Code Optimization

Code Optimization

- Generated code by the code generator or intermediate code need not be efficient.
- Means have been done for getting efficient code using DAG and instruction selection

Criteria for Code-Improving Transformation

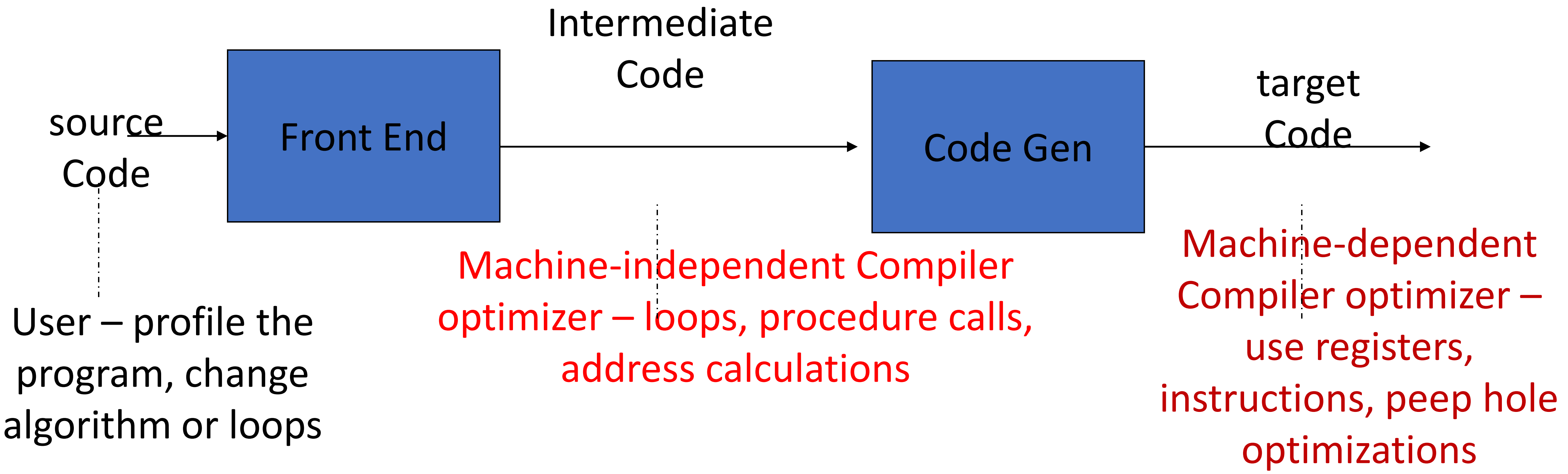
- Should have the most benefit for the least effort
- Transformation should preserve the meaning of the program
  - Optimization should not change the output of the program

Properties of Transformation

- Transformation must on an average speed up the programs by a measurable amount
  - Optimization sometimes would slow down the program. Prevention should be taken for this
- Transformation must be worth the effort
  - Time should not be spent in writing optimization code rather than the actual code of the problem

Position of the Code optimizer

- A main goal is to achieve a better performance



Two levels of optimization

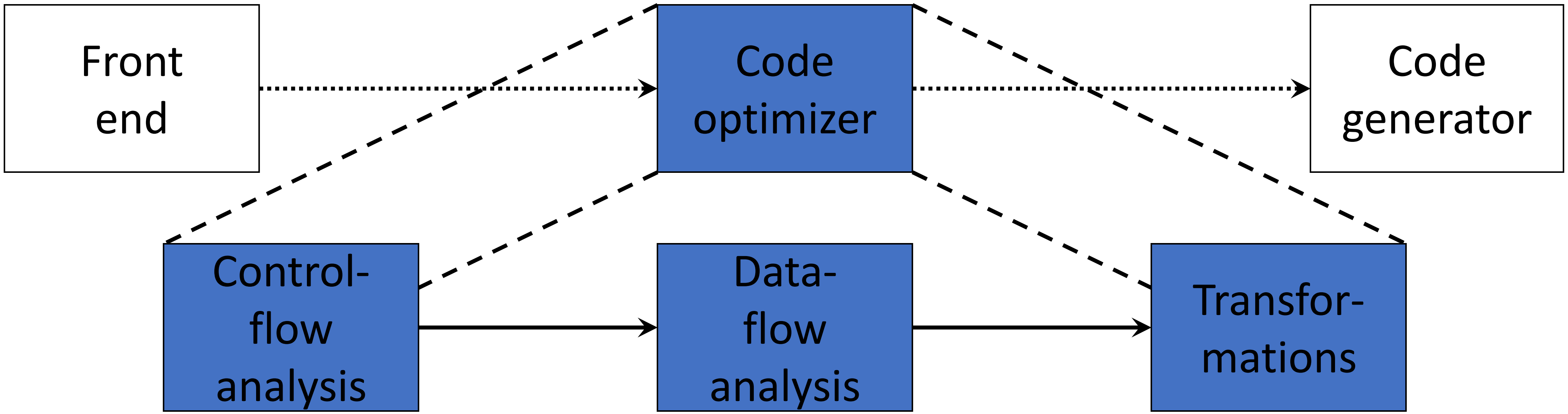
- Machine independent code optimization
  - Control Flow analysis
  - Data Flow analysis
  - Transformation
- Machine dependent code optimization
  - Register allocation
  - Utilization of special instructions.

