CS 4031
Compiler Design
Lecture 14

Mahzaib Younas

Lecturer, Department of Computer Science

FAST NUCES CFD

Techniques of Code Optimization

- Optimization performed within a basic block.
- This is simplest form of optimizations.
- No need to analyze the whole procedure body
- Just analyze the basic block of the procedure

Techniques of Local Code Optimization

- The local optimization techniques include:
    - Constant folding
    - Constant propagation
    - Algebraic Simplification
    - Operator strength reduction
    - Dead code elimination

Constant Folding

- As the name suggests, this technique involves folding the constants by evaluating the expressions that involves the operands having constant values at the compile time.

- Example:
  - Circumference of circle = (22/7) x Diameter
  - Replace (22/7) with 3.14

during compile time and save the execution time.

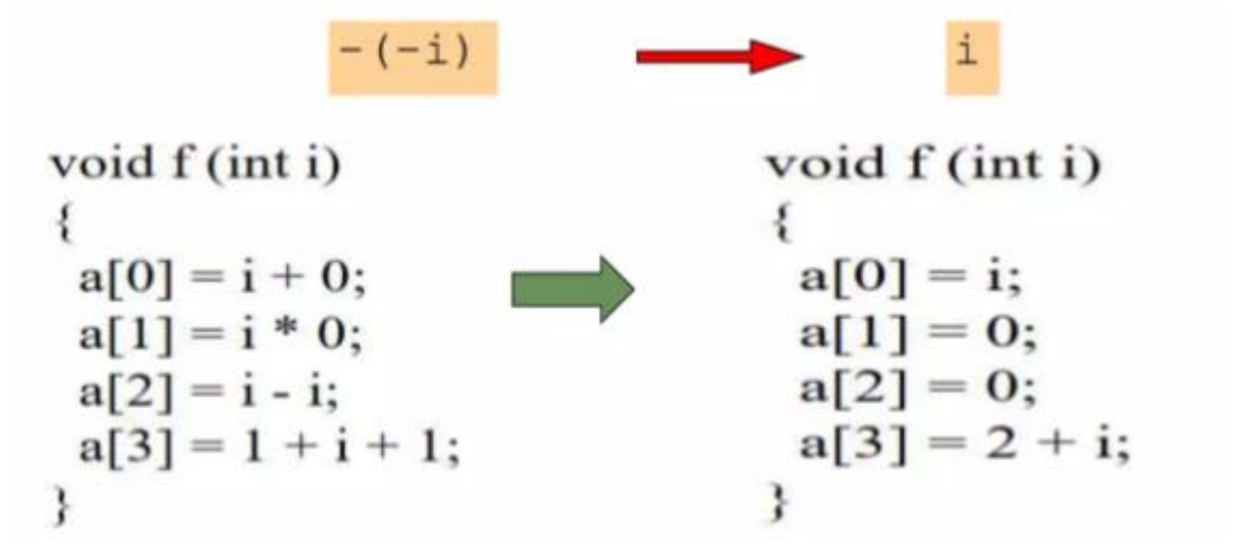
Constant Propagation

- In this technique, if some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program wherever it has been used during compilation, provided that its value does not get alter in between.
- Example:
  - pi = 3.14, radius = 10
  - Area of circle = pi x radius x radius

  Substitutes constants value during compile time and saves the execution time.

Algebraic Simplification

- Use algebraic properties to simplify expressions
- Some expressions can be simplified by replacing them with an equivalent expression that is more efficient



```
-(-i)
```

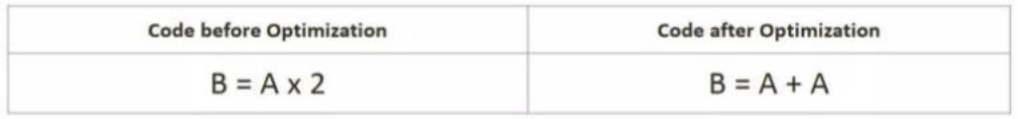
```
i
```

```
void f (int i)
{
  a[0] = i + 0;
  a[1] = i * 0;
  a[2] = i - i;
  a[3] = 1 + i + 1;
}
```

```
void f (int i)
{
  a[0] = i;
  a[1] = 0;
  a[2] = 0;
  a[3] = 2 + i;
}
```

Strength Reeducation

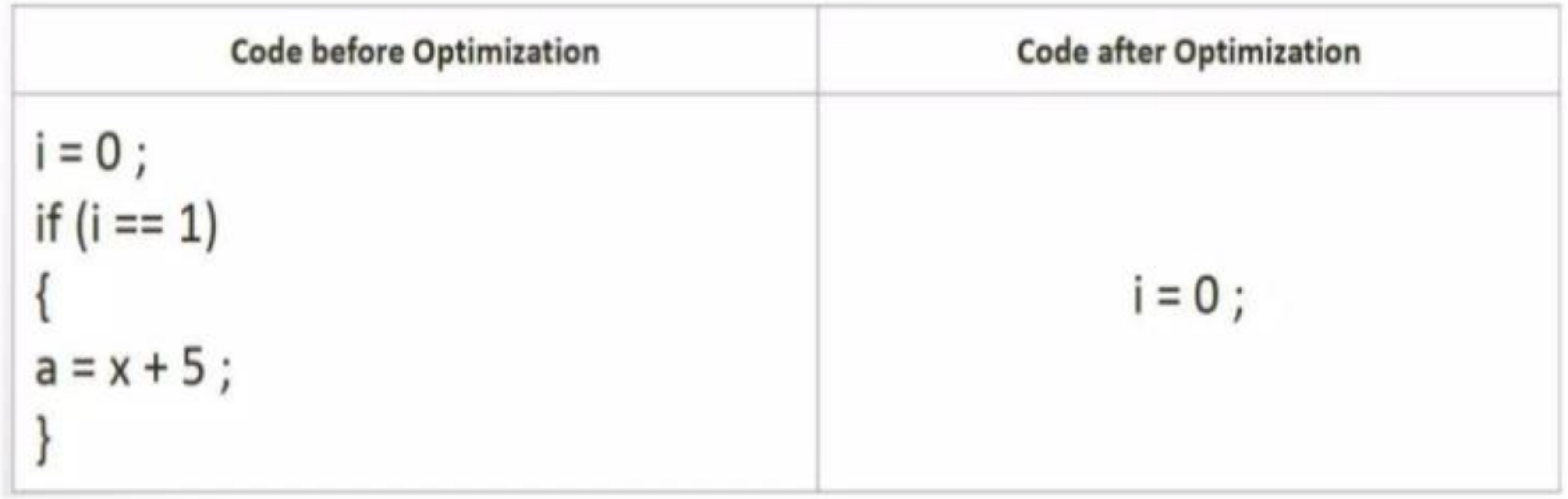
- As the name suggests, this technique involves reducing the strength of the expressions by replacing the expensive and costly operators with the simple and cheaper ones
- Improved Execution Speed
  - Expensive operations, such as multiplication and division, are replaced with simpler ones like addition or bitwise shifts, which execute faster.
  - Example:
  - Instead of X * 2, use X + X, which is computationally cheaper.



Code before Optimization	Code after Optimization
B = A x 2	B = A + A

Dead code Elimination

- Those code are eliminated which either never executes or are not reachable or even if they get execute, their output is never utilized.

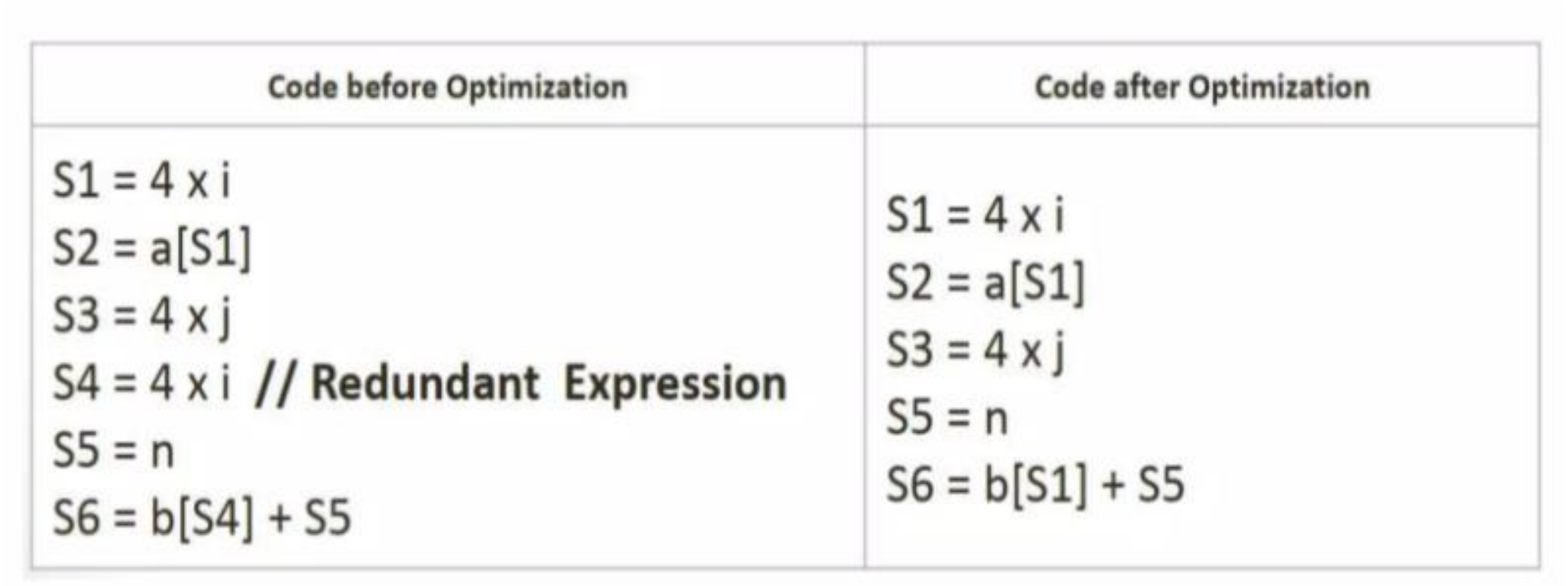


Code before Optimization	Code after Optimization

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

Common sub expression Elimination

- Eliminates the redundant expressions and avoids their computation again and again.



Code before Optimization	Code after Optimization

S1 = 4 x i
S2 = a[S1]
S3 = 4 x j
S4 = 4 x i // Redundant Expression
S5 = n
S6 = b[S4] + S5
	S1 = 4 x i
	S2 = a[S1]
	S3 = 4 x j
	S5 = n
	S6 = b[S1] + S5

Example:

```
• int main() {
•     int a = 5;
•     int b = 10;
•     int c = a * 2;
•     int d = a * 2;  // redundant computation
•     int e = b + c;
•     return e;
• }
```

Solution

```
int main() {
    int a = 5;
    int b = 10;
    int c = a * 2;
    int d = c;
    int e = b + c;
    return e;
}
```

Global Code Optimization

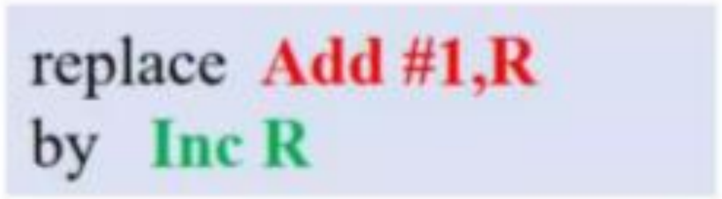
- Optimization across basic blocks within a procedure /function
- Could be restricted to a smaller scope,
  - Example: a loop
- Data-flow analysis is done to perform optimization across basic blocks
- Each basic block is a node in the flow graph of the program.
  - These optimizations can be extended to an entire control - flow graph
- Most of compiler implement global optimizations with well founded theory and practical gains

Peephole optimization

- technique that operates on the one or few instructions at a time.
- Performs machine dependent improvements
- Peeps into a single or sequence of two to three instructions (peephole) and replaces it by most efficient alternative (shorter or faster) instructions.
- Peephole is a small moving window on the target system

