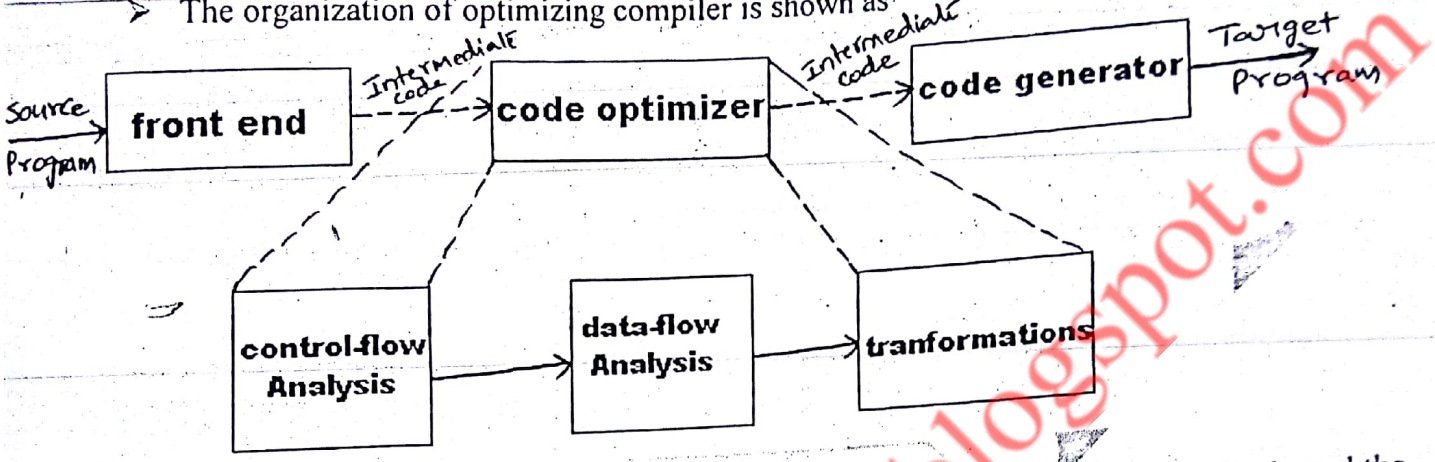
CD-UNIT-6

INTRODUCTION TO CODE OPTIMIZATION:

- The code optimization phase attempts to improve the intermediate code , so that it will result faster-running machine code.
- Compilers that apply code-improving transformations are called *Optimizing Compilers.*
- Optimizing a program is required, so that the compiler can generate
  - Efficient target code along with preserving the meaning of the program.
  - Reduction in size of generated code.
- The organization of optimizing compiler is shown as



- The code optimization phase consists of control-flow analysis, data-flow analysis and the application of transformations.
- The code generator produces the target program from the transformed intermediate code.
- The input and output of code optimizer is an Intermediate code.
- There can be situations where a piece of code cannot be optimized. Hence the output from the code optimizer is the same set of intermediate code.

Advantages of Optimizing Compiler:

- The operations needed to implement high-level constructs are made explicit in the intermediate code, so it is possible to optimize them.
- As the machine independent phases are separated from machine dependent phase (code generator), the code optimizer can easily optimize code with few changes even if the code generator is for any kind of machine.

Machine Dependent Optimization:

- Machine dependent optimization stem from special machine properties that can be exploited to reduce the amount of code or execution time.

Example:
Register allocation, data intermixed with instructions and special machine features, exi-vector operation.

Machine Independent Optimization:

- Machine Independent Optimization depends on only the arithmetic properties of the operations in the language and not on peculiarities of the target machine.

(1) Machine Independent code optimizations:
→ It depends on only arithmetic properties of the operations in the language.

(*) THE PRINCIPLE SOURCES OF OPTIMIZATION:

- A transformation of a program is called local, if it can be performed by looking only at the statements in a basic block, otherwise it is global.
- Many transformations can be performed at both the local and global values.
- Local transformations are usually performed first.
- There are number of ways in which a compiler can improve a program without changing the function it computes. They are,
  - a) Function preserving transformations.
  - b) Loop optimizations.

a). FUNCTION PRESERVING TRANSFORMATIONS: (local Optimization)

- It is also called as local optimization.
- It includes,
  - a) Common Sub-expression Elimination.
  - b) Copy Propagation.
  - c) Dead-Code Elimination.
  - d) Constant Folding.

a) Common Sub-expression Elimination:

- The code optimization can be improved by eliminating common subexpressions from the code.
- An expression whose value was previously computed and the values of variables in the expression are not changed.
- Since its computation can be avoided to recomputed it by using the earlier computed value.

Example:

```
a:=b*c
z:=b*c+d-c
```

-In this code b*c is common for both 'a' and 'z'
-Then this code is replaced by the code, while eliminating b*c subexpression is,

```
t1:=b*c
a:=t1
z:=t1+d-c
```

- It is not possible to eliminate an expression if the value of one of its variable is changed.

