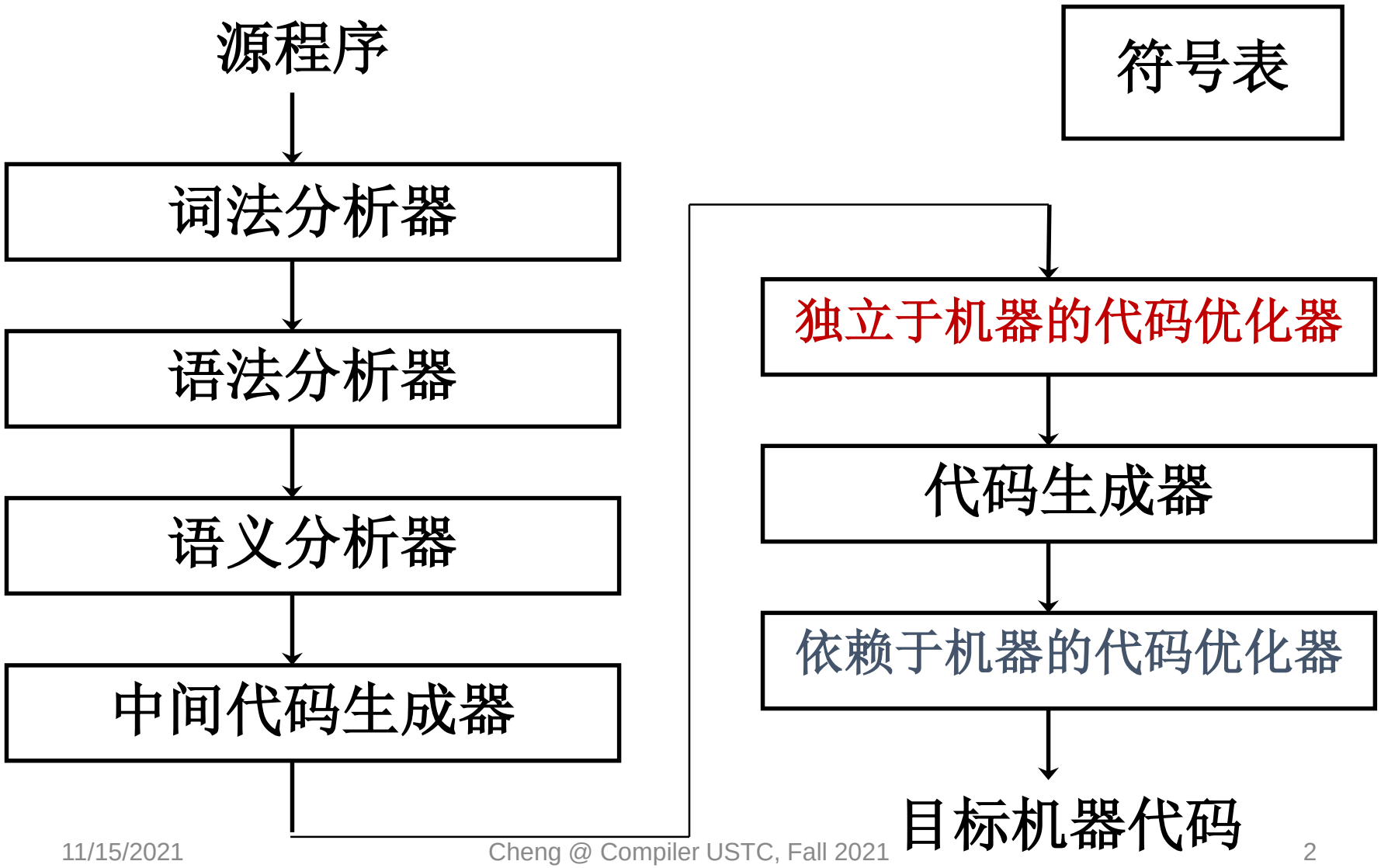
中国科学技术大学
University of Science and Technology of China
USTC, CHINA
ADSLAB
先进数据系统实验室

《编译原理与技术》
独立于机器的优化 I

计算机科学与技术学院
李 诚
2021-11-15

独立于机器的优化


源程序
词法分析器
语法分析器
语义分析器
中间代码生成器
符号表
独立于机器的代码优化器
代码生成器
依赖于机器的代码优化器
目标机器代码

优化的源头和主要种类

- 程序中存在许多程序员无法避免的冗余运算
  - 如A[i][j]和X.f1这样访问数组元素和结构体的域的操作
  - ❖编译后，这些访问操作展开成多步低级算术运算
  - ❖对同一个数据结构多次访问导致许多公共低级运算

机器无关优化概述

- 代码优化
  - 通过程序变换（局部变换和全局变换）来改进程序，称为优化
- 代码优化的种类
  - 基本块内优化、全局优化
  - 公共子表达式删除、复制传播、死代码删除
  - 循环优化
- 代码优化的实现方式
  - 基本块的DAG表示与分析
  - 数据流分析及其一般框架、循环的识别和分析

本章主要内容

- 基本块与流图（revisited）
- 优化策略
  - 公共子表达式删除(common subexpression elimination)
  - 复制传播(copy propagation)
  - 常量合并(constant folding)
  - 死代码删除(dead code elimination)
  - 代码移动(code motion)
  - 强度削弱(strength reduction)
  - 删除归纳变量(induction variable elimination)

图灵奖获得者—法兰·艾伦

- 由于在编译优化方面的杰出贡献被授予2006年计算机图灵奖
- 世界上第一位获得图灵奖的女科学家
- 重要的理论与实践工作有：
  - Program Optimization， 1966
  - Control Flow Analysis， 1970
  - A Basis for Program Optimization, 1970
  - A Catalog of Optimizing Transformations， 1971
  - ……

Frances Allen
(1932-2020)
ACM/IEEE Fellow
美国科学院院士

举例

快速排序程序片段如下

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;

```
(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
```

基本块(Basic block)

- 连续的三地址指令序列，控制流从它的开始进入，并从它的末尾离开，中间没有停止或分支的可能性（末尾除外）

基本块划分算法

- 输入：三地址指令序列
- 输出：基本块列表
- 算法：
  - 首先确定基本块的第一个指令，即首指令(leader)
    - 指令序列的第一条三地址指令是一个首指令
    - 任意转移指令的目标指令是一个首指令
    - 紧跟一个转移指令的指令是一个首指令
  - 然后，每个首指令对应的基本块包括了从它自己开始，直到下一个首指令(不含)或指令序列结尾之间的所有指令

举例


```
(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
```

举例——首指令


```
(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
```

举例——基本块

B_1
(1) i := m –1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]

B_2
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)

B_3
(9) j := j –1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)

B_4
(13) if i >= j goto (23)

B_5
(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)

B_6
(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

流图 (Flow graph)

