# CS 4031
# Compiler Construction
# Lecture 13


Mahzaib Younas

Lecturer, Department of Computer Science

FAST NUCES CFD

# Code Optimization

- It is possible for a programmer to outperform an optimizing compiler by using his or her knowledge in choosing a better algorithm and data items. This is far from practical conditions.
- In such an environment, there is need for an optimizing compiler. 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.

# Need of Efficiency

- Inefficient programming (which forces many invisible instructions to be performed for actual computation.)
- Example:
  - a = a + 0
- The programming constructs for easy programming.
  - example: Iterative loops.
- Compiler generated temporary variables or instructions.

# Constraints

- The following constraints are to be considered while applying the techniques for code improvement:
  - The transformation must preserve the meaning of the program, that is, the target code should ensure semantic equivalence with source program.
  - Program efficiency must be improved by a measurable amount without changing the algorithm used in the program.
  - When the technique is applied on a special format, then it is worth transforming to that format only when it is beneficial enough.
  - Some transformations are applied only after detailed analysis, which is time consuming. Such analysis may not be worthy if the program is run very few number of times.

# Classification of Optimizations

- The optimization can be classified depending on
  - Level of code
  - Programming Language
  - Scope

# Level of Code

- Design level
  - efficiency of code can be improved by making the best use of available resources and selection of suitable algorithm.
- Source code level
  - the user can modify the program and change the algorithm to enhance the performance of the object code.
- Compile level
  - the compiler can enhance the program by improving the loops, optimizing on the procedure calls and address calculations. This is possible when the representation is in three address code.
- Assembly level
  - the compiler optimizes the code based on the machine architecture and is based on the available registers and suitable addressing modes.

# Programming Language

- Machine Independent–
  - the code-improvement techniques that do not consider the features of the target machine are machine-independent techniques.
  - Constant folding, dead code elimination, and constant propagation are examples of machine-independent techniques. These are applied on either high-level language or intermediate representation.
- Machine dependent
  - these techniques require specific information relating to target machine. Register allocation, strength reduction, and use of machine idioms are
  - examples of machine-dependent techniques.