Example:

```
a:=b*c
c:=4+c
z:=b*c+d-c
```

-In this code b*c is not common for both 'a' and 'z' because 'c' is changed after a:=b*c is computed.

b) Copy Propagation:

- It is also called as Variable Propagation.
- If there are copy statements of the form x:=y then in Copy propagation, the use of variable x is replaced by the variable y in the subsequent expression.
- The copy propagation is possible if none of the variable is changed after this arrangement.

Example:

```
a:=b+c
d:=c
e:=b+d-3
```

-This code has common subexpressions b+c and b+d.
-If we replace the variable 'd' by the variable 'c' as both have the same value.
-After applying the copy propagation the code is,

```
a:=b+c
d:=c
e:=b+c-3
```

c) Dead-Code Elimination:

- A piece of code which is not reachable, that is, the values it computes is never used anywhere in the program then it is said to be dead code.
- It can be removed for program safely.
- An assignment to a variable results in dead code, if the value of this variable is not used in the subsequent program.
- Copy propagation often makes the copy statements into dead code.

Example:

```
a:=b+c
d:=c
e:=b+d-3
```

-Here, let us suppose the value of 'd' and 'a' are not used in the subsequent program, then we eliminate the dead code variable d and a, the code is changed after the elimination of dead code is,

```
t1:=b+c
e:=t1-3
```

d) Constant Folding:

- The substitution of values for names whose values are constant is known as Constant Folding.
- Constant folding is simply applied in such a way that values known at compile time to be associated with variables are used to replace certain uses of these variables in the translated program text.

Example:

```
#define PI 3.14
area=PI*r*r;
```

-Here area=PI*r*r; is replaced by area=3.14*r*r; at the time of compilation.

b) LOOP OPTIMIZATION:

- The major source of code optimization is loops, especially inner loops.
- Most of the run-time is spent inside the loops which can be reduced by reducing the number of instructions in an inner loop.
- It includes,
  a) Code Motion
  b) Induction Variable Elimination
  c) Reduction in Strength

a) Code Motion :

- Code Motion reduces the number of instructions in a loop by moving instructions outside a loop.
- It moves loop-invariant computations.
- Loop-invariant computations or expressions that results in the same value independent of the number of times a loop is executed.

Example:

```
while(x!=n-2)
  {
   x=x+2;
  }
```

-Here, expression n-2 is a loop-invariant computation.

- The expression is changed to

```
m=n-2;
while(x!=m)
  {
   x=x+2;
  }
```

b) Induction Variable Elimination:

- An induction variable is a loop control variable or any other variable that depends on the induction variable in some fixed way.
- It can also be defined as a variable which is incremented or decremented by a fixed number in a loop each time, the loop is executed.
- Induction variables are of the form i=i±c, where 'c' is a constant.

Example:

```
int a[10],b[10];
void fun(void)
  {
    int i, j, k;
    for(i=0, j=0 ,k=0; i<10; i++)
            a[j++]=b[k++];
    return;
  }
```

-Here,we have three induction variables i, j, k.

-j and k are not used properly, so we eliminate induction variables j and k, then the change is,

```
int a[10], b[10];
void fun(void)
  {
    int i;
    for(i=0;i<10;i++)
          a[i]=b[i];
    return;
  }
```

- This property will reduce the code and improves the run-time performance of a compiler.

c) Reduction in Strength:

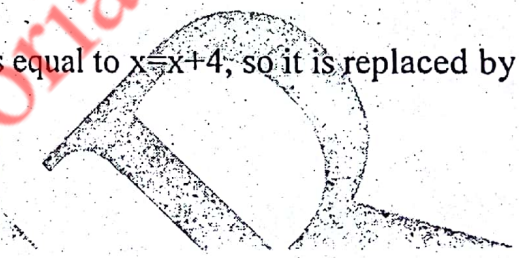
- Strength Reduction is the process of replacing expensive operations by equivalent cheaper operations on the target machine.
- On many machines a multiplication operation takes more time than addition or subtraction.
- On such machines the speed of the object code can be increased by replacing a multiplication by an addition or subtraction. This is called Reduction in Strength.

Example:

```
for(i=1;i<5;i++)
    {
     x=4*i;
    }
```

-Here, the instruction x=4*i is equal to x=x+4, so it is replaced by

```
for(i=1;i<5;i++)
    {
      x=x+4;  //where x=0
    }
```



② Peephole Optimization:

→ A peephole optimization technique is used to improve the target code.

→ Peephole optimization works by finding the peepholes (a short sequence of target Instructions) and replacing them by a shorter (or) faster sequence of Instructions.

→ It can be applied to the intermediate code (or) the object code.

→ It uses the following transformations to improve the code:

a) Elimination of Redundant instructions
b) Optimization of flow of control
c) Simplification of Algebraic expressions
d) Instructions selection.
e) Eliminate Dead code
f) Strength reduction

a) Elimination of Redundant Instructions:

→ One approach to improve the target code is to remove redundant instructions.

Example:

```
(1) MOV R1, A
(2) MOV A, R1
```

→ Here, the first instruction is storing the value of A into register R1 and second instruction storing R1 value into A.

→ These two instructions are redundant, so we eliminate Instruction (2)

→ To perform such a transformation, both instructions must be in the same basic block.