- 流图的结点是一些基本块
- 从基本块B到基本块C之间有一条边，当且仅当C的第一个指令可能紧跟在B的最后一条指令之后执行
  - B是C的前驱(predecessor)
  - C是B的后继(successor)

流图 (Flow graph)

- 流图的结点是一些基本块
- 从基本块B到基本块C之间有一条边，当且仅当C的第一个指令可能紧跟在B的最后一条指令之后执行，**判定方法如下**：
  - 有一个从B的结尾跳转到C的开头的跳转指令
  - 参考原来三地址指令序列中的顺序，C紧跟在B之后，且B的结尾没有无条件跳转指令

举例——流图

B_1
(1) i := m –1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]

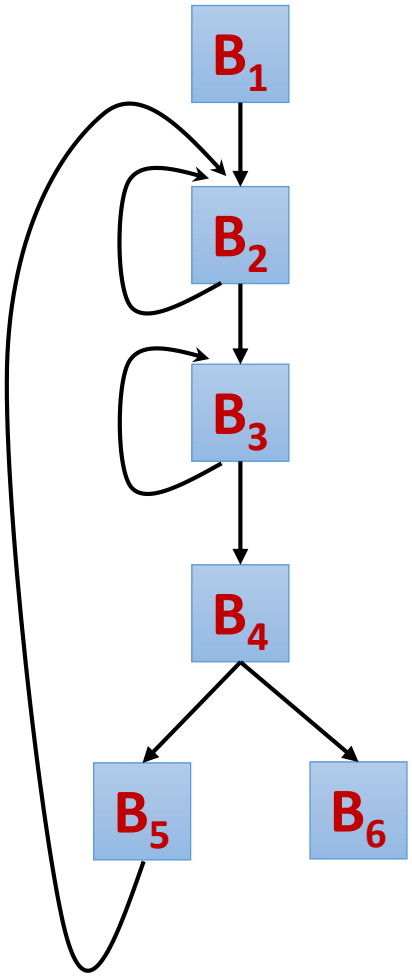
B_2
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)

B_3
(9) j := j –1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)

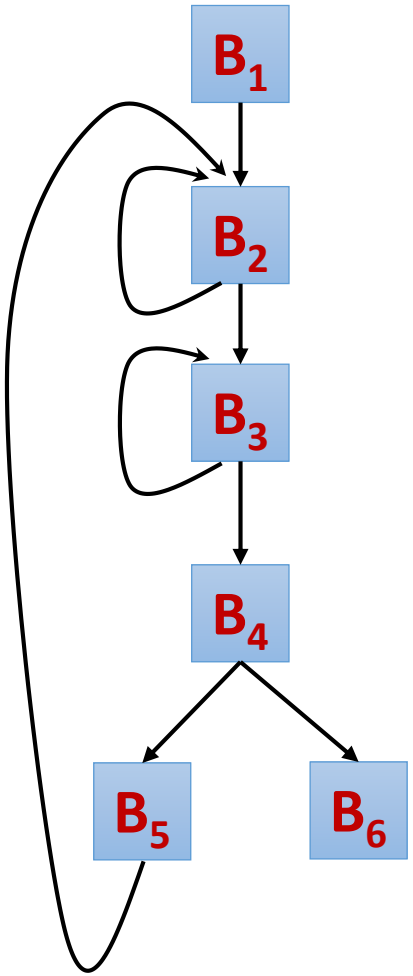
B_4
(13) if i >= j goto (23)

B_5
(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)

B_6
(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



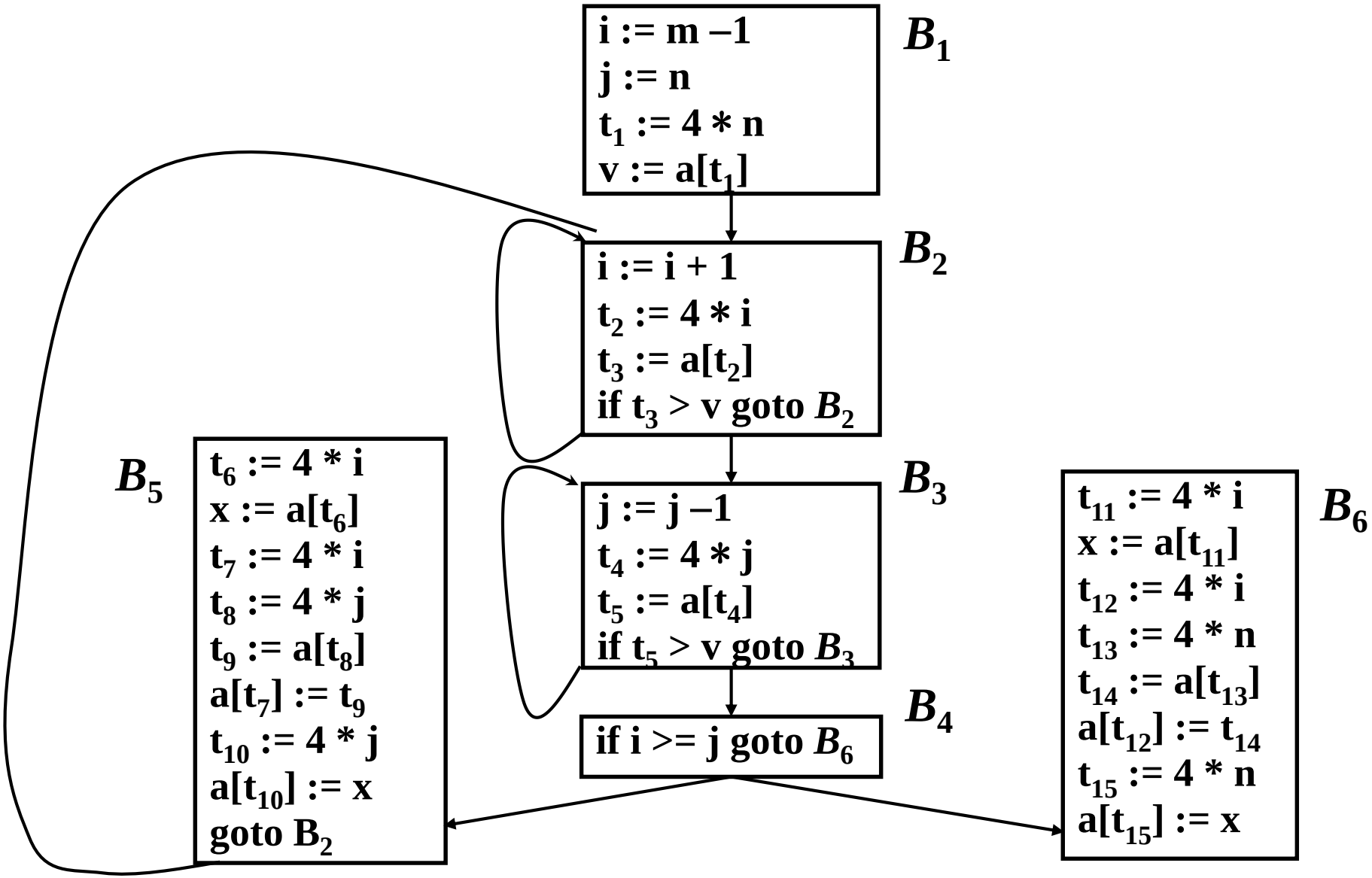
- 流图中的一个结点集合L是一个循环，如果它满足：
  - 该集合有唯一的入口结点
  - 任意结点都有一个到达入口结点的非空路径，且该路径全部在L中
- 不包含其他循环的循环叫做内循环
- 右图中的循环
  - B_2 自身
  - B_3 自身
  - $\{B_2, B_3, B_4, B_5\}$



