dominates its tail is called back edge

All back edges form a loop

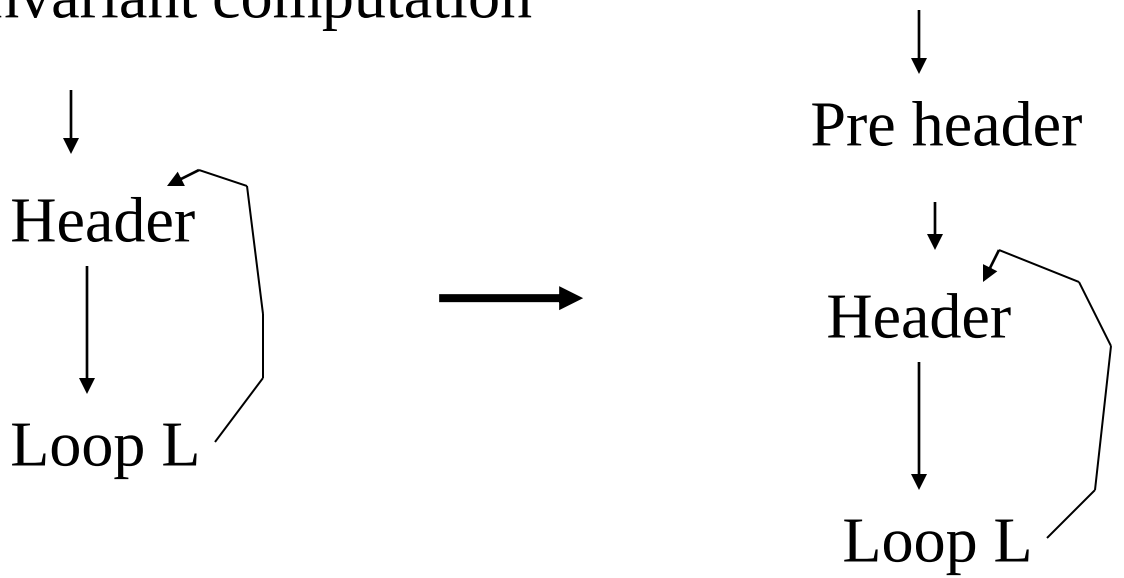
Natural loop- given a back edge n--->d

Natural loop consists of all those nodes which can reach n without going through d plus the node d

For the back edge 9--->1, natural loop consists of entire flow graph



Loop invariant computation



Loop invariant computation Algo.

Input – a loop consisting of a set of basic blocks and u-d chain information for individual statement

1. Mark “invariant” those statements whose operands are all either constant or have all their reaching definitions out side L
2. Repeat
3. Mark “invariant” all those statements not previously marked(in step 1) which have exactly one reaching definition and that definition is a statement in L marked invariant

Loop invariant computation algo. 2 (code motion)

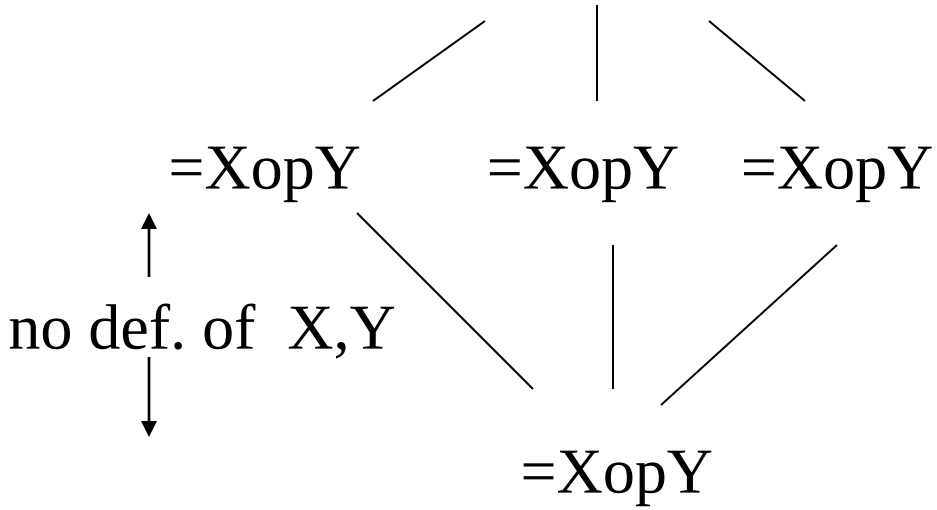
1. Block containing the st. to be moved either dominates all exits of the loop or the name assign is not used outside the loop
2. A loop invariant st. can not be moved to the pre header when it has more than one assignments in the loop
3. A st. assigning to A can not be moved to the pre header if there is a use of A in the loop which is reached by any definition of A other than the st. moved

