Prof. Deepali Raikar

9

Code Optimization

9.1 Introduction

The code optimization is required to produce an efficient target code. There are two important issues that need to be considered while applying the techniques for code optimization and those are -

1. The semantic equivalence of the source program must not be changed.
2. The improvement over the program efficiency must be achieved **without changing the algorithm** of the program.

The code becomes inefficient because of two factors; one is programmer and the other factor is compiler. In this chapter we will discuss the various transformations that could be applied on the code to improve the performance of the program. Sometime a question arises that where exactly the code optimization can be appilied whether it should be before code generation phase or after the code generation phase? Here is a block diagram which itself suggests us the places where exactly code optimization techniques can be applied.

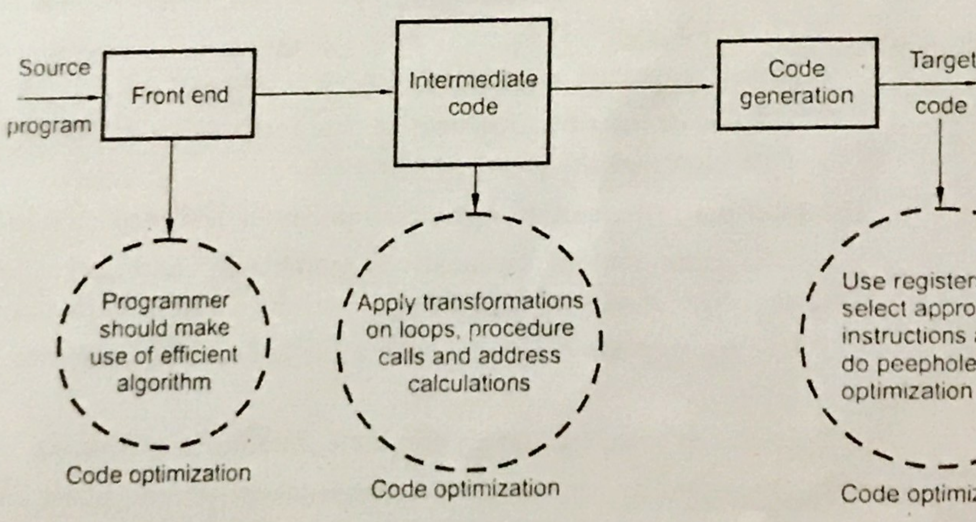


Fig. 9.1 Code optimization

Thus it is the responsibility of both programmer and compiler to generate efficient code using proper usage of target code.

9.2 Classification of Optimization

The classification of optimization can be done in two categories machine dependant optimization and machine independent optimization.

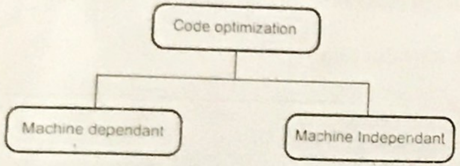


Fig. 9.2 Classificaiton of optimization

The machine dependant optimization is based on characteristics of the target machine for the instruction set used and addressing modes used for the instructions to produce the efficient target code.

The machine dependant optimization is based on the characteristics of the programming languages for appropriate programming structure and usage of efficient arithmetic properties in order to reduce the execution time.

The **machine dependant optimization** can be achieved using following criteria :

1. Allocation of sufficient number of resources to improve the execution efficiency of the program.
2. Using immediate instructions wherever necessary.
3. The use of intermix instructions. The intermixing of instruction along with the data increases the speed of execution.

The **machine independent optimization** can be achieved using following criteria

1. The code should be analyzed completely and use alternative equivalent sequence of source code that will produce a minimum amount of target code.
2. Use appropriate program structure in order to improve the efficiency of target code.
3. From the source program eliminate the unreachable code.
4. Move two or more identical computations at one place and make use of the result instead of each time computing the expressions.

9.3 Principle Sources of Optimization

The optimization can be done locally or globally. If the transformation is applied on the same basic block then that kind of transformation is done locally otherwise transformation is done globally. Generally the local transformations are done first.

While applying the optimizing transformations the semantics of the source program should not be changed.