Code optimizer

- Machine independent code optimization operates at the intermediate code level
- Machine dependent code optimization happens at the assembly level using Peep hole optimization

The Code Optimizer

- Control flow analysis
- Data-flow analysis
- Transformations



Advantages

- Operations need to implement high-level constructs are made explicit in the intermediate code
- Address calculations a[i]
- Intermediate code can be independent of the target machine

Optimizations

- Function Preserving Transformation
  - Common sub-expression elimination, dead code, copy propagation, constant folding,
- Loop Optimization
  - Induction variable elimination, Reduction in strength, code motion

Example for Optimization

```
assume an external input-output array: int a[]
void quicksort( int m, int n ) {
  int i, j, v, x; // temps
  if ( n <= m ) return;
  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;
  } //end while
  x = a[i]; a[i] =a [n]; a[n] = x;
  quicksort( m, j );
  quicksort( i+1, n );
} //end quicksort
```

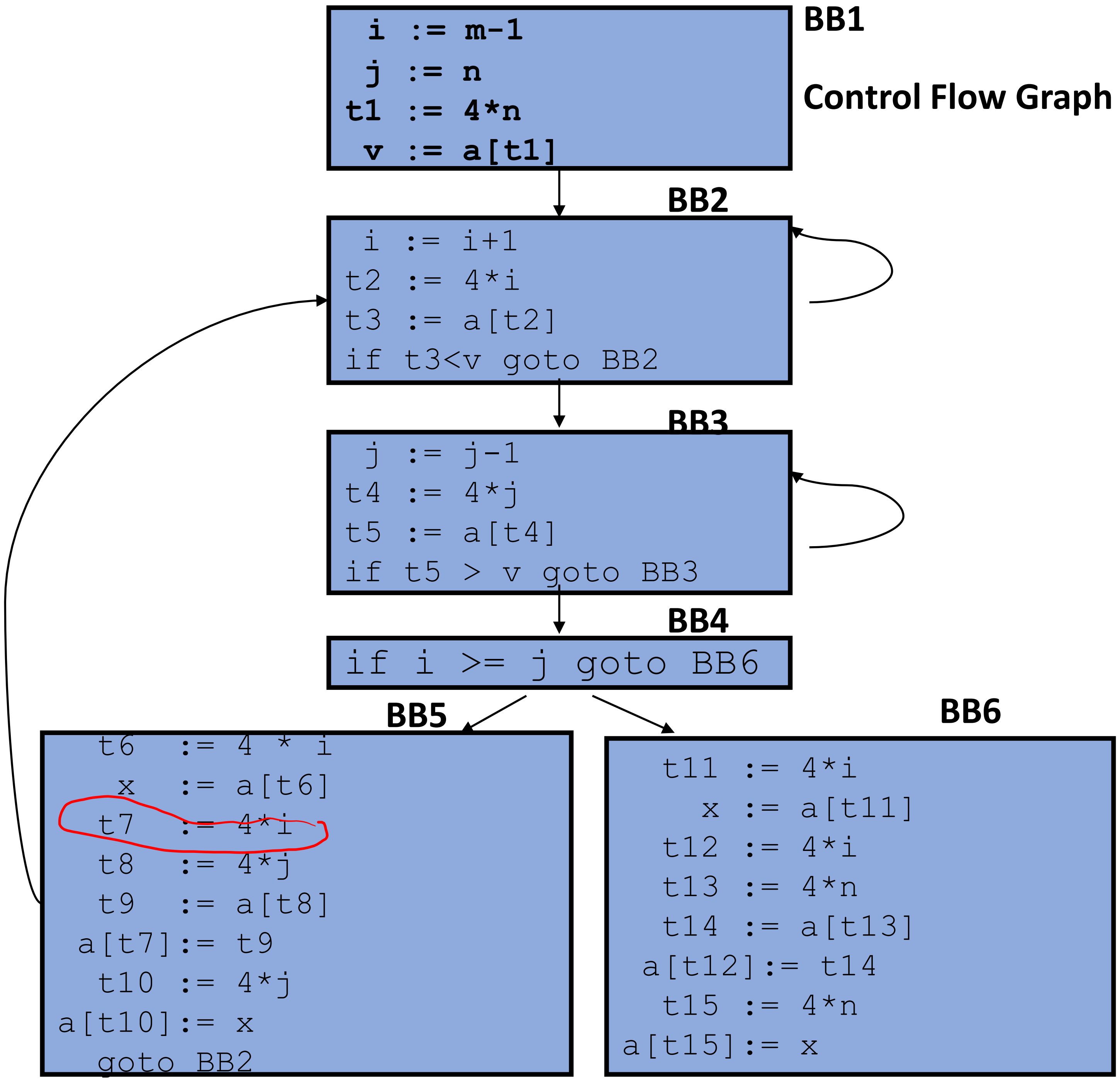
Quicksort IR

```
 (1) i := m-1
 (2) j := n
 (3) t1 := 4*n
 (4) v := a[t1]
L0: L1:
 (5) i := i+1
 (6) t2 := 4*i
 (7) t3 := a[t2]
 (8) if t3<v goto L1
L2:
 (9) j := j-1
 (10) t4 := 4*j
 (11) t5 := a[t4]
 (12) if t5>v goto L2
 (13) if i>=j goto L3
 (14) t6 := 4*i
 (15) x := a[t6]
```