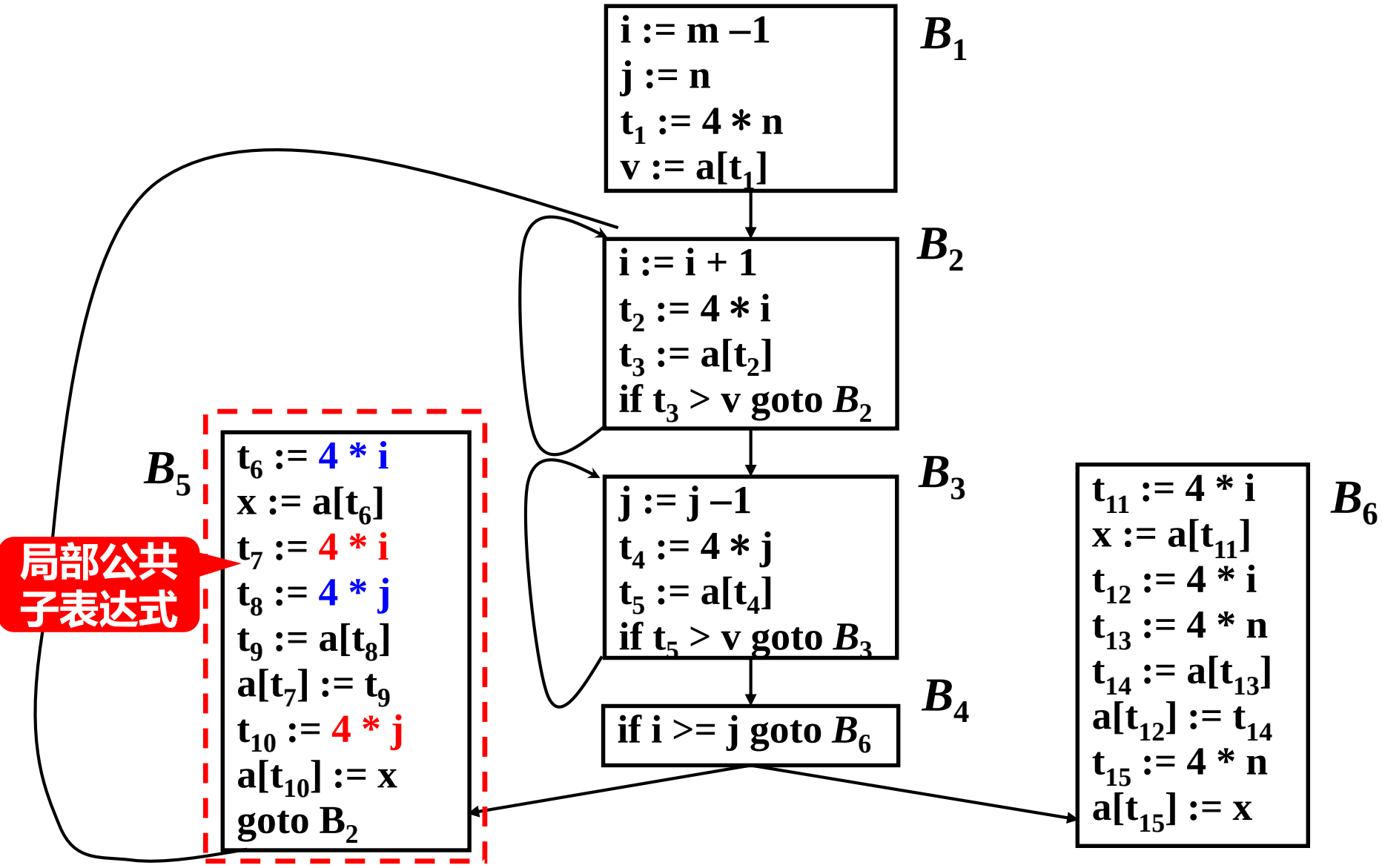
本章主要内容

- 基本块与流图
- 优化策略
  - 公共子表达式删除(common subexpression elimination)
  - 复制传播(copy propagation)
  - 常量合并(constant folding)
  - 死代码删除(dead code elimination)
  - 代码移动(code motion)
  - 强度削弱(strength reduction)
  - 删除归纳变量(induction variable elimination)