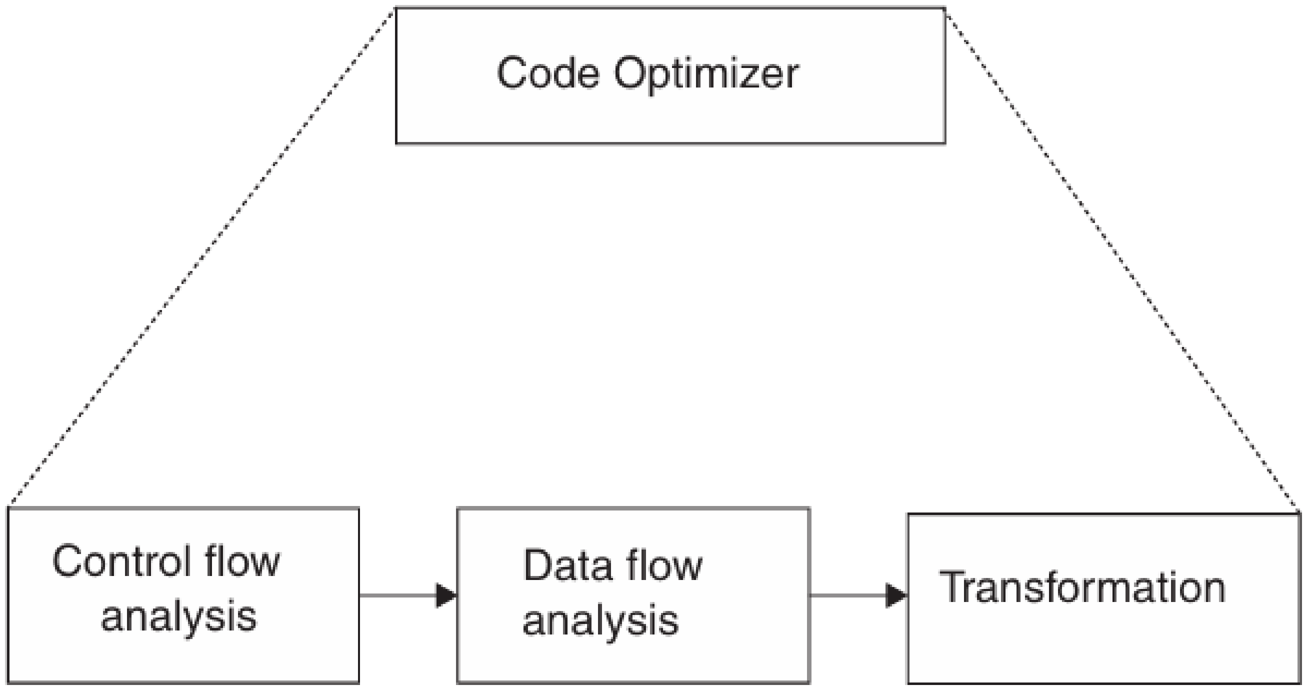
# Scope

- Local
  - 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.
- Global
  - optimization 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

# Where How to Optimize

- 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.

# Code Optimization Model



# Flow Graph

- A graphical representation of three address code is called flow graph.
- The nodes 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 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.

# Example:

- For the statement a = b + c * d/e the corresponding 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.

