# CODE OPTIMIZATION

Zakia Zinat Choudhury
Lecturer
Department of Computer Science & Engineering
University of Rajshahi

1

## Code Optimization

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

2

# Why Code Optimization?

Optimizing an algorithm is beyond the scope of the code optimization phase. So, the program is optimized. And it may involve reducing the size of the code.

So, optimization helps to:

- Reduce the space consumed and increases the speed of compilation.
- Manually analyzing datasets involves a lot of time. Hence, we make use of software like Tableau for data analysis. Similarly, manually performing the optimization is also tedious and is better done using a code optimizer.
- An optimized code often promotes re-usability.

3

# Efforts of Code Optimization

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.

4

# Types of Code Optimization

The optimization process can be broadly classified into two types :

- **Machine Independent Optimization** – This code optimization phase attempts to **improve the intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here **does not involve any CPU registers or absolute memory locations.**

- **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 the memory hierarchy.

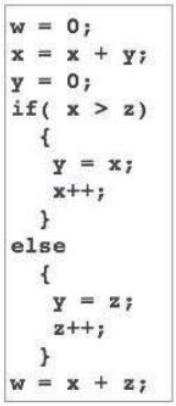
5

# Basic Blocks

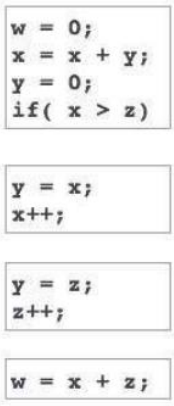
Source codes generally have **a number of instructions**, which are **always executed** in **sequence** and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic 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.

6




Source Code

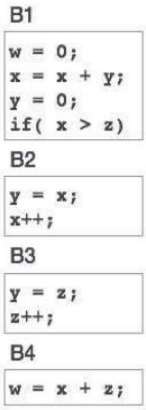



Basic Blocks

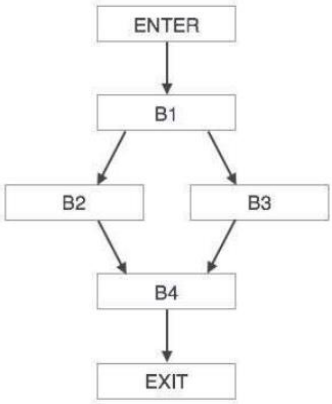
# Basic Blocks

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.

7




Basic Blocks




Flow Graph

# 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

# Sources of Code Optimization

There are several ways in which a compiler can improve a program without changing the function it computes.

They are:

- Common sub expression elimination
- Copy propagation
- Dead-code elimination
- Constant folding

9

# Common Sub Expression Elimination

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

For example

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5
```

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

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of i is not been changed from definition to use.

10

# Copy Propagation

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example:

```
x=Pi;
A=x*r*r;
```

The optimization using copy propagation can be done as follows: A=Pi*r*r;

Here the variable x is eliminated

11

# 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.

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

For example:

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

Here, ‘if’ statement is dead code because this condition will never get satisfied.

12

# Constant Folding

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation.

13

# Loop Optimization

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

**Loop Optimization** is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.

Three techniques are important for loop optimization:

- Code motion
- Induction-variable elimination
- Reduction in strength

14

# Code Motion

An important modification that decreases the amount of code in a loop is **code motion**. This transformation takes an expression that yields the same result independent of the number of times a loop is executed and places the expression before the loop. Note that the notion **"before the loop"** assumes the existence of an entry for the loop.

For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change limit*/
```

Code motion will result in the equivalent of

```
t= limit-2;
while (i<=t) /* statement does not change limit or t */
```

15

# Induction Variable Elimination

Induction-variable elimination which we apply to replace variables from inner loop.

**Induction variable elimination** can reduce the number of additions (or subtractions) in a loop and improve both run-time performance and code space. Some architectures have auto-increment and auto-decrement instructions that can sometimes be used instead of induction variable elimination.

16

# 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.

**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.

17