Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1402-1403

Local Optimization

- **The Use of Algebraic Identities**
  - Arithmetic identities can be applied to **eliminate computations** from a basic block

$$\begin{array}{ll} x+0=0+x=x & x-0=x \\ x\times 1=1\times x=x & x/1=x \end{array}$$

  - **Local reduction in strength**, that is, replacing a more expensive operator by a cheaper one

EXPENSIVE		CHEAPER
x^2	$=$	$x \times x$
$2 \times x$	$=$	$x + x$
$x/2$	$=$	$x \times 0.5$

  - **Constant folding**, that is, evaluating constant expressions at compile time and replace the constant expressions by their values
    - The expression $\mathbf{2 * 3.14}$ would be replaced by $\mathbf{6.28}$

Local Optimization

- **The Use of Algebraic Identities**
  - We can apply algebraic transformations such as **commutativity and associativity**
    - For example, $*$ is commutative; that is, $\boldsymbol{x} * \boldsymbol{y} = \boldsymbol{y} * \boldsymbol{x}$
      - Before we create a new node labeled $*$ with left child $\boldsymbol{M}$ and right child $\boldsymbol{N}$, we always check whether such a node already exists
      - However, because $*$ is commutative, we should then check for a node having operator $*$, left child $\boldsymbol{N}$, and right child $\boldsymbol{M}$
  - The **relational operators** sometimes generate unexpected common subexpressions
    - For example, the condition $\boldsymbol{x} > \boldsymbol{y}$ can also be tested by subtracting
  - **Associative laws** might also be applicable to expose common subexpressions

```
a = b + c;
e = c + d + b;
```

→

```
a = b + c
t = c + d
e = t + b
```

If t is not needed outside this block →

```
a = b + c
e = a + d
```

Local Optimization

- **Representation of Array References**
  - We might be tempted to optimize by replacing the third instruction $\boldsymbol{z} = \boldsymbol{a}[\boldsymbol{i}]$ by the simpler $\boldsymbol{z} = \boldsymbol{x}$
  - However, since $\boldsymbol{j}$ could equal $\boldsymbol{i}$, the middle statement may in fact change the value of $\boldsymbol{a}[\boldsymbol{i}]$; thus, it is not legal to make this change

```
x = a[i]
a[j] = y
z = a[i]
```