(b) Optimize flow of Controls:

→ The target code generated by the code generation algorithm frequently contains unnecessary Jumps such as,

- Jumps to Jumps
- Jumps to Conditional Jumps
- Conditional Jumps to Jumps.

Example: Jump to Jump

```
goto label1
    ==
label1 : goto label2
```

Can be achieved by the sequence

```
goto label2
    ==
label1 : goto label2
    ==
label2 :
    ==
```

where, label1 : goto label2 is removed, using peephole optimization

(c) Simplify Algebraic expressions:

→ There are endless algebraic simplifications that can be achieved through peephole optimization.

→ One of this is to implement algebraic identities that occur frequently in the code.

→ Examples are,

a := 0 + a (0)

a := 1 * a

which can be eliminated easily through peephole optimization.

(d) Instruction Selection:

→ Also called as use of machine idioms.

→ Target machines have hardware instructions that can perform certain operations more efficiently

→ The use of these instructions in the target code significantly reduces the running time of the target program.

→ The addressing modes such as auto-decrement available in some machine causes an operand to be incremented (or) decremented by one automatically.

→ The use of these modes in parameter passing and in pushing (or) popping a stack greatly improves the target code.

→ The statements like,

$$x = x + 1 \text{ (or)}$$
$$x = x - 1$$

Can also be replaced by these addressing modes.

(e) Eliminate Dead code:

→ Another possibility to improve code is to remove unreachable instructions from the target program

→ An unlabeled instruction immediately after the unconditional Jump is, unreachable, so it can be removed.

Example:

```
If FLAG = 1 goto L1 goto L2

L2 :
```

→ peephole optimization replace the above code as,

```
If FLAG ≠ 1 goto L2

L2 :
```

(f) Strength reduction:

→ In Strength reduction expensive operations replaced by equivalent cheaper operations on the target machine.

Example: x^2 is expensive because it needs a call to an exponentiation routine.

→ So it is cheaper to implement it as a multiplication expression, i.e,

```
x * x
```

→ It is cheaper to implement

- a fixed-point multiplication (or) division by a power of two as a shift operation and
- floating-point division by a constant as multiplication by a constant.

3. INTRODUCTION TO DATA FLOW ANALYSIS:

- The code improvement phase consists of control-flow analysis, data-flow analysis and application of transformations.

1.1 Basic Blocks:

- A Basic Block is a collection of three-address statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.
- Every statement in a basic block is a three-address statement.

Example:

a=(b*c)/d+k

The three-address statements are,

t1:=b*c
t2:=t1/d
a:=t2+k

1.2 Algorithm for Partitioning into Basic Blocks:

- This algorithm is used to partition a sequence of three address statements into basic blocks.

Input: A sequence of three-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

- We first determine the set of leaders, i.e, the first statements of basic blocks.
- The rules used are,

  i)The first statement is a leader.

  ii)Any statement that is the target of a conditional or unconditional goto is a leader.

  iii)Any statement that immediately follows a goto or conditional goto statement is a leader.
- A Basic Block is drawn for each leader followed by the set of statements.
- No Basic Block can have more than one leader.

GLOBAL OPTIMIZATION:-

→ The local optimization has a very restricted scope on the other hand the global optimization is applied over a broad scope such as procedure (or) function body.

→ For a global optimization, a program is represented in the form of program flowgraph.

→ The program flowgraph is a graphical representation in which each node represents the basic block and edges represent the flow of control from one block to another.

→ There are two types of analysis performed for global optimizations.

(i) Control-flow Analysis.
(ii) Data-flow Analysis.

→ The control-flow analysis determines the information regarding arrangement of graph nodes (basic blocks), presence of loops, nesting of loops and so on.

→ In control-flow analysis, the analysis is made on the flow of control by carefully examining the program flowgraph.

→ In Data-flow Analysis, the analysis is made on flow of data.

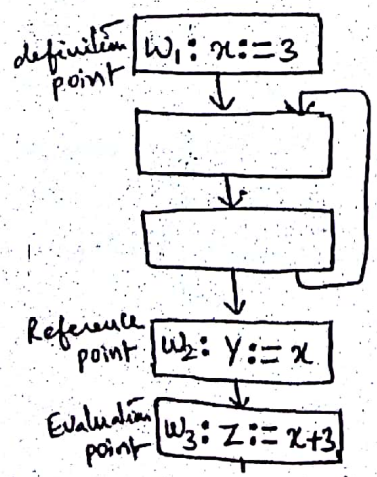
→ The Data-flow Analysis is basically a process in which the values are computed using data-flow properties such as

- Available expressions
- Reaching definitions
- Live variables
- Busy Expressions.

→ The basic terminologies on data flow properties are,

- A program point containing the definition is called definition point.
- A Program point at which a reference to a data item is made is called reference point.
- A Program point at which some evaluating expression is given is called Evaluation point.