BB1: (1)--(4)
BB2: (5)--(8)
BB3: (9)--(12)
BB4: (13)
BB5: (14)--(22)
BB6: (23)--(30)

```
 (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 L0
L3:
 (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
 (31) 2 calls ...
```

Quicksort CFG



t7 = t6

BB5

```
  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 BB2
```

BB6

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

Function Preserving Transformation

- Common Sub-expression Elimination
- Copy Propagation
- Dead-code elimination
- Constant Folding

Common Sub-expression Elimination (CSE)

- E is called a common sub-expression if E was previously computed and the values of variables in E have not changed since the previous computation
- Identifying E is easier with DAG as a basic block – have multiple tags associated

Example

- t7 and t10 have common sub-expressions 4 * i and 4 * j respectively.

```
  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 BB2
```

```
  t6   := 4 * i
   x   := a[t6]
  t8   := 4*j
  t9   := a[t8]
a[t6]:= t9
 a[t8]:= x
 goto BB2
```

Example – BB6

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

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

Global Common sub-expression

- 4 * i computed in block B2 and 4 * j computed in B3 hasn’t changed till B5 or B6.Hence, their values could be reused
- So is the value of a[t2] and a[t4]

Example

BB5

```
t6   := 4 * i
 x   := a[t6]
t8   := 4*j
t9   := a[t8]
a[t6]:= t9
a[t8]:= x
goto BB2
```

```
x    := a[t2]
t9   := a[t4]
a[t2]:= t9
a[t4]:= x
goto BB2
```

```
x    := t3
t9   := t5
a[t2]:= t9
a[t4]:= x
goto BB2
```

Example – BB6

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

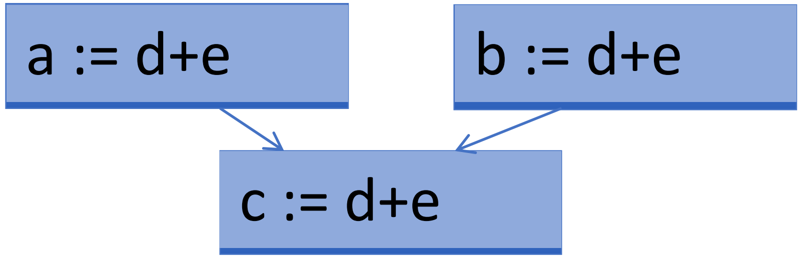
```
x  := a[t2]
t14  := a[t1]
a[t2]:= t14
a[t1]:= x
```

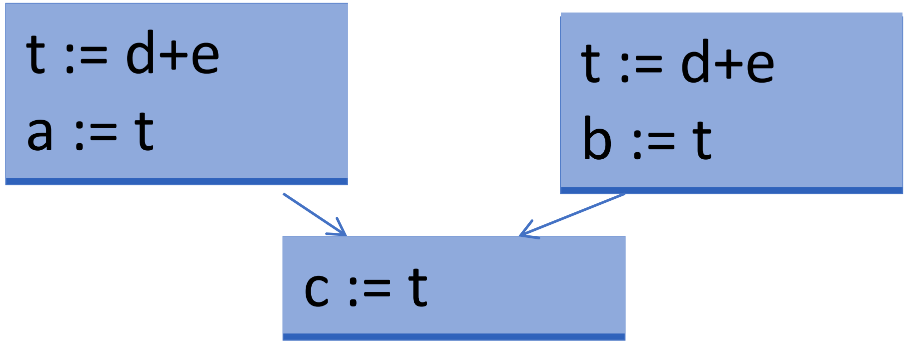
```
x  := t3
t14  := a[t1]
a[t2]:= t14
a[t4]:= x
```

Copy Propagation

- Assignments of the form f := g is the copy statement
- Elimination of common sub-expression leads to creation of copy.
- Assignments f:=g , can be transformed as use “g” for “f”

Copies introduced due to CSE





Example

```
x   := t3
t9  := t5
a[t2]:= t9
a[t4]:= x
goto BB2
```

```
x   := t3
a[t2]:= t5
a[t4]:= t3
goto BB2
```

Example – BB6

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

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

Dead-code elimination

- A variable is live at a point where its value can be used subsequently, else dead at that point
- x := t3 – this is dead, because x is not going to be used and is replaced by t3

Constant Folding

- Deducing during compile time that a value is constant and using the constant instead is known as constant folding
- This would also lead to dead-code

Example

- a := 1
- c := a+ b

----

a:= 1

c := 1 + b

Here, the statement a := 1 becomes redundant

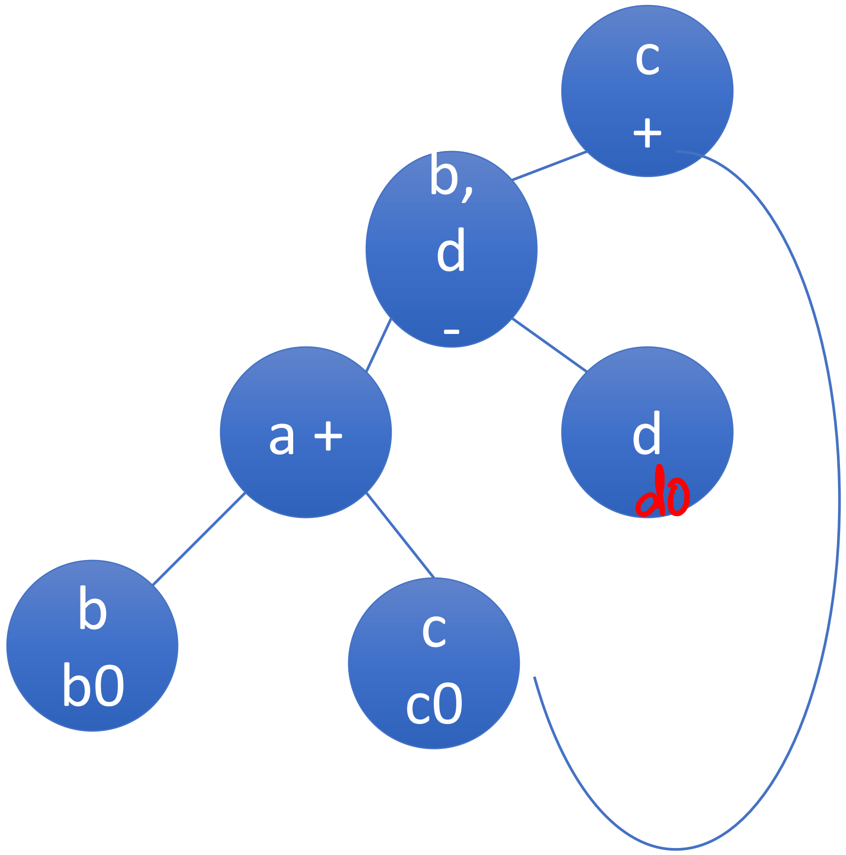
DAG for optimization

a := b+c

b := a-d

c := b+c

d := a - d



Algebraic simplification

- Algebraic identities are used to convert multiplication to addition, exponentiation to multiplication
- Multiplicative identity and additive identity are also applied

Algebraic Simplification

- $x + 0 = 0 + x = x$
- $x – 0 = x$
- $x * 1 = 1 * x = x$
- $x / 1 = x$
- $x ** 2 = x * x$
- $x * 2 = x + x$

Loop Optimization

- Running time of a program may be improved if the number of instructions in the inner loop is reduced
- Outer loops could have more instructions

Code motion

```
Actual Code:
while (i <=  limit - 2)
{    stmt;
}
L1:
     t1 = limit - 2
     if (i > t1) goto L2
     body of loop
     goto L1
L2:
```

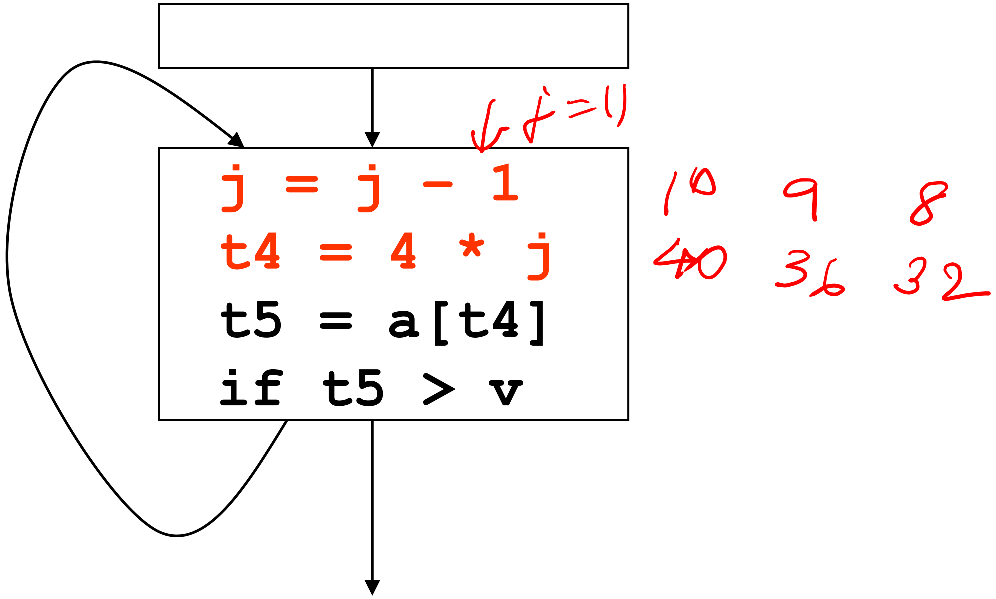
Code motion - Modified Code:

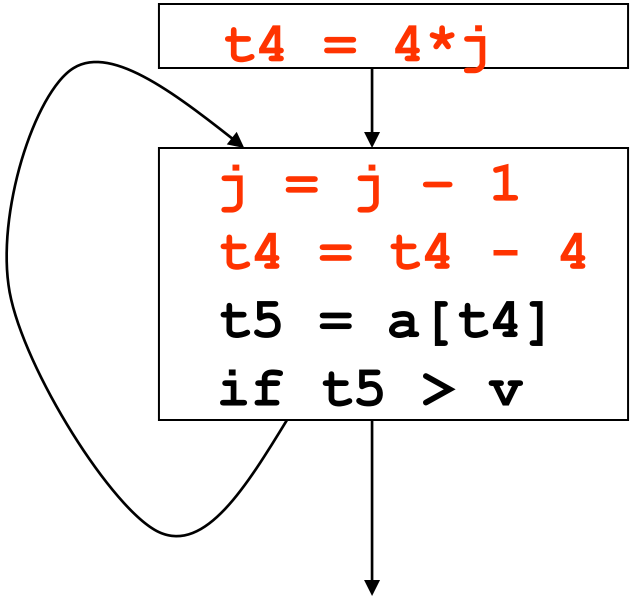
```
t := limit - 2
    while (i <= t)

t1 = limit – 2
L1:
   if (i > t1) goto L2
   body of loop
   goto L1
L2:
```

Strength Reduction

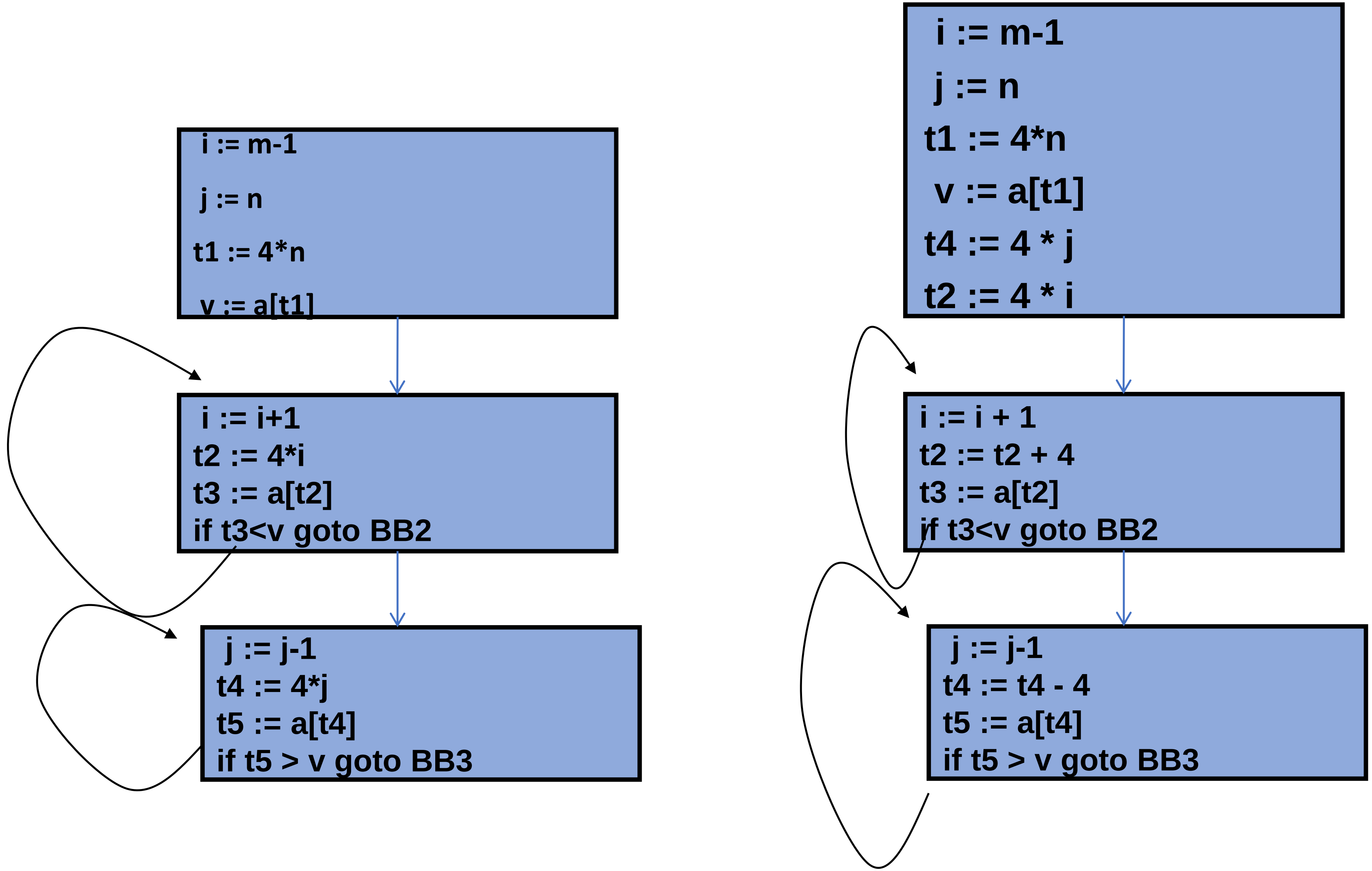
- Induction Variables control loop iterations