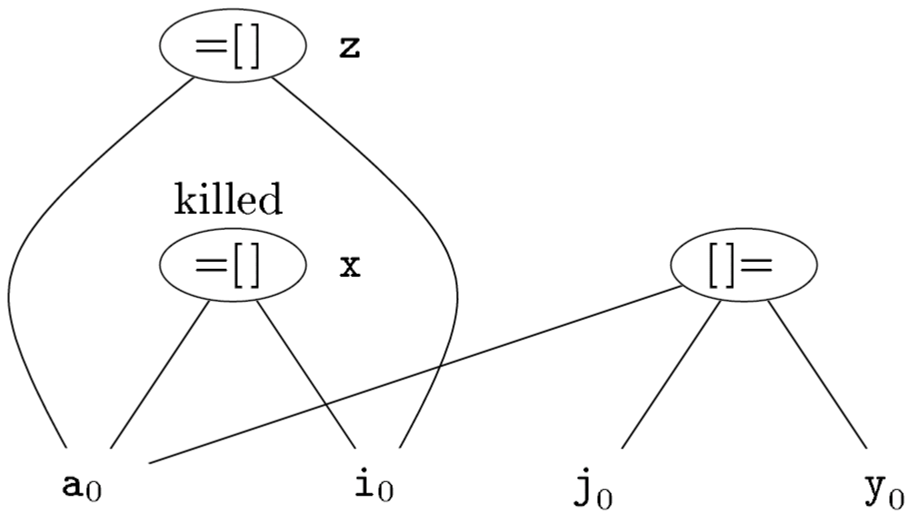
Local Optimization

- **Representation of Array References**
  - The proper way to represent array accesses in a DAG is as follows
    - An assignment from an array, like $\boldsymbol{x} = \boldsymbol{a}[\boldsymbol{i}]$, is represented by creating a node with operator $=[\,]$ and two children representing the initial value of the array, a0 in this case, and the index $\boldsymbol{i}$
      - Variable $\boldsymbol{x}$ becomes a label of this new node
    - An assignment to an array, like $\boldsymbol{a}[\boldsymbol{j}] = \boldsymbol{y}$, is represented by a new node with operator $[\,]=$ and three children representing $\boldsymbol{a0}$, $\boldsymbol{j}$ and $\boldsymbol{y}$
      - There is no variable labeling this node
      - The creation of this node kills all currently constructed nodes whose value depends on $\boldsymbol{a0}$

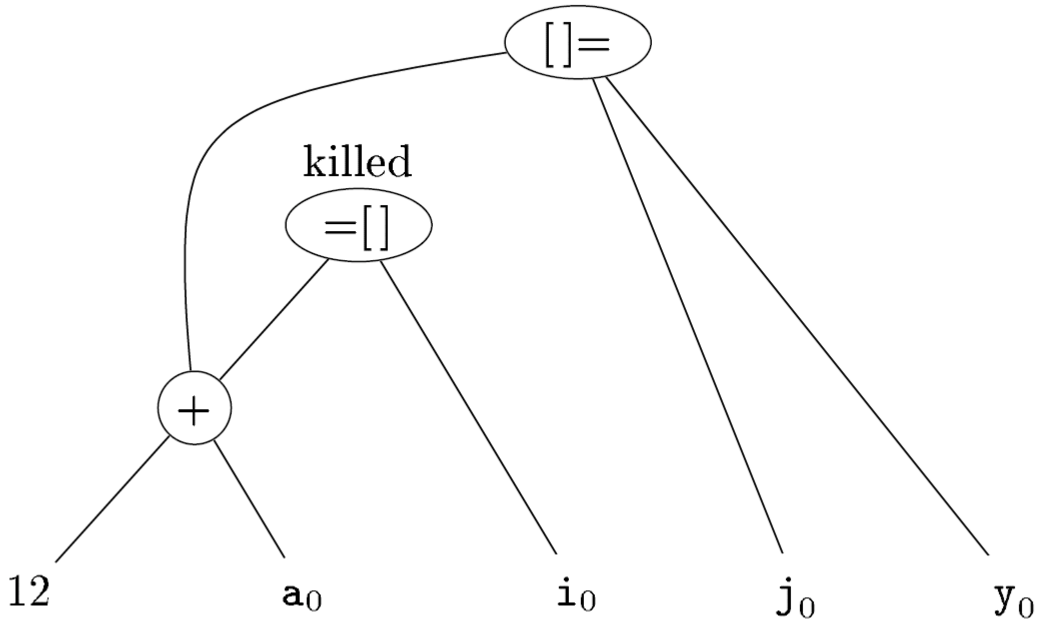
Local Optimization

- **Representation of Array References**
  - **Example**

```
x = a[i]
a[j] = y
z = a[i]
```




```
b = 12 + a
x = b[i]
b[j] = y
```



Local Optimization

- **Reassembling Basic Blocks From DAGs**
  - For each node that has one or more attached variables, we construct a three-address statement that computes the value of one of those variables
  - If we do not have global live-variable information to work from, we need to assume that every variable of the program ***(but not temporaries that are generated by the compiler to process expressions)*** is live on exit from the block
  - **Example**

```
a = b + c
b = a - d
c = b + c
d = a - d
```

If b is not live on exit from the block

```
a = b + c
d = a - d
c = d + c
```

If both b and d are live on exit

```
a = b + c
d = a - d
b = d
c = d + c
```

Global Optimization

- Most global optimizations are based on data-flow analyses
- A compiler optimization must preserve the semantics of the original program
- Except in very special circumstances, the compiler cannot understand enough about the program to replace it with a substantially different and more efficient algorithm
- ***A compiler knows only how to apply relatively low-level semantic transformations***

