11 CODE OPTIMIZATION

11.1 INTRODUCTION

Code Optimization is the optional phase of a compiler which can be applied after Intermediate code generation.

Definition :- It refers to the techniques used by compiler to improve the execution efficiency of the object code.

→ It involves finding out patterns in a program & replacing these patterns by equivalent but more efficient constructs.

→ Increase in efficiency is concerned with size and running time of object program.

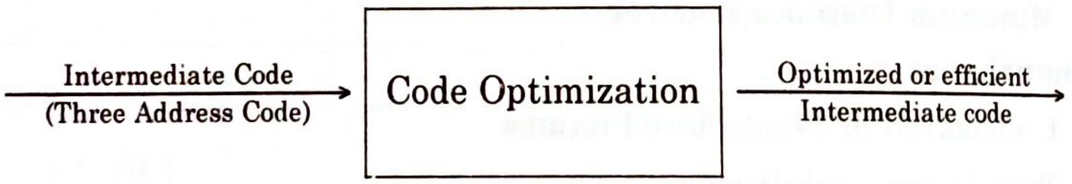


Fig 11.1 : Code Optimization

→ Code optimization is nothing, but actually a code improvement.

Following are the creterias which should be taken care of while optimizing the code :-

1. Code optimization should reduce execution time or space taken by object program.
2. Increase in program efficiency should not change the semantic analysis (meaning) of the program.
3. Algorithm should not be modified in any sense.

1.1 CLASSIFICATION OF CODE OPTIMIZATION

Broadly, code optimization can be classified as :–

(a) Machine Dependent

(b) Machine Independent

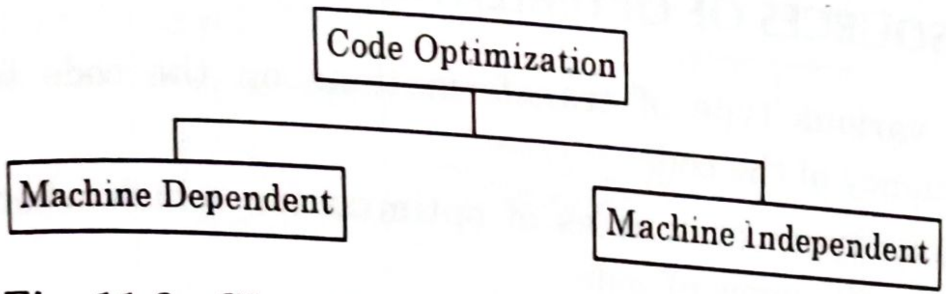


Fig. 11.2 : Classification of Code Optimization

(a) **Machine Dependent Optimization :–** Machine dependent optimization requires knowledge of Target Machine. It depends on target machine for instruction set used & addressing modes used for instructions to produce efficient target code. For example :– **Peephole optimization** is type of machine dependent optimization.

Machine dependent optimization can be achieved using following reterias :–

1. Machine's Registers should be utilized efficiently.
2. Efficient utilization of instruction set of a Machine.
3. Allocation of sufficient number of resources to improve the execution efficiency of the program.
4. Immediate instructions should be used wherever required.

(b) **Machine Independent Optimization :–**

Machine Independent optimization can be performed independently of target achine for which compiler is generating code *i.e.* optimization is not dependent n Machine's platform.

Machine Independent optimization can be achieved using following reteria :–

1. By Elimination of common sub-expressions.
2. Eliminating unreachable code from the program.
3. Elimination of loop invariant computation *i.e.* eliminating code inside the loop which does not affect the execution of loop and keeping it outside the loop.
4. By Elimination of induction variable.
5. By improving the Algorithm structure *i.e.* algorithmic optimization.

11.1.2 ADVANTAGES OF CODE OPTIMIZATION

1. Program after optimization will occupy less memory space.
2. Program will run faster on performing optimization.
3. Produce better object language program.

11.2 PRINCIPAL SOURCES OF OPTIMIZATION

We can apply various type of transformations on the code to increase the performance or efficiency of the code.

There are various types ot sources of optimization which when implemented give rise to increase in efficiency of code.

Following are the principal sources of optimization :–

1. **Local Optimization**
   (*a*) Common Subexpression Elimination
   (*b*) Constant Folding
   (*c*) Dead Code Elimination
2. **Loop Optimization**
   (*a*) Code Motion
   (*b*) Induction Variable Elimination
   (*c*) Reduction in Strength
3. **Data Flow Analysis**
4. **Algorithmic Optimization**

1. **Local Optimization :–** These are optimizations or transformations which are applied with in a straight line.

