UNIT - V

Machine - Independent optimizations :- The principal sources of optimization, Introduction of Data-flow Analysis, foundations of Data-flow Analysis, constant propagation, partial Redundancy Elimination, Loops in flow Graphs.

Optimization is the process of transformation of code to an efficient code. Efficiency is in terms of space requirement and time for its execution without changing the meaning of the given code.

The following constraints are to be considered while applying the techniques for code improvement.

(1) The transformation must preserve the meaning of the program, that is, the target code should ensure semantic equivalence with source program.

(2) program efficiency must be improved by a measurable amount without changing the algorithm used in the program.

(3) when the technique is applied on a special format, then it is worth transforming.

Optimization can be classified as

1) Local optimizations :-

optimizations performed within a single basic block are termed as local optimizations. These techniques are simple to implement and does not require any analysis since we do not require any information relating to how data and control flows.

(2) Global optimizations :-

optimizations performed across basic blocks is called global optimizations. These techniques are complex as it requires additional analysis to be performed across basic blocks. This analysis is called data-flow analysis.

Optimization techniques can be applied to intermediate code or on the final target code. It is a complex and a time-consuming task that involves multiple sub phases, sometimes applied more than once.

* most compilers allow the optimization to be turned off to speed up compilation process.

* To apply optimization it is important to do control flow analysis and data flow analysis followed by transformations.

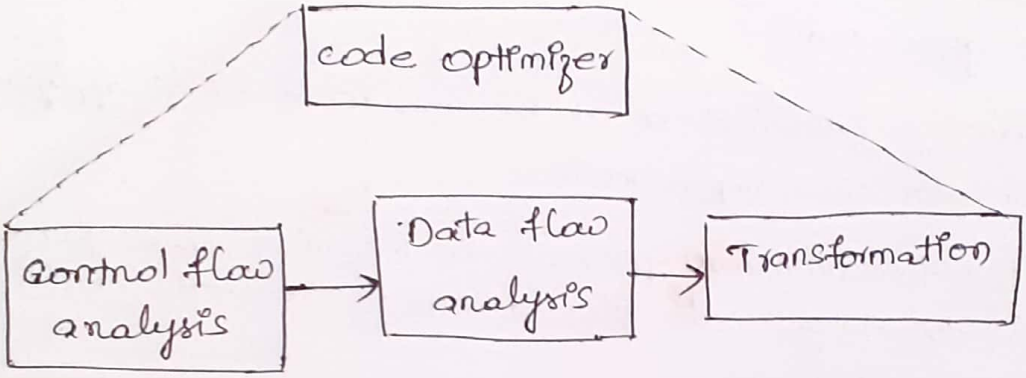


Fig: code optimization model.

Control Flow Analysis:- It determines the control structure of a program and builds a control flow graph.

Data flow Analysis:- It determines the flow of scalar values and builds data flow graphs. The solution to flow analysis propagates data flow information along a flow graph.

Transformation:- Transformations help in improving the code without changing the meaning or functionality.

Flow Graph:-

A graphical representation of three address code is called flow graph. The node in the flow graph represent a single basic block and the edges represent the flow of control.

* These flow graphs are useful in performing the control flow and data flow analysis and to apply local or global optimization and.

code generation.

Basic Block:- A basic block is a set of ~~conseq~~ consecutive statements that are executed sequentially. once the control enters into the block then every statement in the basic block is executed one after the other before leaving the block.

Ex: for the statement a = b+c * d/e the corresponding a set of three address code is.

$t_1 = c * d$
$t_2 = t_1 / e$
$t_3 = b + t_2$
$a = t_3$.

All these statements correspond to a single basic block.

Procedure to identify the Basic Blocks.

Given a three address code, first identify the leader statements and group the leader statements with the statements up to the next leader.

* To identify the leader use the following rules:

1. First statement in the program is a leader.
2. Any statement that is the target of a conditional or unconditional statement is a leader statement.
3. Any statement that immediately follows a conditional/unconditional statement is a leader statement.

