Unit IV

CODE OPTIMIZATION

Optimization –Issues related to optimization –Basic Block – Conversion from basic block to flow graph – loop optimization & its types - DAG – peephole optimization – Dominators - Global data flow analysis

Optimization

Principles of source optimization:

Optimization is a program transformation technique, which tries to improve the code by making it consume less resource (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.

Types of optimization:

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.

Steps before optimization:

1) Source program should be converted to Intermediate code
2) Basic blocks construction
3) Generating flow graph
4) Apply optimization

Basic Block

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.

```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

```
w = x + z;
```

Basic Blocks

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

1) Search header statements of all the basic blocks from where a basic block starts. Following specifications denotes the header statement:

- First statement of a program.
- Statements that are target of any branch (conditional/unconditional).
- Statements that follow any branch statement.

2) Header statements and the statements following them form a basic block.

3) A basic block does not include any header statement of any other basic block.

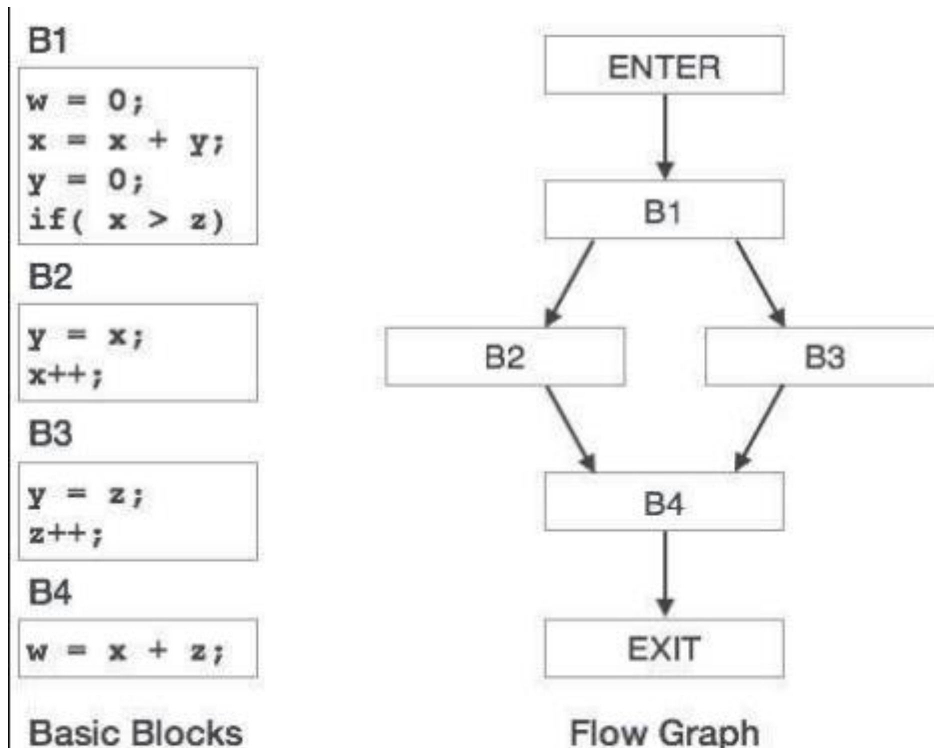
Basic blocks are important concepts from both code generation and optimization point of view.

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.

Conversion from basic block to flow graph

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.



Loop optimization & its types

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 :*
  - If a computation produces the same value in every loop iteration, move it out of the loop.
  - An expression can be moved out of the loop if all its operands are invariant in The loop
  - Constants are loop invariants.
- *Induction analysis*: A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.

  Eg:

```
extern int sum;
int foo(int n)
{
   int i, j;
   j = 5;
   for (i = 0; i < n; ++i) {
```

```
        j += 2;
        sum += j;
    }
    return sum;
}
```

This can be re written as, extern int sum;

```
int foo(int n)
 {
    int i;
    sum = 5;
        for (i = 0; i < n;
                        ++i)
{
 sum += 2 * (i + 1);
    }
    return sum;
}
```

- *Reduction in Strength*: 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.

  ```
  c = 7;
  for (i = 0; i < N; i++)
  {
      y[i] = c * i;
  }
  ```

  can be replaced with successive weaker additions

