CS536

Machine-Independent Optimizations

A Sahu

CSE, IIT Guwahati

Outline

- Machine Independent Optimization
- Standard Optimizations
- Local Optimization: DAG
- Basic Loop Optimization
- Basic Data Flow Analysis

Machine Independent Code Optimization

Causes of Redundancy

- Redundancy is available at the source level
  - Due to recalculations while one calculation is necessary.
- Redundancies in address calculations
  - Redundancy is a side effect of having written the program in a high-level language
  - where referrals to elements of an array or fields in a structure is done through accesses like A[i][j] or X -> f1.

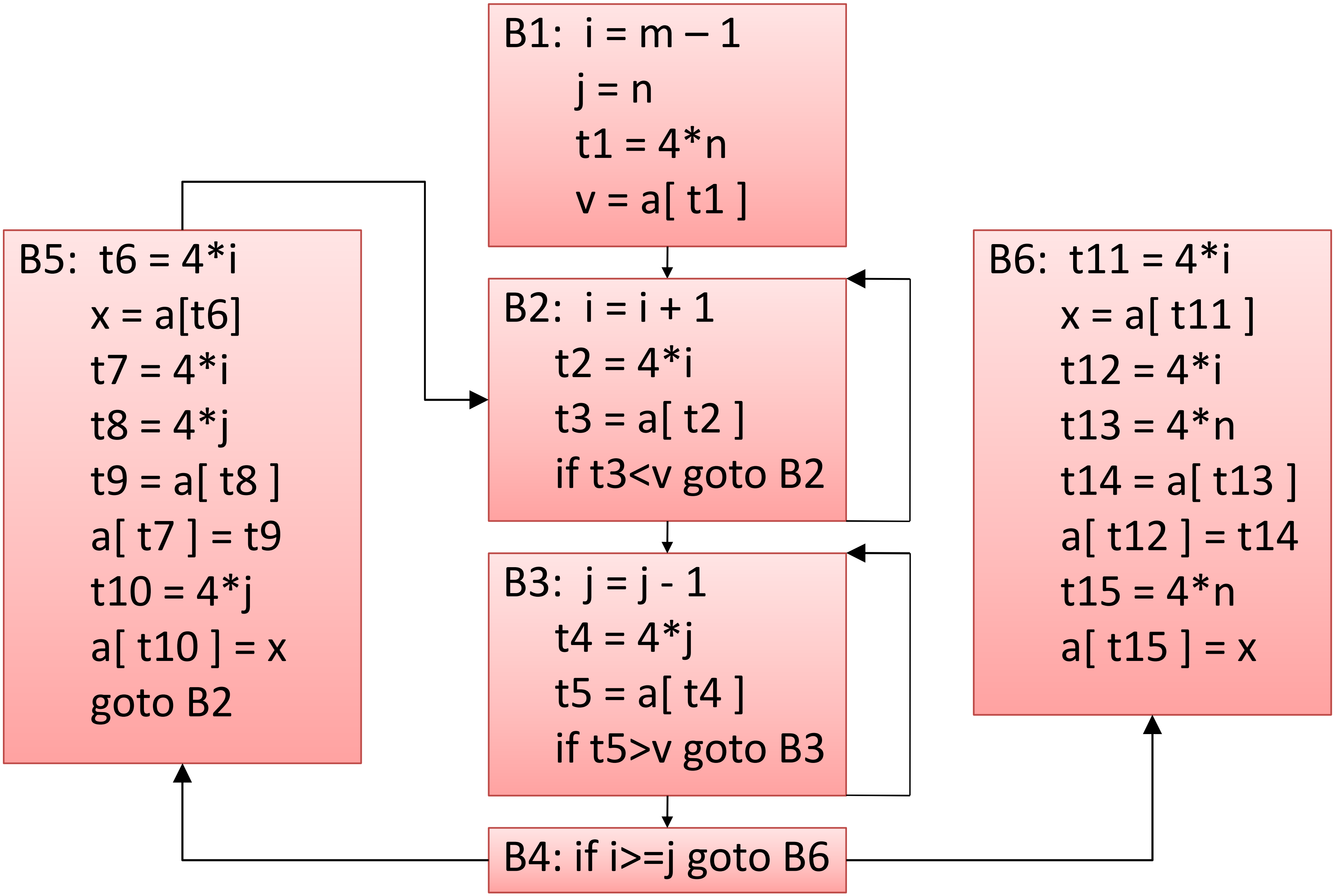
Causes of Redundancy

- As a program is compiled,
- Each of high-level data-structure accesses
  - array access and structure access
- Get expands into a number of low-level arithmetic operations
  - Such as the computation of the location of the [i, j]-th element of a matrix A.
- Accesses to the same data structure often share many common low-level operations.

A Running Example: Quicksort

```
void quicksort (int m, int n) {
  /* recursively sorts a[ m ] through a[ n ] */
      int i , j, 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;
      }
      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 );
  }
```

Flow Graph for Quicksort Fragment



Semantics-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
  - Common-subexpression elimination
  - Copy propagation
  - Dead-code elimination
  - Constant folding
  - Code motion
  - Induction-variable elimination

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.
- Avoid **recomputing** E if can be used its previously computed value;
  - that is, the variable x to which the previous computation of E was assigned has not changed in the interim.

Common Sub Expr. Elimination

```
B1:  i = m – 1
     j = n
     t1 = 4*n
     v = a[ t1 ]
```

