PUNE VIDYARTHI GRIHA's

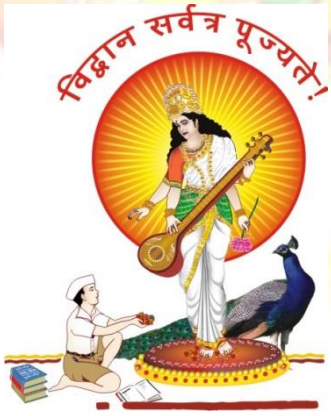
COLLEGE OF ENGINEERING

(Approved by AICTE, Accredited by NAAC, Affiliated to SPPU)

NASHIK – 04.

COMPILER NOTES

UNIT - VI



DEPARTMENT OF COMPUTER ENGINEERING

AY – 2018-19

UNIT - VI

Syllabus - Need for Optimization, local, global and loop optimization, Optimizing transformations, compile time evaluation, common sub-expression elimination, variable propagation, code movement, strength reduction, dead code elimination, DAG based local optimization, Introduction to global data flow analysis, Data flow equations and iterative data flow analysis.

CODE OPTIMIZATION

8.1 Need of Code Optimization :

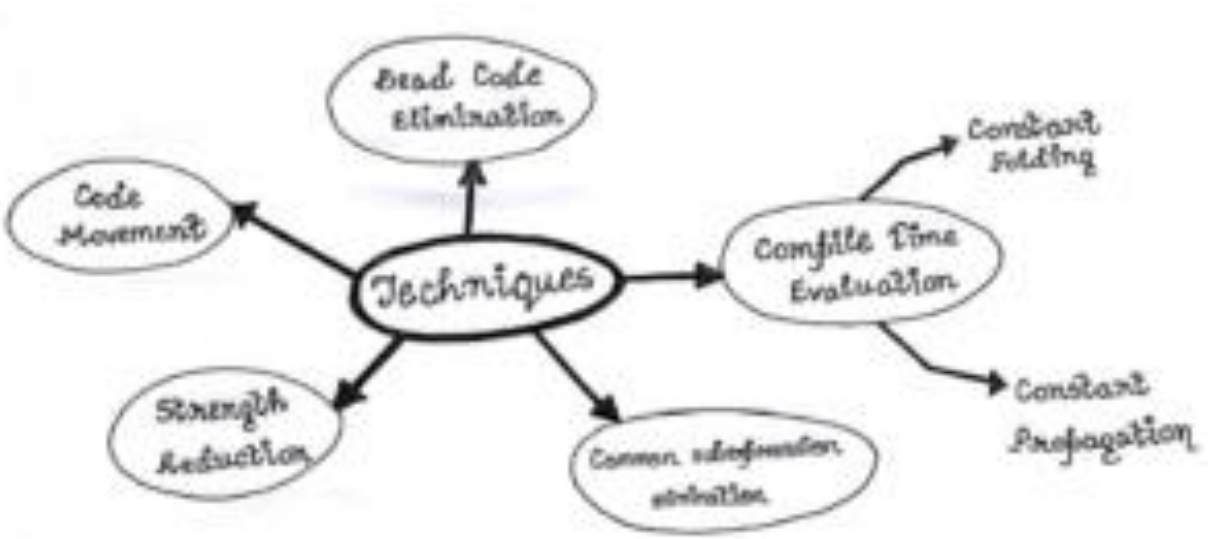
What is Code Optimization?

Code Optimization is an approach for enhancing the performance of the code by improving it through the elimination of unwanted code lines and by rearranging the statements of the code in a manner that increases the execution speed of the code without any wastage of the computer resources.

Advantages of Code Optimization-

- Optimized code has faster execution speed
- Optimized code utilizes the memory efficiently
- Optimized code gives better performance

8.2 Techniques for Code Optimization-



Important code optimization techniques

1. Compile Time Evaluation
2. Common sub-expression elimination
3. Dead Code Elimination
4. Code Movement
5. Strength Reduction

1. Compile Time Evaluation-

Two techniques that falls under compile time evaluation are-

A) Constant folding-

- As the name suggests, this technique involves folding the constants by evaluating the expressions that involves the operands having constant values at the compile time.
- **Example-**

Circumference of circle = (22/7) x Diameter

Here, this technique will evaluate the expression 22/7 and will replace it with its result 3.14 at the compile time which will save the time during the program execution.

B) Constant Propagation-

- In this technique, if some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program wherever it has been used during compilation, provided that its value does not get alter in between.
- **Example-**

pi = 3.14

radius = 10

Area of circle = pi x radius x radius

Here, this technique will substitute the value of the variables ‘pi’ and ‘radius’ at the compile time and then it will evaluate the expression 3.14 x 10 x 10 at the compile time which will save the time during the program execution.

2. Common sub-expression elimination-

The expression which has been already computed before and appears again and again in the code for computation is known as a common sub-expression.