Example:- Identify the basic blocks for the following code fragment

```
main()
{ int i=0; n=10;
  int a[n];
  while (i<=(n-1))
  { a[i] = i * i;
    i=i+1; } return; }.
```

The three address code for initialize function is as follows:

(1) i := 0 ⟶ leader1 using rule 1
(2) n := 10
(3) t1 := n-1 ⟶ leader2 using rule 2
(4) if i > t1 goto (12)
(5) t2 := i*i ⟶ leader 3 using rule 3.
(6) t3 := 4*i
(7) t4 := a[t3]
(8) t4 := t2
(9) t5 := i+1
(10) i := t5
(11) goto (3)
(12) return. ⟶ leader 4. using rule 3.

1) i=0 2) n=10	B1	Basic Block1 includes statements (1) and (2)
(3) t1 := n-1 (4) if i > t1 goto (12	B2.	Basic block2 includes statements (3) and (4)
(5) t2 = i*i (6) t3 = 4*i (7) t4 = a[t3] (8) t4 = t2 (9) t5 = i+1 (10) i := t5 (11) goto (3)	B3.	Basic block3 includes statements (5) - (11)
(12) return.	B4.	Basic block4 includes statemet (12)

Fig: Basic Blocks.

Flow Graph :-

Flow graph shows the relation between the basic block and its preceeding and its successor blocks. The block with the first statement is B1. An edge is placed from block B1 to B2, if block B2 could immediately follow B1 during execution or satisfies the following conditions.

→ The last statement in B1 is either conditional or unconditional jump statement that is followed by the first statement in B2 or

→ The first statement in B2 follows the last statement in B1 and is not an unconditional / conditional jump statement