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

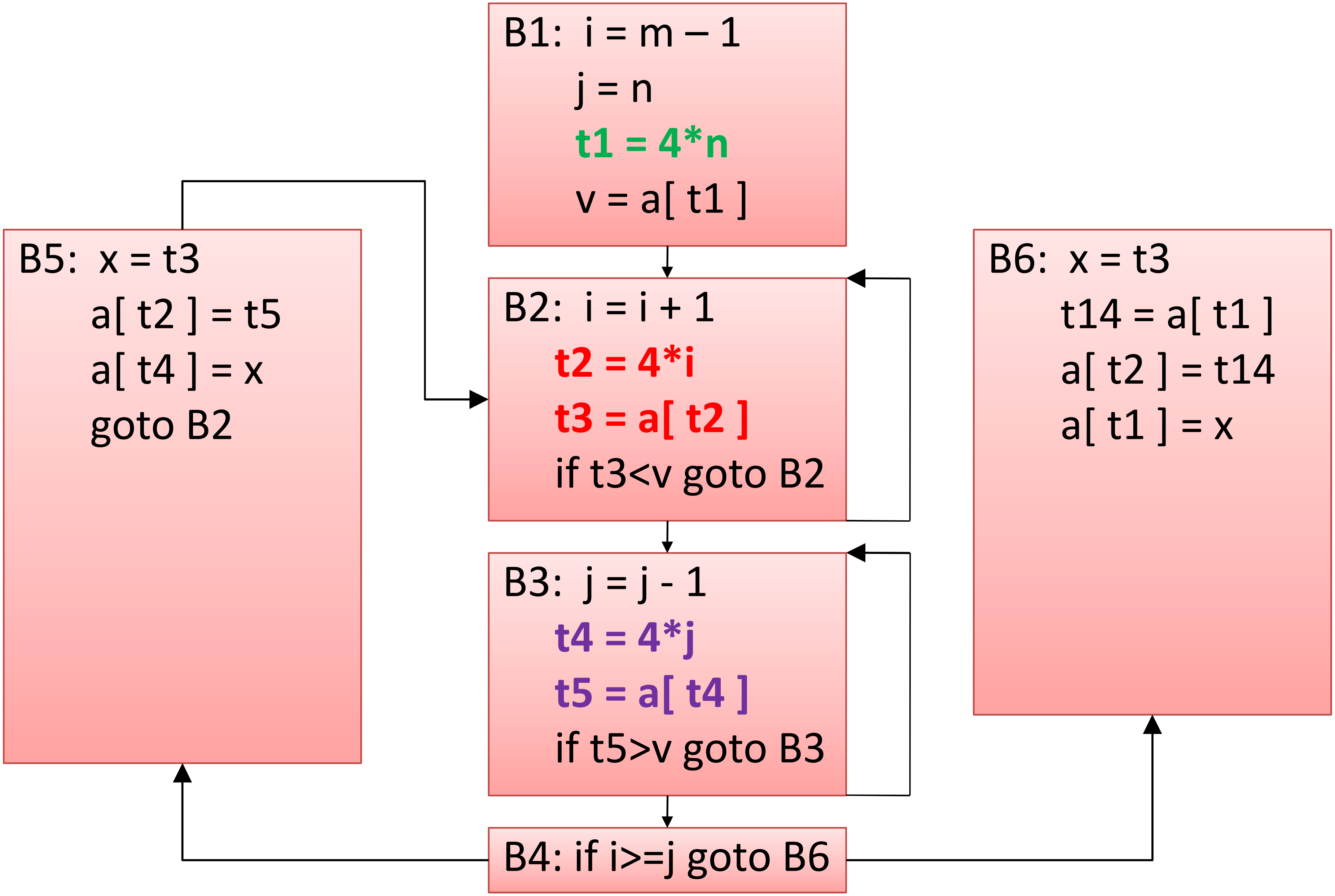
```
B3:  j = j - 1
     t4 = 4*j
     t5 = a[ t4 ]
     if t5>v goto B3
```

```
B4: if i>=j goto B6
```

```
B5:  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
```

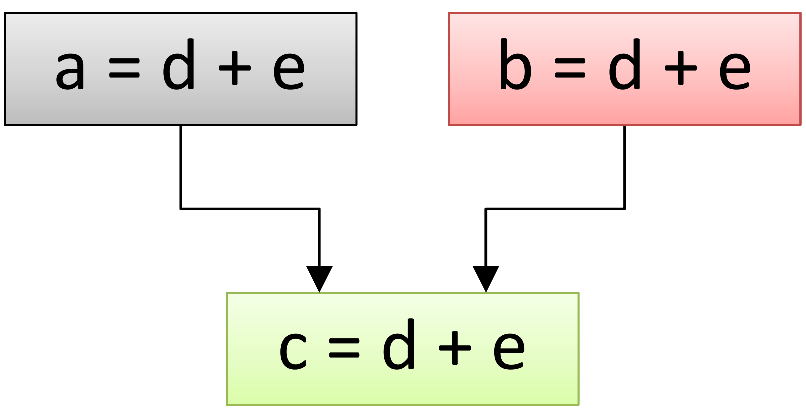
```
B6:  t11 = 4*i
     x = a[ t11 ]
     t12 = 4*i
     t13 = 4*n
     t14 = a[ t13 ]
     a[ t12 ] = t14
     t15 = 4*n
     a[ t15 ] = x
```

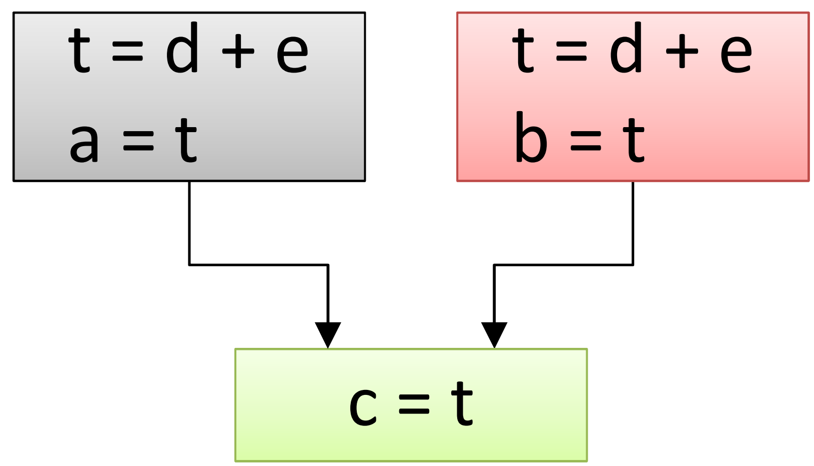
Flow Graph After C.S. Elimination



Copy Propagation

- This optimization concerns assignments of the form u = v called copy statements.
- The idea behind the copy-propagation transformation is to use v for u, wherever possible after the copy statement u = v.
- Copy propagation work example:





Copy Propagation

- The assignment x = t3 in block B5 is a copy.

  Here is the result of copy propagation applied to B5.

```
B5:  x = t3
     a[ t2 ] = t5
     a[ t4 ] = x
     goto B2
```

```
B5:  x = t3
     a[ t2 ] = t5
     a[ t4 ] = t3
     goto B2
```

- **This change may not appear to be an improvement, but it gives the opportunity to eliminate the assignment to x.**
- **One advantage of copy propagation is that it often turns the copy statement into dead code.**

Dead-code Elimination

- Dead code
  - Code that is unreachable or
  - that does not affect the program (e.g. dead stores) can be eliminated.
- While the programmer is unlikely
  - to introduce any dead code intentionally,
  - it may appear as the result of previous transformations.
- Deducing at compile time that the value of an expression is a constant and
  - Using the constant instead is known as constant folding.

Dead-code Elimination: Example

- In the example below,
  - the value assigned to i is never used, and the dead store can be eliminated.
  - The first assignment to global is dead, and
  - the third assignment to global is unreachable; both can be eliminated.

```
int global;
void f () {
    int i;
    i = 1;  /* dead store */
    global = 1; /* dead store */
    global = 2;
    return;
    global = 3; /* unreachable */
}
```

```
int global;
void f ()    {
    global = 2;
    return;
}
```

Code Motion: Loop Invariant

- Code motion decreases the amount of code in a loop.
- 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.

