THEORY OF COMPUTATION AND COMPILERS

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

Local Optimization or Local Transformation:

- A transformation of a program is called "local" if it can be performed by looking only at the statements in a "basic block". Otherwise it is called "global".
- These optimizations can be done easily since we do not consider any control flow information.
- There are two important classes of local transformations that can be applied to basic blocks. These are:
  1. Structure-preserving transformations
  2. Algebraic transformations

Machine - Independent Optimizations

Local Optimization or **Local Transformation**:

What are different types of local optimization techniques?

- The different types of local optimization techniques are:
  1. Local Common Subexpressions
  2. Dead-Code Elimination
  3. Use of Algebraic Identities

Local 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

Local 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

Local Optimization

A Running Example (Quicksort)

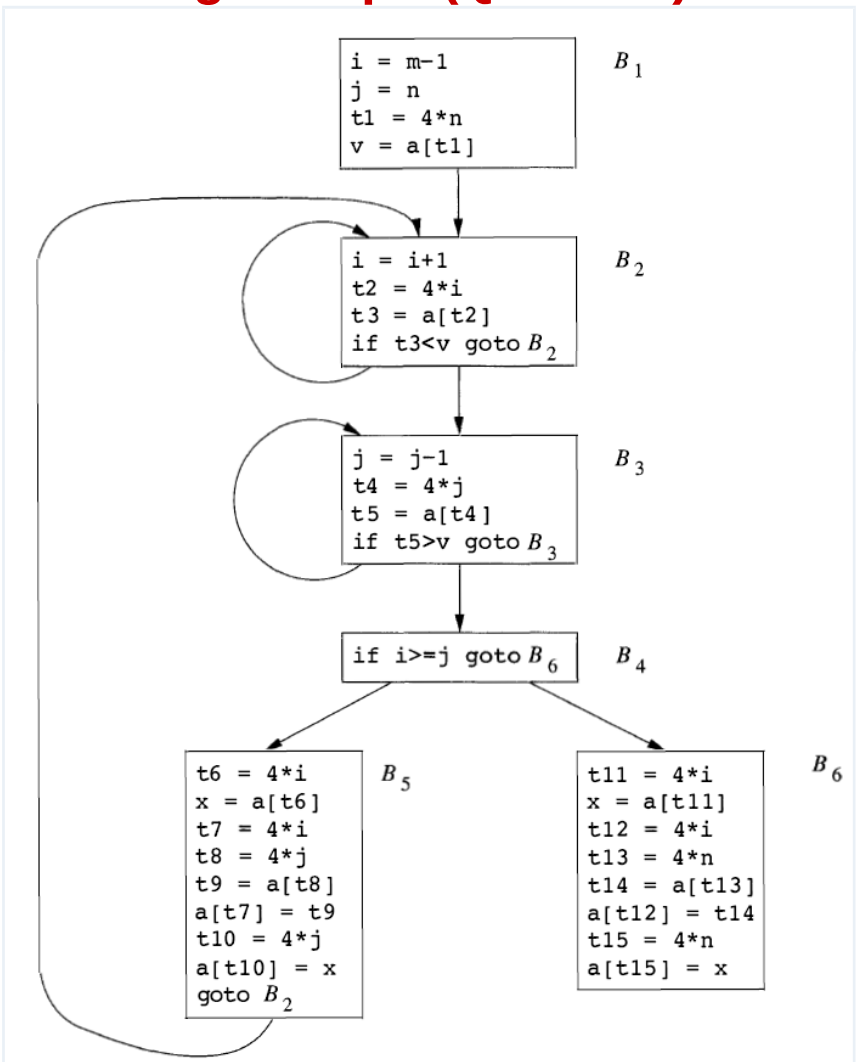


Fig: Flowgraph for above three-address code

Local Optimization

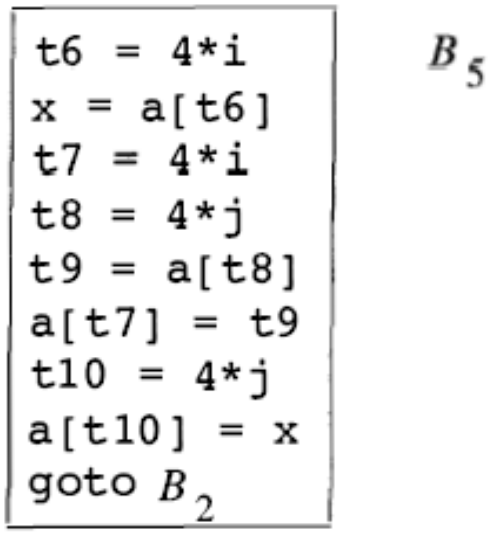
1. Common Subexpression Elimination

- An occurrence of an expression E is called a *common subexpression* if E was previously computed and the values of the variables in E have not changed since the previous computation. We avoid recomputing E if we can use its previously computed value; that is, the variable x to which the previous computation of E was assigned has not changed in the interim.
- An expression, say x+y, is redundant iff along every path from the procedure's entry it has been evaluated and its constituent subexpressions (x, y) have not been redefined.

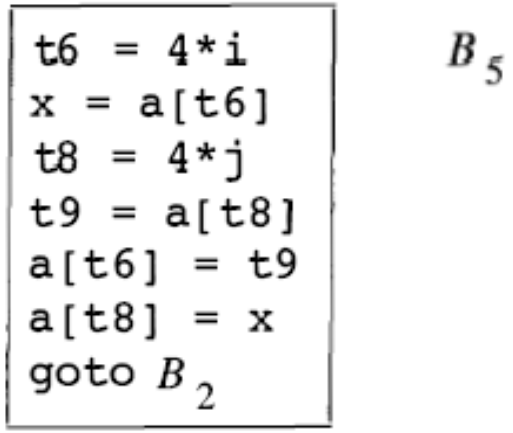
Local Optimization

1. Common Subexpression Elimination

Local:



(a) Before.