# 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;
}
```

# Example:

- The three address code for the initialize function is as follows:

(1). $i := 0$
(2). $n := 10$
(3). $t_1 := n - 1$
(4). If $i > t_1$ goto (12)
(5). $t_2 := i * i$
(6). $t_3 := 4 * i$
(7). $t_4 := a[t_3]$
(8). $t_4 := t_2$
(9). $t_5 := i + 1$
(10). $i := t_5$
(11). goto (3)
(12). return

# Solution

- Identifying leader statements in the above three address code
- Statement (1) is leader using rule 1
- Statement (3) and (12) are leader using rule 2
- Statement (4) and (12) are leaders using rule 3

# Solution

1. $i := 0$ → Leader 1
2. $n := 10$
3. $t_1 := n - 1$ → Leader 2
4. If $i > t_1$ go to (12)
5. $t_2 := i * i$ → Leader 3
6. $t_3 := 4 * i$
7. $t_4 := a[\ t_3\ ]$
8. $t_4 := t_2$
9. $t_5 := i + 1$
10. $i := t_5$
11. go to (3)
12. Return → Leader 4

**Basic block 1 includes statements (1) and (2) Basic block 2 includes statements (3) and (4) Basic block 3 includes statements (5)–(11) Basic block 4 includes statement (12)**

# Flow Graph

(1) $i = 0$
(2) $n = 10$

B1

(3) $t_1 := n - 1$
(4) If $i > t_1$ go to (12)

B2

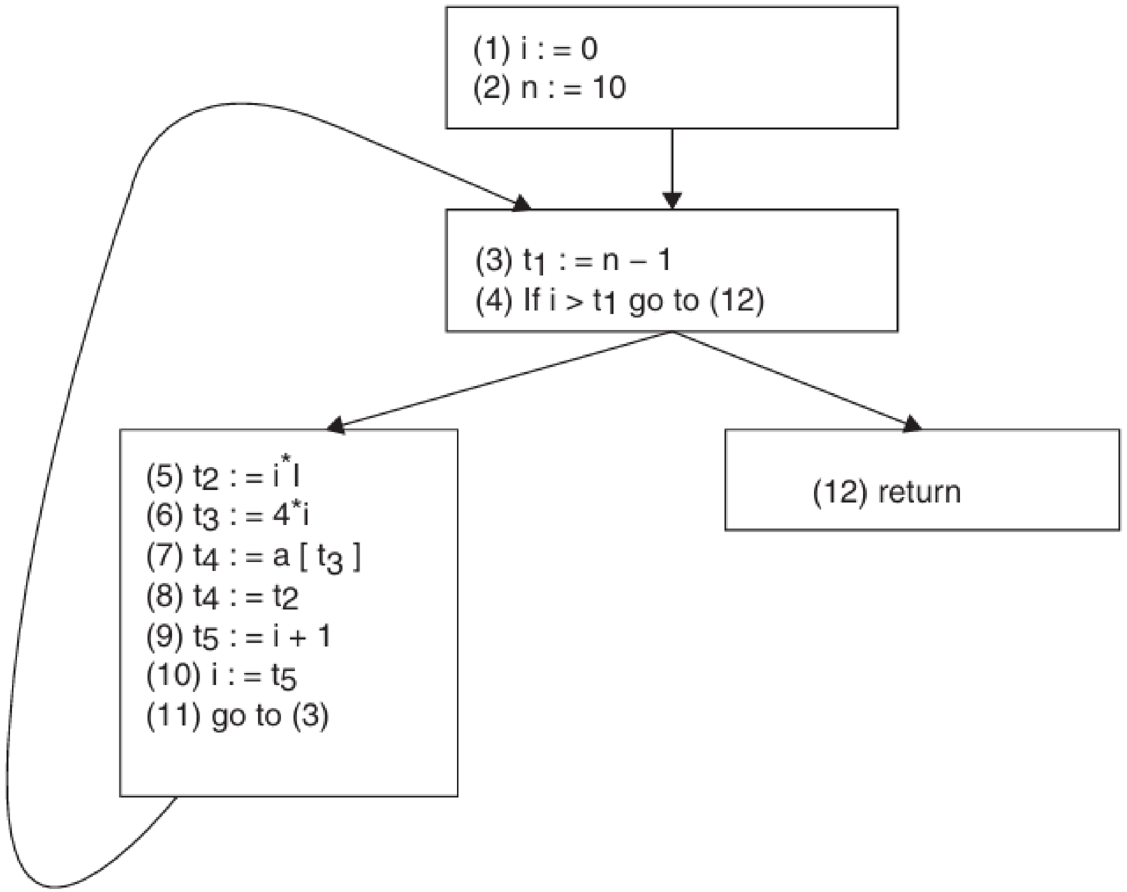
(5) $t_2 = i^* I$
(6) $t_3 = 4^* i$
(7) $t_4 = a [ t_3 ]$
(8) $t_4 = t_2$
(9) $t_5 = i + 1$
(10) $i := t_5$
(11) go to ( 3 )

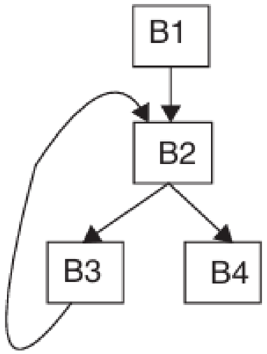