Code Motion

Evaluation of **limit - 2** is a loop-invariant computation in the following while-statement :

```
while ( i <= limit-2){  // stmt does not change limit
        loopbody();//limit is not modified here
}
```

Code motion will result in the equivalent code:

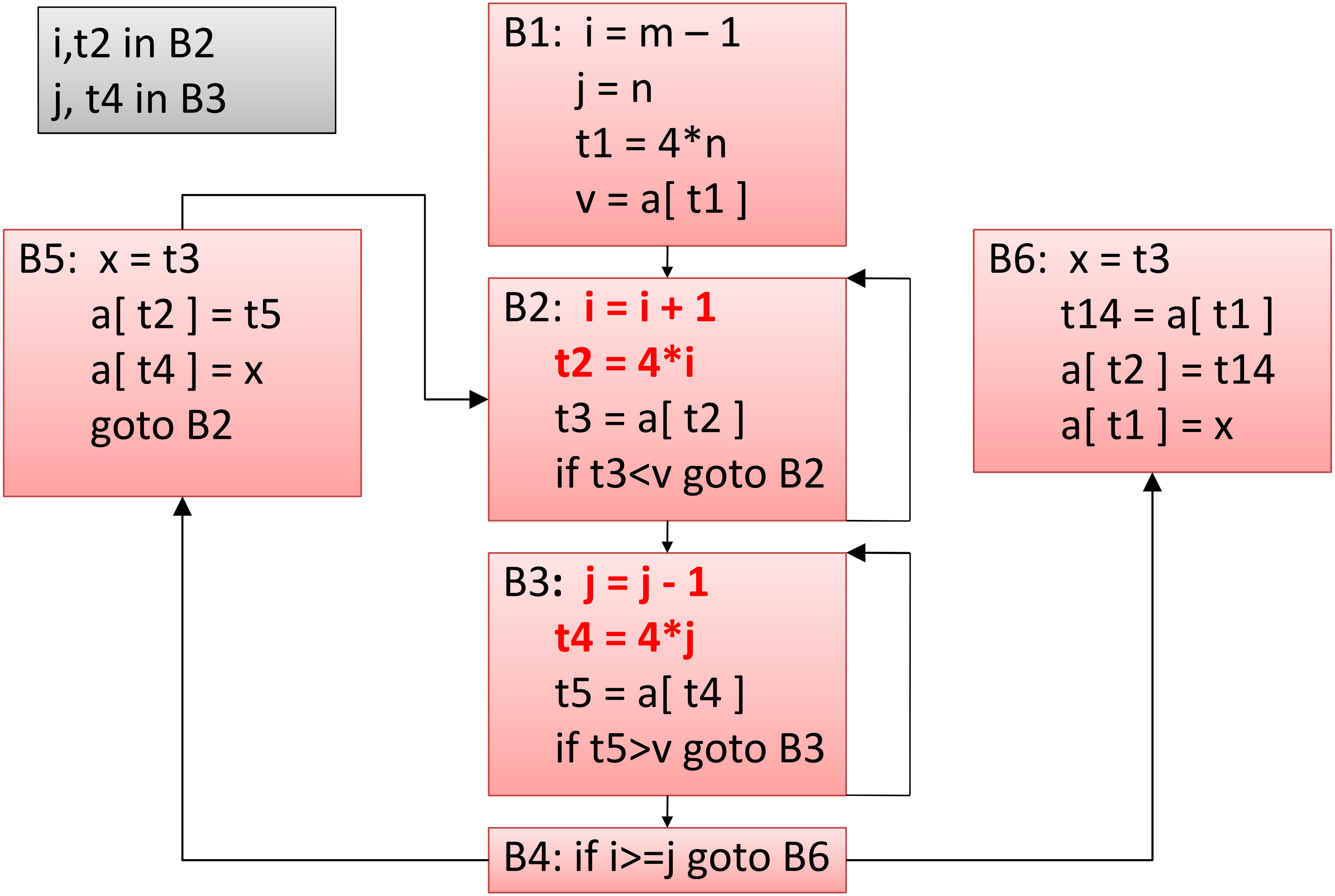
```
t = limit-2;
while ( i <= t ){  // stmt does not change limit and t
  loopbody();//limit is not modified here
}
```

Induction-Variable (IV) Elimination

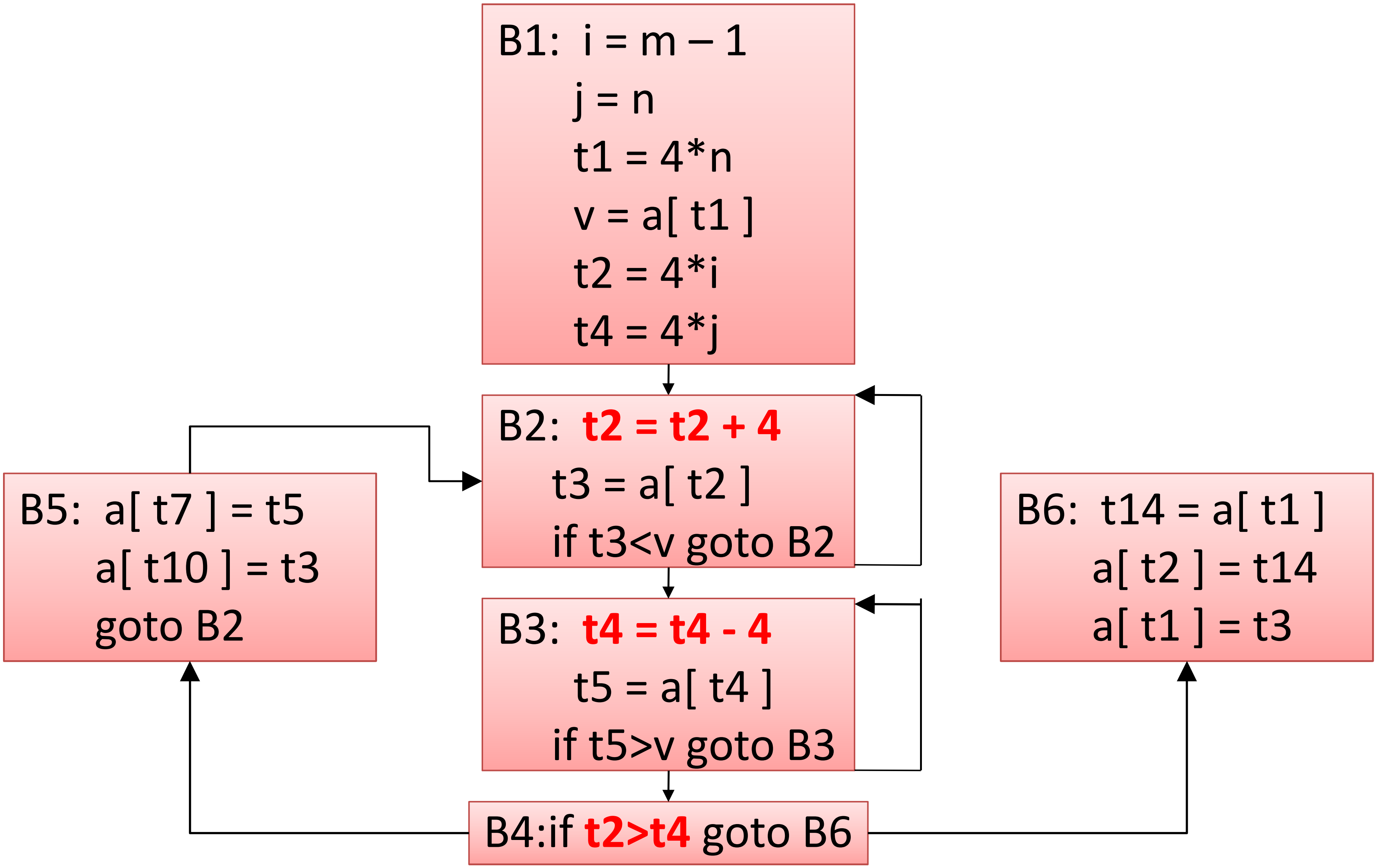
- Variable x is said to be an **"induction variable“**
  - If there is a positive or negative constant c such that
  - Each time x is assigned, its value increases by c.
- Induction variables can be computed
  - With a single increment (addition or subtraction) per loop iteration.
- Transformation of replacing an expensive operation, such as multiplication,
  - By a cheaper one, such as addition, is known as strength reduction.

