Compiler Design

Fatemeh Deldar

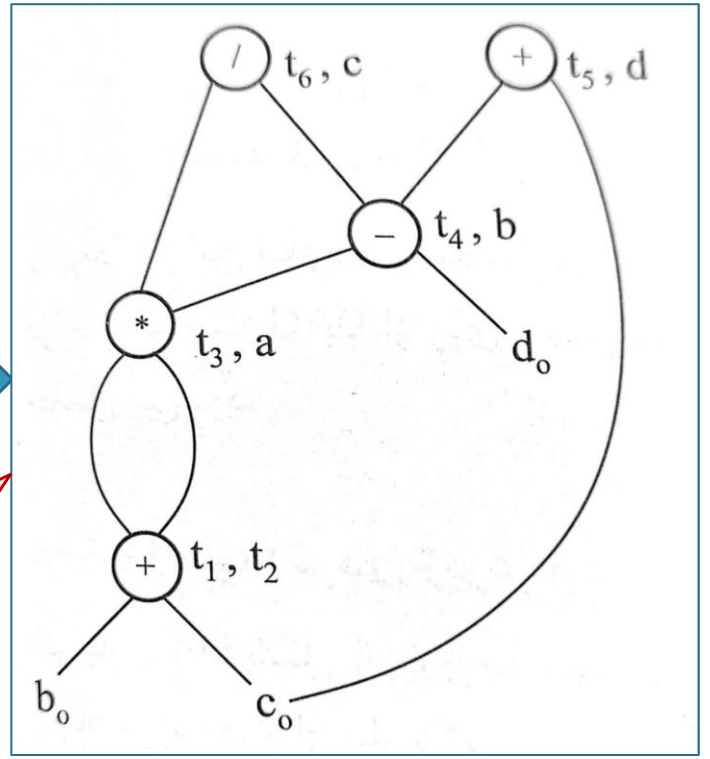
Isfahan University of Technology

1403-1404

Local Optimization

- **Finding Local Common Subexpressions**
  - **Example**

```
t1 = b + c
t2 = b + c
t3 = t1 * t2
a = t3
t4 = a – d
b = t4
t5 = b + c
d = t5
t6 = a / b
c = t6
```

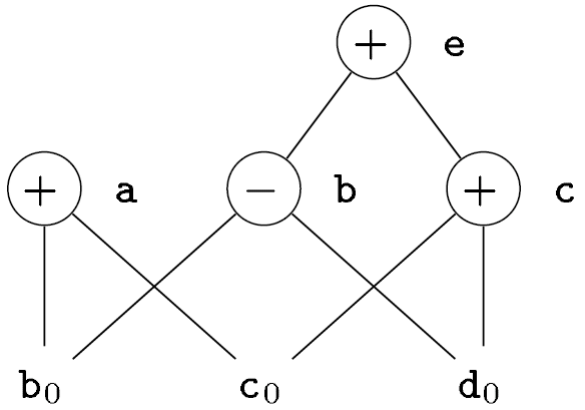


زیرعبارت مشترک b+c در t2 دوباره محاسبه نشده ولی در t5 دوباره محاسبه می‌شود.

Local Optimization

- **Dead Code Elimination**
  - **The operation on DAGs that corresponds to dead-code elimination can be implemented as follows**
    - **Delete from a DAG any root that has no live variables attached**
    - **Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code**
  - **Example**

- **In this example, if *a* and *b* are live but *c* and *e* are not, we can immediately remove the root labeled *e***
- **Then, the node labeled *c* becomes a root and can be removed**
- **The roots labeled *a* and *b* remain, since they each have live variables attached**



Local Optimization

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

$x + 0 = 0 + x = x$	$x - 0 = x$
$x \times 1 = 1 \times x = x$	$x/1 = x$

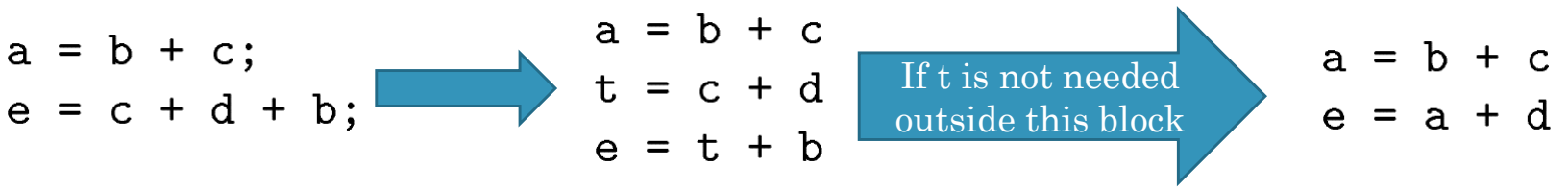
  - **Local reduction in strength**, that is, replacing a more expensive operator by a cheaper one

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 $\boldsymbol{M}$
  - **Associative laws** might also be applicable to expose common subexpressions



Local Optimization

- **Representation of Array References**
  - **Can we optimize the following code by replacing the third instruction $z = a[i]$ by the simpler $z = x$?**

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

- **Since j could equal i, the middle statement may in fact change the value of $a[i]$; thus, it is not legal to make this change**