Ex:-



Program Point (W)

(*) Data-Flow-Properties :-

(i) Available Expressions
(ii) Reaching Definitions
(iii) Live variables
(iv) Busy Expressions

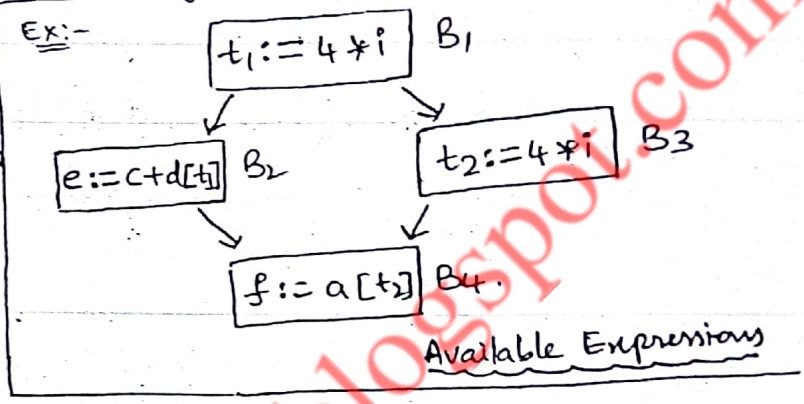
(i) Available Expressions :-

→ An expression is available at a program point 'w' if and only if along all paths are reaching to w.

→ The expression 4*i is the available for B_2, B_3 and B_4, because this expression is not been changed by any of the block before appearing in B_4.

→ The use of available expressions is to eliminate common subexpressions.

Ex:-



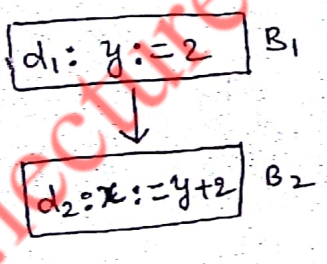
Available Expressions

(ii) Reaching Definitions :-

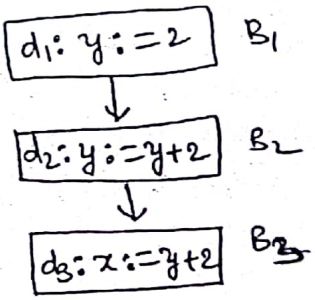
→ A definition 'd' reaches at point 'p' if there is a path from 'd' to 'p' along which 'd' is not killed.

→ A definition 'd' of a variable 'x' is killed when there is a redefinition of 'x'.

Ex:-



Reaching definition d_1



Killing definition d_1

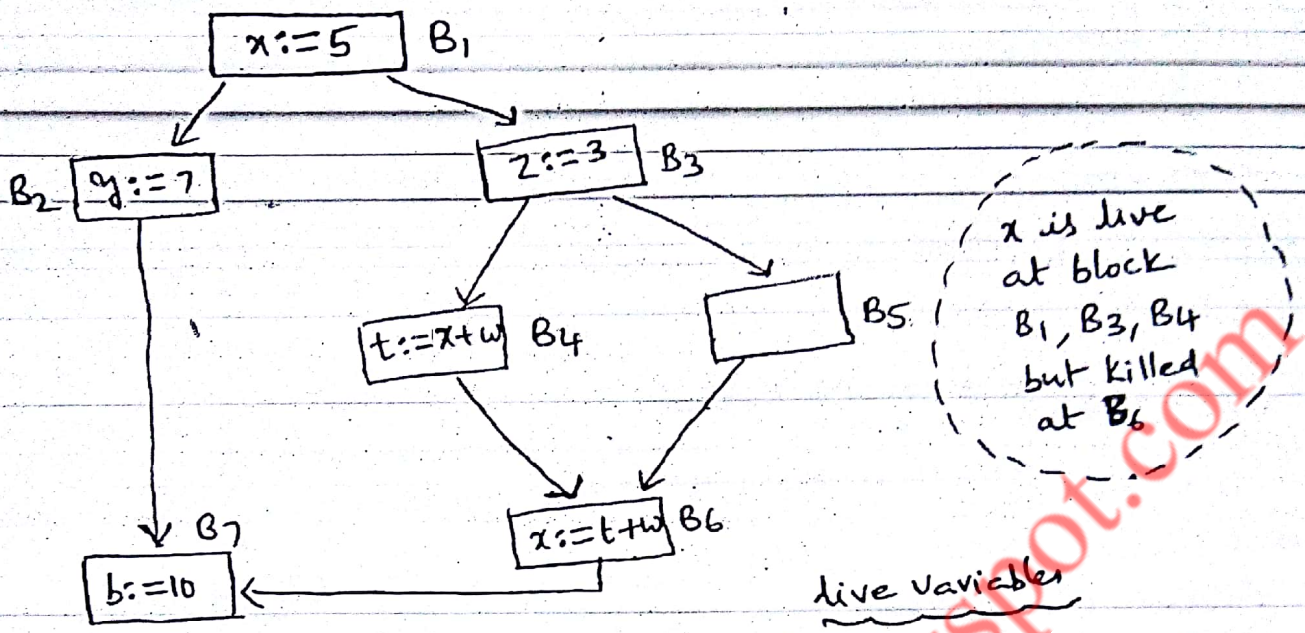
→ Reaching definitions are used in Constant and variable propagation

(iii) Live Variables :-

→ A variable 'x' is live at some point 'p' if there is a path from 'p' to the exit, along which the value of x is used before it is redefined.