Induction variable Example: i, t2, j, t4

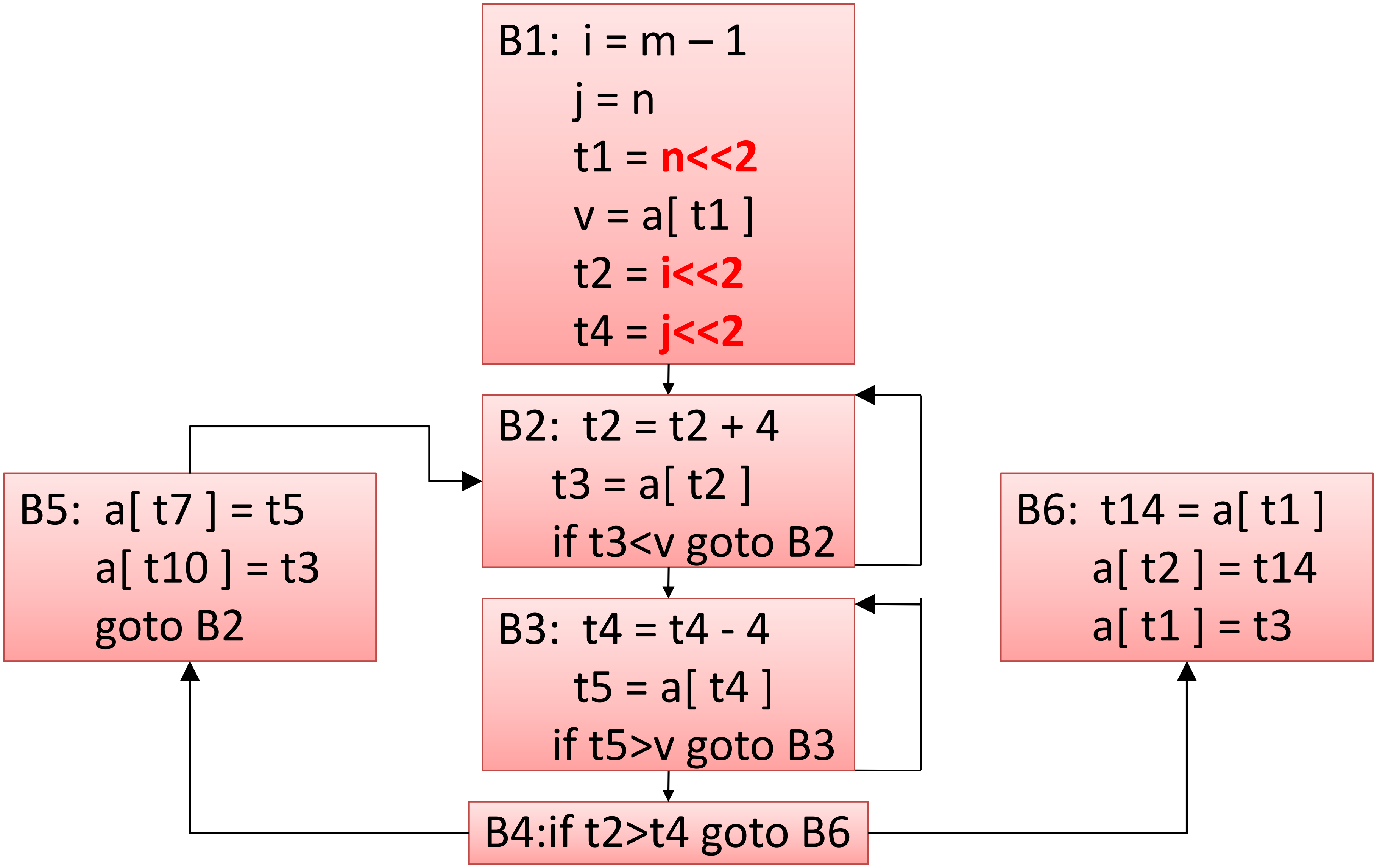
i,t2 in B2
j, t4 in B3



Flow Graph After IV Elimination



Flow Graph Strength Reduction



Flow Analysis

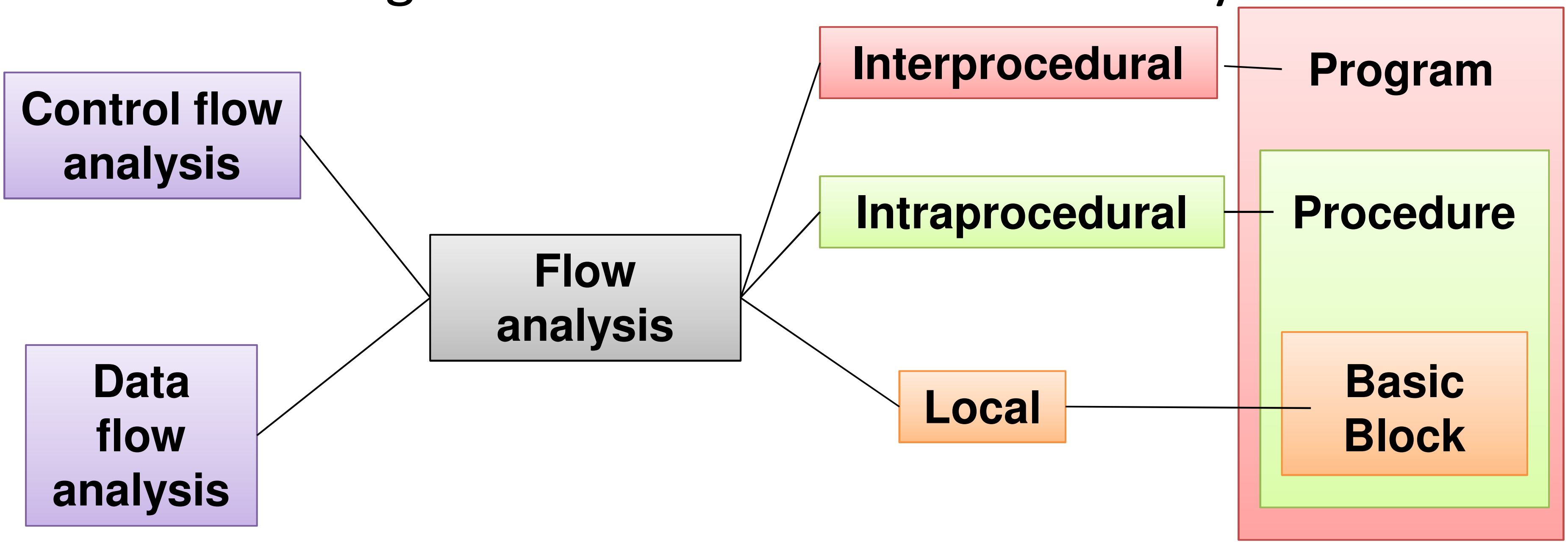
- **Flow analysis** is a fundamental prerequisite
  - For many important types of code improvement.
- Generally control flow analysis precedes data flow analysis

Flow Analysis

- **Control flow analysis** (CFA) represents flow of control usually in form of graphs. CFA constructs:
  - Control flow graph
  - Call graph
- **Data flow analysis (**DFA) is the process of **asserting and collecting information prior to program execution**
  - About the possible modification, preservation, and use of certain entities
  - such as values or attributes of variables in a computer program.

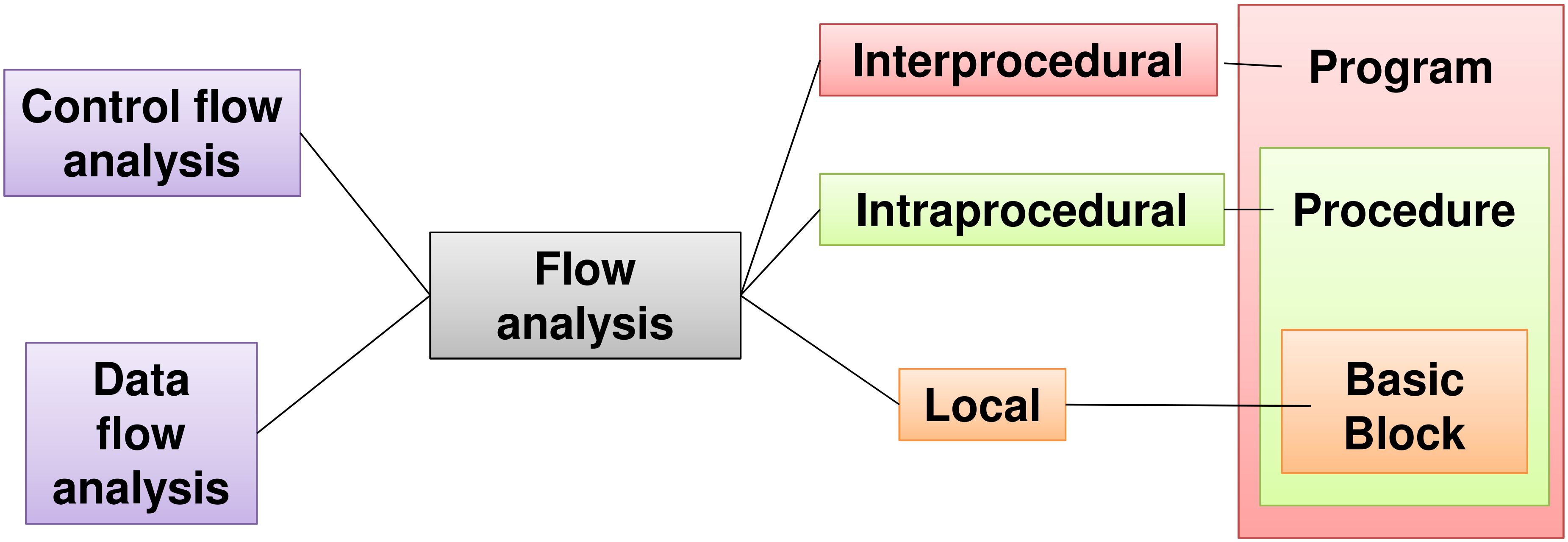
Classification of Flow Analysis

- Two orthogonal classifications of flow analysis:



