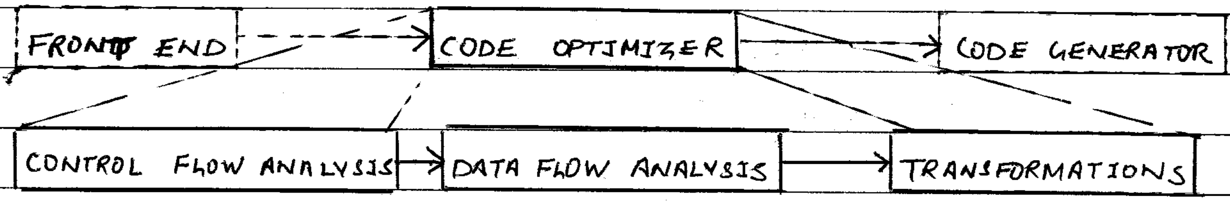
UNIT- 5

CODE OPTIMIZATION

INTRODUCTION TO CODE OPTIMIZATION –

① **Organization of the code optimizer** –



② **Principal source of optimization** –

A transformation of a program is called local if it can be performed by looking only at the statements in the basic block otherwise it is called global.

→ **Function-Preserving Transformation** –

Various types of transformation includes –

(i) Common subexpression elimination – An occurrence of an expression E is called a common subexpression if E was previously computed, and the values of variables in E have not changed since the previous computation.

(ii) Copy propagation – In this transformation is to use g for f, whenever possible after the copy statement f := g.

(iii) Dead Code Elimination – Remove dead or useless code, statements that computes values that never get used.

(iv) Constant folding – Deducing at compile time that the value of an expression is a constant and using the constant ~~code~~ instead.

→ **Loop Optimization** –

Three techniques are –

(i) Code Motion – Decrease the amount of code in a loop by moving code outside.

Eg – while (i<=limit-2) ⇒ t = limit-2; while (i<=t)

(ii) Induction variable elimination – A variable is incremented/decremented by some constant every time then it is called induction variable. If two or more induction variables in a loop, it may be possible to get rid of all ~~to~~ but one.

(iii) Reduction in Strength – It replaces an expensive operation by cheaper one such ~~eg~~ – as a multiplication by an addition.

Eg – for (int i=10; i<10; i++) { ⇒ for (int i = 1; i<10; i++) {
count = i*7; } ~~count = count+i*7~~ count=temp,
~~count = count~~ temp=temp+7; }

myCompanion

(3) <u>Optimization of basic block</u> – Examples (by constructing a DAG from a basic block)

(i) <u>Structure-Preserving transformation</u> – Various transformations are –

(1) Common subexpression elimination

(2) Dead code elimination

(ii) <u>Algebric transformations</u> –

(1) Reduction in strength . Eg → $x * x \Rightarrow x^2$

(2) Use of algebric identities . Eg – $x+0 = 0+x \Rightarrow x$, $x*1 = 1*x \Rightarrow x$

(3) simple algebric transformation .

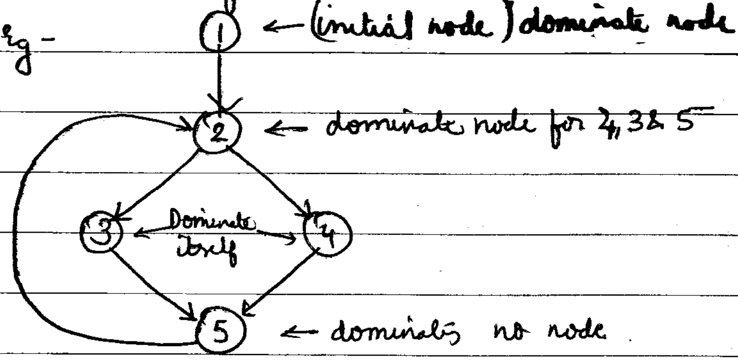
(4) Constant folding . Eg – $2 * 3.14 \Rightarrow 6.28$

(4) <u>Loops in flow graphs</u> –

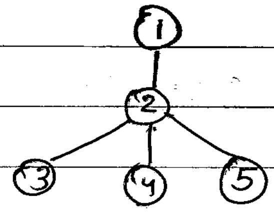
(1) <u>Dominators</u> –

In a flow graph, a node d dominates n if every path to node n from initial node goes through d only. This can be denoted as 'd dom n'. Every initial node dominates all the remaining nodes in the flow graph. Similarly every node dominates itself .