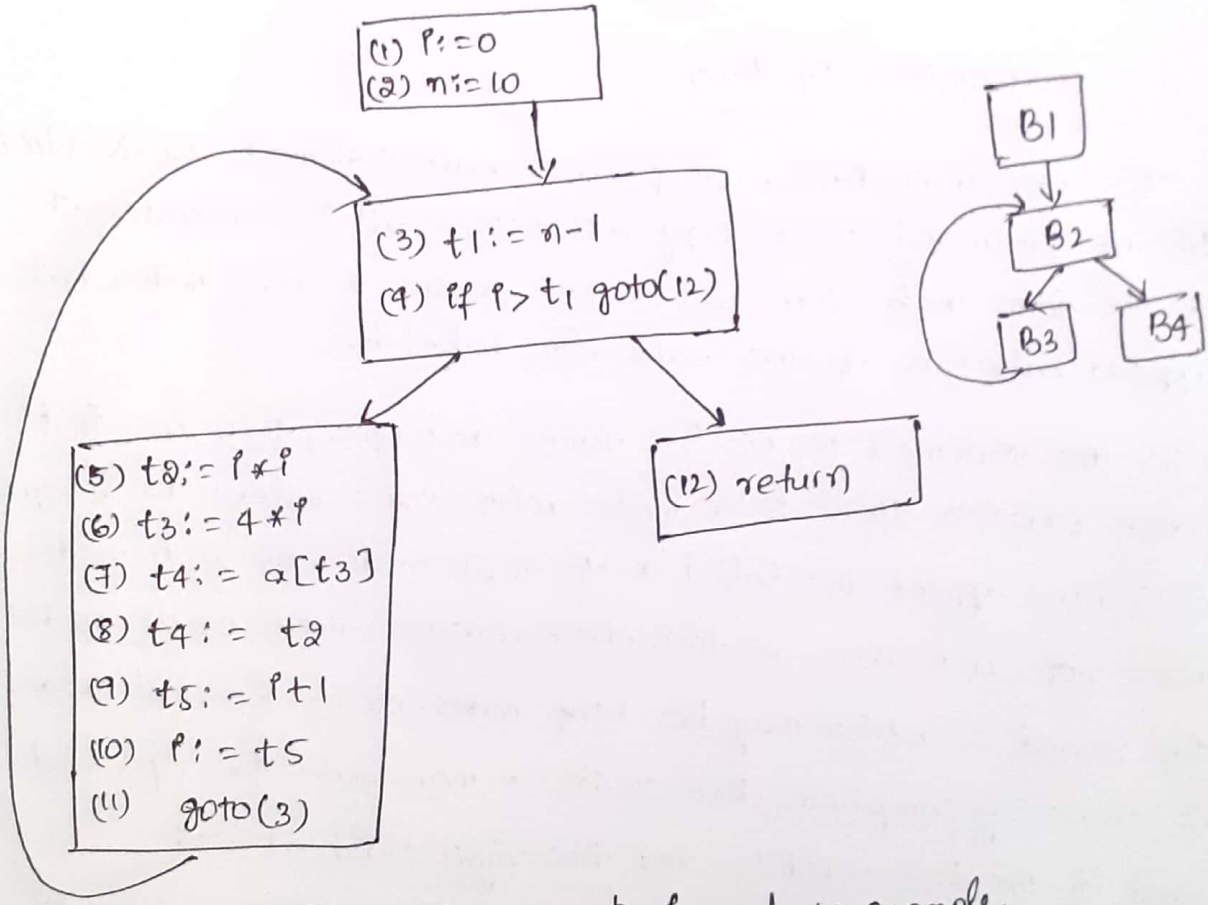


fig: Flow graph for above example.

DAG representation of Basic Block.

A DAG is a useful data structure for implementing transformations within a basic block. It gives pictorial representation of how values computed at one statement are useful in computing the values of other variables.

* It is useful in identifying common sub-expressions within a basic block.

* A DAG has nodes, which are labeled as follows.

(1) The leaf nodes are labeled by either identifiers or constants. If the operators are arithematic then it always requires the r-value

(2) The labels of interior nodes correspond to the operator symbol.

Note: The DAG is not the same as a flow graph. Each node in a flow graph is a basic block, which has a set of statements that can be represented using DAG.

=

Construction of DAG.

To construct DAG, we process each statement of the block. If the statement is a copy statement, that is, a statement of the form $a=b$, then we do not create a new node, but append label a to the node with label b.

* If the statement is of the form $a=b$ op c, then we first check whether there is a node with same values as b op c, if so, we append the label a to that node. If such node does not exist then we first check whether there exists nodes for b and c which may be leaf nodes or an internal nodes if recently computed, then create a new node for 'op' and add it the left child b and the right child as 'c'.