9.3.1 Compile Time Evaluation

Compile time evaluation means shifting of computations from run time to compilation time. There are two methods used to obtain the compile time evaluation.

1. Folding

In the folding technique the computation of constant is done at compile time instead of execution time.

For example : length = (22/7) * d

Here folding is implied by performing the computation of 22/7 at compile time instead of execution time.

2. Constant propagation

In this technique the value of variable is replaced and computation of an expression is done at the compilation time.

For example :

```
pi = 3.14;
r=5;
Area = pi * r*r
```

Here at the compilation time the value of pi is replaced by 3.14 and r by 5 then computation of 3.14 * r * r is done during compilation.

9.3.2 Common Sub Expression Elimination

The common sub expression is an expression appearing repeatedly in the program which is computed previously. Then if the operands of this sub expression do not get changed at all then result of such sub expression is used instead of recomputing it each time.

For example :

$t_1 := 4 * i$

$t_2 := a[t_1]$

$t_3 := 4 * j$

$t_4 := 4 * i$

t_5 := n;

t_6 := b[t_4]+t_5

The above code can be optimized using the common sub expression elimination as,

t_1 := 4 * i

t_2 := a[t_1]

t_3 := 4 * j

t_5 := n;

t_6 := b[t_1]+t_5

The common sub expression t_4 := 4 * i is eliminated as its computation is already in t_1 and value of i is not been changed from definition to use.

9.3.3 Variable Propagation

Variable propagation means use of one variable instead of another.

For example :

```
x=pi;
...
area=x * r * r;
```

The optimization using variable propagation can be done as follows,

```
area = pi * r * r;
```

Here the variable x is eliminated.

9.3.4 Code Movement

There are two basic goals of code movement -

1. To reduce the size of the code i.e. to obtain the space complexity.
2. To reduce the frequency of execution of code i.e. to obtain the time complexity.

For example :

```
for(i=0;i<=10;i++)
{
  x=y*5;
  k=(y*5) + 50;
}
```

This can be optimized as

```
temp = y*z;
```



```
for(i=0;i<=10;i++)
{
  x=z;
  k=z + 50;
}
```

The code for y * 5 will be generated only once. And simply the result of that computation is used wherever necessary.

Loop invariant computation

The loop invariant optimization can be obtained by moving some amount of code outside the loop and placing it just before entering in the loop. This method is also called code motion.

For example :

```
while(i<=Max-1)
{
  sum= sum+a[i];
}
```

The above code can be optimized by removing the computation of Max–1 outside the loop. The optimized code can be,

```
n=Max-1;
while(i<=n)
{
  sum=sum+a[i];
}
```

9.3.5 Strength Reduction

The strength of certain operators is higher than others. For instance strength of * is higher than +. In strength reduction technique the higher strength operators can be replaced by lower strength operators.

For example :

```
for(i=1;i<=50;i++)
{
  ...
  count = i*7;
  ...
}
```

Here we get the count values as 7,14, 21 and so on upto less than 50.

This code can be replaced by using strength reduction as follows -

```
temp=7;
for(i=1;i<=50;i++)
```

```
{
...
count = temp;
temp=temp+7;
}
```

Thus we get the value of count as 7,14, 21 and so on upto less than 50.

The induction variable is integer scalar identifier used in the form of

v = v ± constant, Here v is a induction variable.

The strength reduction is not applied to the floating point expressions because such a use may yield different results.

9.3.6 Dead Code Elimination

A variable is said to be **live** in a program if the value contained into it is used subsequently. On the other hand, the variable is said to be dead at a point in a program if the value contained into it is never been used. The code containing such a variable supposed to be a dead code. And an optimization can be performed by eliminating such a dead code.

For example :

```
i=j;
...
x=i+10;
...
```

The optimization can be performed by eliminating the assignment statement i=j. This assignment statement is called **dead assignment**.

Another example :

```
i=0;
if(i=1)
{
  a=x+5;
}
```

Here **if** statement is a dead code as this condition will never get satisfied hence if statement can be eliminated and optimization can be done.