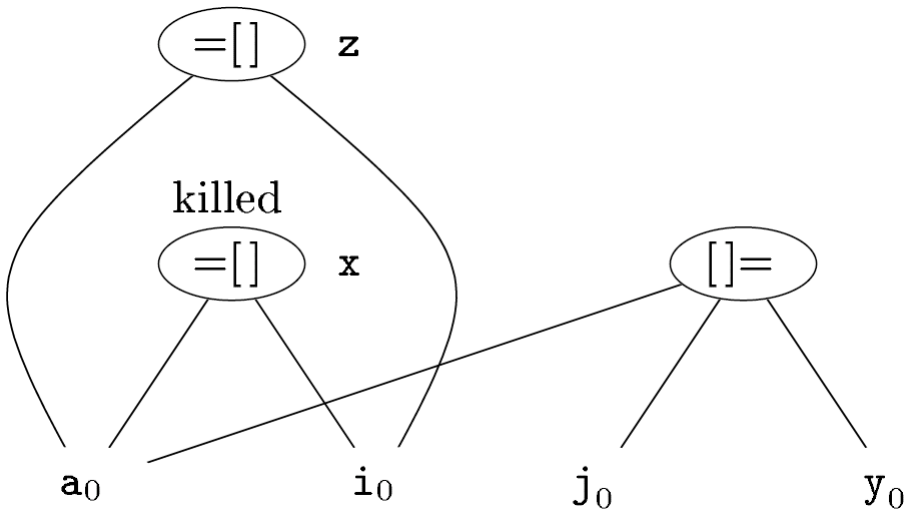
Local Optimization

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

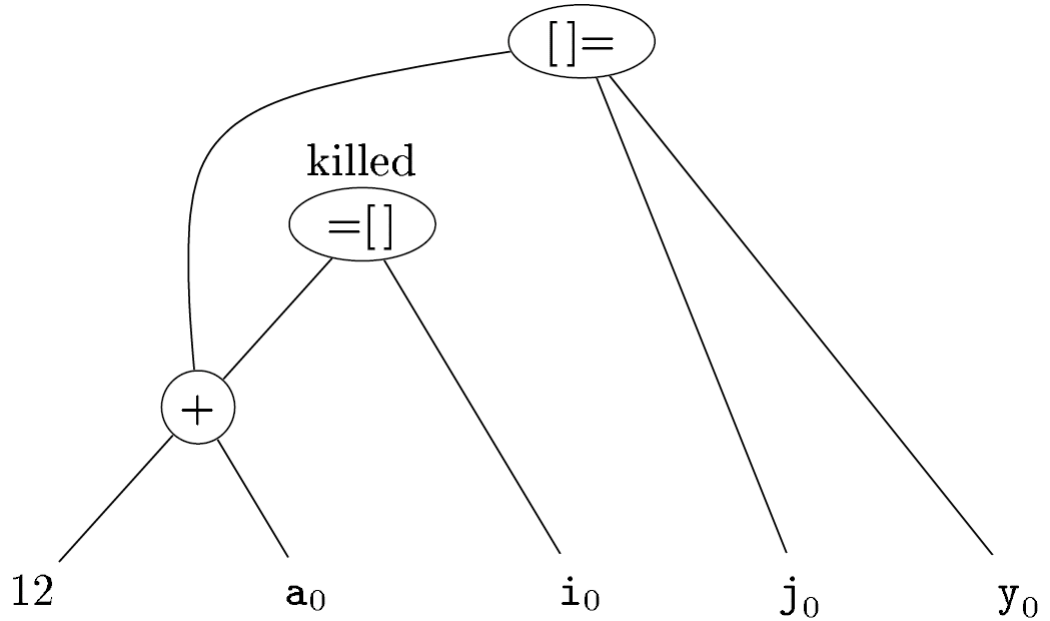
Local Optimization

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

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



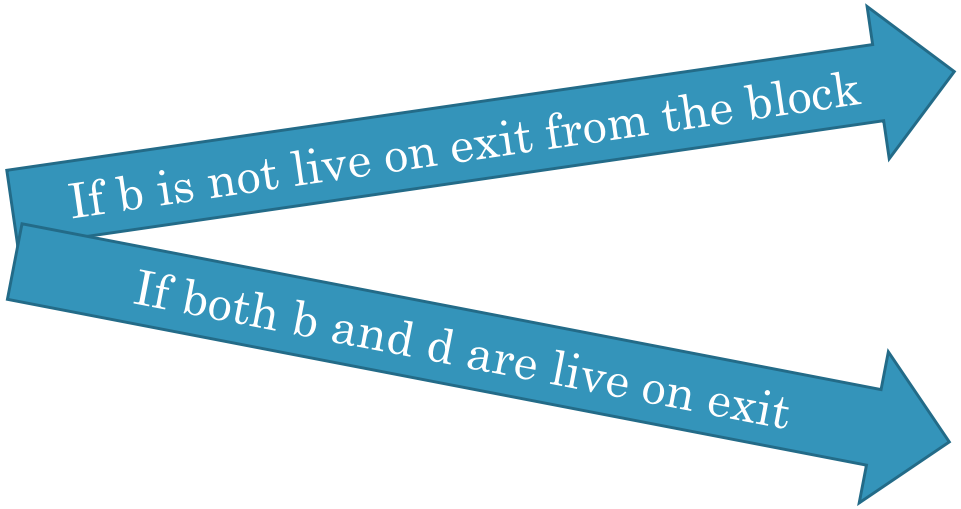

```
b = 12 + a
x = b[i]
b[j] = y
```



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
```




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

```
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

- **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**