* Label this new node with a. This would become the value of a to be used for next statements; hence, we mark the previously marked nodes with a as a_0.

=

Example:- Let us consider Basic block B1 to B4 and construct DAG for each block step by step

B1
(1) i := 0
(2) n := 10

B2
(3) t1 := n-1
(4) If i > t1 goto (5)

B3
(5) t8 := i * i
(6) t3 := 4 * i
(7) t4 := a[t3]
(8) t9 := t2
(9) t5 := i+1
(10) i := t5
(11) goto (3)

B4
(12) return.

For Block B1

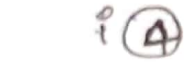


Fig: DAG for Block B1

* For the statement in B1, first we create a leaf labeled 4 and attach identifier i to it

For Block B2

For first statement, which we first create nodes for n and 1, then create node for operator (-) and label it as t1.

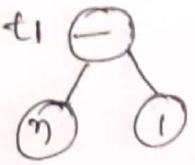
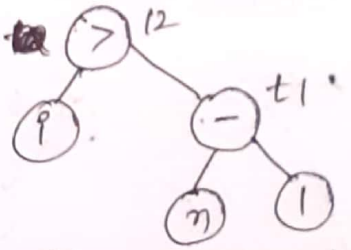


Fig (a)
DAG for first statement in Block B2



Fig(b): DAG for Block B2.

* fig(b) shows the, second statement, which is a conditional statement, we create a node for operator > and label it as 12 as when this condition is satisfied it should go to statement 12.

For Block B3

* For the first statement, we create nodes for i and use as right child, then create node for operator (*) and label it as t2.

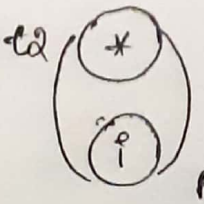


Fig: (a) DAG for Block B3.

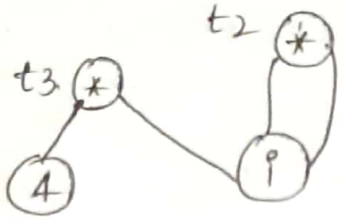
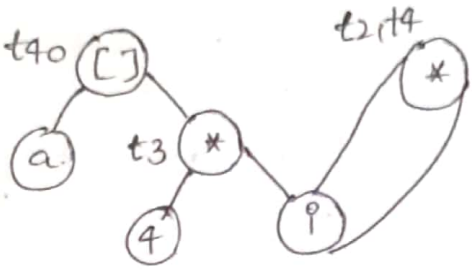
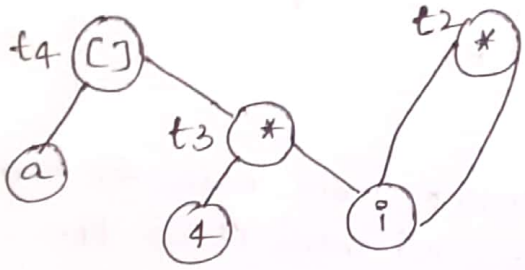