As the name suggests, this technique involves eliminating the redundant expressions to avoid their computation again and again. The already computed result is used in the further program wherever its required.

Example-

Code before Optimization	**Code after Optimization**
S1 = 4 x i S2 = a[S1] S3 = 4 x j S4 = 4 x i // **Redundant Expression** S5 = n S6 = b[S4] + S5	S1 = 4 x i S2 = a[S1] S3 = 4 x j S5 = n S6 = b[S1] + S5

3. Code Movement-

As the name suggests, this technique involves the movement of the code where the code is moved out of the loop if it does not matter whether it is present inside the loop or it is present outside the loop.

Such a code unnecessarily gets executed again and again with each iteration of the loop, thus wasting the time during the program execution.

Example-

Code before Optimization	**Code after Optimization**

for (int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
	x = y + z ;
	for (int j = 0 ; j < n ; j ++)
	{
	a[j] = 6 x j;
	}

4. Dead code elimination-

As the name suggests, this technique involves eliminating the dead code where those statements from the code are eliminated which either never executes or are not reachable or even if they get execute, their output is never utilized.

Example-

Code before Optimization	**Code after Optimization**

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

5. Strength reduction-

As the name suggests, this technique involves reducing the strength of the expressions by replacing the expensive and costly operators with the simple and cheaper ones.

Example-

Code before Optimization	**Code after Optimization**
B = A x 2	B = A + A

Here, the expression “A x 2” has been replaced with the expression “A + A” because the cost of multiplication operator is higher than the cost of addition operator.

8.3 Optimization Transformation :

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
  item = 10;
  value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
  value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

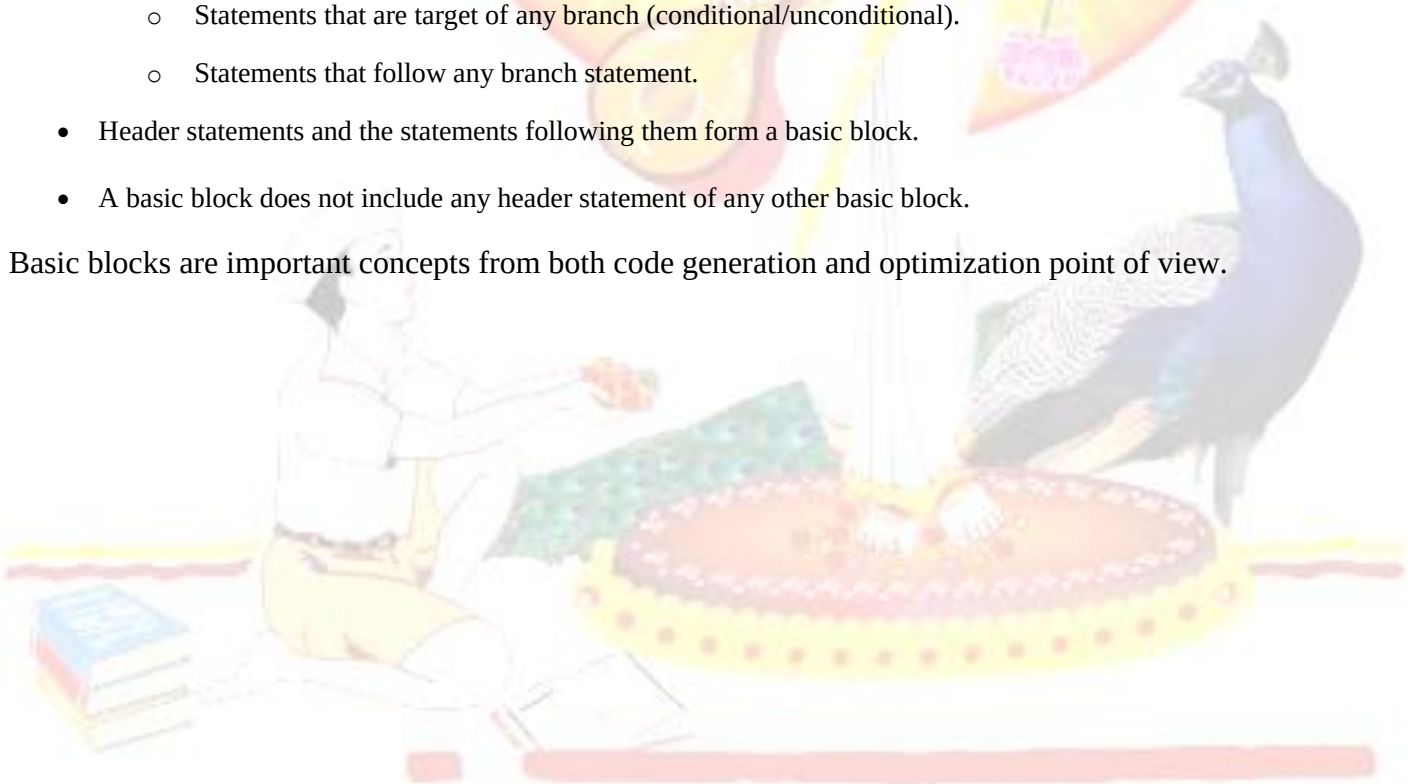
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

