

Course: Compiler Construction

Week 11: Code Optimization

Lecturer: Martha Gichuki

Learning outcomes Week 11: Code Optimization

At the end of the lecture, you will be able to:

- i. Describe code optimization
- ii. Describe the objectives of code optimization
- iii. Describe various techniques used to optimize code

1. Introduction:

- ✓ In compiler design, the term code optimization refers to *attempts* made by the compiler to produce more efficient code compared to the obvious code.
- ✓ Code optimization is therefore a contradiction since the code produced by a compiler *cannot be guaranteed* to be as fast or faster than any other code that performs the same task¹.
- ✓ Code optimization or improvement is the attempt to *remove unnecessary instructions in object code*, or the *replacement* of one sequence of instructions by a faster sequence of instructions that does the same thing.

- ✓ The objectives of code improvement are:
 - i. To consume fewer resources i.e. Central Processing Unit (CPU) and Memory
 - ii. Deliver high speed²

¹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 40

² Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 608

1.1. Basic Blocks

- ✓ Source codes contain multiple instructions, which are always executed in sequence and are considered as the basic blocks of the program.
- ✓ 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, such as *IF-THEN-ELSE*, *SWITCH-CASE* conditional statements and loops such as *DO-WHILE*, *FOR*, and *REPEAT-UNTIL*, among others³.

1.2. Identifying Basic Blocks

- ✓ To identify the basic blocks in a program, the following algorithm can be used:
 - i. 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.
 - ii. Header statements and the statements following them form a basic block.
 - iii. A basic block does not include any header statement of any other basic block⁴.
- ✓ 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 execution⁵

³ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 550

⁴ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 551

⁵ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 552

```

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

Code optimization can be accomplished either locally or globally: -

- i. **Local code optimization** - code improvement happens within a basic block
- ii. **Global code optimization** - where code improvements take into account whatever is happening across basic blocks⁶.

Three rules to be followed in Code Optimization process

- i. Output code should not change the program meaning.
- ii. Optimization should increase program speed and use a smaller number of resources
- iii. Optimization process should be fast and should not delay the entire compiling process⁷

Two Categories of Code Optimization processes⁸

A. Machine-Independent Optimization

- ✓ The compiler takes the intermediate code and transforms a part of the code that *does not involve any CPU registers and/or absolute memory locations*.

⁶ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 608

⁷ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 608

⁸ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 608

✓ **Example:**

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

- ✓ This code involves repeated assignment of the identifier item⁹
- ✓ In order to save the CPU cycles and use the code on any processor, we re-write the code as follows:-

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

B. Machine-Dependent Optimization

- ✓ This is done after the target code has been generated and when the code is transformed according to the target machine architecture¹⁰.
- ✓ 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.

2. Towards Optimized Code¹¹

- ✓ Improved Intermediate code can be generated by the compiler through *address calculations and improving loops*
- ✓ While generating target code, the compiler can *use memory hierarchy and CPU register*
- ✓ Users can *change/rearrange the code or use better algorithms* to write the code
- ✓ *Dead code* can be eliminated and elimination of *code redundancy* is also possible

⁹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 608

¹⁰ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 608

¹¹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 608

2.1. What causes Code Redundancy?

- ✓ Presence of several redundant operations in a typical program.
- ✓ Sometimes the redundancy happens at the source level e.g. programmers may find it more direct and convenient to recalculate some result, leaving it to the compiler to recognize that only one such calculation is necessary.
- ✓ Code redundancy is mostly a side effect of writing programs in a high-level language.
- ✓ In most languages (other than C or C++, where pointer arithmetic is allowed), programmers have no choice but to refer to elements of an array or fields in a structure¹²

2.2. Redundant instruction elimination

- ✓ At compilation level, the compiler searches for redundant instructions.
- ✓ Multiple loading and storing instructions may carry the same meaning even if some of them are removed¹³.
- ✓ Example: Consider the two instructions below:-

MOV x, R0

MOV R0, R1

- ✓ We remove the first instruction and re-write the instruction as:

MOV x, R1

2.3. Data-flow analysis

- ✓ This is a key part of code optimization.
- ✓ Data-Flow analysis engines is one of the compiler construction tools we discussed in lecture 1
- ✓ The engines facilitate the gathering of information about how values are transmitted from one part of a program to the other parts¹⁴.

2.4. Dead Code Elimination

- ✓ Dead code plays no role in any program execution and it refers to statements that are either:-
 - i. Never executed or unreachable

¹² Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 639

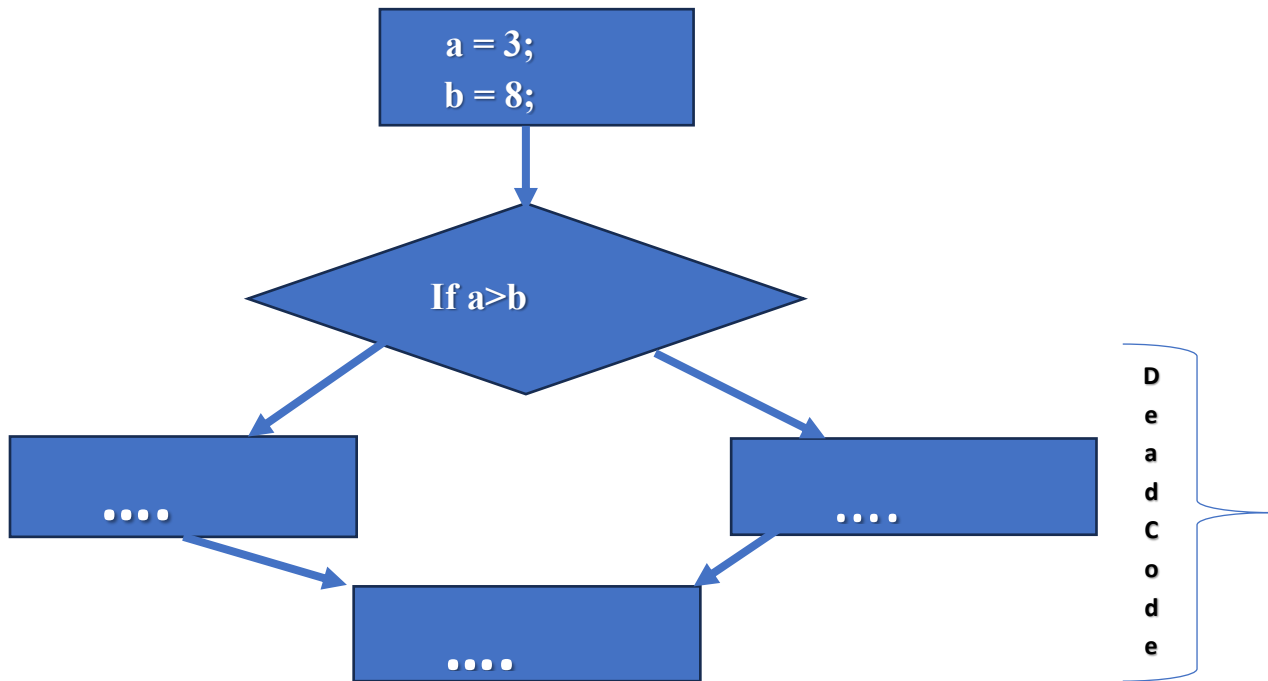
¹³ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 639

¹⁴ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 628

ii. Or, if executed their output is never utilized

✓ Partially dead code is that whose computed values are used only under certain circumstances¹⁵

Dead Code Elimination¹⁶



2.5. Loop Optimization

✓ Loop optimization saves on CPU cycles and memory and it can be achieved using the following techniques:-

- i. Strength reduction - Some expressions consume more CPU cycles, time and memory than others. These should be replaced with cheaper expressions without compromising the output quality
- ii. Loop Invariant Code - is a code fragment residing in the loop, which computes the same value at each iteration. This code can be moved out of the loop by saving it to be computed only once instead of with each iteration

¹⁵ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 560, 655

¹⁶ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 616

- iii. Induction analysis - An induction variable is one whose value is altered within the loop by a loop-invariant value¹⁷

a) Strength Reduction

- ✓ There are operations that consume more time and space.
- ✓ Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result. Consider the following examples:-
 - i. $x * 2$ is computationally expensive and can be replaced by $x \ll 1$, which involves only one left shift. Left bit shifting to multiply by any power of two
 - ii. $x = x * 8$ can be written as $x \ll 3$ (since 2 to the power of 3 is 8).
 - iii. $x = x / 2$ can be written as $x \gg 1$. Right bit shifting to divide by any power of two.
 - iv. $a * a$ can be written as a^2 . The output is the same, but a^2 is much more efficient to implement¹⁸.

b) Loop invariant Code

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

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

- ✓ Assume that the values of operands (y and z) are not changed from assignment of variable a to variable c.
- ✓ If the condition statement is true, then y OP z is computed twice, otherwise once¹⁹

¹⁷ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 667

¹⁸ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 577, 713

¹⁹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 667

Code motion

- ✓ Code motion can be used to eliminate this redundancy as follows:-

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

- ✓ Here, regardless of whether the condition is true or false; y OP z should be computed only once²⁰.

2.6. Peephole Optimization

- ✓ This optimization technique works locally on the source code to transform it into an optimized code.
- ✓ Working locally, means a small portion of the code block at hand is worked on.
- ✓ These methods can be applied on intermediate codes as well as on target codes.
- ✓ A batch of statements is analyzed and are checked for possible optimization²¹

2.7. Unreachable code

- ✓ Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidentally written pieces of code that can never be reached.

²⁰ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 644

²¹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 574

✓ **Example²²:**