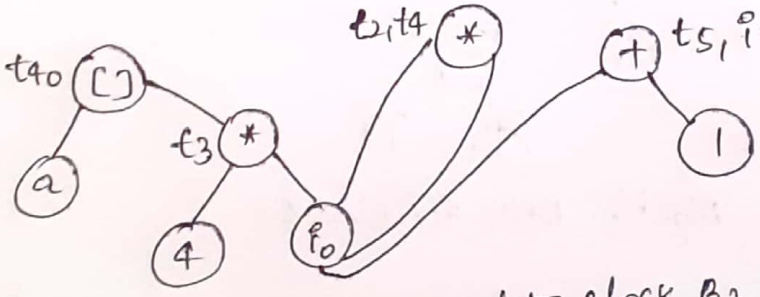
Fig: DAG for two statements in Block B3.



Fig(c): DAG for three statements in Block B3.



Fig(d): DAG for four statements in Block B3.



Fig(e): DAG for complete Block B3.

Principle sources of optimization.

A transformation of a program is called local if it is applied within a basic block and global if applied across basic blocks.

* there are different types of transformations to improve the code and these transformations depend on the kind of optimization required.

(1) Function – preserving transformations are those transformations that are performed without changing the function it computes.

* these are primarily used when global optimizations are performed.

(2) Structure – preserving transformations are those that are performed without changing the set of expressions computed by the block.

* many of these transformations are applied locally.

(3) Algebraic transformations are used to simplify the computation of expression set using algebraic identities.

* These can replace expensive operations by cheaper ones, for instance multiplication by 2 can be replaced by left shift.

Function preserving Transformations.

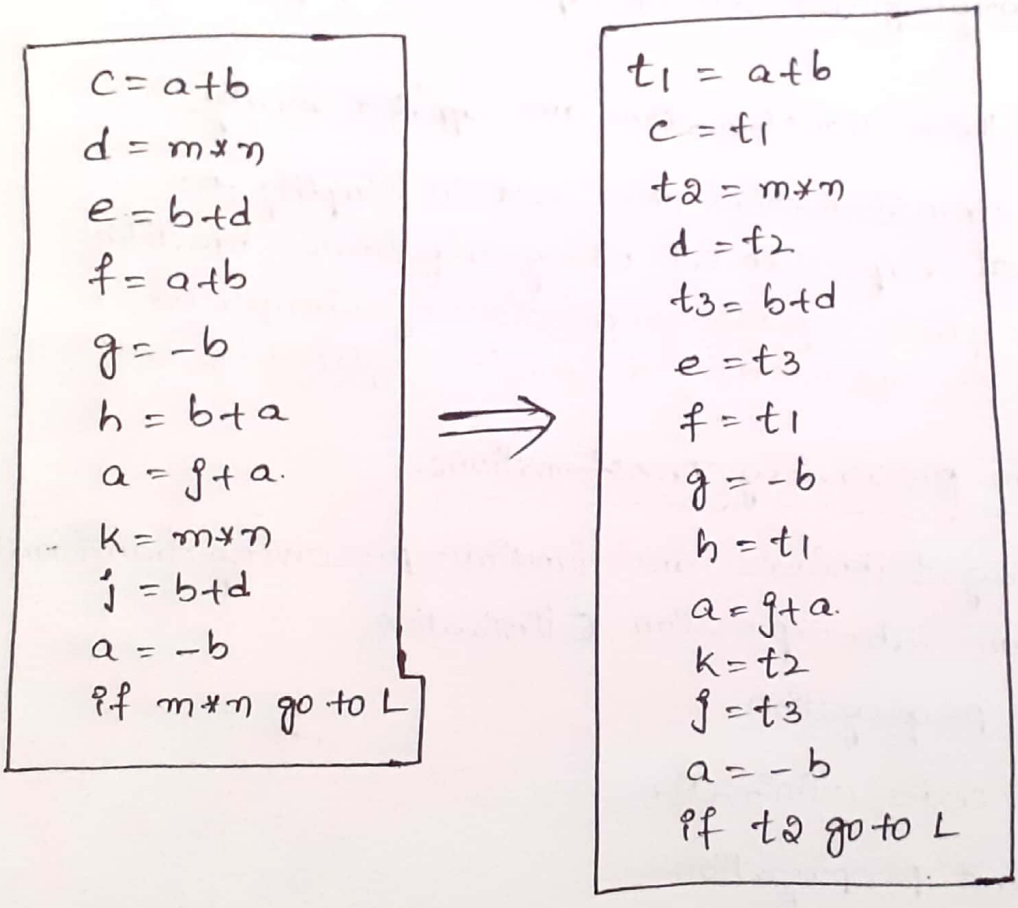
The following techniques are function-preserving transformations