Global Optimization

- **Causes of Redundancy**
  - Sometimes the redundancy is available at the **source level**
    - For instance, a programmer may find it more direct and convenient to recalculate some result
  - But more often, the redundancy is a side effect of **having written the program in a high-level language**

Global Optimization

- **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);
}
```

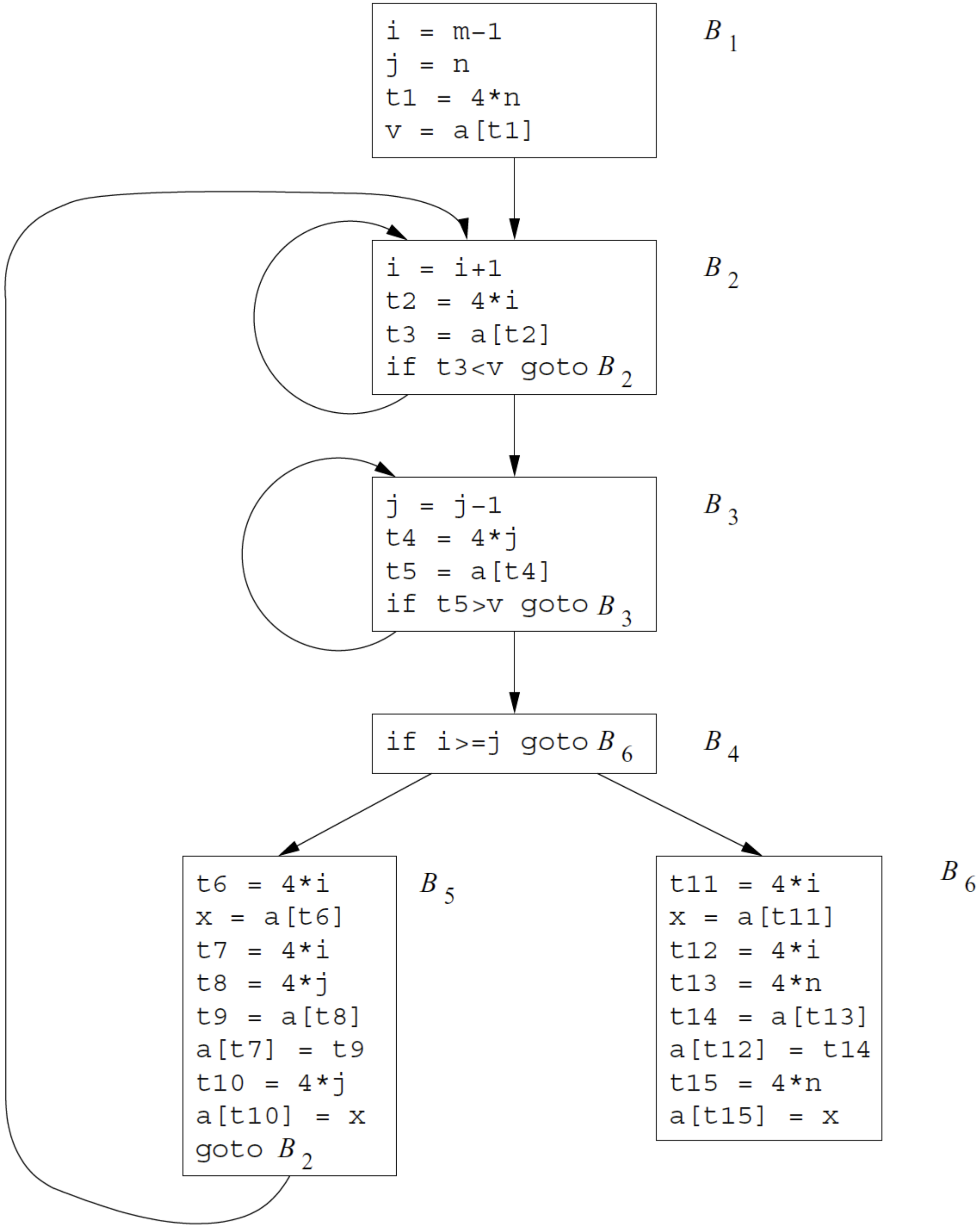
Global Optimization

- **Three-address code for the quicksort fragment**

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

Global Optimization