- Interprocedural optimizations usually require a call graph.
- In a call graph each node represents a procedure and an edge from one node to another indicates that one procedure may directly call another.

In This Course



1. Local Flow analysis in a Basic Block
2. Control Flow Analysis Assuming Basic Block as Black Box (BB as BB)
3. Global Data Flow Analysis

Local Optimization

Optimization of Basic Blocks

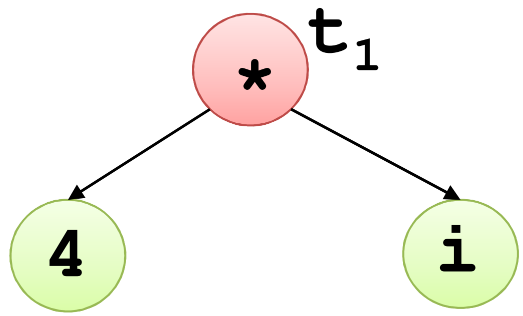
- Many structure preserving transformations can be implemented by construction of DAGs of basic blocks

DAG representation of Basic Block (BB)

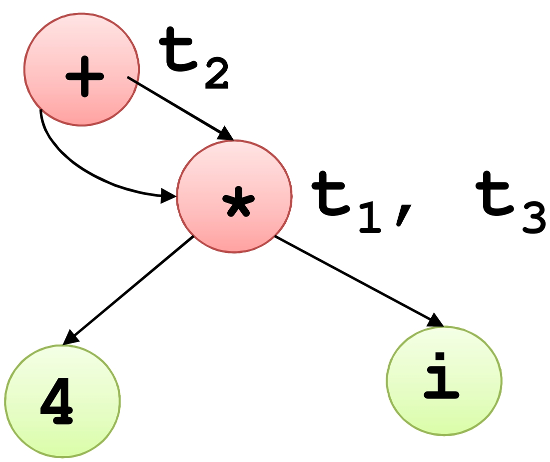
- Leaves are labeled with unique identifier (var name or const)
- Interior nodes are labeled by an operator symbol
- Nodes optionally have a list of labels (identifiers)
- Edges relates operands to the operator (interior nodes are operator)
- Interior node represents computed value
  - Identifier in the label are deemed to hold the value