1) common sub-expression Elimination
2) copy propagation.
3) Dead code elimination
4) constant propagation.

(1) common sub-expression Elimination :-

An expression E is said to be common sub expression if E is computed before and the variables in the expression are not modified since its ~~comp~~ computation If such expression is present then the re-computation can be avoided by using the result of the previous computation

* This technique can be applied both locally and globally
* we need to maintain a table to store the details of expressions evaluated so far and use this information to identify and eliminate the re-computation of the same expression.
* The common sub-expression elimination can be done within basic block by analyzing and storing the information of expression in the table until the operands in the expression are redefined
* If any operand in the expression is redefined, then remove the expression from the table



program code before and after common sub expression elimination

* The above figure shows the optimized code on applying common sub expression elimination technique locally.

=

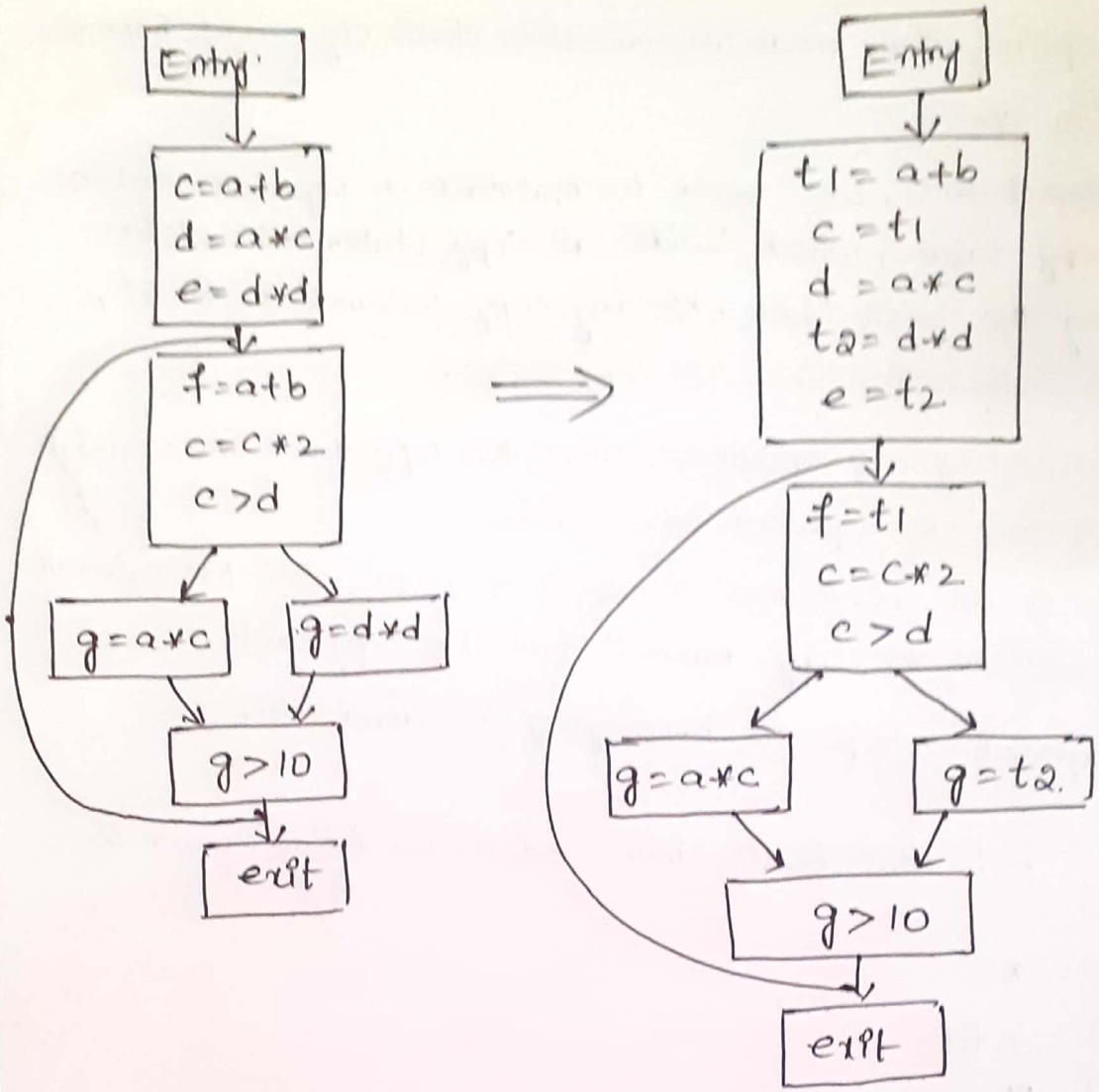


Fig: Example for global common sub Expression Elimination.

* The above example shows the common-sub expression elimination globally by replacing the evaluated expression with a new temporary variable t1 and t2 and these variables are used wherever the same expression is used.

=

(ii) Copy propagation :-

A copy statement is a statement that is in the form a=b.

* copy propagation technique is applied to replace the later use of variable 'a' with the use of 'b' if original values of 'a' do not change after this assignment.
* This technique can be applied locally and globally.

* This optimization reduces runtime stack size and improves execution speed.

* To implement this, we need to maintain a separate table called copy table, which holds all copy statements. While traversing the basic block, if any copy statement is found, add this information into the copy table.

* While processing any statement, check the copy table for variable a, which can be replaced variable b.

* If there is an assignment statement that computes the value of a then remove the copy statement from the copy table.

* Copy propagation helps in identifying the dead code.

Example:- Let the basic block contain the following set of statements.

$Y = X$
$Z = Y + 1$
$W = Y$
$Y = W + Z$
$Y = W$

Statement NO	Instruction	Updated Instruction	copy table content
1.	$Y = X$	$Y = X$	$\{(Y, X)\}$
2.	$Z = Y + 1$	$Z = X + 1$	$\{(Y, X)\}$
3.	$W = Y$	$W = X$	$\{(Y, X), (W, X)\}$
4.	$Y = W + Z$	$Y = X + Z$	$\{(W, X)\}$
5.	$Y = W$	$Y = W$	$\{(W, X), (Y, X)\}$.

* On the first statement 1, since it is a copy statement, the information is added in to the copy table.

* Statement 2 uses the value x indirectly from Y, hence GET(Y, table) could return x and the statement is modified

replacing Y with X.

* The third statement is a copy statement and since Y is to replace X, we insert into the copy table. 'x' to be replaced by X
* When statement 4 is processed, Y value is defined. and Hence, details of Y are removed from the copy table.
* The fifth statement is a copy statement and is inserted into the copy table.

(iii) Dead code Elimination:-

Code that is unreachable or does not affect the program is said to be dead code. Such code requires unnecessary cpu time, which can be identified and eliminated using this technique.

Example program:-

```
1)  Int Var1;
2)  Void Sample()
3)  {
4)      Int i;
5)      i = 1;    /* dead store */
6)      Var1 = 1; /* dead store */
7)      Var2 = 2;
8)      return;
9)      Var1 = 3; /* unreachable */
10) }
```

The code fragment after dead code elimination

```
Int Var1;
Void Sample()
{
    Var1 = 2;
    return;
}
```

→ Here the value of i is never used, and the value assigned to Var1 variable in statement 6 is dead store and the statement 9 is unreachable. These statements can be eliminated as they do not affect the program execution.

(iv) Constant propagation :-

Constant propagation is an approach that propagates the constant value assigned to a variable at the place of its use.

For example, given an assignment statement $x=c$, where c is a constant, replace later uses of x with uses of c, provided there are no intervening assignments to x.