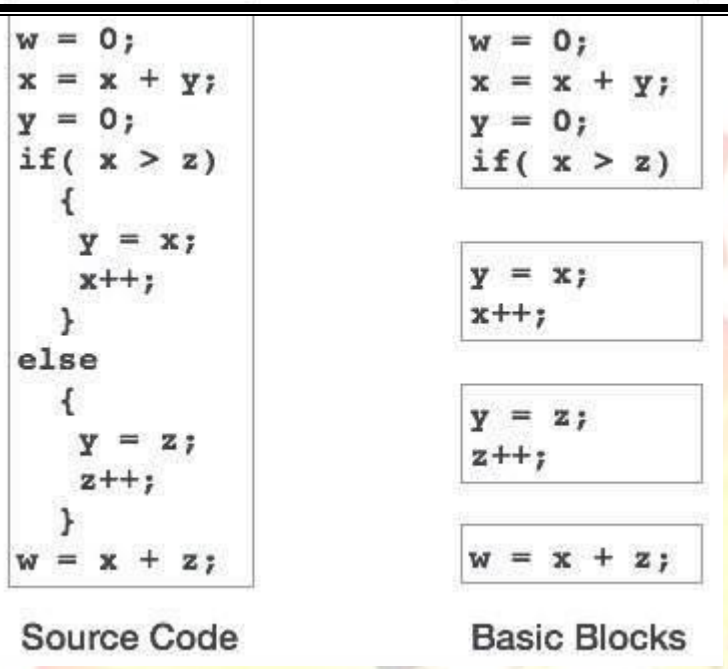
Basic block identification

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
    - First statement of a program.
    - Statements that are target of any branch (conditional/unconditional).
    - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.

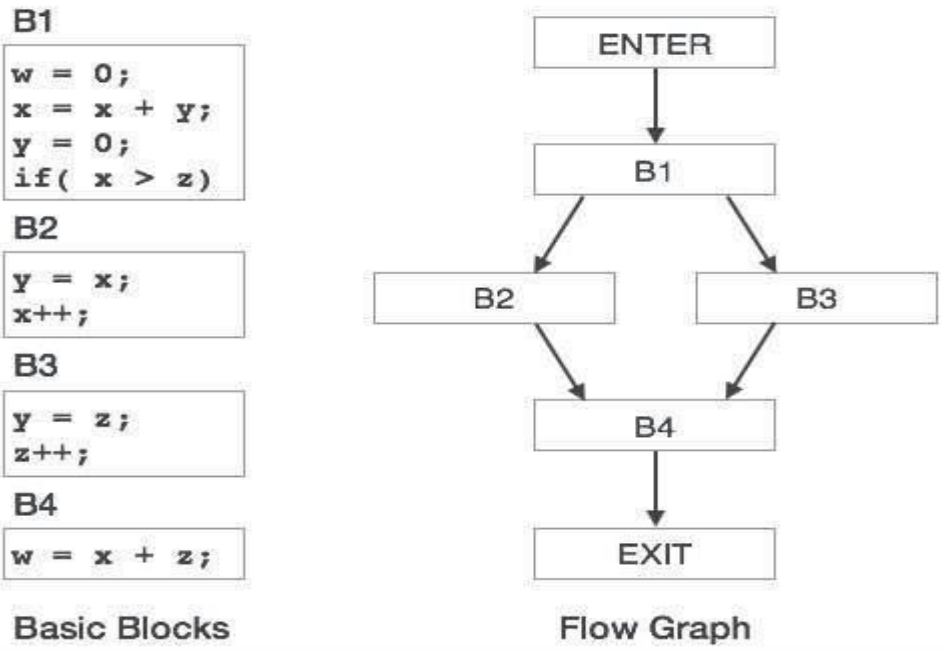




Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



8.4 Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- **Invariant code** : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
- **Induction analysis** : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- **Strength reduction** : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

8.5 Dead-code Elimination

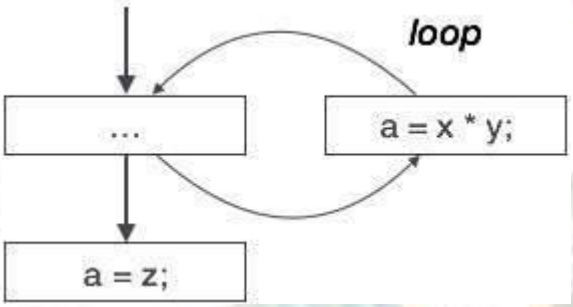
Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