- **Flow graph for the quicksort fragment**



Global Optimization

- **Semantics-Preserving Transformations**
  - A program will include several calculations of the same value
  - **Example:** Local common-subexpression elimination

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

B_5

(a) Before.

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

B_5

(b) After.

Global Optimization

- **Global Common Subexpressions**
  - 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

Global Optimization

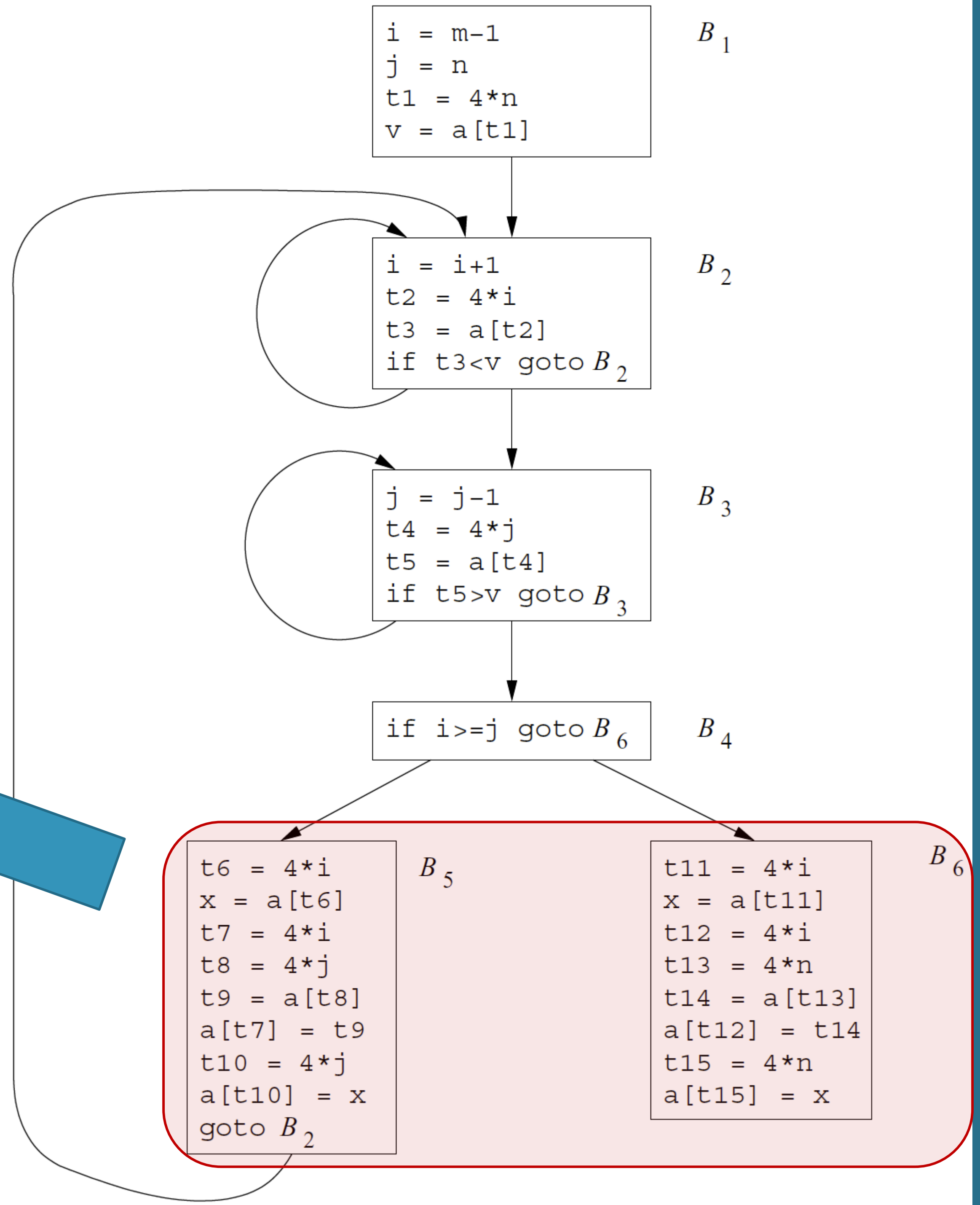
- **Example:** Eliminating both global and local common subexpressions from blocks B5 and B6