B3

(12) return

B4

# Flow Graph Control



# DAG Representation of Basic Block

- A DAG has nodes, which are labeled as follows:
  - The leaf nodes are labeled by either identifiers or constants. If the operators are arithmetic then it always requires the r- value.
  - The labels of interator nodes correspond to the operator symbol.
  - Some nodes are sometimes referred to by the sequence of identifiers for labels.
  - The interior nodes represent computed values.

# Construction of DAG

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

B1

(3) $t_1$ : = n – 1
(4) If i > $t_1$ go to (12)

B2

(5) $t_2$ : = i*I
(6) $t_3$ : = 4*i
(7) $t_4$ : = a [ t3 ]
(8) $t_4$ : = $t_2$
(9) $t_5$ : = i + 1
(10) i : = t5
(11) go to ( 3 )

B3

(12) return

B4

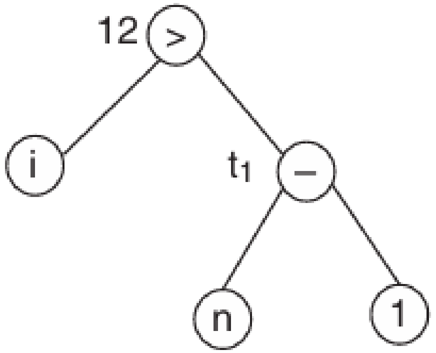



**Figure 10.6(b)** DAG for Block B2

# Construction of DAG

B1

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

B2

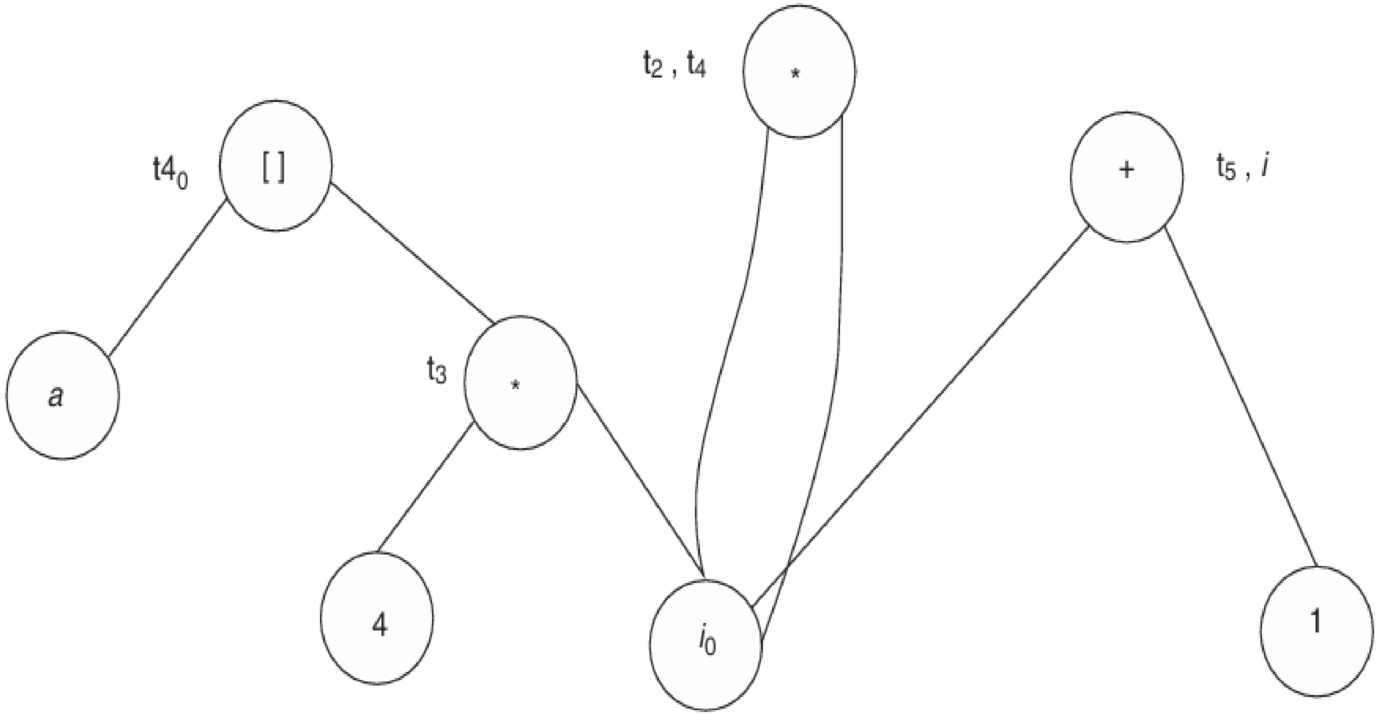
(3) $t_1$ : = n – 1
(4) If i > $t_1$ go to (12)

B3

(5) $t_2$ : = i*I
(6) $t_3$ : = 4*i
(7) $t_4$ : = a [ t3 ]
(8) $t_4$ : = $t_2$
(9) $t_5$ : = i + 1
(10) i : = t5
(11) go to ( 3 )

B4

(12) return



# Example:

Create DAG for the following code:

```
t1 = 4 * i
t2 = a[t1 ]
t3 = b[t1 ]
t4 = t2 * t3
pr = pr + t4
i = i + 1
if i <=20
goto (1)
```

# Solution: