# Chapter # 12
# Compiler Code Optimizations

Dr. Shaukat Ali
Department of Computer Science
University of Peshawar

# Compiler Code Optimizations

- Introduction
  - Optimized code
    - Executes faster
    - efficient memory usage
    - yielding better performance.
  - Compilers can be designed to provide code optimization.
  - Users should only focus on optimizations not provided by the compiler such as choosing a faster and/or less memory intensive algorithm.

# Compiler Code Optimizations

- A Code optimizer sits between the front end and the code generator.
  - Works with intermediate code.
  - Can do control flow analysis.
  - Can do data flow analysis.
  - Does transformations to improve the intermediate code.

# Compiler Code Optimizations

- Optimizations provided by a compiler includes:
  - Inlining small functions
  - Code hoisting
  - Dead store elimination
  - Eliminating common sub-expressions
  - Loop unrolling
  - Loop optimizations: Code motion, Induction variable elimination, and Reduction in strength.

# Compiler Code Optimizations

- Inlining small functions
  - Repeatedly inserting the function code instead of calling it, saves the calling overhead and enable further optimizations.
  - Inlining large functions will make the executable too large.

# Compiler Code Optimizations

- Code hoisting
  - Moving computations outside loops
  - Saves computing time

# Compiler Code Optimizations

- Code hoisting
  - In the following example (2.0 * PI) is an invariant expression there is no reason to recompute it 100 times.

```
DO I = 1, 100
        ARRAY(I) = 2.0 * PI * I
ENDDO
```

  - By introducing a temporary variable 't' it can be transformed to:

```
t = 2.0 * PI
DO I = 1, 100
        ARRAY(I) = t * I
END DO
```

# Compiler Code Optimizations

- Dead store elimination
  - If the compiler detects variables that are never used, it may safely ignore many of the operations that compute their values.

# Compiler Code Optimizations

- Eliminating common sub-expressions
  - Optimization compilers are able to perform quite well:

```
X = A * LOG(Y) + (LOG(Y) ** 2)
```

  - Introduce an explicit temporary variable t:

```
t = LOG(Y)
X = A * t + (t ** 2)
```

  - Saves one 'heavy' function call, by an elimination of the common sub-expression LOG(Y), the exponentiation now is:

```
X = (A + t) * t
```

# Compiler Code Optimizations

- Loop unrolling
  - The loop exit checks cost CPU time.
  - Loop unrolling tries to get rid of the checks completely or to reduce the number of checks.
  - If you know a loop is only performed a certain number of times, or if you know the number of times it will be repeated is a multiple of a constant you can unroll this loop.

# Compiler Code Optimizations

- Loop unrolling
  - Example:

```
// old loop
for(int i=0; i<3; i++) {
        color_map[n+i] = i;
}
// unrolled version
int i = 0;
colormap[n+i] = i;
i++;
colormap[n+i] = i;
i++;
colormap[n+i] = i;
```

# Compiler Code Optimizations

- Code Motion
  - Any code inside a loop that always computes the same value can be moved before the loop.
  - Example:

```
while (i <= limit-2)
do {loop code}
```

where the loop code doesn't change the limit variable. The subtraction, limit-2, will be inside the loop. Code motion would substitute:

```
t = limit-2;
while (i <= t)
do {loop code}
```

# Compiler Code Optimizations

- Conclusion
  - Compilers can provide some code optimization.
  - Programmers do have to worry about such optimizations.
  - Program definition must be preserved.

- End of Chapter # 12