```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```

B_5

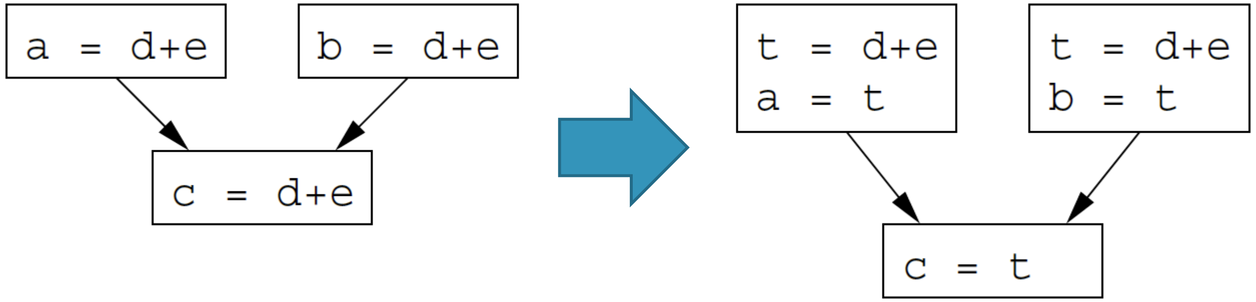
```
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
```

B_6



Global Optimization

- **Copy Propagation**
  - The idea behind the copy-propagation transformation is to use v for u, wherever possible after the copy statement u = v
  - **Example**
    - Since control may reach c = d+e either after the assignment to a or after the assignment to b, it would be incorrect to replace c = d+e by either c = a or by c = b



Global Optimization

- **Copy Propagation**

```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```

Basic block B5 after copy propagation

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

Global Optimization

- **Dead-Code Elimination**
  - A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point
  - **Example:**
    - It may be possible for the compiler to deduce that each time the program reaches this statement, the value of debug is FALSE
    - We can eliminate both the test and the print operation from the object code

```
if (debug) print ...
```

  - Deducing at compile time that the value of an expression is a constant and using the constant instead is known as ***constant folding***
  - One advantage of copy propagation is that it often turns the copy statement into dead code

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

$\Rightarrow$