优化举例-公共子表达式删除



优化举例-公共子表达式删除



公共子表达式删除

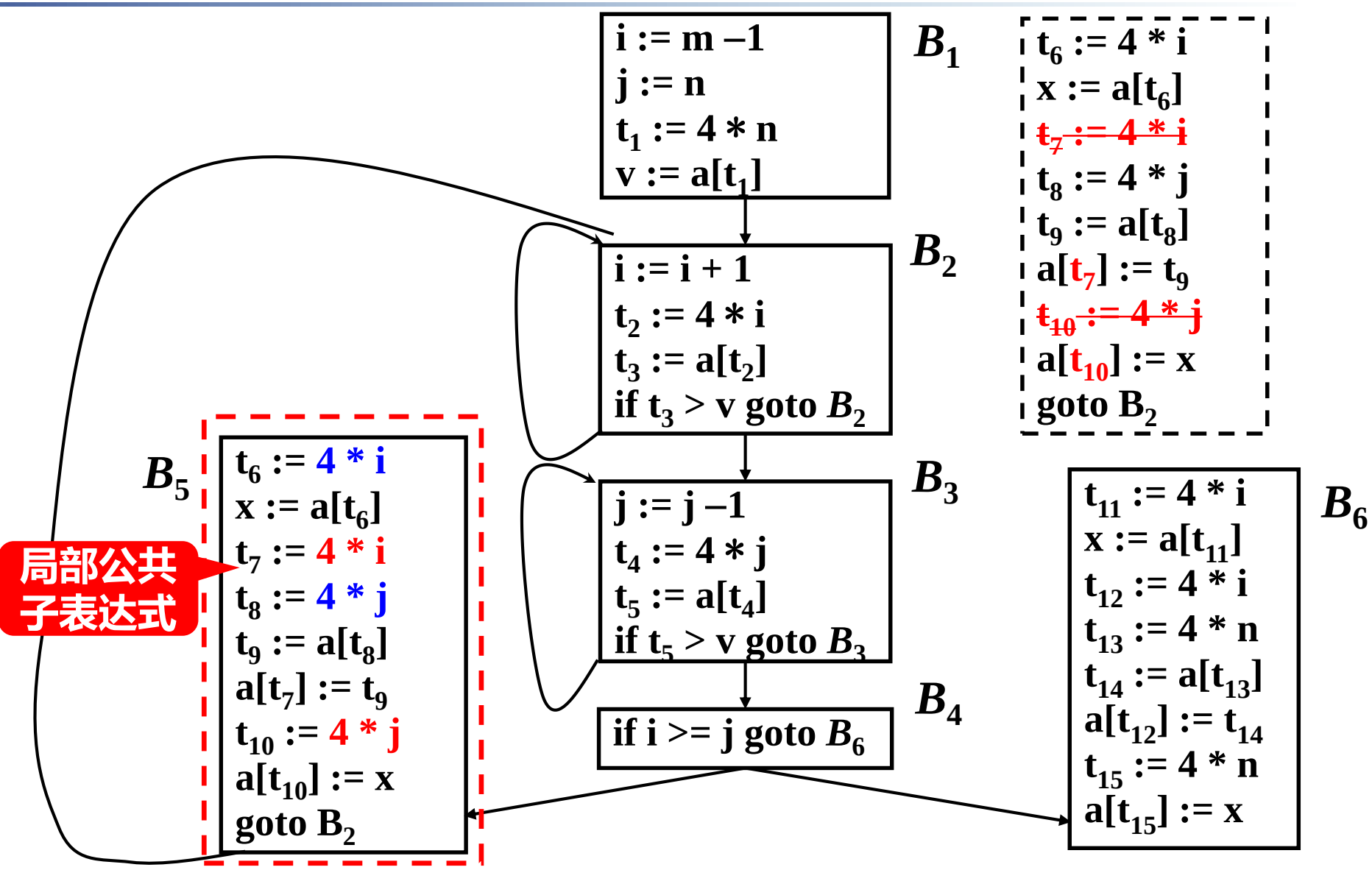
❑公共子表达式

- ❖如果表达式x op y之前已经被计算过，并且到现在为止，x op