  ```
  c = 7;
  k = 0;
  for (i = 0; i < N; i++)
  {
      y[i] = k;
      k = k + c;
  }
  ```

- *Constant folding and constant propagation*

  Constant folding: It is the process of recognizing and evaluating statements with constant expressions (i=22+222+2222 to i=2466), string concatenation (“abc”+”def” to “abcdef”) and expressions with arithmetic identities (x=0; z=x*y[i]+x*2 to x=0; z=0;) at compile time rather than at execution time.

  Constant propagation:

  It is the process of substituting values of known constants in an expression at compile time.

  Eg:int x=14; int y=7+x/2;

  return

  y*(28/x+2);

  Applying constant folding and constant

  propagation, int x=14;

  int y=14;

  return 56;

Direct Acyclic Graph (DAG):

DAG is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

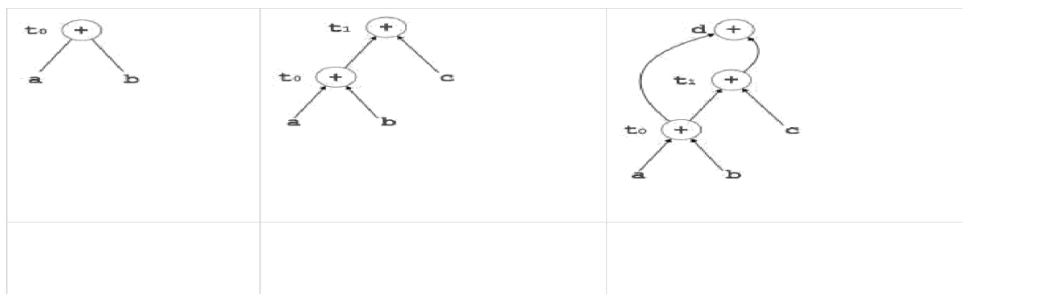
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

$t_0 = a + b$

$t_1 = t_0 + c$

$d = t_0 + t_1$



Data flow analysis:

In order to find out the value of any variable (A) of a program at any point (P) of the program, Global data flow analysis is required.

To process global data flow analysis on a basic block (B), we need to perform following functions:

(i) KILL(B)
(ii) GEN(B)
(iii) IN(B)
(iv) OUT(B)

GEN(B) : Set of definitions generated inside the block B

KILL(B) : Set of definitions outside the block B, which also has definition inside B

IN(B) : Set of definitions taken as an input to block B on a call to B

OUT(B) : Set of definitions outputted from B

- Let 'u' be the statement with definitions a,b. u=a+b

  If the definition for 'a' and 'b' are not found in Uout(P), then IN(B) is computed using GEN(B) as follows,

  IN(B) += GEN(B)

  - OUT(B) is computed from IN(B), GEN(B) and

  KILL(B) i.e) OUT(B) = [IN(B) – KILL(B)] U GEN(B)

Data Flow equations:

IN(B) = Uout[P]

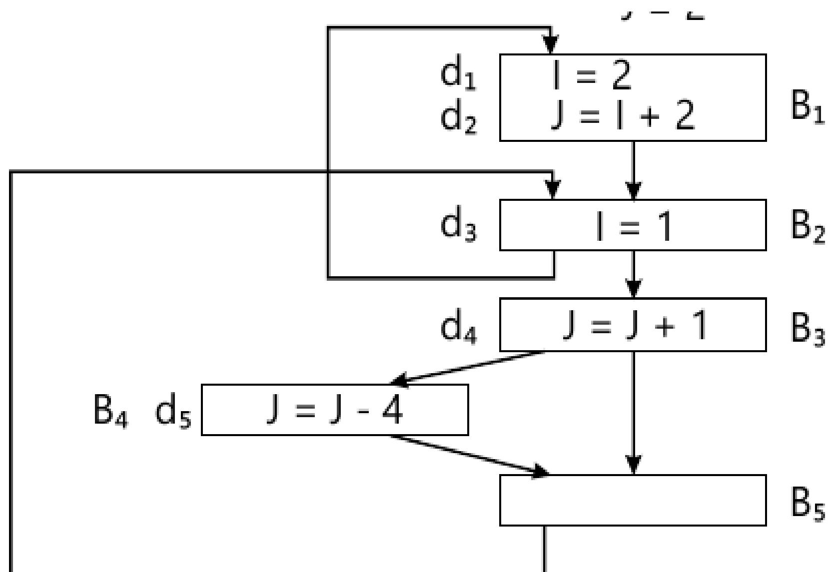
OUT(B) = [IN(B) – KILL(B)] U GEN(B)

Algorithm for reaching definition:

```
I/P: GEN(B), KILL(B)
O/P: IN(B), OUT(B)
Begin
   For each basic block, do
   Begin
         IN(B) = NULL
         OUT(B) = GEN(B)
   END
   Change = TRUE
```