```
a[t2] = t5
a[t4] = t3
goto B2
```

Global Optimization

- **Code Motion**
  - Loops are a very important place for optimizations
    - Especially the inner loops where programs tend to spend the bulk of their time
    - 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
  - An important modification that decreases the amount of code in a loop is code motion
    - This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and evaluates the expression before the loop
  - **Example:** Evaluation of limit-2 is a loop-invariant computation

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

n+1 calculations of limit-2

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

1 calculations of limit-2

Global Optimization

- **Induction Variables and Reduction in Strength**
  - Another important optimization is to find induction variables in loops and optimize their computation
  - A variable x is said to be an **"induction variable"** if there is a positive or negative constant c such that each time x is assigned, its value increases by c
  - **Example:** i and t2 are induction variables in the loop containing B2

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

B_2

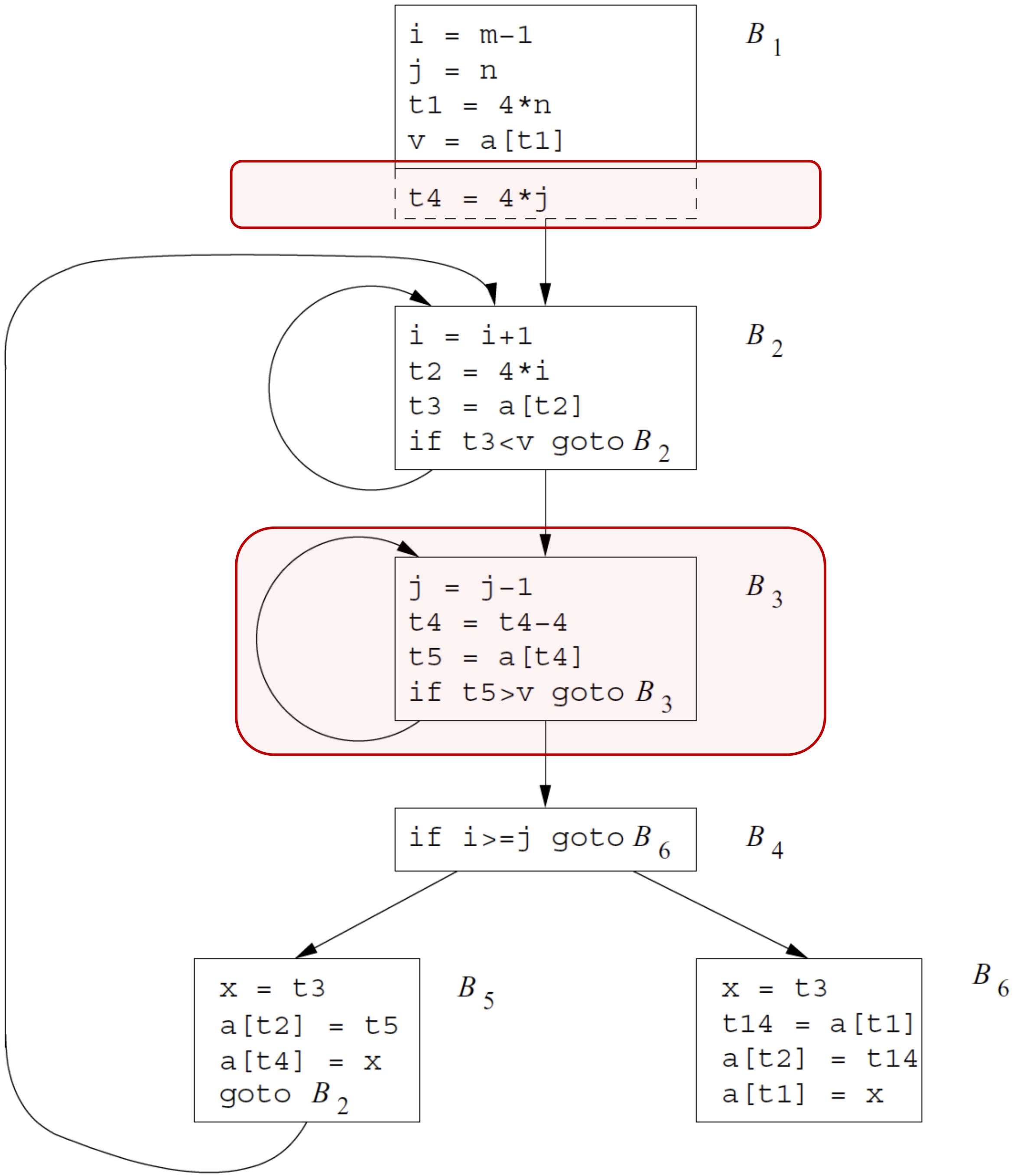
  - The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as **"strength reduction"**

Global Optimization

- **Induction Variables and Reduction in Strength**
  - When processing loops, it is useful to work **"inside-out"**; that is, we shall start with the inner loops and proceed to progressively larger, surrounding loops
  - When there are two or more induction variables in a loop, it may be possible to get rid of all but one
  - However, we can illustrate reduction in strength

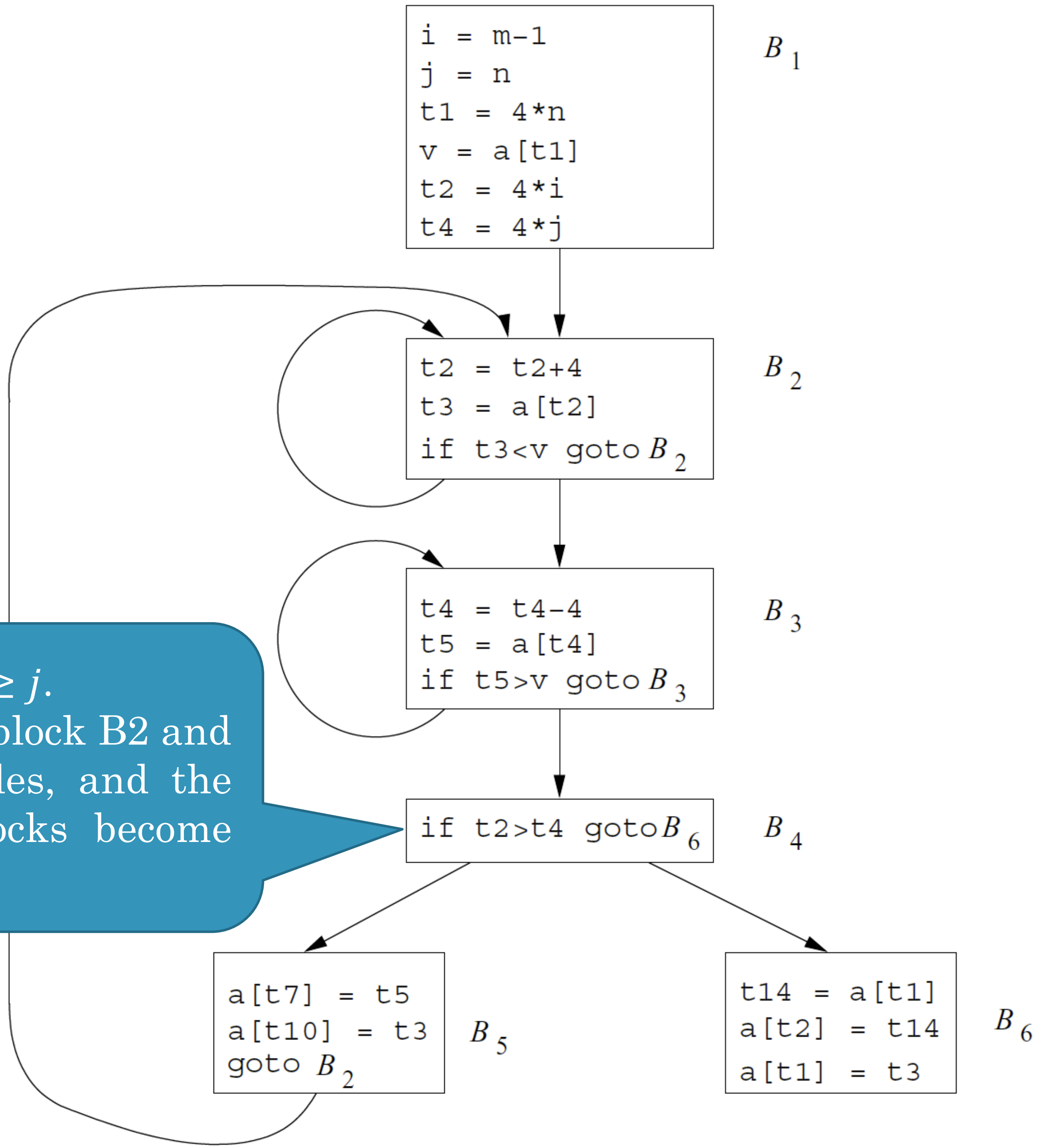
Global Optimization

- **Induction Variables and Reduction in Strength**
  - **Example:** Strength reduction applied to 4*j in block B3



Global Optimization

- **Induction Variables and Reduction in Strength**
  - **Example:** Flow graph after induction-variable elimination



- The test $t2 \geq t4$ can substitute for i $\geq j$.
- Once this replacement is made, i in block B2 and j in block B3 become dead variables, and the assignments to them in these blocks become dead code that can be eliminated