y中的变量的值没有发生改变，那么该表达式的本次出现就成为公共子表达式

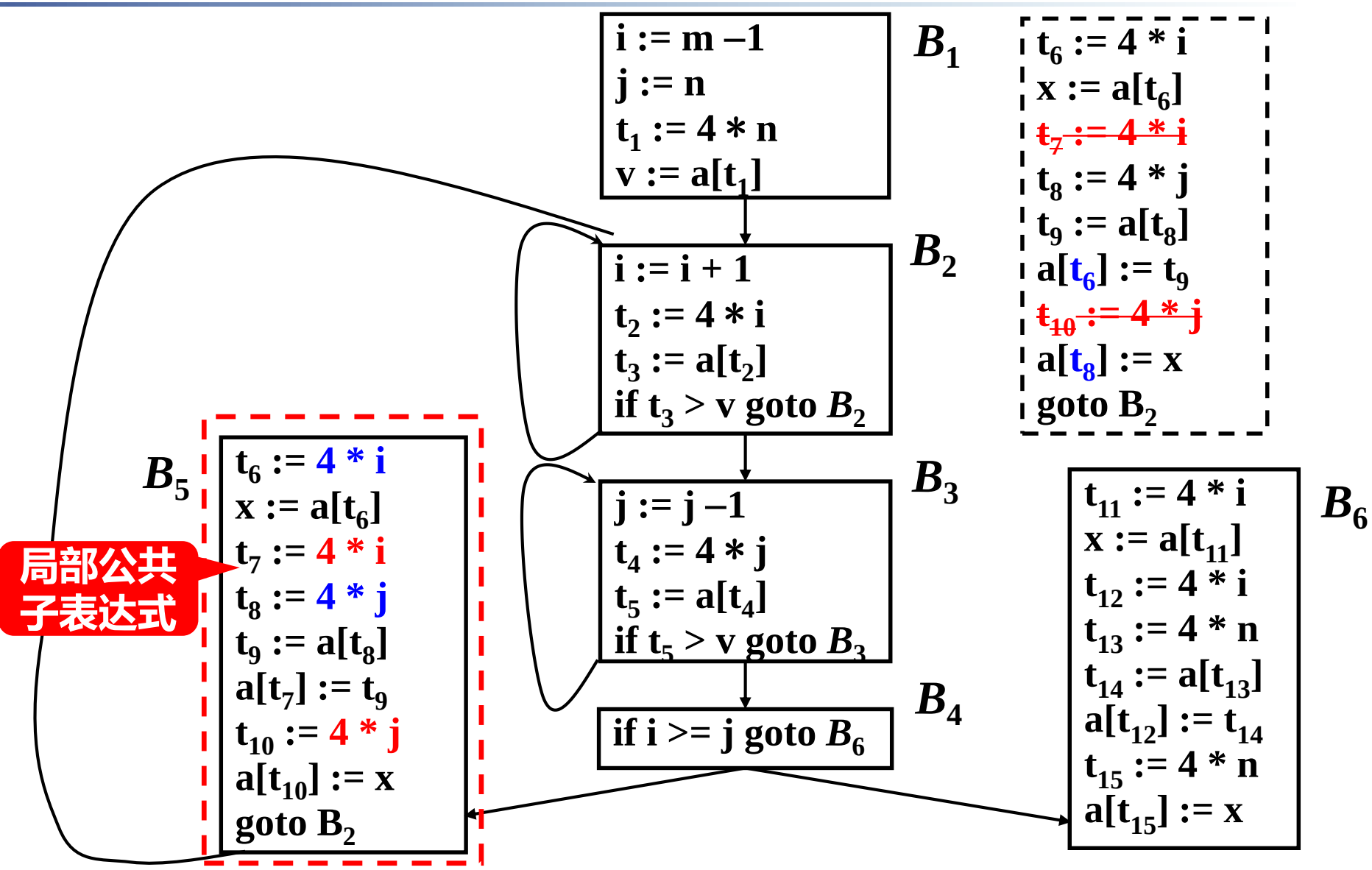
❑潜在优化可能：公共子表达式的删除

- ❖公共子表达式可被上一次计算值替代