9.3.7 Loop Optimization

The code optimization can be significantly done in **loops** of the program. Specially inner loop is a place where program spends large amount of time. Hence if number of instructions are less in inner loop then the running time of the program will get decreased to a large extent. Hence loop optimization is a technique in which code optimization performed on inner loops. The loop optimization is carried out by following methods -

1. Code motion.
2. Induction variable and strength reduction.
3. Loop invariant method.
4. Loop unrolling.
5. Loop fusion.

Let us discuss these methods with the help of example.

1. Code motion

Code motion is a technique which moves the code outside the loop. Hence is the name. If there lies some expression in the loop whose result remains unchanged even after executing the loop for several times, then such an expression should be placed just before the loop (i.e. outside the loop). Here before the loop means at the entry of the loop.

For example :

```
While (i<=Max-1)
{
    sum=sum+a[i];
}
```

The above code can be optimized by removing the computation of Max −1 outside the loop. Hence the optimized code can be -

```
n = Max-1;
While (i<=n)
{
 sum=sum+a[i];
}
```

2. Induction variables and reduction in strength

A variable x is called an induction variable of loop L if the value of variable gets changed every time. It is either decremented or incremented by some constant.

For example :

Consider the block given below.

B1

$$\begin{array}{l} i := i+1 \\ t_1 := 4 * j \\ t_2 := a[t_1] \\ \text{if } t_2 < 10 \text{ goto B1} \end{array}$$

- The structure preserving transformation is a DAG based transformation. That means a DAG is constructed for the basic block then the above said transformations can be applied.
- The common subexpression can be easily detected by observing the DAG for corresponding basic block.

For example :

Consider ,

m:= n * p

n:= m+q

p:= n * p

q:= m+q

if we assume the values of

n=1

p=2 and

q=3 then the expressions becomes

m:=n * p ⟵

:=1 * 2

∴ m:=2

n:=m+q ⟵

:=2+3

∴ n:=5

p:=n * p ⟵

:=5 * 2

∴ p:=10

q:=m+q ⟵

:=2+3

∴ q:=5

(Note that the values are given for understanding of common *sub expressions*).

Thus the above sequence of instructions contain two common sub expressions such as n * p and m+q. But for the common sub expression n * p the value of n gets changed when it reappears. Hence n * p is not a common sub expression. But the expression m+q gives the same result in repetitive appearance and values of m and q are consistent each time. Therefore m+q is supposed to be the common sub

expression. Ultimately n and q are the same. The DAG is constructed and the common expressions can be identified.

For us the common sub expressions means the expressions that are guaranteed to compute the same value. The DAG construction method for optimization is effective for dead code elimination. We can delete any node from the DAG, that has no ancestors. This results in dead code elimination.

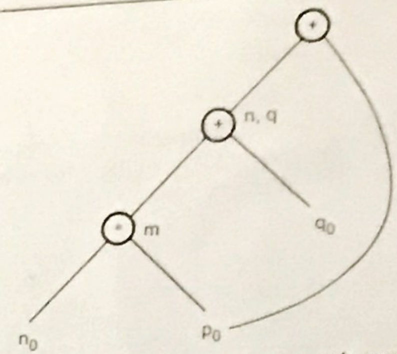


Fig. 9.3 DAG for identifying common sub expression

Use of Algebraic Identities

- Algebraic identities are used in peephole optimization techniques. Also some simple transformations can be applied in order to optimize the code.

For example :

a + 0 = a

a * 1 = a

a / 1 = a

Thus corresponding identities can be applied on corresponding algebraic expressions.

- The algebraic transformation can be obtained using the strength reduction technique.

For example :

Instead of using 2 * a we can use a+a.

Instead of using a/2 we can use a * 0.5

Thus use of lower strength operator instead of higher strength operator makes the code efficient.

- The constant folding technique can be applied to achieve the algebraic transformations.

For example :

Instead of using a= 2 * 5.4 we can use a=10.8. This saves the efforts of compiler in doing computations.

- The use of common sub expressions elimination, use of associativity and commutativity is to apply algebraic transformations on basic blocks.

For example : Consider a block

x:=y * z

t:=z * r * y

Here commutative law can be applied as y * z = z * y and from second expression we can replace z * y by x hence optimized block becomes

x:=y * z

t:=x * r

.5 Loops in Flow Graphs

Let us get introduced with some common terminologies being used for loops in flow graph.

