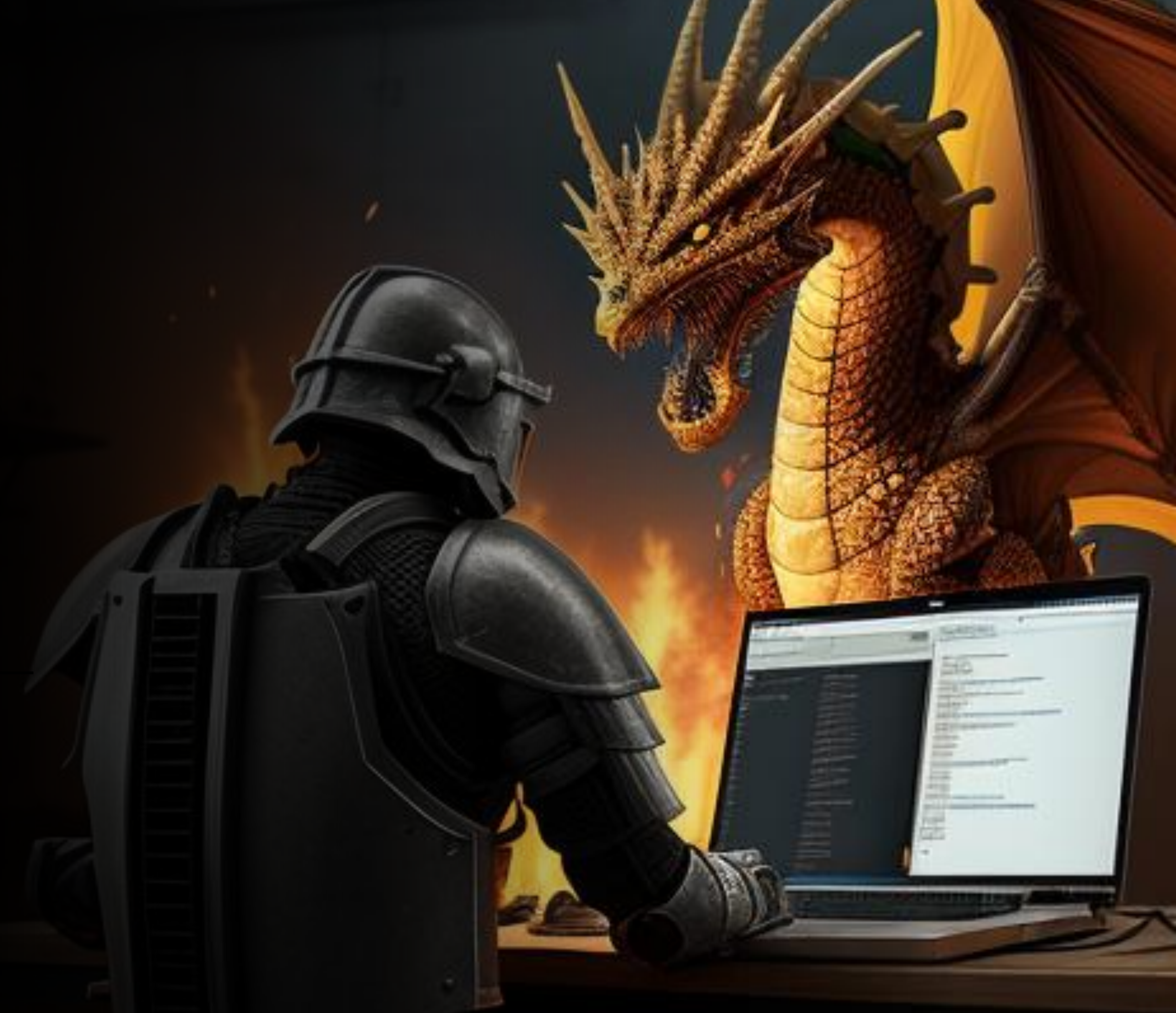
Code Optimization



Baivab Das

Lecturer

Department of CSE

University of Asia Pacific

Introduction

- Code optimization phase is used for improving the intermediate code
- Some transformations are applied to improve/optimize the intermediate code (We have already seen some of the transformations)
- It reduces the space and time of the program
- The code optimization techniques must preserve the meaning of the program
- Code optimization can be machine dependent or machine independent

Code Optimization Techniques

- Common Subexpression Elimination
- Constant Folding
- Copy Propagation
- Dead Code Elimination
- Code Motion
- Induction Variable Elimination and Reduction in Strength

Common subexpression elimination

- If an expression **previously computed** and the **value was not changed** then they are called common subexpressions

```
    1)  x  =  y+z
    2)  y  =  x-w
    3)  z  =  y+z
B1  4)  w  =  x-w
```

```
    1)  x  =  y+z
    2)  y  =  x-w
    3)  z  =  y+z
B1  4)  w  =   y
```

- Observe above, statement 1 and 3 might have the same operations but the value of y gets changed in statement 2
- Statement 2 and 3 are common and they are optimized