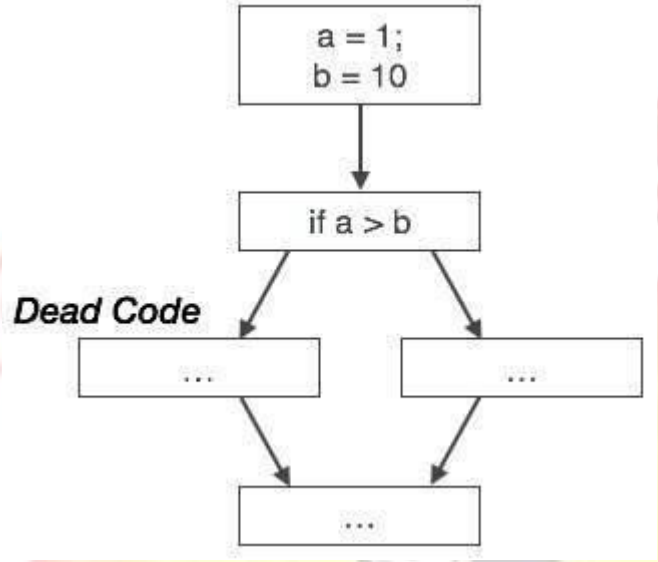
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code

There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.



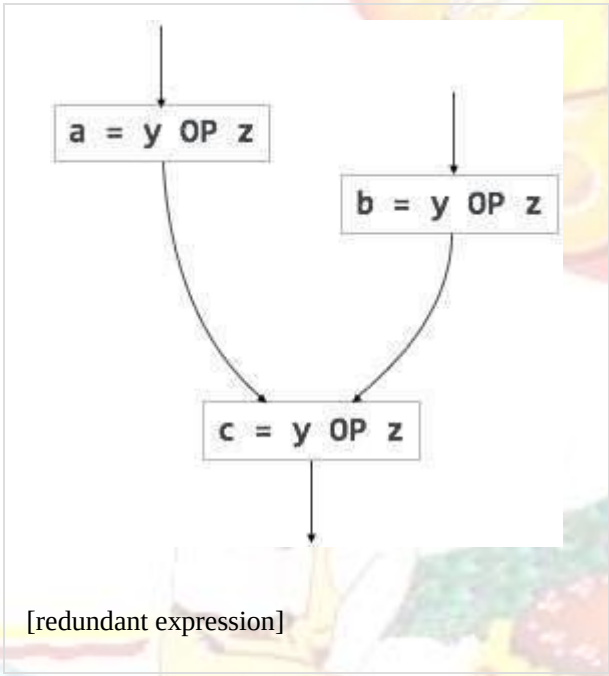
The above control flow graph depicts a chunk of program where variable ‘a’ is used to assign the output of expression ‘x * y’. Let us assume that the value assigned to ‘a’ is never used inside the loop.Immediately after the control leaves the loop, ‘a’ is assigned the value of variable ‘z’, which would be used later in the program. We conclude here that the assignment code of ‘a’ is never used anywhere, therefore it is eligible to be eliminated.



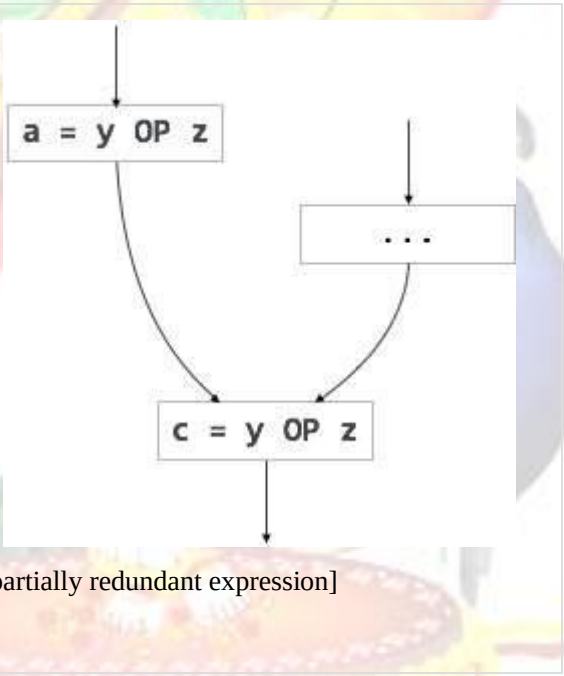
Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands.whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,



[redundant expression]



[partially redundant expression]

Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

```
If (condition)
{
  a = y OP z;
}
else
{
  ...
}
c = y OP z;
```