* This approach is similar to copy propagation and is applied at first stage of optimization. This method can analyze by propagating constant value in conditional statement, to determine whether a branch should be executed or not, that is, identifies the dead code.

Example: Let us consider the following example.

```
pi = 22/7
void area_per(int r)
{
    float area, perimeter;
    area = pi*r*r;
    perimeter = 2*pi*r;
    print area, perimeter;
}.
```

In this example, we can notice some simple constant propagation results, which are as follows.

→ In line 1 the variable pi is constant and this value is the result of 22/7, which can be computed at compile time and has the value 3.413.

→ In line 5 the variable pi can be replaced with the constant value 3.413.

→ In line 6 since the value of pi is constant, the partial result of the statement can be computed and the statement can be modified as perimeter = 6.285*r;

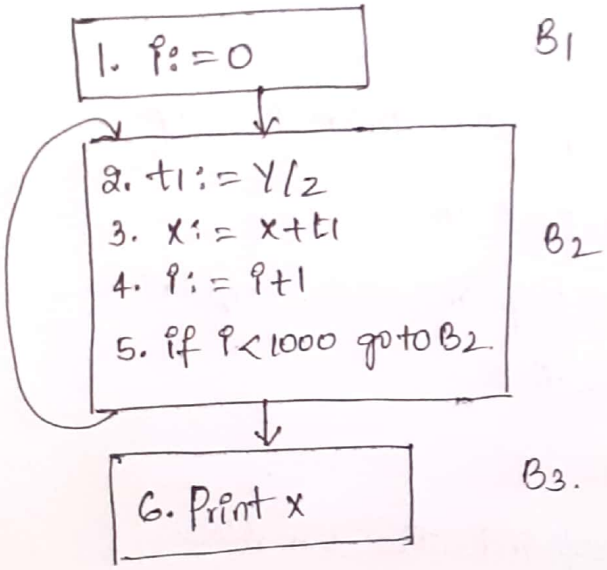
Loop optimization Techniques.

In most of the programs, 90% of execution time is inside the loops; hence loop optimization is the most valuable machine - independent optimization and are good candidates for code improvement.

Let us consider the example of a program that has a loop statement

```
for (int i=0; i<1000; i++)
    x = x+y/z;
print x;
```

The optimized code of this program is represented on flow graph. as



Flow graph After code optimization.

The total number of statements that are executed are as follows.

Statement Number	Frequency of Execution	Total No. of Statements
1	1	1
2-5	1000	4000
6	1	1

on the code, the number of statements that are executed are 4002. Instead if the code is written as follows, then the total number of statements that are executed are only 1002.

```
1.   t1 = Y/2
2.   x = x+t1
3.   x = x+t1
4.   x = x+t1
5.   ---
6.   ---
1001. x = x+t1
1002. print x.
```

The execution is three times better for later than the first form, but the space requirements is more. Inefficient code is generated in the loops for various reasons like

→ Induction variable usage to keep track of iteration
→ unnecessary computations made inside the iterative loops that are not effective.
→ use of high strength operators inside the loop.

The important loop optimizations are

① Elimination of loop invariant computations.
② Strength reduction
③ code motion
④ Elimination of induction variables.

=

(1) A Loop Invariant computation :—

A loop invariant computation is one that evaluates the same value every time the statements in loop are executed. * moving such loops out/side computational statements outside. the loop leads to a reduction in the execution time.

Algorithm for elimination of loop invariant code:

Step1: Identify loop-invariant code.

Step2: An instruction is loop-invariant if, for each operand:

(i) the operand is constant; OR

(ii) All definitions for all the operands that reach the instruction inside the loop are made outside the loop, OR

(iii) there is exactly one definition made using loop invariant variables inside the loop for an operand that reaches the instruction.

Step3:- move it outside the loop

① move each instruction i to newly created preheader if and only if it satisfies these requirements of the loop.

Example:

```
for (int i=0; i<1000; i++)
    x = x + y/z;
Print x;
```

In the above example, the value y and z remains unchanged as there is no definition for these variables in the loop. Hence, the computation of y/z remains unchanged, which can be moved above the loop and the same code can be rewritten as follows.

```
t1 = y/z;
for (int i=0; i<1000; i++)
    x = x + t1;
Print x;
```

(2) Induction Variables :—

Induction variables are those variables used in a loop, their values are in lock-step, and hence, it may be possible to eliminate all except one.