AVAILABLE EXPRESSION

An expression ***X op Y*** is available at point p if every path(may include a cycle) form initial node to p evaluate ***X opY*** and after the last such evaluation prior to reaching p there are no subsequent assignments to X or Y

(Used to find and compute common sub expression globally)

Computation of available exp.



Data flow equations

OUT[n]=IN[n] – E_KILL[n] U E_GEN[n]

IN[n]= $\bigcap$ OUT[p], p is predecessor of n

E_GEN[n] : a block generates expression XopY if it evaluates XopY and does not subsequently redefine X or Y

E_KILL[n] : a block kills expression XopY if it assigns X or Y and does not subsequently recompute XopY

Initialization IN[n1] = *fi* & OUT[n1] = E_GEN[n1]

For i =2,N

IN[n_i] = U (universal set of all expressions)

OUT [n_i] = U – E_KILL[n_i]

IN[n_i] results in the available expression in each block n_i

After getting the available expression *copy propagation* takes place

T := BopC where D = BOPC is obtained as available expression

D: = T

replace statement **s** (assigning to A) by A := T

Constant folding

Compute and replace expression by its value if it can be computed at compile time

Algorithm

Input: expressions ***s*** and u-d chain information

For all statements ***s*** of program do

Begin

for each operand ***B*** of ***s*** do

Begin if there is unique definition of ***B*** that reaches ***s*** and
that is of the form ***B=c***, for a constant ***c*** then replace ***B*** by ***c*** in ***s***

if all operands of ***s*** are now constant then replace ***s*** by ***A = e,***
where ***e*** is the computed value of the RHS of expression and ***A***
is assigned this value by ***s***

end

end

Backward flow problem

Live variable – for a name A and a point p, if the value of a at p could be used along some path in the flow graph starting at p then a is live at p otherwise A is dead at p

analysis is extensively used in object code generation

Data flow equations for live variable analysis

IN[n] = OUT[n] – DEF[n] U USE[N]

OUT[n] = U IN[s], s: successor of n

DEF[n] : Set of names assigned values prior to any use of that name in n

USE[n] : Set of names used in n prior to any prior to any definition

Initialization

$IN[n_i]$ = ***fi*** for i = 1..n

nodes can be ordered in depth first

Computation in reverse direction (backwards) i.e from node n to 1

Busy expression computation

Expression *B op C* is very busy at point p if along every path from p, we come to a computation of *B op C* before any definition of *B* or *C*

Only one computation of busy expression is required and it is copied all computations of *B op C* from point p onwards

Data flow equations

$$IN[n] = OUT[n] - V\_DEF[n] \cup V\_USE[N]$$

$$OUT[n] = \bigcap IN[s], \text{ s: successor of}$$

U be the universal set of all expressions computed somewhere in the program

Initalization $IN[n_i] = U$, for all nodes N

CODE HOISTING

Putting(hoisting) all computations of an expression say *B* op *C* to a point say p which dominates all these computations of *B* op *C*

Algo. [after getting busy exp. Information form OUT[n]

For each occurrence *s* : *A= BopC* trace backwards from the node containing statement s to check that

1. There are no definition of *B* or *C* that reach s without without going through node n
2. There is a path from n to to s that contain def. of *B* or *C* or use of *B* op *C*
3. If steps 2 and 3 are true, compute T = ***BopC*** at point p (end of node n) and then replace all computations ***BopC*** by T