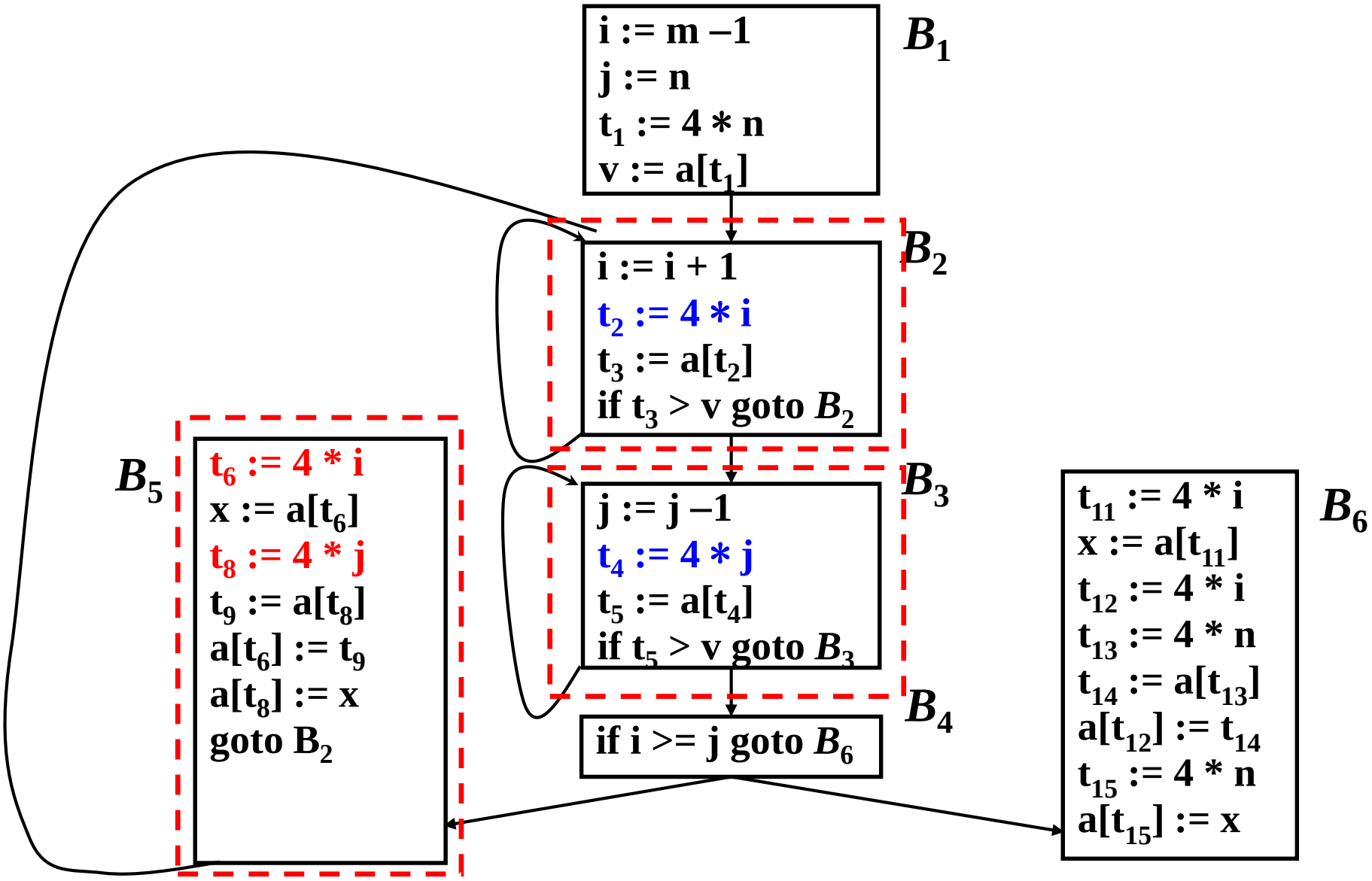
优化举例-公共子表达式删除



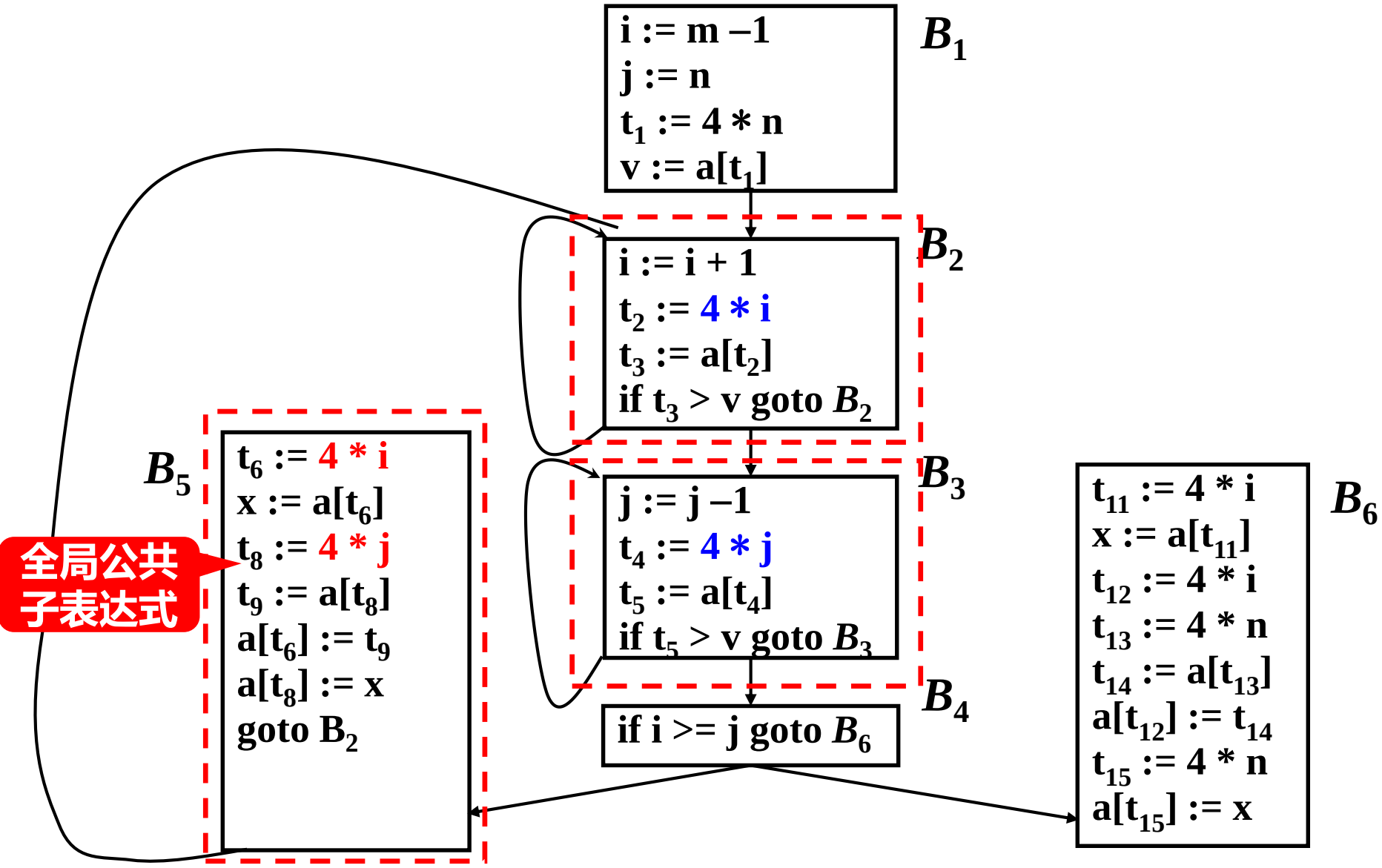
优化举例-公共子表达式删除



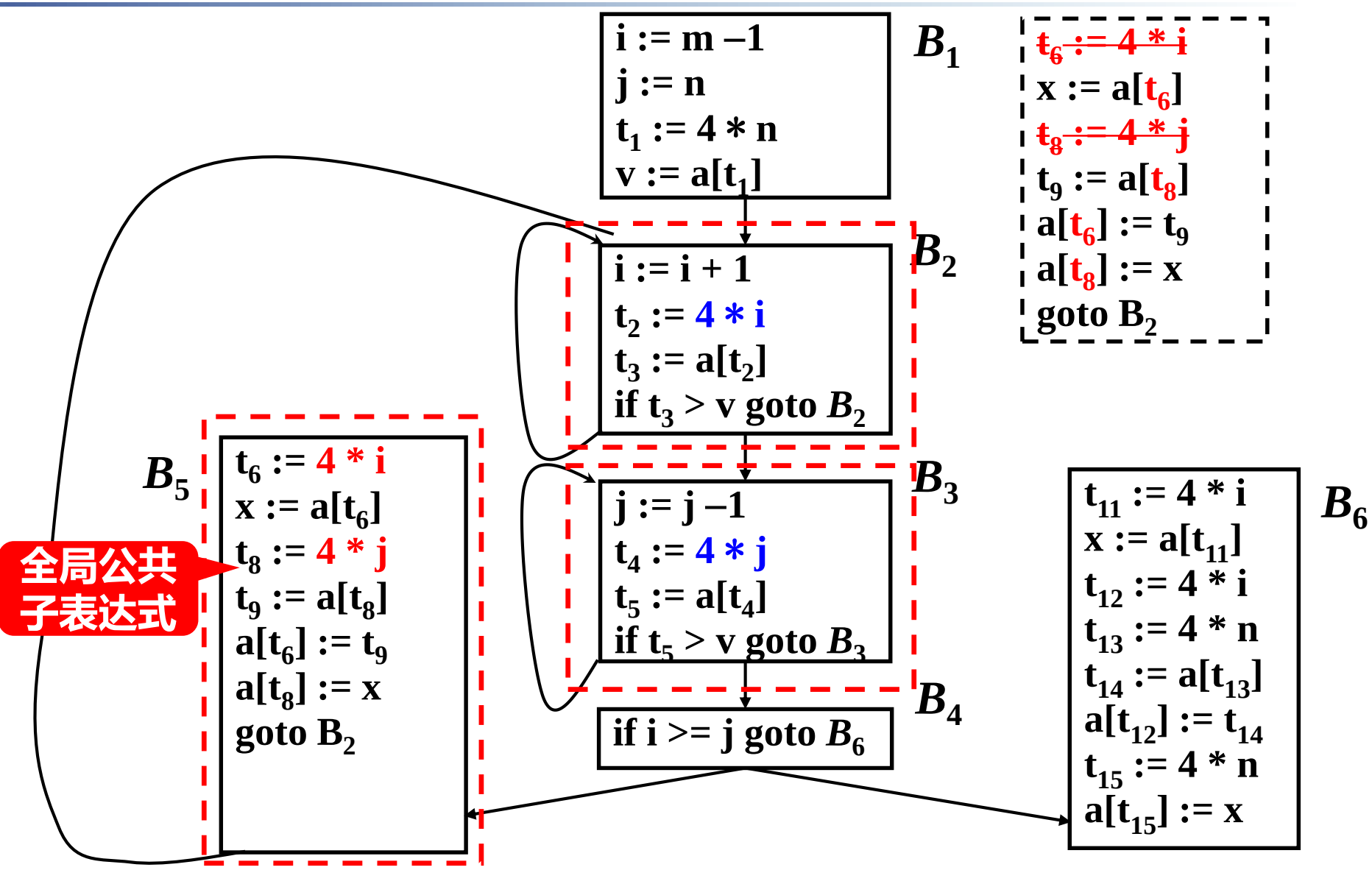