```
void add_ten(int x)
{
return x + 10;
printf("value of x is %d", x);
}
```

- ✓ In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

2.8. Flow of control optimization²³

- ✓ There are instances in a code where the program control jumps back and forth without performing any significant task. To achieve optimization, these jumps should be eliminated.
- ✓ Consider the following piece of code:

```
...
MOV R1, R2
GOTO L1
...
L1 : GOTO L2
L2 : INC R1
```

- ✓ In this code, label L1 can be removed as it passes the control to L2.
- ✓ Instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
MOV R1, R2
GOTO L2
...
L2 : INC R1
```

²² Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 575

²³ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 576

2.9. Algebraic Expression Simplification

- ✓ Sometimes algebraic expressions can be simplified.
- ✓ Examples:-
 - i. The expression $a = a + 0$ can be replaced by a itself
 - ii. The expression $a = a + 1$ can simply be replaced by $INC\ a^{24}$.

2.10. Accessing Machine Instructions

- ✓ The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently.
- ✓ If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results²⁵.

Content Covered in Week 11: Code Optimization

At the end of the lecture, we were able to:

- i. Describe code optimization
- ii. Describe the objectives of code optimization
- iii. Describe various techniques used to optimize code

Course Text Books

1. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; *Addison- Wesley Pub Co*, ISBN: 0201100886 (2007))
2. Compiler Construction: Principles and practices; Kenneth C. Loudon; *Cengage Learning*; 1st edition, ISBN-10 : 0534939724 (1997)
3. Basics of Compiler Design: Torben Mogensen; *DIKU University of Copenhagen Universitetsparken 1 DK-2100 Copenhagen DENMARK* (2007)
4. Engineering a Compiler: 2nd edition; Keith D. Cooper and Linda Torczon; *Morgan Kaufmann Publishers* ISBN: 978-0-12-088478-0 (2003)
5. Compiler Design: Santanu Chattopadhyay; *PHI Learning Publishers*, ISBN 812032725X, 9788120327252 (2005)

²⁴ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 577

²⁵ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 577