Eg –



<u>Dominator tree</u> –



(2) <u>Natural loops</u> –

(b dom a)

The heads dominate their tails i.e. If $a \rightarrow b$ is an edge, b is the head and a is tail . then this edge is called <u>back edge</u>

Given a back edge $n \rightarrow d$, we define the <u>natural loop</u> of the edge to be d plus the set of nodes that can reach n without going through d . Node d is the header of the group.

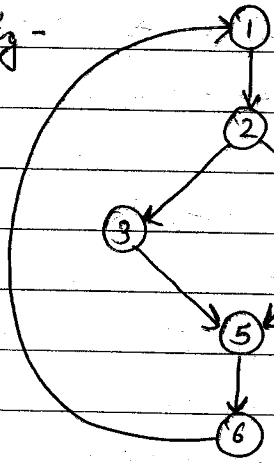
→ <u>Algorithm for constructing the 'natural loop' is given as</u> –

procedure insert(m);

if m is not in loop then begin

```
        loop := loop U {m};
        push m onto stack
    end;
    /* main program follows */
    stack := empty;
    loop := {d};
    insert(n);
    while stack is not empty do begin
        pop m, the first element of stack, off stack;
        for each predecessor p of m do insert(p)
    end
```

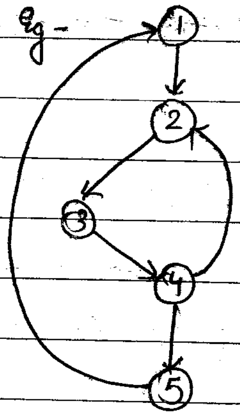
Eg -



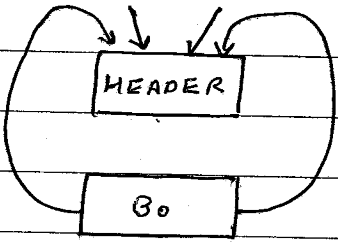
6 → 1 is a natural loop because we can reach 6 from every nodes present in the flow graph without going through 1

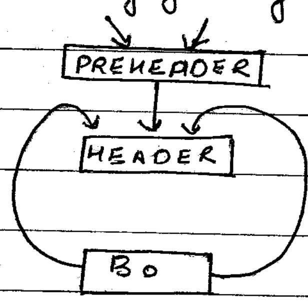
(3) Inner loops - It is a loop that contains no loop.

Eg -



4→2 is ~~inner~~ inner loop that means edge given by 2-3-4





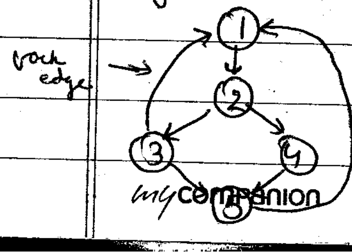
(4) Pre-Header - The pre-header is a new block created such that successor of this block is the header block. All the computations that can be made before the header block can be made before the pre-header block.

(5) Reducible flow graphs - It is flow graph in which there are two types of edges forward edges and backward edges. These edges have following properties -

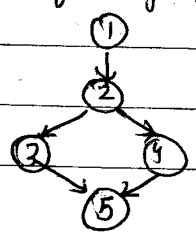
(i) The forward edge form an acyclic graph

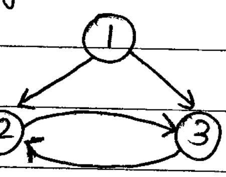
(ii) The back edges are such edges whose head dominates their tail.

The program structure in which there is exclusive use of if-then, while-do or goto statements generates a flow graph which is always reducible.



⇒





Non-reducible graph.

(5) <u>Introduction to global data flow analysis</u> -

An optimizing collects ~~by a process~~ data flow information by a process known as data flow analysis.

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

It reads as the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flow through the statement. Such equations are called data-flow equations.

Three factors on how data-flow equations are set up and solved are -

(i) Notions of generating & killing depend on the desired information.

(ii) Data-flow analysis is affected by the control constructs in the program.

(iii) There are subtleties like procedure calls, pointer variables & array variables.

→ Points & Paths -

Within a basic block, we talk ~~about~~ of the point between two adjacent statements, as well as the point ~~between~~ before the first statement & after the last.

→ Reaching definitions -

A definition of a variable x is a statement that assigns, or may assign, a value to x. Those statements certainly define x called as unambiguous statements. If those statements may define a value of x called as ambiguous statements like -

(1) A call of a procedure with x as parameter.

(2) An assignment through a pointer that could refer to x.

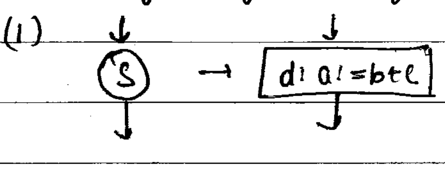
By defining reaching definitions as we have, we sometimes allow inaccuracies. However, they are all in the "safe" or "conservative" direction.

A decision is conservative if it never leads to a change in what the program computes. In applications of reaching definitions, it is normally conservative to assume that a definition can reach a point even if it might not.

→ Data flow analysis of structured programs -

→ Data flow equations for reaching definitions -

(1)

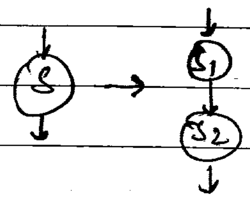


$gen[S] = \{d\}$

$kill[S] = D_a - \{d\}$

$out[S] = gen[S] \cup [in[S] - kill[S]]$

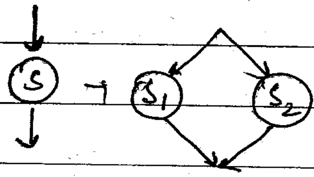
** *** This is related to definition d whether d reach the end or not

(2)


gen[s] = gen[s2] ∪ (gen[s1] − kill[s2])

kill[s] = kill[s2] ∪ (kill[s1] − gen[s2])

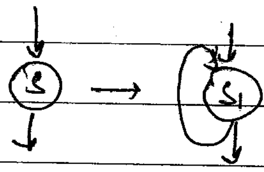
in[s1] = in[s]   in[s2] = out[s1]   out[s] = out[s2]

(3)


gen[s] = gen[s1] ∪ gen[s2]

kill[s] = kill[s1] ∩ kill[s2]

in[s1] = in[s]   in[s2] = in[s]   out[s] = out[s1] ∪ out[s2]

(4)


gen[s] = gen[s1]

kill[s] = kill[s1]

in[s1] = in[s] ∪ gen[s1]'   out[s] = out[s1]

→ **Conservative Estimation of data-flow information** –

Conservative refers to making safe assumptions when insufficient information is available at compile time, i.e. the compiler has to guarantee not to change the meaning of the optimized code

Safe refers to the fact that a superset of reaching definitions is safe

Accuracy → the larger the superset of reaching definitions, the less information we have to apply code optimization.

→ **Representation of sets** –

Set of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interests in the flow graph. Then the bit vector representing a set of definitions will have 1 in position i if and only if the definition numbered i is in the set.

(c) **Code improving transformations** –

→ Elimination of global common subexpression – consider only expression generated by block ~~and~~ and not with whether it is recomputed several times within a block. Eg –

t2 : 4*i t3 : a[t2]	⇒	u := 4*i t2 := u t3 := a[t2]	OR	(15) := 4*i (18) := a[(15)]
↓		↓		↓
t6 : 4*i t7 : a[t6]		t6 : u t7 : a[t6]		t6 := (15) t7 := (18)

→ Copy propagation elimination - statement ~~tick~~ like x := y.
→ Detection of loop-invariant computation -
Loop-invariant are those whose value does not change as long as controls within the loop, which can be remove by performing code motion.
→ Elimination of induction variables -
A variable x is called induction variable of a loop L if every time the variable x changes values, it is incremented / decremented by some constant. It can be eliminated by strength reduction.

(7) Data flow analysis of structured flow graph -
Depth first search / ordering.
Depth first spanning tree gives depth of a flow graph.
Interval partitions
Interval graphs
Node splitting
A Region in a flow graph is a set of nodes N that includes a header, which dominates all the other nodes in a region.

(8) Symbolic debugging of optimized code -
A symbolic debugger is a system that allows us to look at a program's data while that program is running.
The debugger is usually called when a program error occurs.
Deducing values of variables in basic blocks
Effects of Global optimization
- induction-variable elimination
- Global common subexpression elimination
- Code Motion