优化举例-公共子表达式删除



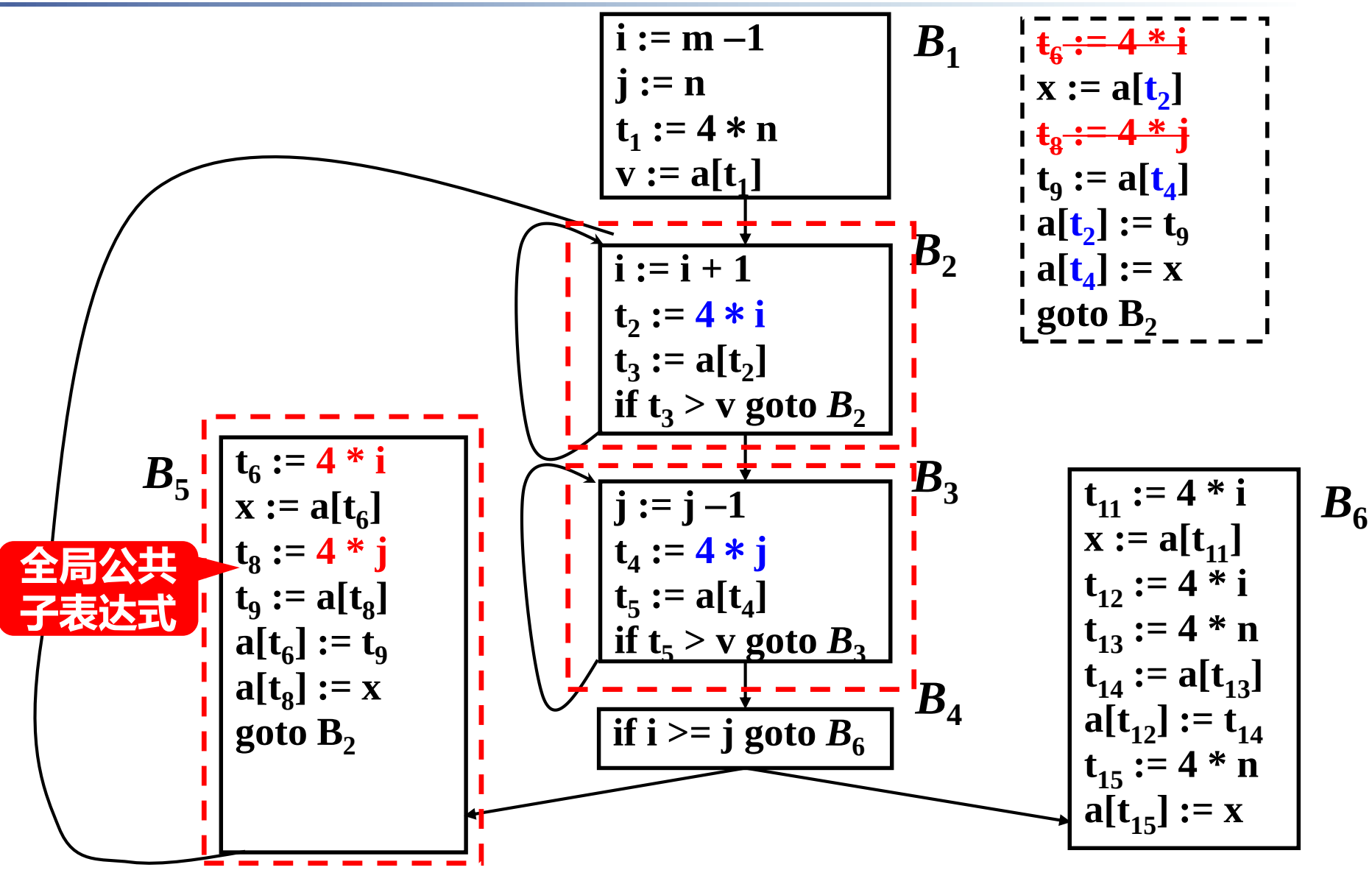
优化举例-公共子表达式删除



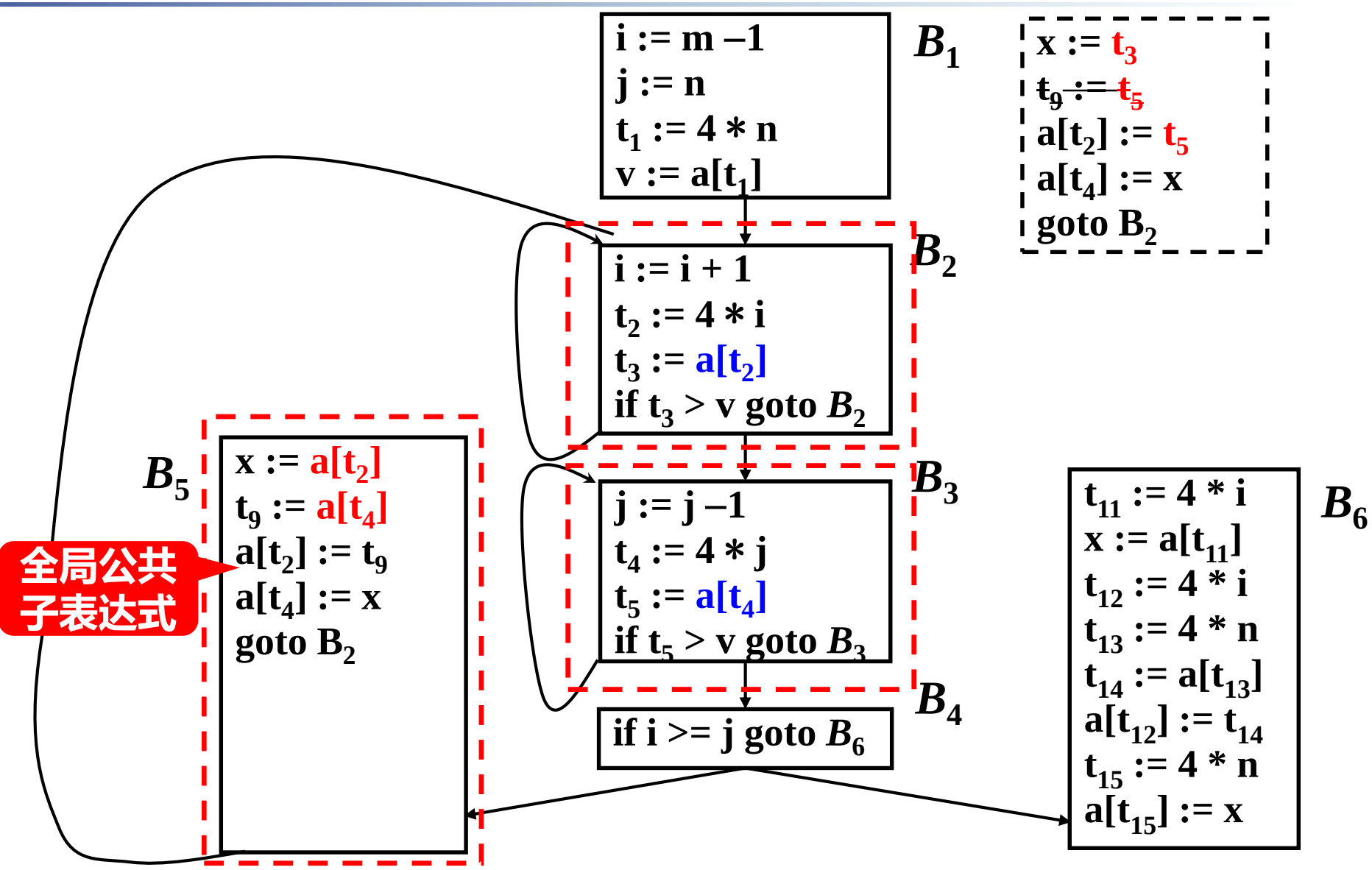