→ Otherwise the variable is said to be dead at that point.

Ex:-



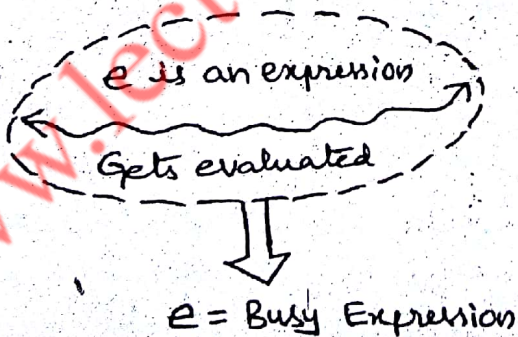
live variables

→ Live variables are useful in
- register allocation
- Dead code elimination.

(iv) Busy Expressions :-

→ An expression 'e' is said to be a busy expression along some path P_i ------ P_j if and only if an evaluation of e exists along some path P_i ----- P_j

→ No definition of any operand exists before its evaluation along the path.



Busy Expressions

→ Busy Expressions are useful in performing code movement optimization.

Q15. Write an algorithm to compute reaching definition Informatory for a flow graph.

Answer :

A statement that assigns a value to a variable V is a definition of V. A definition d of a variable V is said to reach a point P, if there exists a path from d to p along which V is not redefined. The reaching definition problem is to compute for each block all definitions of each variable which reach the beginning of the block.

The data flow equations for reaching definitions are as follows,

$$BGN[B] = \bigcup_{\substack{P \text{ is a} \\ \text{Predecessor} \\ \text{of B}}} END[P]$$

$$END[B] = GEN[B] \cup (BGN[B] - KILL[B])$$

The union operator ($\cup$) in equation for BGN[B] indicates that as definition reaches a block if it reaches the end of any of its predecessors.

Algorithm

The algorithm for computing reaching definitions takes a flow graph as input for which the sets GEN[B] and KILL[B] have been computed for each block B. The output of the algorithm is the sets BGN[B] and END[B] computed for each block B.

The algorithm initially assumes that BGN[B] = ϕ for each block B. It propagates reaching definitions information as long as they are not killed the algorithm is as follows.

1. Initialize END[B] := GEN[B] for each block B on the assumption that initially BGN[B] = ϕ for all B.
2. Do following until there are no changes in any of the END[B] sets.
   (i) For each block B calculate

$$BGN[B] := \bigcup_{\substack{P \text{ is a} \\ \text{Predecessor} \\ \text{of B}}} END[P]$$

   PREV_END[B] := END[B]

   END[B] := GEN[B] $\cup$ (BGN[B] – KILL[B]);

   (ii) Check whether there are any changes in END[B] sets by comparing PREV_END[B] and END[B]. Record the result this helps in terminating the algorithm.

Q16. Explain the working of the above algorithm using a suitable example.

Answer :

Model Paper-III, Q7(b)

Consider the graph shown below,

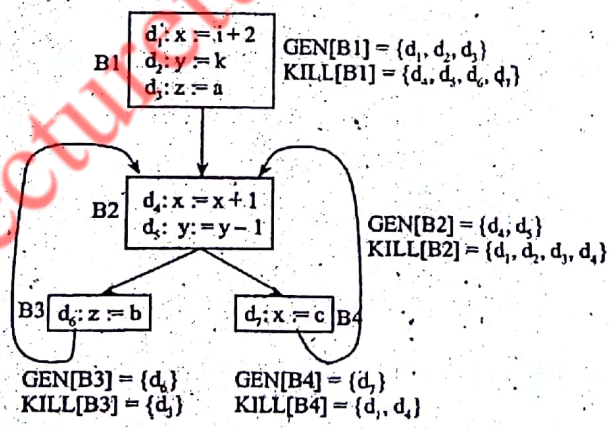


Figure: Flow Graph

The above figure shows GEN and KILL sets computed for each block. In the above flow graph there are seven definitions $d_1, d_2, \ldots, d_7$ defining the variables x, y and z.

We use bit vectors to represent the set of definitions. In a bit vector, the bit i contains 1 if and only if the definition d_i is in the set.

The first step of the algorithm initializes END[B] = GEN[B] for each block B on the assumption that initially BGN[B] = ϕ for all B. These initial values of END[B] are shown in the table given below.

The algorithm enters into the while loop and starts first iteration. Suppose in the inner for loop B takes B1, B2, B3 and B4 in that order with B = B1.

BGN[B1] = ϕ represented by 000 0000 since B1 is an initial node which have no predecessors and

END[B1] = GEN[B1] .

Which also remains equal to GEN[B1] since PREV_END[B1] = END[B1] the variable any changes is not set to true.

Next for loop takes B = B2 for which BGN and END sets are computed as follows,

BGN[B2] = END[B1] $\cup$ END[B3] $\cup$ END[B4]

= 1110000 + 0000 010 + 0000 001

= 1110 011

END[B2] = GEN[B2] $\cup$ (BGN[B2] – KILL[B2])

= 0001 100 + (1110 011 – 1100 001)

= 0011 110

Similarly BGN and END sets for B3 and B4 are computed the table given below shows these computations.