1. Dominators

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.

For example :

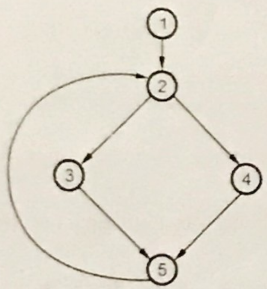


Fig. 9.4 Flow graph

In above flow graph,

Node 1 is initial node and it dominates every node as it is a initial node.

Node 2 dominates 3, 4 and 5 as there exists only one path from initial node to node 2 which is going through 2(it is 1-2-3). Similarly for node 4 the only path exists is 1-2-4 and for node 5 the only path existing is 1-2-4-5 or 1-2-3-5 which is going through node 2.

Node 3 dominates itself similarly node 4 dominates itself. The reason going to remaining node 5 we have another path 1-4-5 which is not through 3 dominates itself) Similary for going to node 5 there is another path 1-3-5 wh not through 4(hence 4 dominates itself).

Node 5 dominates no node.

2. Natural loops

Loop in a flow graph can be denoted by n → d such that d dom n. These edges are called back edges and for a loop there can be more than one back edges. If there is p → q then q is a head and p is a tail. And head dominates tail.

For example :

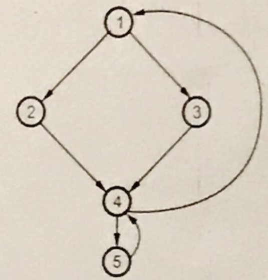


Fig. 9.5 Flow graph with loops

The loops in the above graph can be denoted by 4 → 1 i.e. 1 dom 4. Similarly 5 → 4 i.e. 4 dom 5.

The natural loop can be defined by a back edge n → d such that there exists a collection of all the nodes that can reach to n without going through d and at the same time d also can be added to this collection.

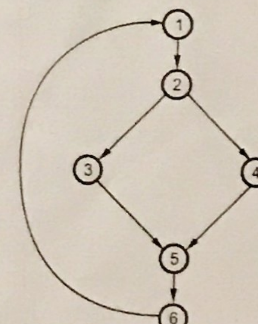


Fig. 9.6 Flow graph of natural loop

For example :

6 →1 is a natural loop because we can reach to all the remaining nodes from 6.

By observing all the predecessors of node 6 we can obtain a natural loop. Hence 2-3-4-5-6-1 is a collection in natural loop.

3. Inner loops

The inner loop is a loop that contains no other loop.

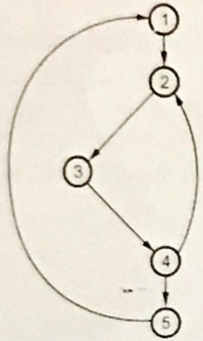


Fig. 9.7 Flow graph for inner loop

Here the inner loop is 4 → 2 that means edge given by 2-3-4.

4. Pre-header

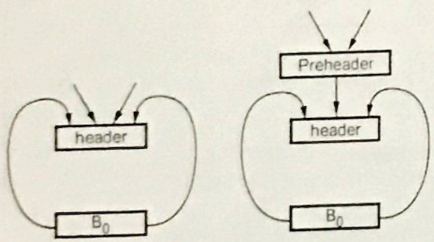


Fig. 9.8 Pre-header

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.

The pre-header can be as shown in above Fig. 9.8.

5. Reducible flow graphs

The reducible graph is a 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.

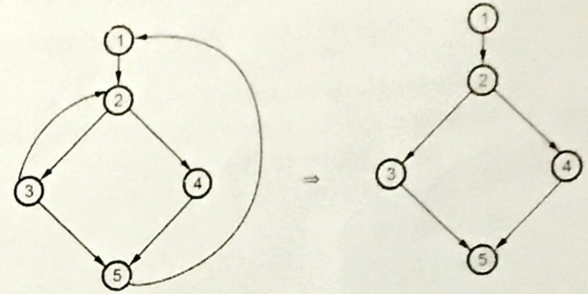


Fig. 9.9 Flow graph

The above flow graph is reducible. We can reduce this graph by removing the back edge from 3 to 2 edge. Similarly by removing the backedge from 5 to 1 we can reduce the above flow graph. And the resultant graph is a cyclic graph.