```
While change, do
Begin
    Change = FALSE
    For each basic block, do
    Begin
        NewIN(B) = Vout(P)
        //P is the precedence of block B
        If NewIN(B) != IN(B), then
            Change = TRUE
            IN(B) = NewIN(B)
            OUT(B) = [IN(B) – KILL(B)] U GEN(B)
    END
END
END
```

Sample Problem:



Solution:

i) Generate GEN(B) and KILL(B) for all blocks:

GEN(B) = {d1 d2 d3 d4 d5}, where 'd' refers to a definition GEN(B1) = {1 1 0 0 0}

GEN(B2) = {0 0 1
0 0} GEN(B3) = {0
0 0 1 1} GEN(B4)
= {0 0 0 0 1}
GEN(B5) = {0 0 0
0 0} Hence,

KILL(B1) = {0 0 1
1 1} KILL(B2) =
{1 1 0 1 1}
KILL(B3) = {1 1 1
0 0} KILL(B4) =
{1 1 1 1 0}
KILL(B5) = {1 1 1
1 1}

ii) Process algorithm to compute IN(B) and
OUT(B) Iteration 1:

	IN(B) = NULL	OUT(B) = GEN(B)
B1	0 0 0 0 0	1 1 0 0 0
B2	0 0 0 0 0	0 0 1 0 0
B3	0 0 0 0 0	0 0 0 1 1
B4	0 0 0 0 0	0 0 0 0 1
B5	0 0 0 0 0	0 0 0 0 0

Change = TRUE

Iteration 2:

Change = FALSE

IN(B1) = {0 0 0 0 0}

Change = TRUE

OUT(B1) = (0 0 1 0 0 – 0 0 1 1 1) U (1 1 0 0 0)

= (0 0 1 0 0) & (1 1 0 0 0) | (1 1 0 0 0)

OUT(B1) = (1 1 0 0 0)

Proceeding the algorithm further,

	IN(B) = Vout(P)	OUT(B) = [IN(B) – KILL(B)] U GEN(B)
B1	0 0 0 0 0	1 1 0 0 0
B2	1 1 0 0 0`	0 1 1 0 0
B3	0 1 1 0 0	0 0 1 1 0
B4	0 0 1 1 0	0 0 1 0 1
B5	0 0 1 1 1	0 0 1 1 1

Iteration 3:

	IN(B) = Vout(P)	OUT(B) = [IN(B) – KILL(B)] U GEN(B)
B1	0 1 1 0 0	1 1 0 0 0
B2	1 1 1 1 1	0 1 1 1 1
B3	0 1 1 1 1	0 0 1 1 0
B4	0 0 1 1 0	0 0 1 0 1
B5	0 0 1 1 1	0 0 1 1 1

Iteration 4:

	IN(B) = Vout(P)	OUT(B) = [IN(B) – KILL(B)] U GEN(B)
B1	0 1 1 1 1	1 1 0 0 0
B2	1 1 1 1 1	0 1 1 1 1
B3	0 1 1 1 1	0 0 1 1 0
B4	0 0 1 1 0	0 0 1 0 1
B5	0 0 1 1 1	0 0 1 1 1

Iteration 5:

	IN(B) = Vout(P)	OUT(B) = [IN(B) – KILL(B)] U GEN(B)
B1	0 1 1 1 1	1 1 0 0 0
B2	1 1 1 1 1	0 1 1 1 1
B3	0 1 1 1 1	0 0 1 1 0
B4	0 0 1 1 0	0 0 1 0 1
B5	0 0 1 1 1	0 0 1 1 1

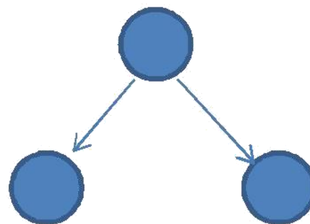
As iteration 4 and 5 produces same values, the process ends. Hence the value of definition anywhere at any point of time is deduced.

Dominators

In order to deduct the control flow within basic blocks, it's required to compute dominators. In the flow graph consisting a node D and E, if path leaves from D to E, then node D is said as dominating E.

Dominator Tree:

Dominator tree represents the hierarchy of the blocks and its execution flow. Here, the initial node is taken as the root and every further parent is said to be intermediate dominator of child node.



In above tree, node 1 dominates 2 and 3.

Dominator computing algorithm:

Begin

D(n0) = {n0}//Dominator(Node 0) = {Node 0}, where n0 is the root node

For each node 'n' in N–{n0}, // N – Set of nodes in the graph

D(n) = N;

Change = TRUE

While change, do

Begin

Change = FALSE

For each node 'n' in N–{n0}

Begin

NEWD = n U [P(n)] // P(n) – Predecessor of n

If D(n) != NEWD, then

Change = TRUE

Change = NEWD

END

END

END

Peephole Optimization

- Optimizing a small portion of the code.
- These methods can be applied on intermediate codes as well as on target codes.
- A bunch of statements is analyzed and are checked for the following possible optimization

(1) Redundant instruction elimination

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

```
MOV x, R0
MOV R0, R1
```

First instruction can be rewritten as

```
MOV x,R1
```

(2) Unreachable Code

It is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
….

If ( debug ) {
Print debugging information

}
```

In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L1 goto L2

L1: print debugging information L2: ………………………… (a)
```

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of debug; (a) can be replaced by:

```
If debug ≠1 goto L2
Print debugging information
L2: …………………………… (b)
```

If debug $\neq 0$ goto L2
Print debugging information
L2: …………………………… (c)

As the argument of the statement of (c) evaluates to a constant true it can be replaced

By goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

(3) Flow of control optimization

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1
….

L1: gotoL2 (d)
by the sequence
goto L2
….
L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1
….
L1: goto L2 (e)

can be replaced by
If a < b goto L2

….
L1: goto L2

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

L1: if a < b goto L2 (f) L3:

may be replaced by

If a < b goto L2
goto L3

…….

L3:

While the number of instructions in(e) and (f) is the same, we sometimes skip the unconditional jump in (f), but never in (e).Thus (f) is superior to (e) in execution time

(4) Algebraic Simplification

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

x := x+0 or
x := x * 1

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

(5) Reduction in Strength

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

X2 → X*X

(6) Use of Machine Idioms

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : =i+1.

i:=i+1 → i++
i:=i-1 → i- -