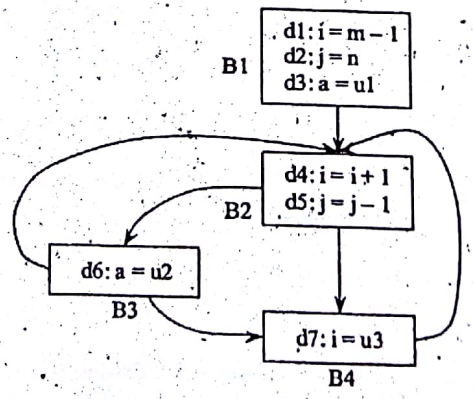
Block B	Initial Values		Iteration 1		Iteration 2	
	BGN[B]	END[B]	BGN[B]	END[B]	BGN[B]	END[B]
B1	0000000	1110000	0000000	1110000	0000000	1110000
B2	0000000	0001100	1110011	0011110	1111111	0011110
B3	0000000	0000010	0011110	0001110	0011110	0001110
B4	0000000	0000001	0011110	0010111	0011110	0010111

Table: Computation of BGN and END Sets for Example Flow Graph

At the end of the first iteration the set END[B4] = 0010111 represents that the definition d_7 is generated in B4 and definitions d_3, d_5 and d_6 reach B4 without being killed in B4.

The second iteration of while loop starts. In this iteration there are no changes to any of the END sets so the algorithm terminates.

Q17. Write the iterative algorithm for reaching definition. Compute in and out for the following figure.

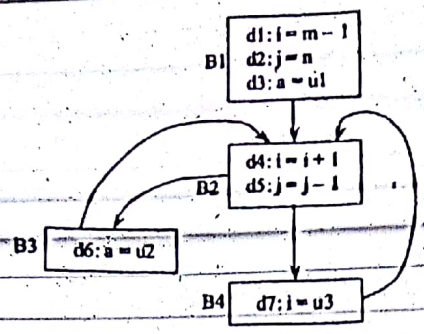


Figure

Answer :

Iterative Algorithm for Reaching Definition

Model Paper-II, Q7(b)



Figure

In the figure of flow graph, there are seven definitions d_1, d_2 ..... d_7 defining the variables i, j and a.

The GEN and KILL sets for each block are as follows,

For B1: GEN[B1] = $\{d_1, d_2, d_3\}$

KILL[B1] = $\{d_4, d_5, d_6, d_7\}$

For B2: GEN[B2] = $\{d_4, d_5\}$

KILL[B2] = $\{d_1, d_2, d_7\}$

For B3: GEN[B3] = $\{d_6\}$

KILL[B3] = $\{d_3\}$

For B4: GEN[B4] = $\{d_7\}$

KILL[B4] = $\{d_1, d_4\}$

We use bit vectors to represent the set of definitions. In a bit vector, the bit i contains 1 if and only if the definition d_i is in the set.

The first step of the algorithm initializes END[B] = GEN[B] for each block B on the assumption that initially BGN[B] = ϕ for all B. These initial values of END[B] are shown in the table.

First Iteration

The algorithm enters into the while loop and *starts first iteration*. Suppose in the inner for loop B takes B1, B2, B3 and B4 in that order with B = B1.

BGN[B1] = ϕ represented by 000 0000 since B1 is an initial node which have no predecessors and

END[B1] = GEN[B1] = 1110000

Which also remains equal to GEN[B1] since PREV_END[B1]=END[B1] the variable any changes is not set to true.

Next, for loop takes B = B2 for which BGN and END sets are computed as follows,

BGN[B2] = END[B1] $\cup$ END[B3] $\cup$ END[B4]

= GEN[B1] $\cup$ 0000000 $\cup$ 0000000

= 1110000 + 0000000 + 0000000

= 1110000

END[B2] = GEN[B2] $\cup$ (BGN[B2] − KILL[B2])

= 0001100 + (1110000 − 1100001)

= 0001100 + 0010000

= 0011100

Now, for loop takes B = B3 for which BGN and END sets are computed as follows,

BGN[B3] = END[B2]

= 0011100

END[B3] = GEN[B3] ∪ (BGN[B3] – KILL[B3])

= 0000010 + (0011100 – 0010000)

= 0000010 + 0001100

= 0001110

Now, for loop takes B = B4 for which BGN and END sets are computed as follows,

BGN[B4] = END[B2]

= 0011100

END[B4] = GEN[B4] ∪ (BGN[B4] – KILL[B4])

= 0000001 + (0011100 – 1001000)

= 0000001 + 0010100

= 0010101

Second Iteration

The algorithm enters into the while loop again and *starts second iteration*. Here, we consider the BGN and END sets of each block obtained in first iteration.