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.

Example 9.1 : *Consider the following flow graph and find*

a) Dominators for each basic block.

b) Detect all the loops in the graph. For each loop find the back edge and header block information.

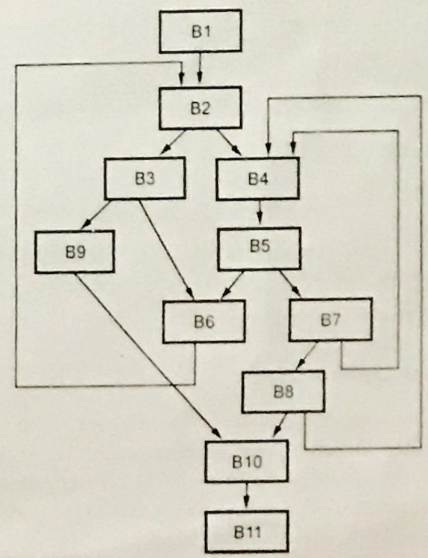


Fig. 9.10

Solution : In a flow graph node 'd' dominates n if every path to node 'n' from initial node goes through 'd' node (Block) only.

Every node (Block) dominates itself.

B1 = As it is an initial node it dominates all the nodes.

B2 = B3, B4, B5, B6, B7, B8, B9, B10, B11.

B3 = B9

B4 = B5, B7, B8

B5 = B7, B8

B6 = dominates itself because there exist another path also to go to other remaining nodes.

B7 = B8

B8 = dominates itself

B9 = dominates itself

B10 = dominates node B11 because to reach to B11 we have to go through B10.

B11 = dominates no node

Loop 1

Back edge : B6 - B2

Header : B2

Loop 2

Back edge : B7 - B4

Header : B4

Loop 3

Back edge : B8 - B4

Header : B4

9.6 Local Optimization

- The local optimization is done at restricted scope. In other words local optimization is a kind of optimization that can be done on some sequence of statements. Typically local optimization can be done within the specific basic blocks.

```
........
a: = b+c
d: = e*f
p: = b+c
q: = e*f
........
```

Fig. 9.11 Basic block

- A DAG is constructed for the basic block and then optimization can be d
- Peephole technique is effective method of optimization for doing the loca optimization.

9.6.1 DAG Based Local Optimization

As mentioned earlier, a DAG can be constructed for a block and certain transformations such as common sub expression elimination; dead code elimination can be applied for performing the local optimization.

For example : Construct the DAG for the following block

a := b * c

d := b

e := d * c

b := e

f := b + c

g := f + d

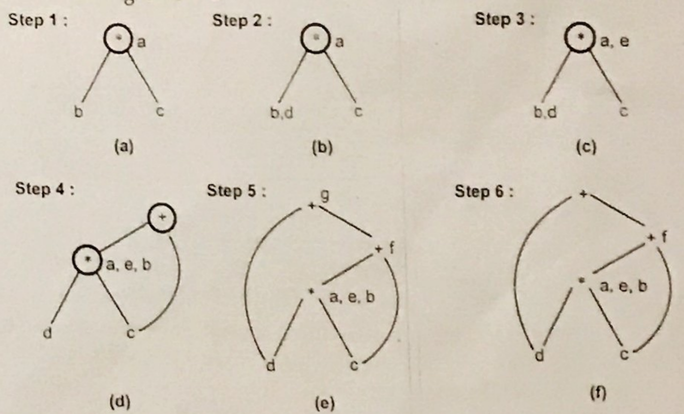


Fig. 9.12 Construction of DAG

The optimized code can be generated by traversing the DAG.

The local optimization on the above block can be done.

1. A common sub expression e=d*c which is actually b * c(since d:=b) is eliminated.
2. A dead code for b=e is eliminated.