Characteristics of Peephole optimization

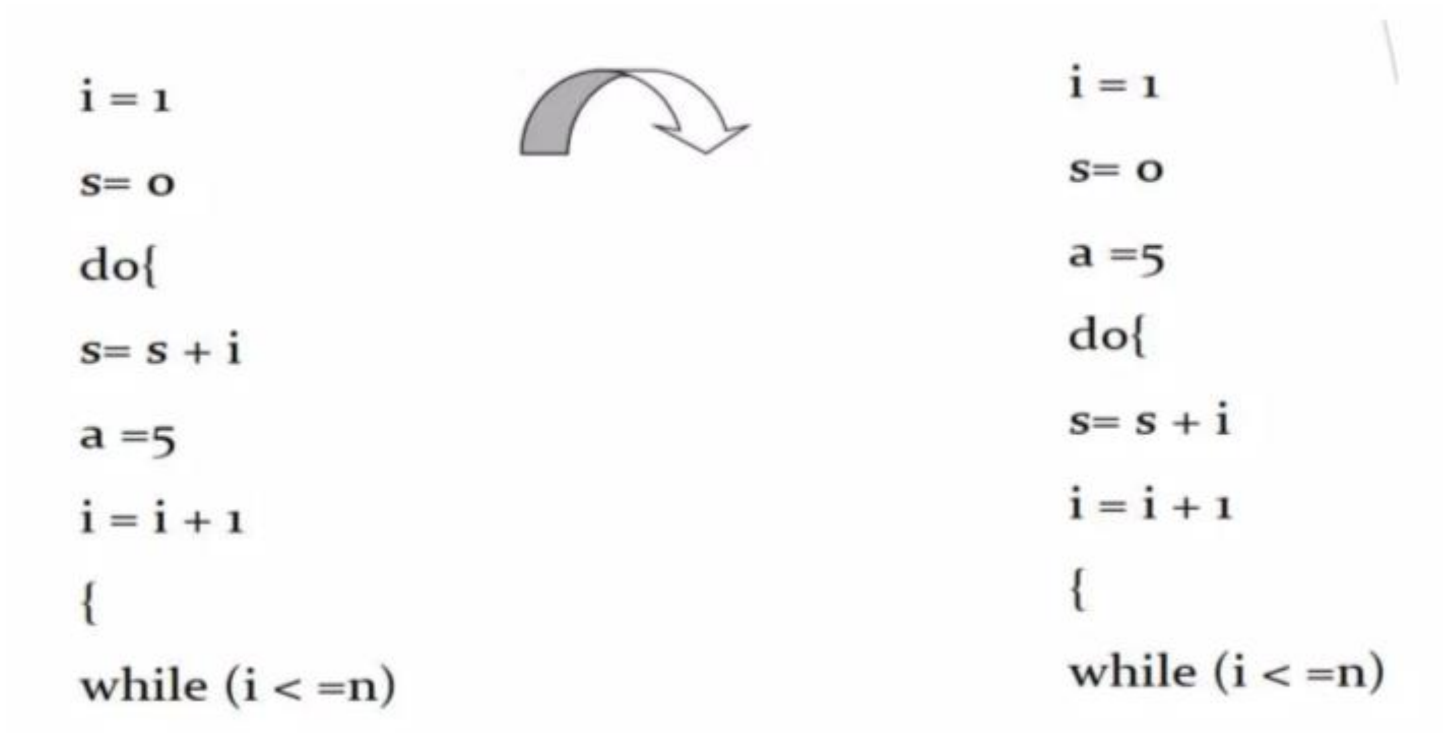
- Redundant-instruction (loads and stores)elimination
- Flow-of-control optimizations
  - Elimination of multiple jumps (i.e. goto statements)
- Elimination of unreachable code
- Algebraic simplifications
- Reducing operator strength
- Use of machine idioms
  - Provide low level code


replace **Add #1,R**
by **Inc R**

Loop Optimization

- optimization plays an important role in improving the performance of the source code by reducing overheads associated with executing loops.
- Loop Optimization can be done by removing:
  - Loop invariant
  - Induction variables