For B = B1; BGN[B1] = ϕ = 0000000

END[B1] = GEN[B1]

= 1110000

For B = B2; BGN[B2] = END[B1] ∪ END[B3] ∪ END[B4]

= 1110000 + 0001110 + 0010101

= 1111111

END[B2] = GEN[B2] ∪ (BGN [B2] – KILL[B2])

= 0001100 + (1111111 – 1100001)

= 0001100 + 0011110

= 0011110

For B = B3; BGN[B3] = END[B2]

= 0011110

END[B3] = GEN[B3] ∪ (BGN[B3] – KILL[B3])

= 0000010 + (0011110 – 0010000)

= 0000010 + 0001110

= 0001110

For B = B4; BGN[B4]= END[B2]

= 0011110

END[B4]= GEN[B4] ∪ (BGN[B4] – KILL[B4])

= 0000001 + (0011110 – 1001000)

= 0000001 + (0010110)

= 0010111

Third Iteration

The algorithm enters into the while loop again and starts third iteration. Here, we consider the BGN and END sets of each block obtained in second iteration.

For B = B1; BGN[B1]= ϕ = 0000000

END[B1]= GEN[B1]

= 1110000

For B = B2; BGN[B2]= END[B1] ∪ END[B3] ∪ END[B4]

= 1110000 + 0001110 + 0010111

= 1111111

END[B2]= GEN[B2] ∪ (BGN[B2] – KILL[B2])

= 0001100 + (1111111 – 1100001)

= 0001100 + 0011110

= 0011110

For B = B3; BGN[B3]= END[B2]

= 0011110

END[B3]= GEN[B3] ∪ (BGN[B3] – KILL[B3])

= 0000010 + (0011110 – 0010000)

= 0001110

For B = B4; BGN[B4]= END[B2]

= 0011110

END[B4]= GEN[B4] ∪ (BGN[B4] – KILL[B4])

= 000001 + (0011110 – 1001000)

= 000001 + 0010110

= 0010111

In the third iteration of while loop there are no changes to any of the END sets. So, the algorithm terminates.

The table shows all the BGN and END sets for the given flow graph:

Block	Initial Values		Iteration 1		Iteration 2	
B	BGN[B]	END[B]	BGN[B]	END[B]	BGN[B]	END[B]
B1	0000000	1110000	0000000	1110000	0000000	1110000
B2	0000000	0001100	1110011	0011110	1111111	0011110
B3	0000000	0000010	0011110	0001110	0011110	0001110
B4	0000000	0000001	0011110	0010111	0011110	0010111

SHORT QUESTIONS WITH SOLUTIONS

Q1. Discuss briefly about,

(i) Local optimizations

(ii) Global optimizations.

Answer : Model Paper-I, Q1(f)

(i) Local Optimizations

These are the optimizations carried out within a single basic block. This technique do not require the information regarding the data and flow of control. Thus, implementation of this technique is simple.

(ii) Global Optimizations

These are the optimizations carried out across basic blocks, instead of single basic block. This analysis is also known as data-flow analysis. In this technique, additional analysis is required across basic blocks. Thus, implementation of this technique is complex.

Q2. Briefly explain how global common sub expressions are eliminated.

Answer :

Eliminating Global Common Subexpression

The code can be improved by eliminating common subexpressions from the code. An expression whose value was previously computed and the values of variables in the expression are not changed, since its computation can be avoided to recompute it by using the earlier computed value.

Example

Consider the following sequence of code.

```
a := b * c
:
z := b * c + d – c
```

In the above code the assignment to z have the common subexpression $b * c$. Since its value is not changed after the point it was computed, and its use in the expression z, we can avoid recomputing it by replacing the above code as follows:

```
t1 := b * c
a := t1
:
z := t1 + d – c
```

However, it is not possible to eliminate an expression if the value of one of its variable is changed.

Consider the following code:

```
a := b * c
c := 4 + c
:
z := b * c + d
```

Here, the expression $b * c$ is not common because the value of variable c is changed after computing $b * c$. Therefore, we cannot eliminate this expression.

Q3. Explain briefly elimination of redundant instruction in reephole optimization.

Answer :

One approach to improve the target code is to remove redundant instructions. For example, consider the following instructions.

1. MOV R1, A
2. MOV A, R1.

Here the first instruction is storing the value of A into register R1 and second instruction is loading R1 value into A. These two instructions are redundant so we eliminate instruction (2), because whenever instruction (2), is executed after (1), it is ensured that the register R1 contains A value. But of the instruction (2), had a label then we could not delete it because we are not sure that (1) will always be executed before (2). To perform such a transformation both instructions must be in the same basic block.

Q4. Discuss briefly about,

(i) Simplification of algebraic expressions

(ii) The use of machine instructions.

Answer :

(i) Simplification of Algebraic Expressions **Model Paper-IV, Q1(f)**

There are endless algebraic simplifications that can be achieved through peephole optimization. One of this is to implement algebraic identities that occur frequently in the code. For example, the intermediate code generator often produces statements such as,

$a := 0 + a$ or $a = 1 * a$

Which can be eliminated easily through peephole optimization.