We assume that the values of operands (**y** and **z**) are not changed from assignment of variable **a** to variable **c**. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{
  ...
  tmp = y OP z;
  a = tmp;
  ...
}
else
{
  ...
  tmp = y OP z;
}
c = tmp;
```

Here, whether the condition is true or false; y OP z should be computed only once.

8.6 Types of Code Optimization :

- **Peephole optimizations**

  Usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like "looking through a peephole" at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions. For instance, a multiplication of a value by 2 might be more efficiently executed by left-shifting the value or by adding the value to itself (this example is also an instance of strength reduction).

- **Local optimizations**

  These only consider information local to a basic block.[1] Since basic blocks have no control flow, these optimizations need very little analysis (saving time and reducing storage requirements), but this also means that no information is preserved across jumps.

- **Global optimizations**

  These are also called "intraprocedural methods" and act on whole functions.[1] This gives them more information to work with but often makes expensive computations necessary. Worst case assumptions have to be made when function calls occur or global variables are accessed (because little information about them is available).

- **Loop optimizations**

  These act on the statements which make up a loop, such as a *for* loop (e.g., loop-invariant code motion). Loop optimizations can have a significant impact because many programs spend a large percentage of their time inside loops.

- **Prescient store optimizations**

  Allow store operations to occur earlier than would otherwise be permitted in the context of threads and locks. The process needs some way of knowing ahead of time what value will be stored by the assignment that it should have followed. The purpose of this relaxation is to allow compiler optimization to perform certain kinds of code rearrangement that preserve the semantics of properly synchronized programs.[2]\

- **Interprocedural, whole-program or link-time optimization**

  These analyze all of a program's source code. The greater quantity of information extracted means that optimizations can be more effective compared to when they only have access to local information (i.e., within a single function). This kind of optimization can also allow new techniques to be performed. For instance function inlining, where a call to a function is replaced by a copy of the function body.

- **Machine code optimization**

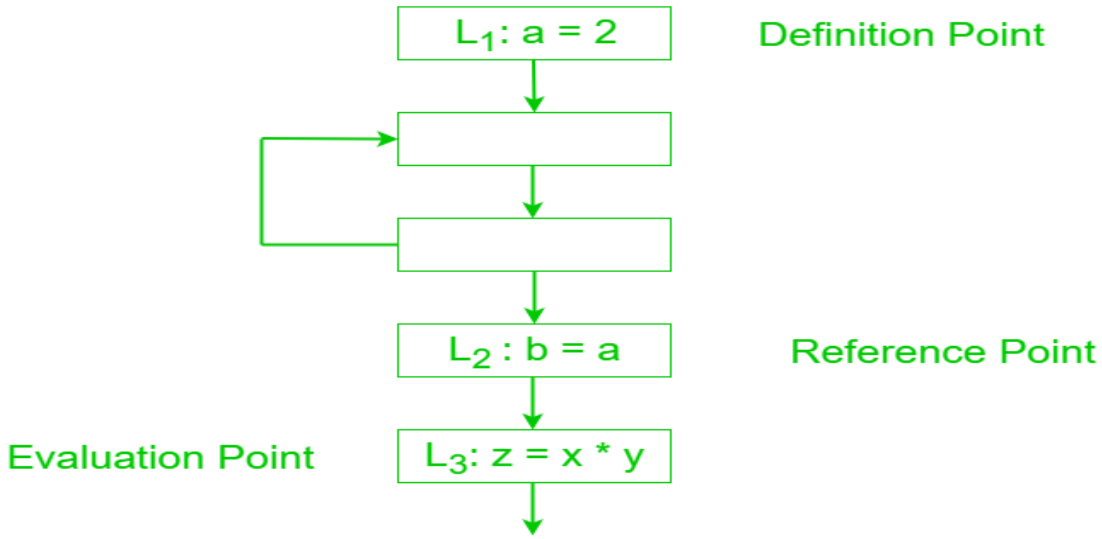
These analyze the executable task image of the program after all of an executable machine code has been linked. Some of the techniques that can be applied in a more limited scope, such as macro compression (which saves space by collapsing common sequences of instructions), are more effective when the entire executable task image is available for analysis.[3]

8.7 Data Flow Analysis :

t is the analysis of flow of data in control flow graph, i.e., the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis optimization can be done. In general, its process in which values are computed using data flow analysis.The data flow property represents information which can be used for optimization.

Basic Terminologies –

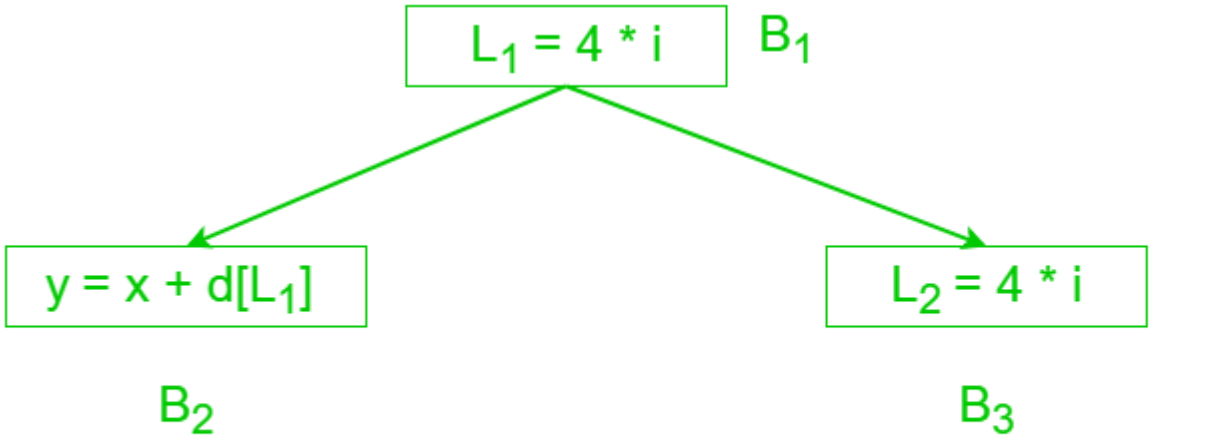
- **Definition Point:** a point in a program containing some definition.
- **Reference Point:** a point in a program containing a reference to a data item.
- **Evaluation Point:** a point in a program containing evaluation of expression.



Data Flow Properties –