Constant Folding

- It is normally performed in compile time
- If the **value of an expression is constant** then use the constant value instead of the expression

```
PI = 22/7
```
→
```
PI = 3.14
```

- Here instead of using `22/7` as the value of `PI`, we can simply write 3.14

```
1) x = 10
2) y = 20
3) z = x+y
```
→
```
1) x = 10
2) y = 20
3) z = 30
```

- The compiler can fold the expression `x+y` at compile time and replace it with the constant value `30`

Copy Propagation

- Copy propagation in compiler design is a technique used to optimize code by eliminating redundant variable assignments
- It works by identifying occurrences of variables that are assigned a constant value and replacing them with the constant value
- This can reduce the number of instructions that need to be executed and improve the performance of the code
- If we have a copy statement such as `f=g` then we can use `g` for `f` wherever possible after the copy statement `f=g`

```
1) x = a
2) y = x*b
3) z = x*c
```

```
1) x = a
2) y = a*b
3) z = a*c
```

Dead Code Elimination

- Dead code elimination is a compiler optimization technique that removes code that is never executed.
- Dead code elimination is a powerful compiler optimization technique that can improve the performance, size, and readability of code
- Dead code can be created in a number of ways, such as:
  - **Unreachable code**: Code that is never reached because of a conditional branch
  - **Unused variables**: Variables that are declared but never used
  - **Unused functions**: Functions that are defined but never called
  - **Unused labels**: Labels that are defined but never jumped to
- The compiler can identify dead code by performing control flow analysis

Example: DCE

```
int main() {
  int x = 10;
  int y = 20;

  if (x > y) {
    //dead code as it is always false
    return 1;
  } else {
    return 0;
  }
}
```

- If we observe the example code, the condition

```
if (x > y)
```

is always false

- We can use DCE on this part

Example: DCE (Cont.)

```
// Intermediate code
1. label 1:
2.   if x > y:
3.     goto label 2
4.   return 0
5. label 2:
6.   return 1
```

```
// IC after DCE
1. label 1:
2.   if x > y:
3.     goto 2
4.   return 0
```

- Note that the `goto` statement is actually unreachable, since the condition `x > y` is **always false**
- Therefore, the optimizer could actually remove the entire `if` statement, resulting in the following optimized intermediate code:

```
// Further
Optimized IC
1. return 0
```

Code Motion

- Code motion is a compiler optimization technique that moves code from one location to another in order to improve the performance of the program
- This can be done by moving code from a loop body to outside the loop, or by moving code from one conditional branch to another
- Code motion is a powerful compiler optimization technique that can improve the performance, size, and readability of code

Example: Code Motion

```
while(i<10)
{
    x=y+z;
    i=i+1;
}
```

```
x=y+z;
while(i<10)
{
i=i+1;
}
```

- As the statement `x=y+z` is not dependent on the loop, we can move this statement outside of the loop
- So, before optimization the `x=y+z` statement was getting executed 10 times with the same result

Code Motion: When Not Apply

- Code motion **cannot be applied** in the following cases:
  - The code that needs to be moved is **interdependent** with other code. For example, if the code that needs to be moved writes to a variable that is also read by other code inside the loop, then the code cannot be moved outside the loop
  - The code that needs to be moved is **affected by side effects**. For example, if the code that needs to be moved calls a function that has side effects, then the code cannot be moved outside the loop
  - The code that needs to be moved is **too large or complex**. The compiler may not be able to determine whether it is safe to move the code, or the cost of moving the code may outweigh the benefits

Induction Variable Elimination and Reduction in Strength

- Induction variables are variables that are used to control the iterations of a loop
- They are typically incremented or decremented by a constant value on each iteration of the loop
- Induction variable elimination is a technique that removes induction variables from loops. This can be done by replacing the induction variable with a constant value that is calculated outside the loop
- Reduction in strength is a technique that replaces complex arithmetic operations with simpler ones. This can be done by using identities and other mathematical properties

Example of IVE and Reduction in Strength

```
i=1;
while(i<10)
{
    t=i*4;
    i=i+1;
}
```

```
t=4;
while(t<40)
{
    t=t+4;
}
```

- The loop is executed 9 times and each time the `t` value is incremented by 4
- This multiplication of `t=i*4` can be replaced by addition
- Induction variable `i` can also be removed by `t`
- During the first time as we don't know the value of `t` we initialize it before the while statement

Example of Induction Variable Elimination

```
int j = 0;
for (int i = 0; i < 100; i++) {
    j = 2*i;
}
return j;
```

- j is an induction variable dervied by applying a multiplication to another IV, i. This makes it a perfect candidate for strength reduction. Each iteration we set j to a brand new value computed with that multiplication. Instead, every iteration we can increment j by two times whatever we increment i by
- To simplify this optimization this is usually done by introducing a new variable to represent the 2*i value for each iteration

```
int j = 0;
int s = 0; //2*i when i == 0
for (int i = 0; i < 100; i++) {
  j = s;
  s = s + 2; //+2 since i gets incremented by 1 each iteration
}
Aft
```