(ii) The Use of Machine Instructions

In strength reduction expensive operations replaced by equivalent cheaper operations on the target machine. For example, the expression x^2 is expensive because it needs a call to an exponentiation routine. So it is cheaper to implement it as a multiplication expression i.e., $x * x$. It is cheaper to implement a fixed-point multiplication or division by a power of two as a shift operation and floating-point division by a constant as multiplication by a constant.

Target machines have hardware instructions that can perform certain operations more efficiently. The use of these instructions in the target code significantly reduces the running time of the target program. The addressing modes such as auto-decrement available in some machine causes an operand to be incremented or decremented by one automatically. The use of these modes in parameter passing and in pushing or popping a stack greatly improves the target code. The statements like $x = x + 1$ or $x = x - 1$ can also be replaced by these addressing modes. This technique is also called use of machine idioms.

The difficulty with this technique is that there are many ways to perform a computation on a typical target machine.

Mention the issues to be considered while applying the techniques for code optimization.

(Model Paper-II, Q1(f) | Nov.-15, Set-1, Q1(f))

Answer :

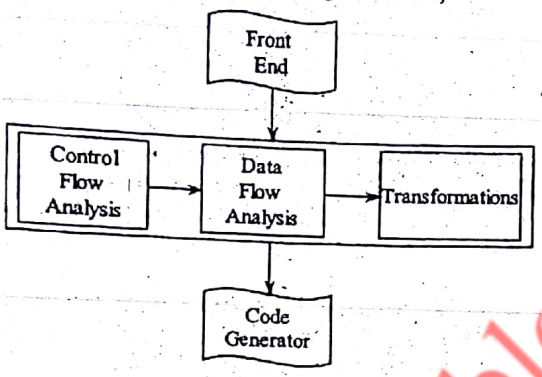
Optimizing a program is required, so that the compiler can generate.

(i) Efficient target code along with preserving the meaning of the program

(ii) Reduction in the size of generated code.

Thus, getting better performance in terms of space and time, is the idea behind optimization.

- A compiler which spends some more amount of time on code optimization, when compared to the rest phases is known as an "optimizing compiler".
- The organization of an optimizing compiler is given in the figure below,



Figure

The code optimizer in the above figure uses the intermediate code and,

- Control flow analysis.
- Data flow analysis.
- Transformations to generate an optimized code.
- The input and output of the code optimizer is an intermediate code. However, the code is an optimized set of intermediate statements. In between, the code optimizer uses control flow analysis and then data flow analysis and then performs some transformations to generate optimized code.
- There can be situations where a piece of code cannot be optimized. Hence, the output from the code optimizer is the same set of intermediate code statements.

Q6. Distinguish between machine dependent and machine independent optimization.

Answer :

	Machine Dependent		Machine Independent
1.	It is dependent on the instruction set and addressing modes to be used.	1.	It is independent of the target machine, but depends on the source language characteristics.
2.	The efficiency of the program is improved by allocating sufficient number of resources.	2.	The efficiency of the target code is improved by using appropriate program structure.
3.	Intermixed instructions with data increases the speed of execution.	3.	Elimination of dead code increases the speed of execution.
4.	Intermediate instructions are used wherever necessary.	4.	Identical computations are moved to one place, thus avoiding the repeated computation of an expression.

Q7. Write a short note on,

(I) Points

(II) Paths.

Answer :

(I) Points

A point is the one that comes between two contiguous statements and the statements that comes one after the other. The following figure illustrates the use of points in basic block.

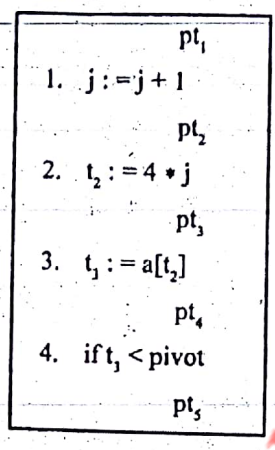


Figure: Basic Block Containing Different Points

As shown in above figure, if block contains four statements then there exists exactly five points (i.e., $pt_1, pt_2, pt_3, pt_4, pt_5$).

(ii) Paths

A path is considered as a set of points, say, $pt_1, pt_2, ..., pt_n$ which are arranged in sequential order providing the global view of the flow of control and satisfying one of the following two conditions for each 'i' where $1 \le i \le n$.

1. If point 'pt_i' comes exactly before the statement then point 'pt_{i+1}' comes exactly after the statement in the same block, instead of two other blocks.
2. If point 'pt_i' comes exactly at the end of one block then point 'pt_{i+1}' comes exactly at the beginning of the next block or following block, instead of the same block.

Q8. Explain different types of transformations used to improve the code.

Answer :

Model Paper-III, Q1(f)

The different types of transformations used for improving the code are as follows,

1. Function preserving transformations
2. Structure preserving transformations
3. Algebraic transformations.

1. Function Preserving Transformations

These are the transformations carried out without making change in the computing function. In general, these are applied on global transformations.

2. Structure Preserving Transformations

These are the transformations carried out without making change in the set of expressions it computes. Usually, these are applied on local optimizations.

3. Algebraic Transformations

These are the transformations carried out to simplify the process of computation for set of expressions using algebraic identities. In this, expensive operations like multiplication by 2 is replaced with cheaper one i.e., left shift.