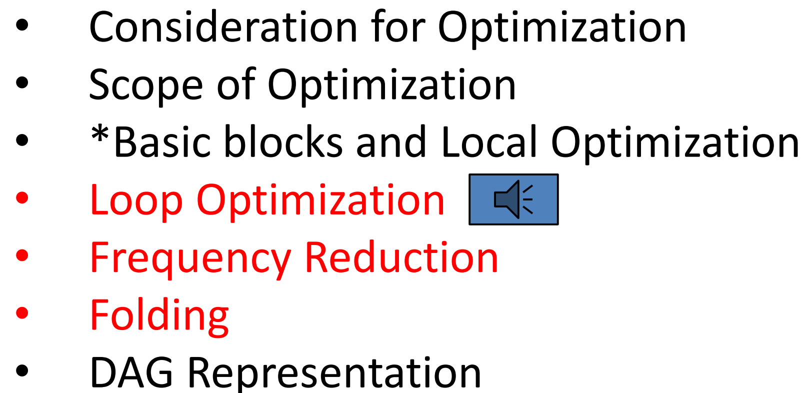
Unit - IV

Code Optimization | Data Flow Analysis

Code Optimization

- Consideration for Optimization
- Scope of Optimization
- *Basic blocks and Local Optimization
- Loop Optimization 
- Frequency Reduction
- Folding
- DAG Representation

Data Flow Analysis

- Flow Graph
- Data Flow Equation
- Global Optimization
- Redundant Sub Expression Elimination
- Induction Variable Elements
- Live Variable Analysis
- Copy Propagation

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

Code Optimization Classification

- By Scope:
  - Local Optimization: within a single basic block.
  - Peephole Optimization : on a window of instructions (usually local)
  - Loop-level Optimization : on one or more loops or loop nests.
  - Global: for an entire procedure
  - Interprocedural: across multiple procedures or whole program.
- By machine information used:
  - Machine-independent versus machine-dependent.
- By effect on program structure:
  - Algebraic transformations (e.g., `x+0`, `x*1`, `3*z*4`, …)
  - Reordering transformations (change the order of 2 computations)
    - Loop transformations: loop-level reordering transformations.

Machine - Independent Optimizations

Loop Optimization:

- Optimizations has to be done within loops especially within inner loops.
- The running time of a program may be improved, if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

Machine - Independent Optimizations

Loop Optimization:

- There are five techniques for loop optimizations. These are:
  1. Code motion or frequency reduction
  2. Induction variable elimination
  3. Reduction in strength
  4. Loop unrolling
  5. Loop jamming

Loop Optimization

A Running Example (Quicksort)

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Fig: C code for quicksort

Loop Optimization

A Running Example (Quicksort)

```
(1)   i = m-1
(2)   j = n
(3)   t1 = 4*n
(4)   v = a[t1]
(5)   i = i+1
(6)   t2 = 4*i
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
(9)   j = j-1
(10)  t4 = 4*j
(11)  t5 = a[t4]
(12)  if t5>v goto (9)
(13)  if i>=j goto (23)
(14)  t6 = 4*i
(15)  x = a[t6]
(16)  t7 = 4*i
(17)  t8 = 4*j
(18)  t9 = a[t8]
(19)  a[t7] = t9
(20)  t10 = 4*j
(21)  a[t10] = x
(22)  goto (5)
(23)  t11 = 4*i
(24)  x = a[t11]
(25)  t12 = 4*i
(26)  t13 = 4*n
(27)  t14 = a[t13]
(28)  a[t12] = t14
(29)  t15 = 4*n
(30)  a[t15] = x
```

Fig: Three- address code for above C fragment

Loop Optimization

A Running Example (Quicksort)