- **Available Expression –** A expression is said to be available at a program point x iff along paths its reaching to x. A Expression is available at its evaluation point.
  A expression a+b is said to be available if none of the operands gets modified before their use.

Example –

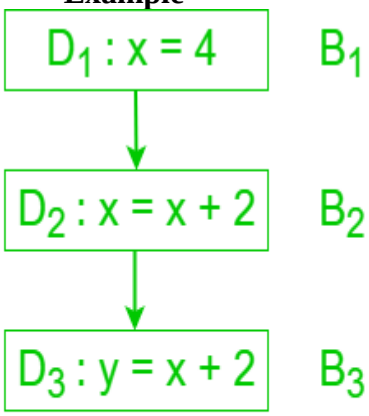


Expression $4 * i$ is available for block B_2 , B_3

Advantage –
It is used to eliminate common sub expressions.

- **Reaching Definition –** A definition D is reaches a point x if there is path from D to x in which D is not killed, i.e., not redefined.

Example –

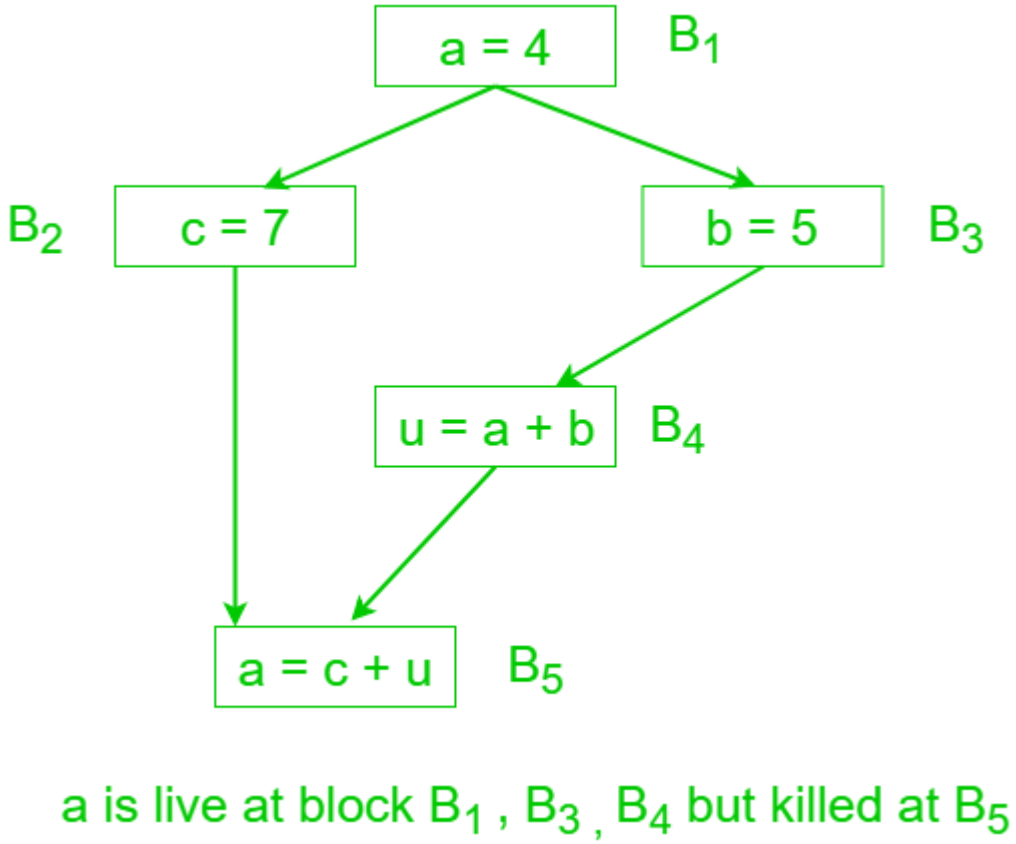


D_1 is reaching definition for B_2 but not for B_3 since it is killed by D_2

Advantage –
It is used in constant and variable propagation.

- **Live variable –** A variable is said to be live at some point p if from p to end the variable is used before it is redefined else it becomes dead.

Example –



Advantage –

1. It is useful for register allocation.
2. It is used in dead code elimination.

- **Busy Expression –** An expression is busy along a path iff its evaluation exists along that path and none of its operand definition exists before its evaluation along the path.

  Advantage –

  It is used for performing code movement optimization.

8.8 Loop Optimization

Loop optimization is most valuable machine-independent optimization because program's inner loop takes bulk to time of a programmer.

If we decrease the number of instructions in an inner loop then the running time of a program may be improved even if we increase the amount of code outside that loop.

For loop optimization the following three techniques are important:

1. Code motion
2. Induction-variable elimination

3. Strength reduction

1.Code Motion:

Code motion is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

For example

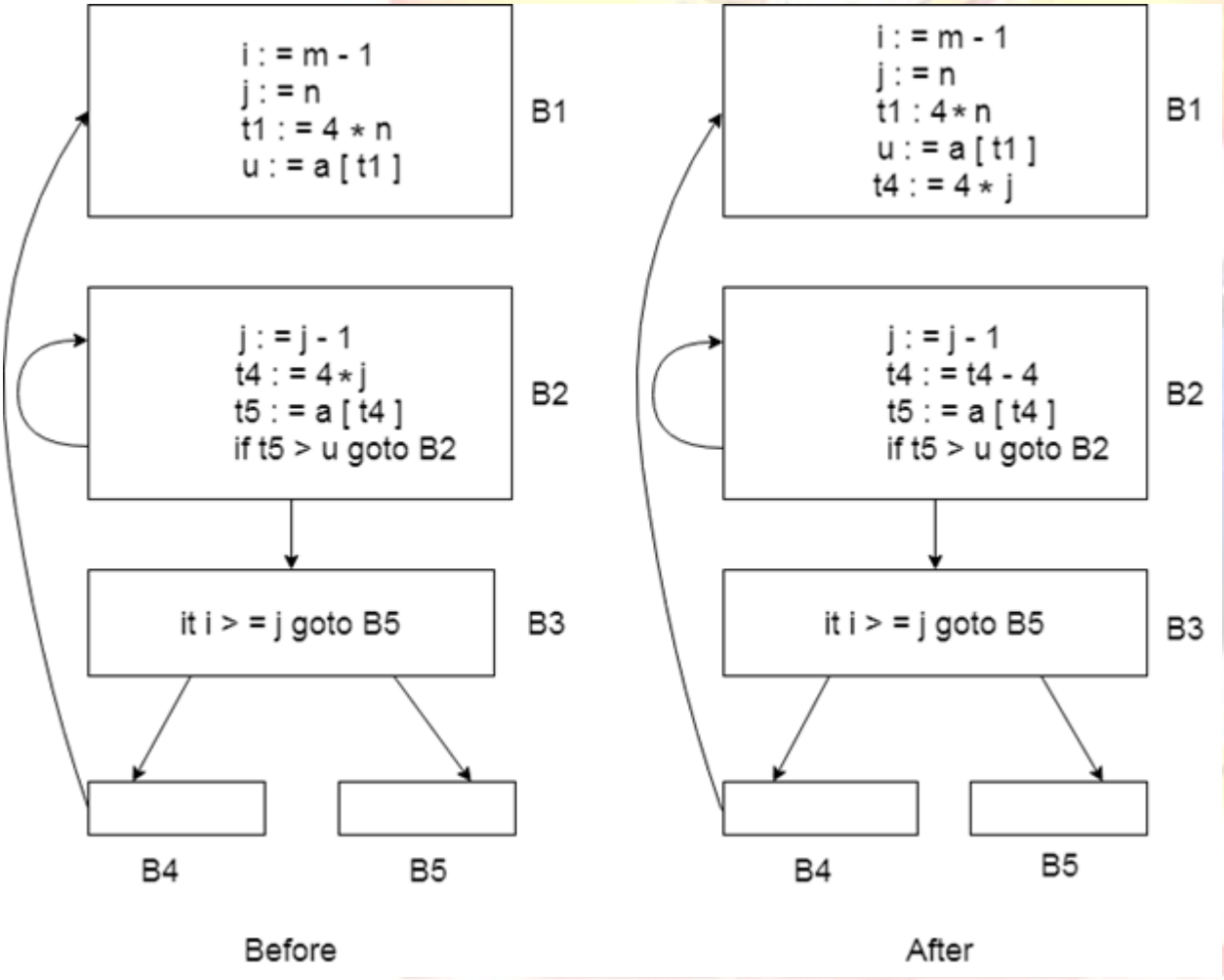
In the while statement, the limit-2 equation is a loop invariant equation.