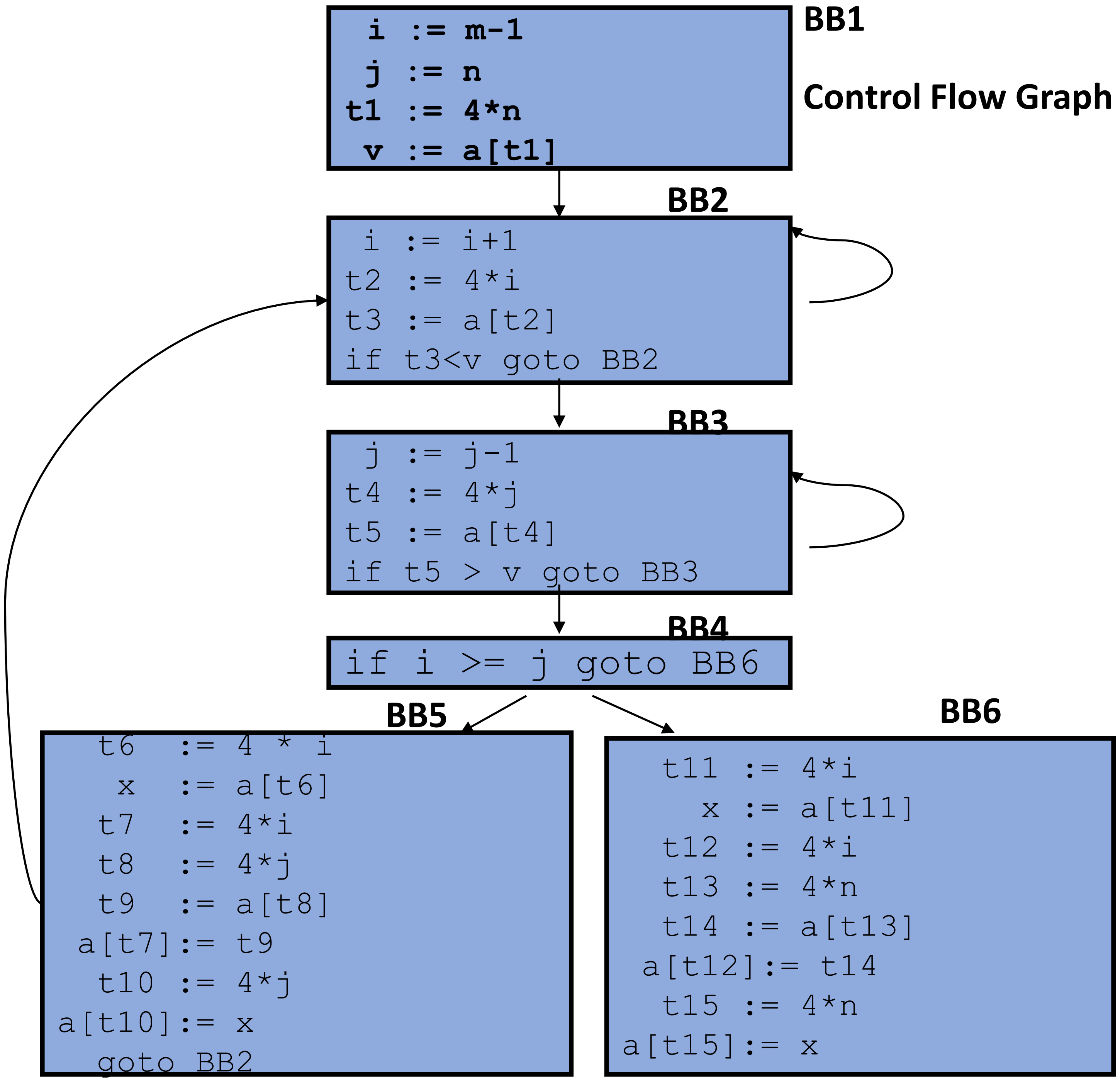


Induction variable Elimination

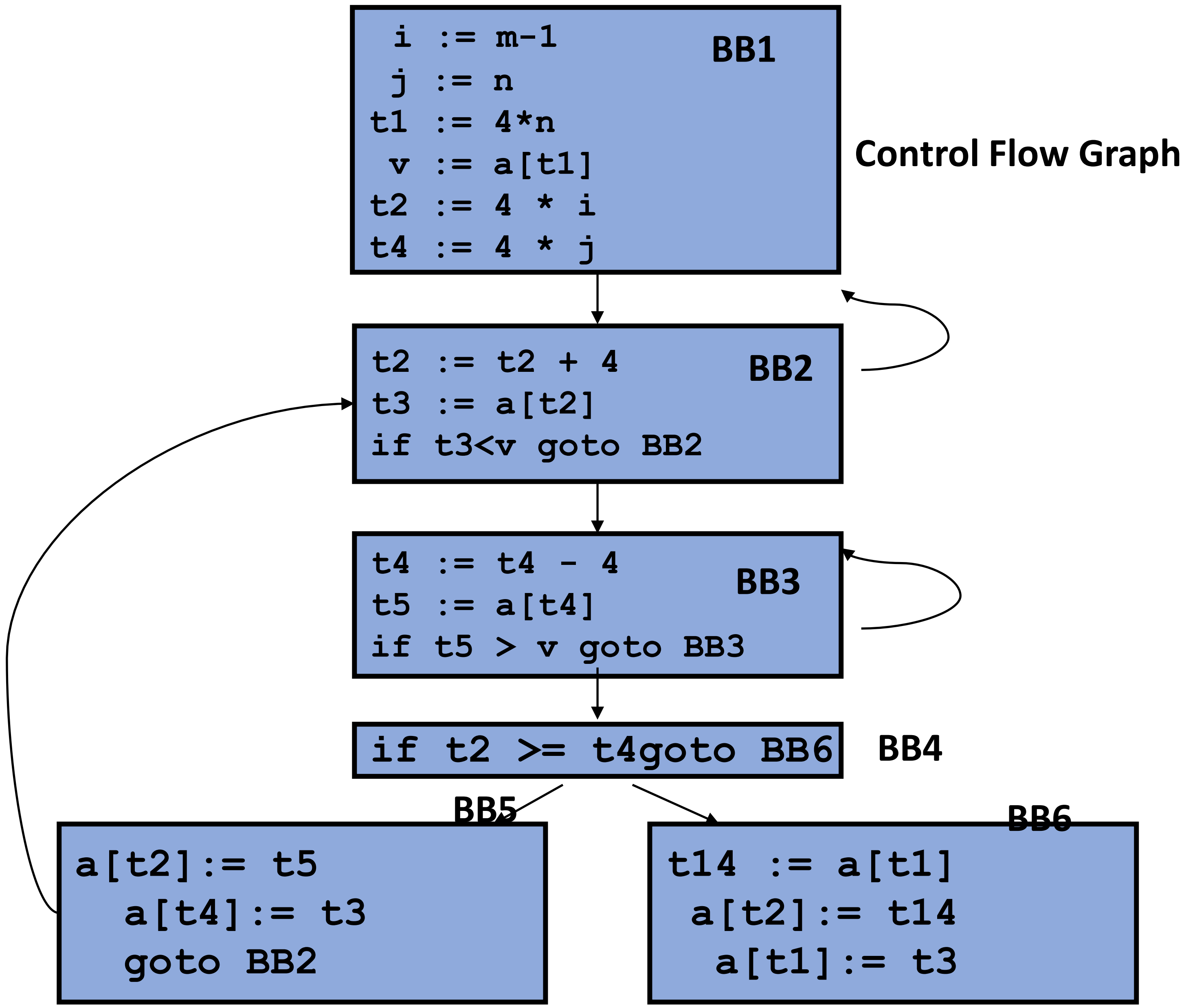
- Loop indices ‘i‘, ‘j’ could be eliminated if the value of ‘i‘ and ‘j’ is not going to be used later


i := m-1
j := n
t1 := 4*n
v := a[t1]
i := i+1
t2 := 4*i
t3 := a[t2]
if t3<v goto BB2
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto BB3
i := m-1
j := n
t1 := 4*n
v := a[t1]
t4 := 4 * j
t2 := 4 * i
i := i + 1
t2 := t2 + 4
t3 := a[t2]
if t3<v goto BB2
j := j-1
t4 := t4 - 4
t5 := a[t4]
if t5 > v goto BB3

Quicksort CFG



Quicksort CFG



Summary

- Example for code optimization
- Basic function preserving transformation
- Algebraic transformation
- Loop Optimization