(b) After.

Local Optimization

2. Dead-Code Elimination

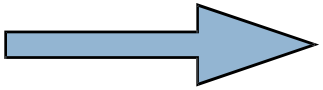
- *Live variable*
  - A variable is live at a point in a program if its value can be used subsequently;
  - otherwise, it is dead at that point.
- *Constant folding*
  - Deducing at compile time that the value of an expression is a constant and using the constant instead

Local Optimization

2. Dead-Code Elimination

- Remove unnecessary code
- e.g. variables assigned but never read

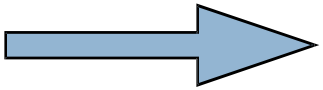
```
b := 3
c := 1 + 3
d := 3 + c
```



```
c := 1 + 3
d := 3 + c
```

- Remove code never reached

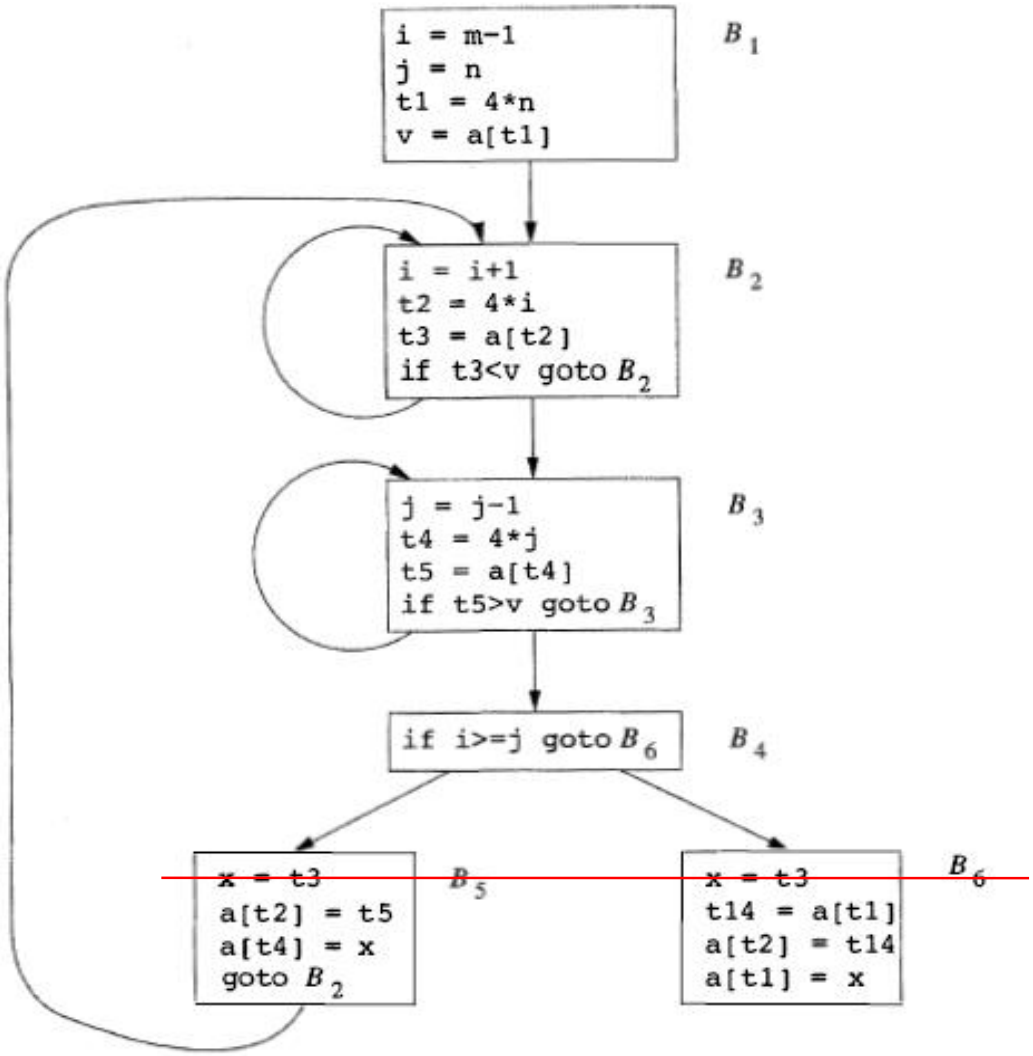
```
if (false)
{a := 5}
```



```
if (false)
{}
```

Local Optimization

2. Dead-Code Elimination



Local Optimization

3. Use of Algebraic Identities

- Algebraic Properties

```
x + 0 = 0 + x  = x
x - 0 = x
x * 1 = 1 * x = x
x / 1 = x
```

- Strength Reduction

```
x^2 = x * x
2 * x = x + x
x / 2 = x * 0.5
```

- Constant Folding

```
2 * 3.14 =  6.28
```

Local Optimization

3. Use of Algebraic Identities

- Commutativity and Associativity

  - DAG construction can help us here.
  - Apart from checking *left op right*, we could also check *right op left* for commutativity.

  E.g.,

  1. (a + b) + (b + a)
  2. a = b + c; e = c + d + b

- Some Algebraic Laws are not obvious

  E.g., Can we optimize

  If (x > y)a = b + x + c – y ?

  However, we need to worry about underflows.