```
1. while (i<=limit-2)     /*statement does not change limit*/
2. After code motion the result is as follows:
3.         a= limit-2;
4.         while(i<=a)    /*statement does not change limit or a*/
```

2.Induction-Variable Elimination

Induction variable elimination is used to replace variable from inner loop.

It can reduce the number of additions in a loop. It improves both code space and run time performance.



In this figure, we can replace the assignment t4:=4*j by t4:=t4+4. The only problem which will be arose that t4 does not have a value when we enter block B2 for the first time. So we place a relation t4=4*j on entry to the block B2.

3.Reduction in Strength

- Strength reduction is used to replace the expensive operation by the cheaper once on the target machine.
- Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.
- Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

Example:

```
1. while (i<10)
2.       {
3. j= 3 * i+1;
4. a[j]=a[j]-2;
5. i=i+2;
6.       }
```

After strength reduction the code will be:

```
1. s= 3*i+1;
2.     while (i<10)
3.      {
4.         j=s;
5.         a[j]= a[j]-2;
6.         i=i+2;
7.         s=s+6;
8.      }
```

In the above code, it is cheaper to compute s=s+6 than j=3 *i

8.9 Global data flow analysis :

- To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.
- Certain optimization can only be achieved by examining the entire program. It can't be achieve by examining just a portion of the program.
- For this kind of optimization user defined chaining is one particular problem.

- Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

Based on the local information a compiler can perform some optimizations. For example, consider the following code:

```
1. x = a + b;
2.   x = 6 * 3
```

- In this code, the first assignment of x is useless. The value computer for x is never used in the program.
- At compile time the expression 6*3 will be computed, simplifying the second assignment statement to x = 18;

Some optimization needs more global information. For example, consider the following code:

```
1. a = 1;
2.   b = 2;
3.   c = 3;
4.   if (....) x = a + 5;
5.   else x = b + 4;
6.   c = x + 1;
```

In this code, at line 3 the initial assignment is useless and x +1 expression can be simplified as 7.

But it is less obvious that how a compiler can discover these facts by looking only at one or two consecutive statements. A more global analysis is required so that the compiler knows the following things at each point in the program:

- Which variables are guaranteed to have constant values
- Which variables will be used before being redefined

Data flow analysis is used to discover this kind of property. The data flow analysis can be performed on the program's control flow graph (CFG).

The control flow graph of a program is used to determine those parts of a program to which a particular value assigned to a variable might propagate.

Iterative Dataflow Analysis :

To code up a particular analysis we need to take the following steps. First, we decide what sort of information we are interested in processing. This is going to determine the transfer function and the joining operator, as well as any initial conditions that need to be set up. Second, we decide on the appropriate direction for the analysis. In the forward direction, we: – Need to get the inputs from the previous instructions – Since we don't know exactly which instruction preceeded the current one, we use the join over

all possible predecessors. – Once we have the input, we apply the transfer function, which generates an output. – Iterate the process. – Mathematically:

In the backward direction, we: – Need get the outputs from the successor instructions. – Use the join since there are many successors. – Use the transfer function to get the inputs. – Iterate the process. – For reverse analyses:

Some simple iterative algorithms for data flow analysis The data flow analysis above was done assuming a structured program and operated on an abstract syntax tree. One simple approach that works even when the flow graph is not reducible is to use iterative algorithms. These algorithms try to find a solution to a system of equations. For example, for the reaching definitions problem, we have the following equations defined on each node B (a basic block) of the flow graph: or, in terms of in alone: Notice that gen and kill are defined for basic blocks. Also, in[S] is assumed to be the empty set. S is the source node of the flow graph. These equations do not have a unique solution. What we want is the smallest solution. Otherwise the result would not be as accurate as possible. in B[ ] out C[ ] C PREDECESSOR B ∈ ( ) = ∪ in B[ ] gen C[ ] ∪ ( ) in C[ ] – kill C[ ] C PRED B ∈ ( ) = ∪ out B[ ] = gen B[ ] ∪ ( ) in B

TEXT BOOKS:

1. Compilers, Principles Techniques and Tools- Alfred V Aho, Monical S Lam, Ravi Sethi, Jeffrey D. Ullman,2nd ed, Pearson,2007.
2. Principles of compiler design, V. Raghavan, 2nd ed, TMH, 2011.
3. Principles of compiler design, 2nd ed, Nandini Prasad, Elsevier

REFERENCE BOOKS:

1. http://www.nptel.iitm.ac.in/downloads/106108052/
2. Compiler construction, Principles and Practice, Kenneth C Louden, CENGAGE
3. Implementations of Compiler, A new approach to Compilers including the algebraic methods, Yunlinsu, SPRINGER