Example: DAG for BB

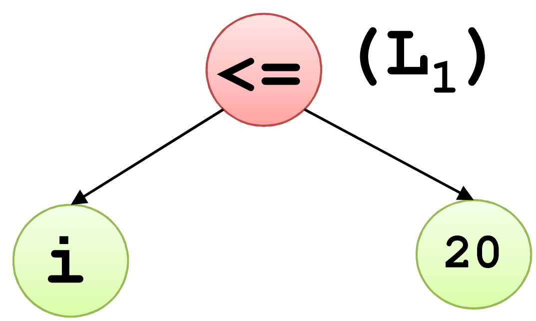
$t_1 := 4 * i$



$t_1 := 4 * i$
$t_3 := 4 * i$
$t_2 := t_1 + t_3$



if (i <= 20)goto L_1



Construction of DAGs for BB

- I/p: Basic block, *B*
- O/p: A DAG for *B* containing the following information:
  1) A label for each node
  2) For leaves the labels are ids or consts
  3) For interior nodes the labels are operators
  4) For each node a list of attached ids (possible empty list, no consts)

Construction of DAGs for BB

- Data structure and functions:
  - Node:
    1) Label: label of the node
    2) Left: pointer to the left child node
    3) Right: pointer to the right child node
    4) List: list of additional labels (empty for leaves)
  - **Node (*id*)**: returns the most recent node created for *id*. Else return *undef*
  - **Create(*id,l,r*)**: create a node with label *id* with *l* as left child and *r* as right child. *l* and *r* are optional params.

Construction of DAGs for BB

- Method:

  For each 3AC, A in B

  A if of the following forms:

  1. $x := y$ op z
  2. $x :=$ op y
  3. $x := y$

  1. if $((n_y = \text{node}(y)) == \mathit{undef})$

     n_y = Create (y);

     if (A == type 1)

     and $((n_z = \text{node}(z)) == \mathit{undef})$

     n_z = Create(z);

Construction of DAGs for BB

2. If (A == type 1)

   Find a node labelled '*op*' with left and right as n_y and n_z respectively [determination of common sub-expression]

   If (not found) n = Create (op, n_y, n_z);

   If (A == type 2)

   Find a node labelled '*op*' with a single child as n_y

   If (not found) n = Create (op, n_y);

   If (A == type 3) n = Node (y);

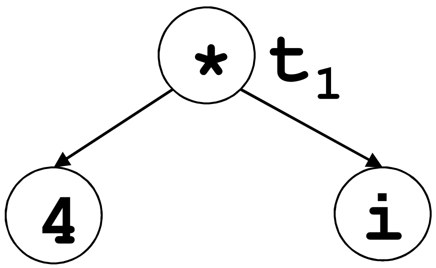
3. Remove x from Node(x).list

   Add x in n.list

   Node(x) = n;

Example: DAG construction from BB

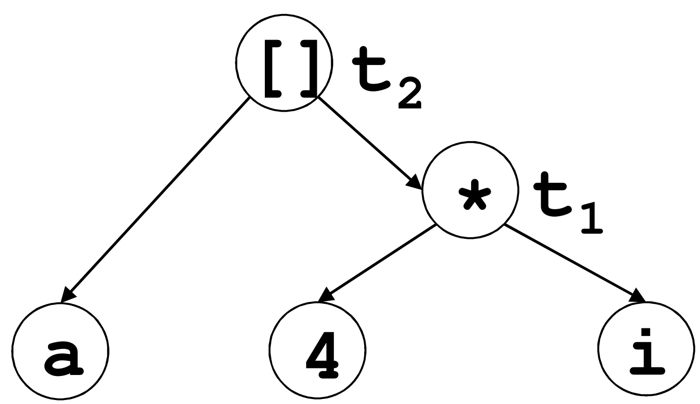
$t_1 := 4 * i$



Example: DAG construction from BB

t_1 := 4 * i

t_2 := a [t_1]

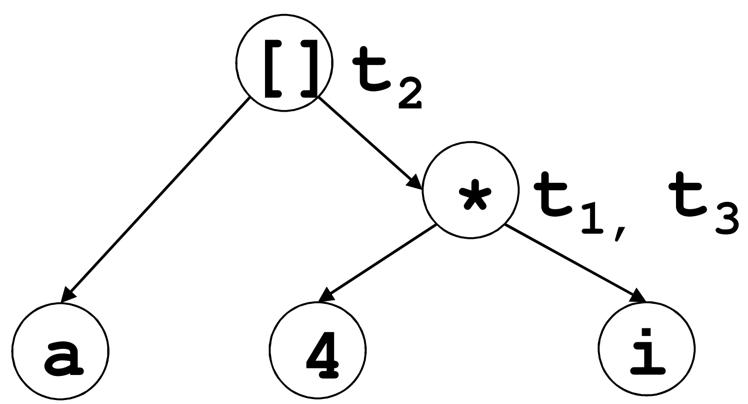


Example: DAG construction from BB

t_1 := 4 * i

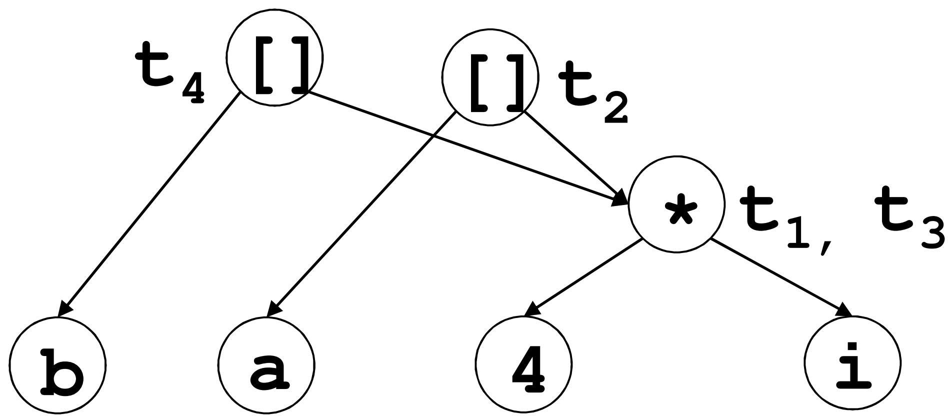
t_2 := a [t_1]