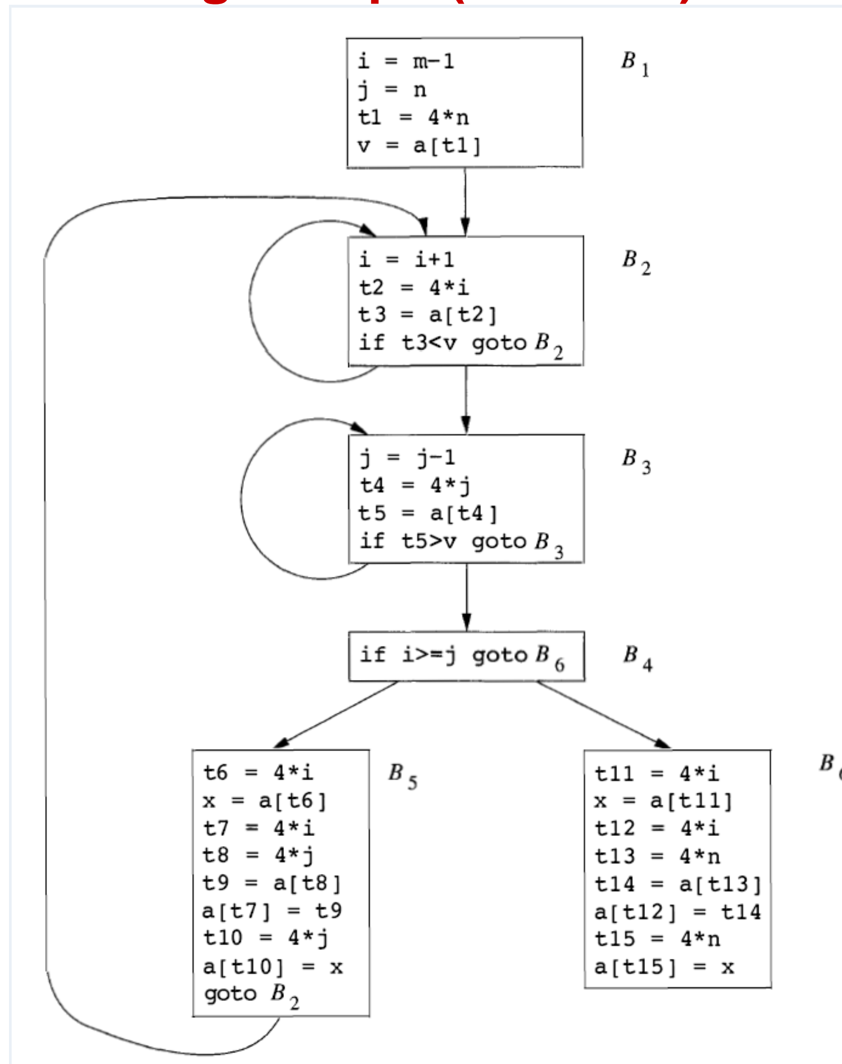


Fig: Flowgraph for above three-address code

Loop Optimization

1. Code Motion or Frequency Reduction

- An important modification that decreases the amount of code in a loop
- *Loop-invariant computation*
  - An expression that yields the same result independent of the number of times a loop is executed
- Code Motion takes loop-invariant computation before its loop

```
while (i <= limit-2)
```

```
t = limit -2  while
(i <= t)
```

2. Induction Variables and Reduction in Strength

- Induction variable
  - For an induction variable x, there is a positive or negative constant c such that each time x is assigned, its value increases by c
- Induction variables can be computed with a single increment (addition or subtraction) per loop iteration
- Strength reduction
  - The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition
- Induction variables lead to
  - strength reduction
  - eliminate computation

Loop Optimization

3. Strength Reduction

- Replace expensive operations with simpler ones
- Example: Multiplications replaced by additions