Example: Removal of Loop Data

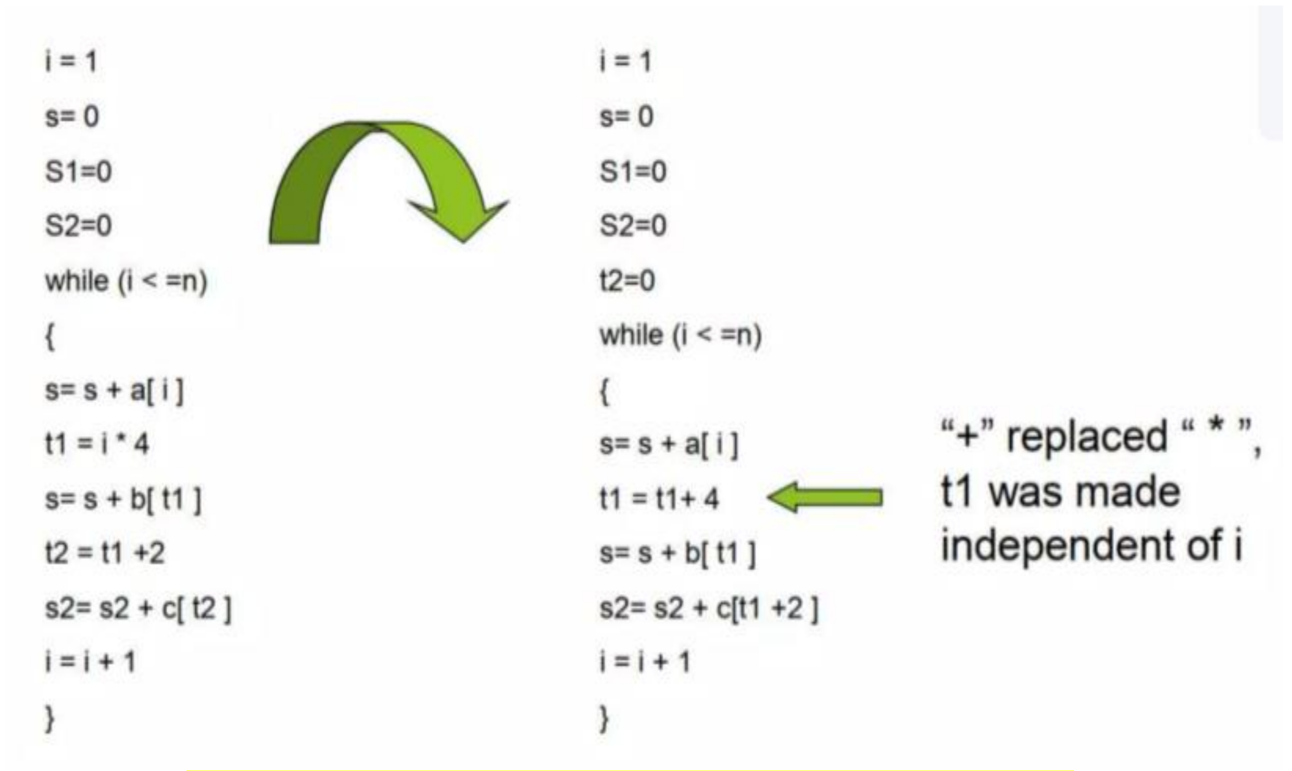


```
i = 1
s= 0
do{
s= s + i
a =5
i = i + 1
{
while (i < =n)
```

```
i = 1
s= 0
a =5
do{
s= s + i
i = i + 1
{
while (i < =n)
```

In this case, the assignment a = 5 is redundant within the loop—it does not change during iterations. By moving it outside the loop, the computation is performed only once instead of repeatedly during each iteration.

Example: Induction Variable



```
i = 1
s= 0
S1=0
S2=0
while (i < =n)
{
s= s + a[ i ]
t1 = i * 4
s= s + b[ t1 ]
t2 = t1 +2
s2= s2 + c[ t2 ]
i = i + 1
}
```

```
i = 1
s= 0
S1=0
S2=0
t2=0
while (i < =n)
{
s= s + a[ i ]
t1 = t1+ 4
s= s + b[ t1 ]
s2= s2 + c[t1 +2 ]
i = i + 1
}
```

“+” replaced “ * ”, t1 was made independent of i

Instead of performing multiplication (t1 = i * 4), which is relatively expensive, the optimized code uses addition (t1 = t1 + 4), which is faster.

Example:

```
int main() {
    int sum = 0;
    int x = 10;
    for (int i = 0; i < 1000; i++) {
        sum += x * 2;  // x*2 is computed in every loop iteration
    }
    return sum;
}
```

Solution:

```
int main() {
    int sum = 0;
    int x = 10;
    int temp = x * 2;  // moved out of the loop
    for (int i = 0; i < 1000; i++) {
        sum += temp;
    }
    return sum;
}
```