The optimized code and basic block is -

$a := b * c$

$d := b$

$f := a + c$

$g := f + d$

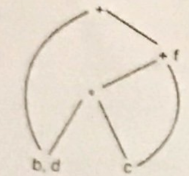
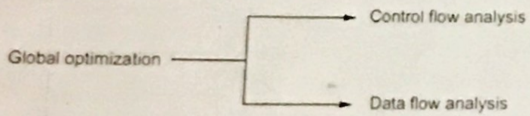


Fig. 9.13 Optimized code and DAG

9.7 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 flow graph. The program flow graph 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 : control flow analysis and data flow analysis.



9.7.1 Control and Data Flow Analysis

The control flow analysis determines the information regarding **arrangement of graph nodes** (basic blocks), **presence of loops, nesting of loops** and **nodes visited before** execution of a specific node. Using this analysis optimization can be performed.

Thus in control flow analysis the analysis is made on the **flow of control** by carefully examining the program flow graph.

In data flow analysis the analysis is made on the **data flow**. That is the data flow analysis determines the information regarding the definition and use of the data in the program. Using this kind of analysis optimization can be done. The data flow analysis is basically a process in which the values are computed using data flow properties.

Example 9.2 : *Consider following program fragment*

```
if  (i>=0)
{
     sum = B[0];
     i=0;
L1:  if (A[i]<B[i])
     {
          j=i;
L:2  :
     if(B[i]>=0)
     {
          sum=sum+B[j];
     }
     j=j+1;
     if(j<N) goto L2;
     }
     i=i + 1;
     }
     i=i + 1;
     if (i < N) goto L1;
     }
     printf ("sum = %d\n", sum);
```

Draw a control flow graph for this code. Show the basic blocks clearly in your control flow graph.

Solution :

Fig. 9.14 (See Fig. on next page)

Example 9.3 : *Build the control flow graph for following code*

```
L:   a : = k + 2
     c : = d - b
     d : = a + b
     if (d>i) goto E
     f : = b - d
     k : = d - 2
     b : = a + f
     if(d<i) goto B
     d : = b*2
     g : = 2*d
```

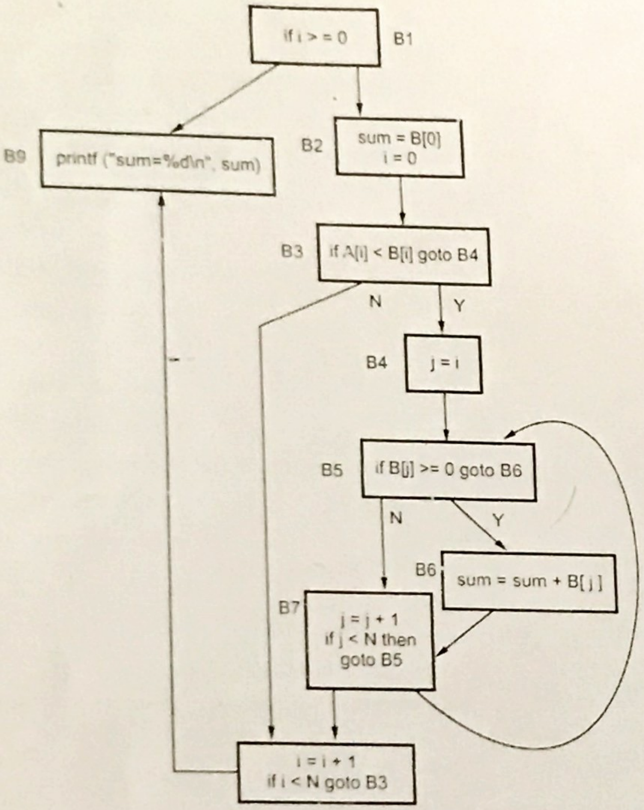


Fig. 9.14

```
B:  i : = i + 1
    b : = d - 1
    goto L
E:  k : = a - e
    f : = e + k
    c : = d + b
```


Solution :