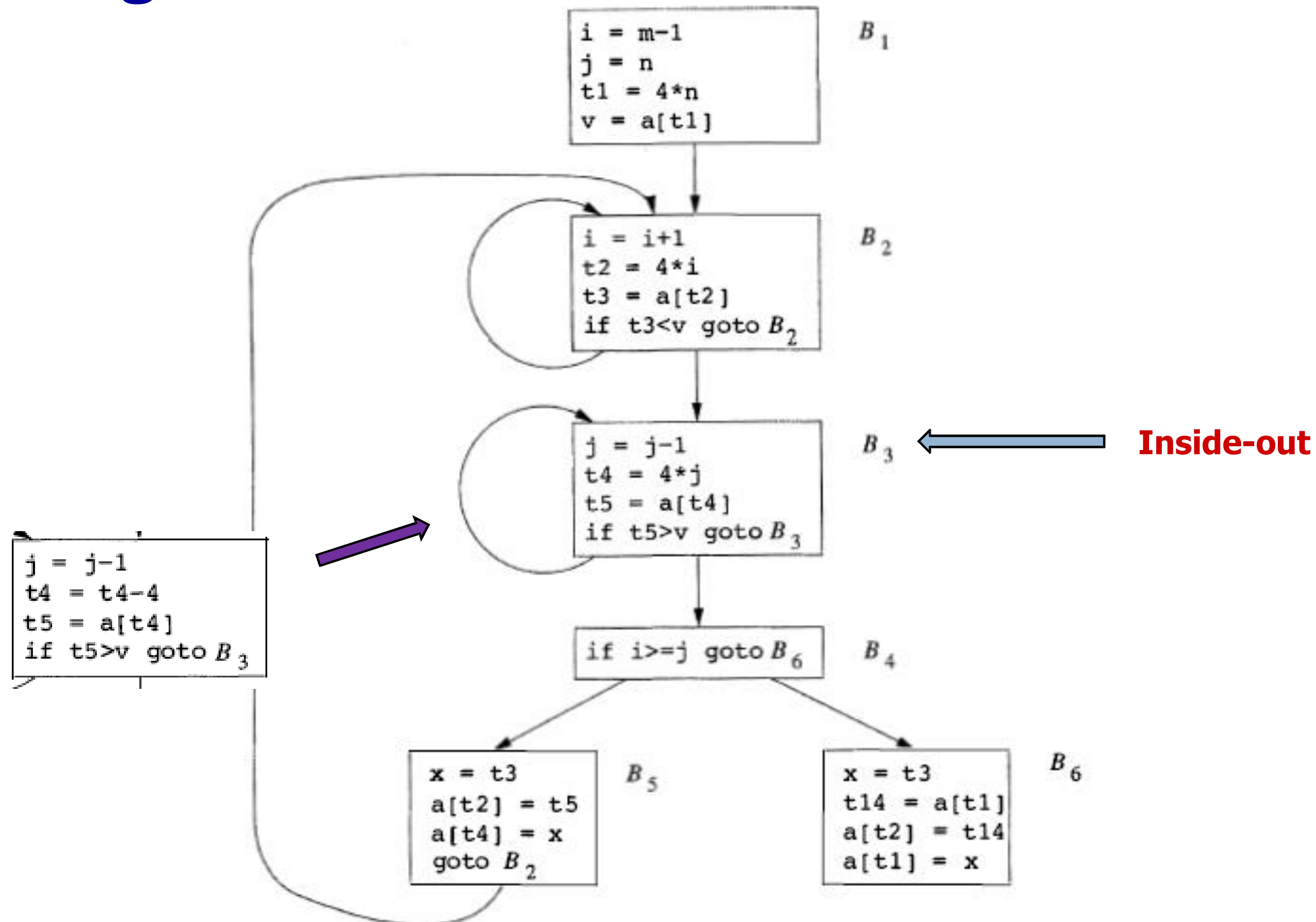
```
y := x * 2
```

```
y := x + x
```

Peephole optimizations are often strength reductions

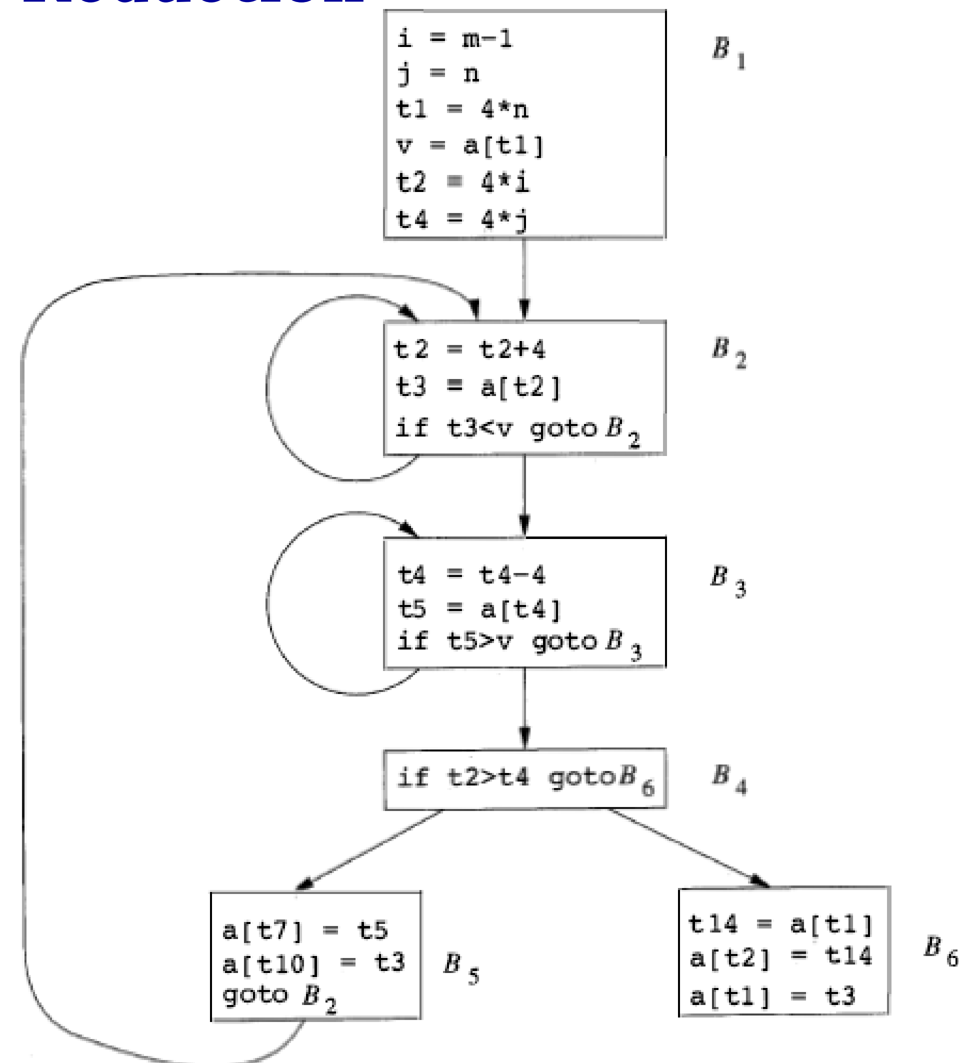
Loop Optimization

Strength Reduction



Loop Optimization

Strength Reduction



Loop Optimization

4. Loop Unrolling

- Loop unrolling involves replicating the body of the loop to reduce the number of tests required to be carried out if the number of iterations are constant.
- Ex:

```
i = 1
while (i <=100)
      {
        x[i] =0;
        i++
      }
```

Loop Optimization

4. Loop Unrolling

Ex:

```
i = 1
while (i <=100)
      {
         x[i] =0;
         i++
      }
```

In this case the test `i <=100` will be performed 100 times, but if the body of the loop is replicated, then the number of times this test need to be performed will be 50 (i.e., 100/2 = 50).

Loop Optimization

4. Loop Unrolling

Ex:

The loop after replication of body will be:

```
i = 1
while (i <=50)
      {
        x[i] =0;
        i++
        x[i] =0;
        i++
      }
```

5. Loop Jamming

- This is the technique of merging the bodies of the two loops, if the two loops have the same number of iterations and then uses the same indices.
- This eliminates the test of one loop.