* There are two types of induction variables — basic and derived.

* Basic induction variables are those that are of the form.

$$I = I \pm C$$

where I is loop variable and C is some constant.

* Derived induction variables are those that are defined only once in the loop and their value is linear function of the basic induction variable.

for example :—

$$J = A * I + B$$

Here J is a variable that is dependent on the basic induction variable I and the constants A and B. This is represented as a triplet (I, A, B)

* Induction variable elimination involves three steps.

1) Detecting induction variable.

2) Reducing the strength of induction variable.

3) Eliminating induction variable.

Induction Variable Elimination :—

Some loops contain two or more induction variables that can be combined into one induction variable.

Example:- The code fragment below has three induction variables (i1, i2 and i3) that can be replaced with one induction variable, thus eliminating two induction variables.

```
int a[SIZE];
int b[SIZE];
void f(void)
{
  int i1, i2, i3;
  for (i1=0, i2=0, i3=0; i1 < SIZE; i1++)
      a[i2++] = b[i3++];
  return;
}
```

After induction variable elimination the code is rewritten as

```
int a[SIZE]
int b[SIZE]
void f(void)
{
  int i1;
  for (i1=0; i1 < SIZE; i1++)
      a[i1] = b[i1];
  return;
}
```

* Induction variable elimination can reduce the number of additions (or subtractions) in a loop, and improve both runtime performance and code space.

Machine - Dependent optimization.

This optimization can be applied on target machine instructions. This includes register allocation, use of addressing modes, and peep hole optimization.

* Instructions involving register operands are faster and shorter; hence, if we make use of more registers during target code generation, efficient code will be generated.

Peephole optimization :-

Generally code generation algorithms produce code, statement by statement. this may contain redundant instructions and suboptimal constructs. The efficiency of such code can be improved by applying peephole optimization, which is simple but effective optimization on target code.

* The peephole is considered a small moving window on the target code. The code in peephole need not be contiguous. It improves the performance of the target program by examining and transforming a short sequence of target instructions.
* The advantage of peephole optimization is that each improvement applied increas opportunities and shows additional improvements. It may need repeated passes to be applied over the target code to get the maximum benefit.

=

(i) Redundant loads and stores

The code generation algorithm produces the target code, which is either represented with single operand or two operands or three operands.

* Let us assume the instructions are with two operands. The following is an example that gives the assembly code for the statement

$x = y + z$.

```
1. MOV y, R0
2. ADD z, R0
3. MOV R0, x
```

Instruction 1 moves the value of y to Register R_0, second instruction performs the addition of value in z with the register content and the result of the operation is stored in the register.

* The third instruction copies the register content to the location x. At this point the value of x is available in both location of x and the register R_0.

If the above algorithm is applied on the code a=b+c, d=a+e then it generates the code given below.

1. MOV b, R0
2. ADD c, R0
3. mov R0, a
4. mov a, R0
5. ADD e, R0.
6. mov R0, d.

Here we can say that 3 and 4 are redundant load and store instructions. These instructions will not affect the values before or after their execution. Such redundant statements can be eliminated and the resultant code is as follows.

1. mov b, R0
2. ADD c, R0.
3. ADD e, R0
4. mov R0, d.

(ii) Algebraic simplification :—

There are few algebraic identities that occur frequently enough and are worth considering

Look at the following statements.

x := x+0
x := x+1

They do not alter the value of x. If we keep them as it is, later when code generation algorithm is applied on it, it may produce sil statements that are of no use. Hence, such statements whether they are in three address code or target code can be removed.

(iii) Dead code Elimination :—

Removal of unreachable code is an opportunity for peephole optimization. A statement immediately after an

unconditional jump or a statement that never get a chance to be executed can be identified and eliminated. Such code is called the dead code.

Eg: #define x=0
...
if(x)
{
... print value.
}

If this is translated to target code as

If x=1 goto L1
goto L2
L1: print value
L2: ....

here, value will be never printed. So whatever code inside the body of "if(x)" is dead code; hence it can be removed.

(iv) Reduction in strength:-

This optimization mainly deals with replacing expensive operations by cheaper ones for example

→ $x^2 \Rightarrow x*x$.
→ fixed-point multiplication and division by a power of 2.
⇒ Shift
→ Floating point division by a constant ⇒ floating-point multiplication by a constant.

=