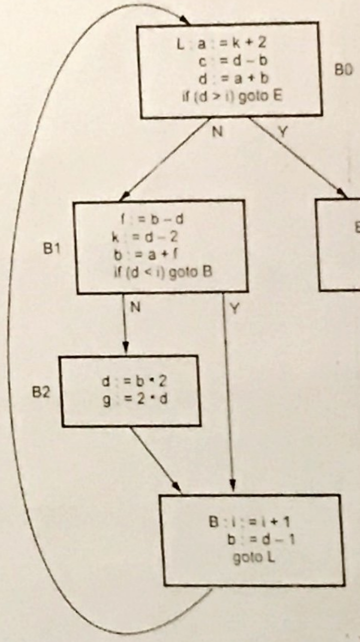


Fig. 9.15

➠ **Example 9.4 :** *Apply dead code elimination to the input code given in (CFG) control flow graph.*

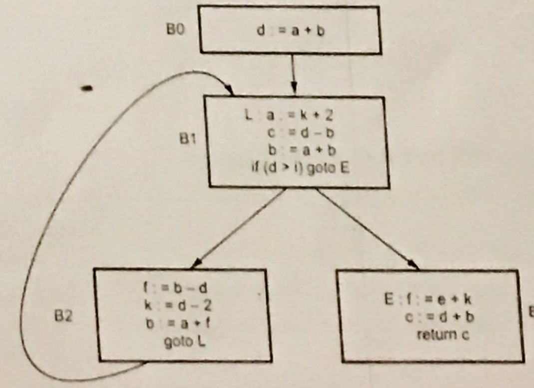


Fig. 9.16(a)

Solution : In the given flow graph various expressions that we come across are -

a + b, k + 2, d – b, b – d, d – 2, a + f, e + k, d + b

In block B1 we get c : = d – b but c is not used further. In block B3 we use c but in that block we freshly obtain c : = d + b. Hence we can eliminate a dead code c : = d – b. Similarly in block B3, computing of f : = e + k has no meaning because there is not link from B3, further to any other block and value of 'f' is not used in B3 at all. Hence we can eliminate f : = e + k. Finally after eliminating the dead code we get a flow graph as shown in Fig. 9.16 (b)

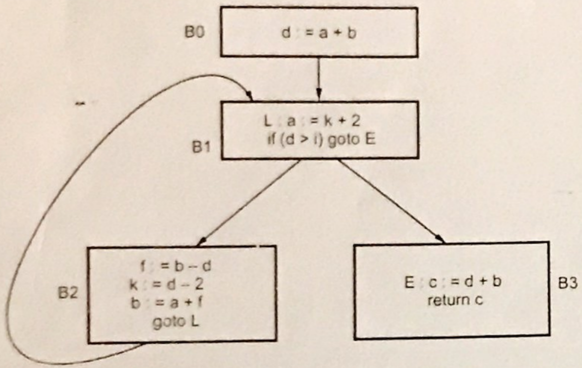


Fig. 9.16 (b)

9.8 Computing Global Data Flow Information

9.8.1 Data Flow Analysis

The data flow property represents the certain information regarding usefulness of the data items for the purpose of optimization. These data flow properties are -

1. Available expressions,
2. Reaching definitions,
3. Live variables,
4. busy expressions.

The data flow analysis is a process of computing values of data flow properties.

In this section we will discuss some of the data flow properties in detail.

9.8.2 Data Flow Properties

Before discussing the data flow properties consider some basic terminologies that will be used while giving the data flow property.

- 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**.

For example :

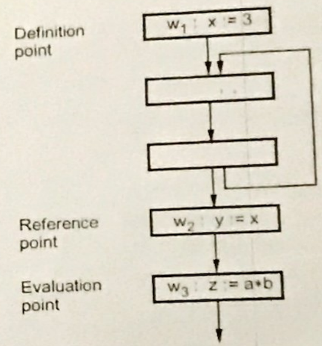


Fig. 9.17 Program points

1 Available expression

An expression x+y is available at a program point w if and only if along all paths reaching to w.

1. The expression x+y is said to be available at its evaluation point.
2. The expression x+y is said to be available if no definition of any operand of expression (here either x or y) follows its last evaluation along the path. In other word, if neither of the two operands get modified before their use.

For example : Fig. 9.18 see on next page.

The expression 4*i is the available expression 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.

Advantage of available expression

- The use of available expression is to eliminate common sub expressions.