Ex:

```
{
for (i = 0; i < 10; i++)
for (j = 0; j < 10; j++)
      x[i, j] = 0;
for (i = 0; i < 10; i++)
      x[i, j] = 1;
}
```

Loop Optimization

5. Loop Jamming

Ex:

```
{
    for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
          x[i, j] = 0;
    for (i = 0; i < 10; i++)
          x[i, j] = 1;
}
```

Here the bodies of the loops on `i` can be concatenated. The result of loop jamming will be:

Loop Optimization

5. Loop Jamming

Ex:

Here the bodies of the loops on i can be concatenated. The result of loop jamming will be:

```
{
  for (i = 0; i < 10; i++)
      {
         for (j = 0; j < 10; j++)
          x[i, j] = 0;
          x[i, j] = 1;
      }
}
```

Folding

Folding

- Constant folding is one of the common example of function-preserving (or semantic-preserving) transformation.
- Deducing at compile time that the value of an expression is a constant and using the constant instead
- Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.
- Terms in constant expressions are typically simple literals, such as the integer literal, but they may also be variables whose values are known at compile time.

Folding

Folding

Ex-1:

Before optimization:

```
tmp=5*3+8-12/2
```

After optimization:

```
tmp=17
```

Folding

Folding

Ex-2:

In the code fragment below, the expression `(6 + 4)` can be evaluated at compile time and replaced with the constant `10`.

```
int f (void)
{
  return 6 + 4;
}
```

Below is the code fragment after constant folding:

```
int f (void)
{
  return 10;
}
```

Folding

When is Constant Folding Applied in Compiler Design

Constant Folding is applied:

- During the Intermediate Code Generation phase of the compiler, which generates an intermediate representation of source code.
- After other optimizations that generate constant expressions, which can be eliminated by constant folding.

Folding

Advantages of Constant Folding

- Constant Folding is used to decrease the execution time.
- Constant Folding optimizes the code.
- Constant Folding also reduces Lines of Code.
- Constant Folding helps to avoid redundant computations in the code, hence making it more efficient.
- Constant Folding also reduces power consumption.
- Constant Folding also helps in efficient memory management.
- Constant Folding makes hardware usage more efficient.

Folding

Constant Propagation in Compiler Design

- Constant propagation is a local optimization technique that substitutes the values of variables and expressions whose values are known beforehand.

Folding

Constant Propagation

- Constant propagation is the process of substituting the values of known constants in expressions at compile time.
- Such constants also include intrinsic functions applied to constant values.

Ex: Consider the following pseudocode:

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

Folding

Constant Propagation

Ex: Consider the following pseudocode:

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

Propagating `x` yields:

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

Folding

Constant Propagation

Ex:

Propagating `x` yields:

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

Continuing to propagate yields the following (which would likely be further optimized by dead-code elimination of both `x` and `y`:

```
int x = 14;
int y = 0;
return 0;
```

Folding

Constant Propagation

- Constant propagation is implemented in compilers using reaching definition analysis results.
- If all variable's reaching definitions are the same assignment which assigns a same constant to the variable, then the variable has a constant value and can be replaced with the constant.
- Constant propagation can also cause conditional branches to simplify to one or more unconditional statements, when the conditional expression can be evaluated to true or false at compile time to determine the only possible outcome.

Folding

Difference between Constant Propagation and Constant Folding

Or

Are constant folding and constant propagation the same?

- No, constant folding and constant propagation are not the same, but they are related compiler optimization techniques.
- Constant propagation replaces the bound variable with a constant expression it is bound to.
- On the other hand, constant folding evaluates the expression with all compile-time inputs.
- In Constant Propagation, the variable is substituted with its assigned constant where as in Constant Folding, the variables whose values can be computed at compile time are considered and computed.