t_3 := 4 * i



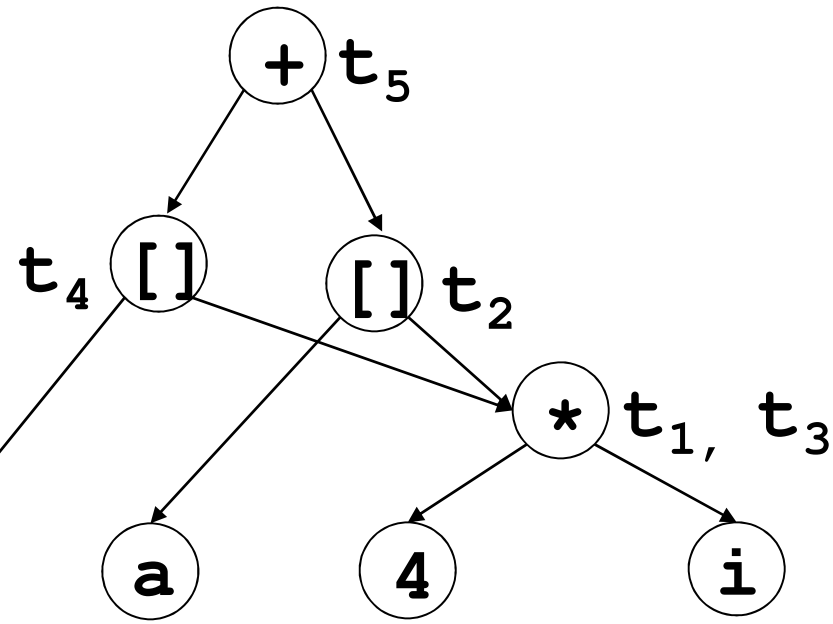
Example: DAG construction from BB

```
t1 := 4 * i
t2 := a [ t1 ]
t3 := 4 * i
t4 := b [ t3 ]
```



Example: DAG construction from BB

```
t1 := 4 * i
t2 := a [ t1 ]
t3 := 4 * i
t4 := b [ t3 ]
t5 := t2 + t4
```



Example: DAG construction from BB

t_1 := 4 * i

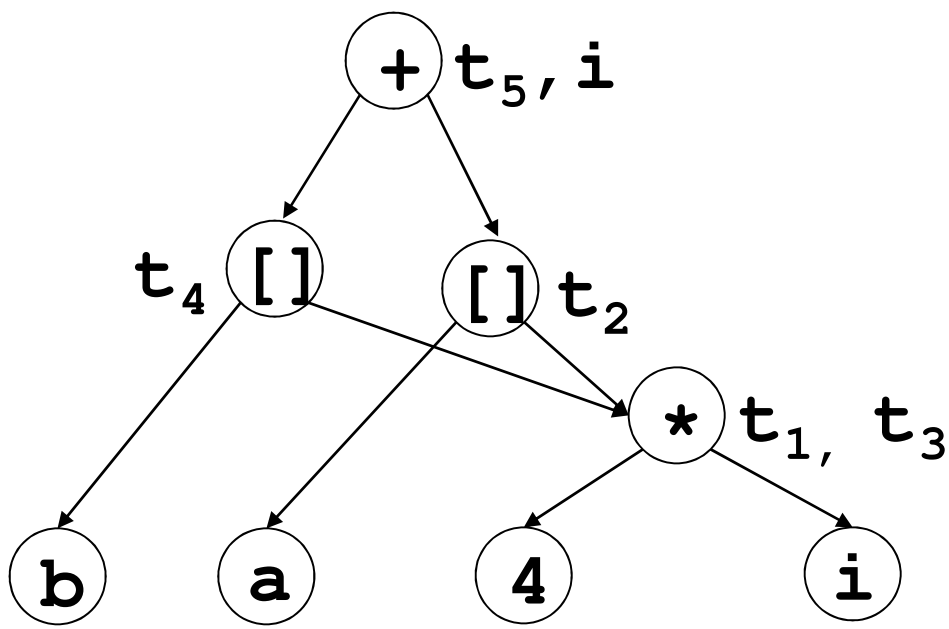
t_2 := a [t_1]

t_3 := 4 * i

t_4 := b [t_3]

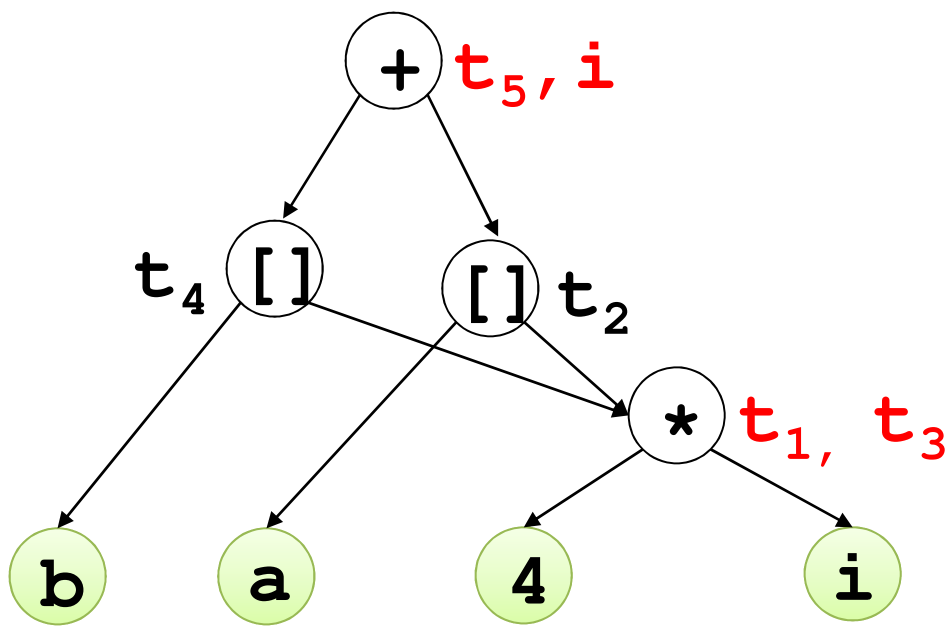
t_5 := t_2 + t_4

i := t_5



Observation

```
t1 := 4 * i
t2 := a [ t1 ]
t3 := 4 * i
t4 := b [ t3 ]
t5 := t2 + t4
i := t5
```



DAG of a Basic Block

- Observations:
  - A leaf node for the initial value of an id
  - A node *n* for each statement *s*
  - **The children of node *n* are the last definition (prior to *s*) of the operands of *n***

Optimization of Basic Blocks

- Common sub-expression elimination: by construction of DAG
  - Note: for common sub-expression elimination, we are actually targeting for expressions that compute the same value.