Following are various transformations which come under local optimization.

(*a*) Common Sub expression Elimination :– An expression E is called common sub expression if it was previously computed. Here, we can avoid recomputing of the expression, if its value is already been computed.

For example :–

$$\boxed{\begin{array}{l} A = B + C + D \\ E = B + C + M \\ \\ \\ \end{array}} \xrightarrow[\text{Elimination}]{\text{Common Sub Expression}} \boxed{\begin{array}{l} T_1 = B + C \\ A = T_1 + D \\ E = T_1 + M \\ \\ \end{array}}$$

Here, $B + C$ is common subexpression.

(*b*) **Dead Code Elimination :–** The Dead code is useless code which can be removed from the Program.

For example :–

```
    ⋮
 a = 1
If (a = = 2)
  {
   a = a + 2 ;
  }
    ⋮
```

$\xrightarrow[\text{Elimination}]{\text{Dead Code}}$

```
a = 1 ;
  ⋮
  ⋮
  ⋮
```

Since, condition if $(a = = 2)$ will never be true as $a = 1$.

$\therefore$ If statement, is a dead code which never gets satisfied. Hence, it can be eliminated.

(*c*) **Constant Folding :–**

In this method of optimization, constant expressions are calculated during compilation. For example :–

```
const  a = 5 ;
 b = a + 2 ;
```

$\xrightarrow[\text{Folding}]{\text{Constant}}$

```
b = 7 ;
```

2. **Loop Optimization :–** Inner loops are the greatest source of optimization.

90 - 10 Rule → Inner loops in a program is a place where program spend large amount of time.

90 - 10 Rule states that 90% of the time is spent in 10% of the code.

Thus, inner loops are most heavily travelled part of program.

Therefore loops are the greatest source of optimization.

So, if number of instructions are less in Inner Loops. The running time will be less.

There are three methods to implement loop optimization :–

1. Code Motion
2. Elimination of Induction Variables
3. Reduction in Strength

(a) Code Motion → It is a technique which moves the code outside the loop. If expression lying in loop remains unchanged even after executing the loop for several times. Then, that expression can be placed outside the loop.

For example :–

```
a = 1, b = 2 ;
for (i = 1 ; i< = 10 ; i++ )
{
    x = a * b ;
    sum = sum + i ;
}
```

Code Motion →

```
a = 1, b = 2 ;
x = a * b ;
for (i = 1 ; i< = 10 ; i++ )
{
    sum = sum + i ;
}
```

As, keeping $x = a * b$ outside the loop will not make any change in result of execution. As x, a, b are independent from loop parameter i.

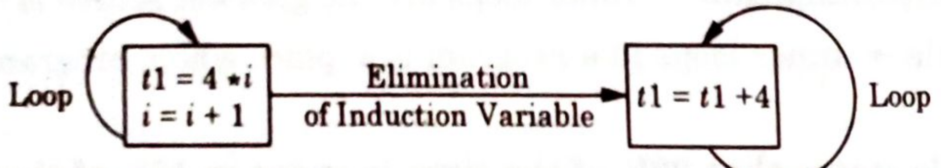
(b) Elimination of Induction Variables → This optimization will decrease the total number of instructions and will decrease the execution time of loop.

∴ x, a, b are loop invariant variables which do not very during execution of loop

∴ Code Motion is also Called Eliminating Loop Invariant Computation.

A variable i is called **induction variable** of a loop if every time it changes its value, it is incremented or decremented by some constant.

For example :–



Here, each increment in value of i lead to increment of $t1$ by 4.

∴ i and $t1$ are in locked state.

Hence i, $t1$ are inductions variable. Above, we have eliminated induction variable i.

(c) Reduction in Strength → Some operators have different strength. For e.g Strength of $*$ is higher than $+$. Higher strength operators can be replaced by lower strength operators.

For example :–

```
for (i =1 ;  i< = 5 ; i++ )
 {
   a = i * 2 ;
 }
```

$\xrightarrow{\text{Reduction in Strength}}$

```
int  a = 0 ;
for (i=1 ; i<=5 ;  i++)
   {
      a = a + 2 ;
   }
```

'+' operation is much cheaper than '*' operation which is costly.

3. **Algorithm Optimization →** Algorithm optimization is the important source f optimization in the running time of a program. We can't achieve optimization until and nless we don't have a good algorithm. **Good Algorithm** means having less time omplexity