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

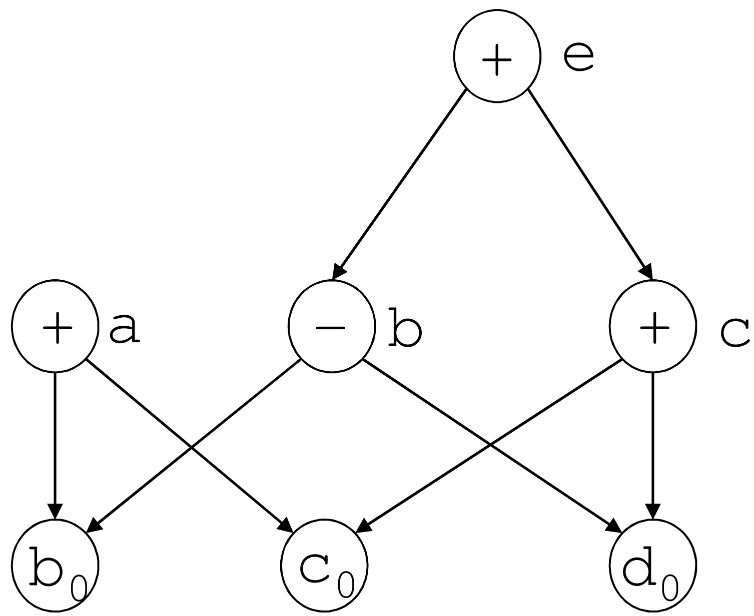
Common expressions

But do not generate the same result

Optimization of Basic Blocks

- DAG representation identifies expressions that yield the same result

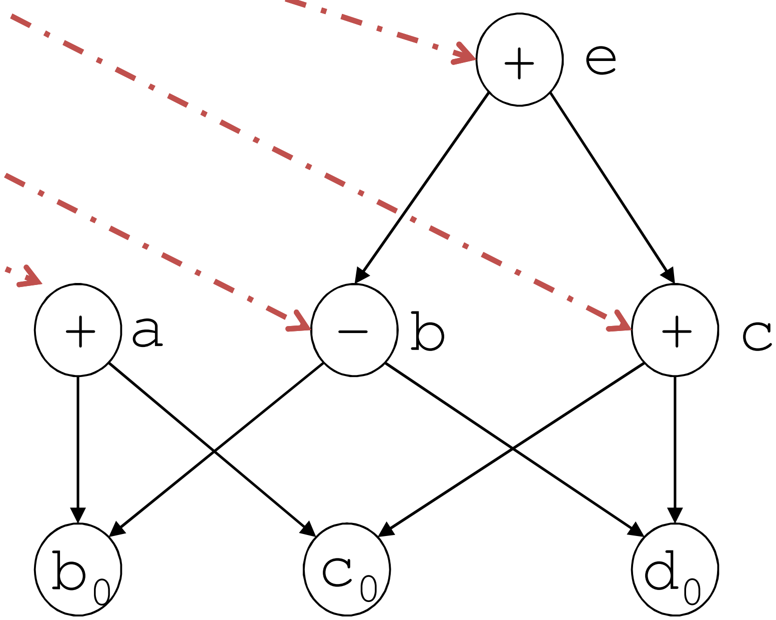
```
a=b+c; b=b-d; c=c+d; e=b+c;
```



Optimization of Basic Blocks

- DAG representation identifies expressions that yield the same result

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

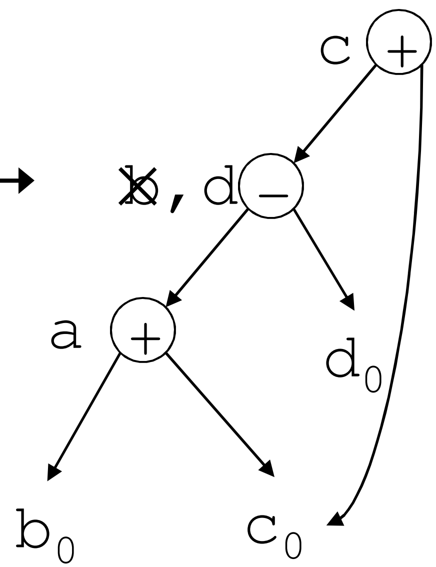


Optimization of Basic Blocks

- Dead code elimination: Code generation from DAG eliminates dead code.

```
a  :=  b  +  c
b  :=  a  −  d
d  :=  a  −  d
c  :=  d  +  c
```

b is not live




```
a  :=  b  +  c
d  :=  a  −  d
c  :=  d  +  c
```

Loop Optimization

Loop Optimization Example

- Basic LO
  - Loop invariant code removal
  - Induction variable strength reduction
  - Induction variable reduction
- Advance LO
  - Loop Interchange
  - Loop Splitting: Peeling Special Case
  - Loop Fusion/Jamming
  - Loop Fission/Distribution
  - Loop Unrolling

Loop Fusion

```
for (i = 0; i < 300; i++)
        a[i] = a[i] + 3;
for (i = 0; i < 300; i++)
        b[i] = b[i] + 4;
```

```
for (i = 0; i < 300; i++) {
        a[i] = a[i] + 3;
        b[i] = b[i] + 4;
}
```

Reduces branches
Improve parallelism
Create bigger basic block

Loop Fission/Split

```
for (i = 0; i < 1000; i++) {
  if(i%2==0)
          a[i] = a[i] + 10;
   else  a[i]= a[i] + 20;
}
```

```
for (i = 0; i < 1000; i=i+2)
          a[i]=a[i]+10;
for (i = 1; i < 1000; i=i+2)
           a[i]=a[i]+20;
```

Reduces branches (of if/else)
Both loop in total do for 1000
Improve parallelism

Loop Peeling

```
int p = 100;
for (int i=0; i<100; ++i) {
        y[i] = x[i] + x[p];
        p = i;
}
```

```
y[0] = x[0] + x[100];
for (int i=1; i<100; ++i) {
        y[i] = x[i] + x[i-1];
}
```

p = 100 only for the first iteration, and
for all other iterations, p = i - 1

Loop unrolling

```
for (x = 0; x < 100; x++) {
    A[x]=x*2+5;
}
```

```
for (x = 0; x < 100; x += 4 ) {
        A[x]=x*2+5;
        A[x+1]=(x+1)*2+5;
        A[x+2]=(x+2)*2+5;
        A[x+3]=(x+3)*2+5;
}
```

It improve parallelization

Increase size of the BB

Loop unrolling

```
for (x = 0; x < 100; x++) {
        process(x);
}
```

```
for (x = 0; x < 100; x += 4 ) {
        process(x);
        process (x + 1);
        process (x + 2);
        process (x + 3);
}
```

It improve parallelization

Suppose you have 4 worker unroll for 4 in a batch

Suppose you have 6 worker unroll for 6 in a batch