优化举例-公共子表达式删除



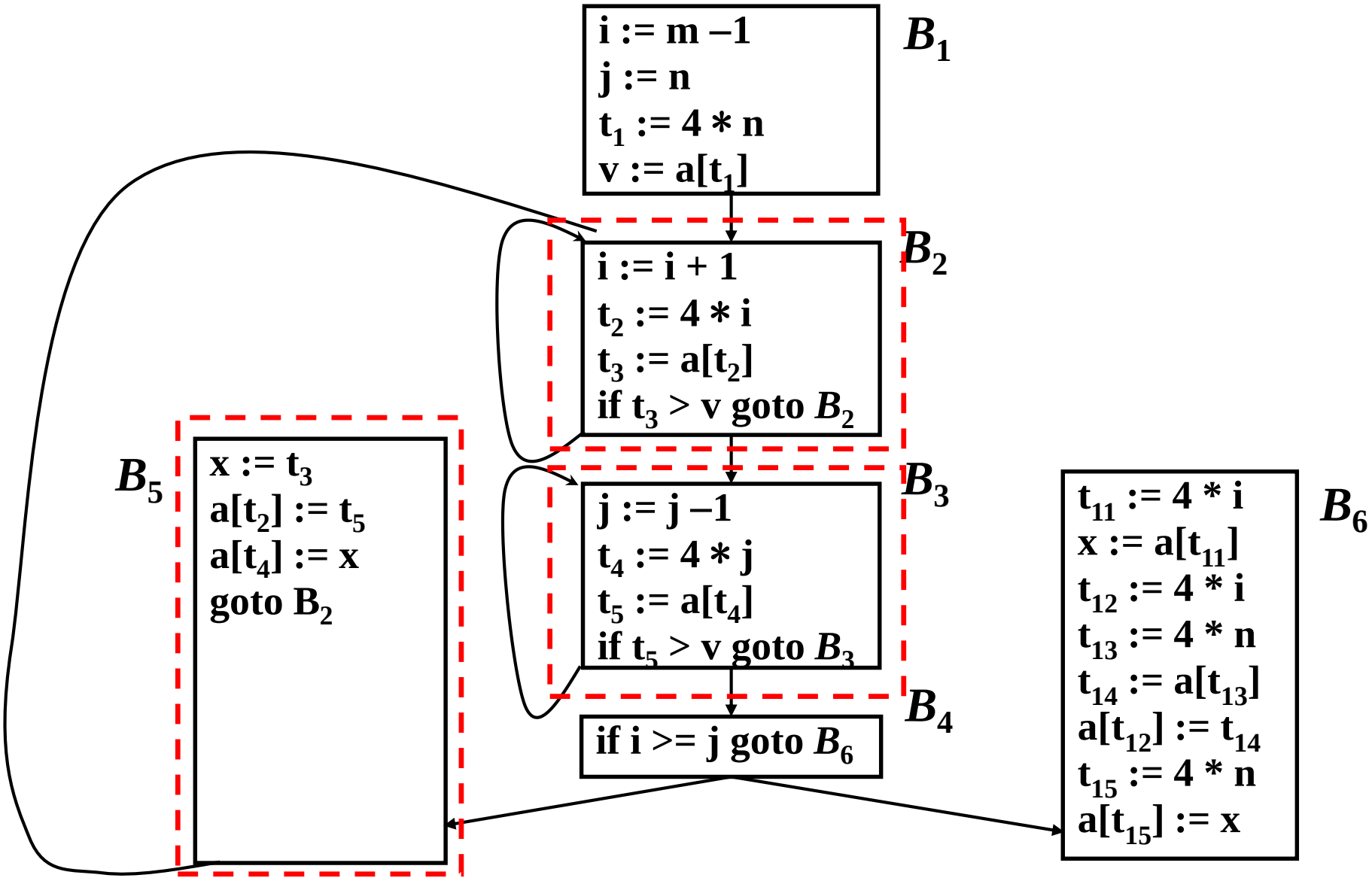
优化举例-公共子表达式删除



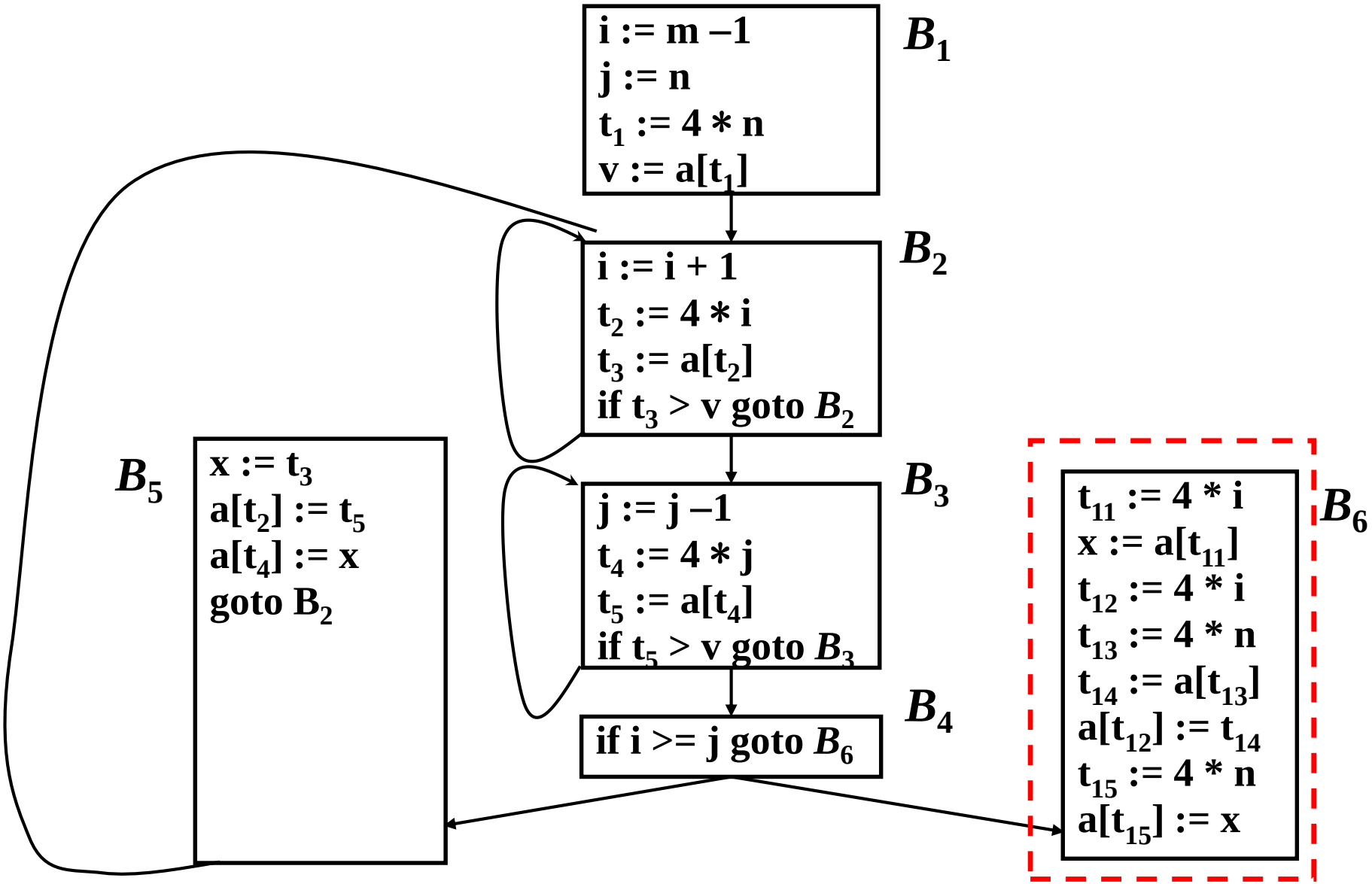
优化举例-公共子表达式删除



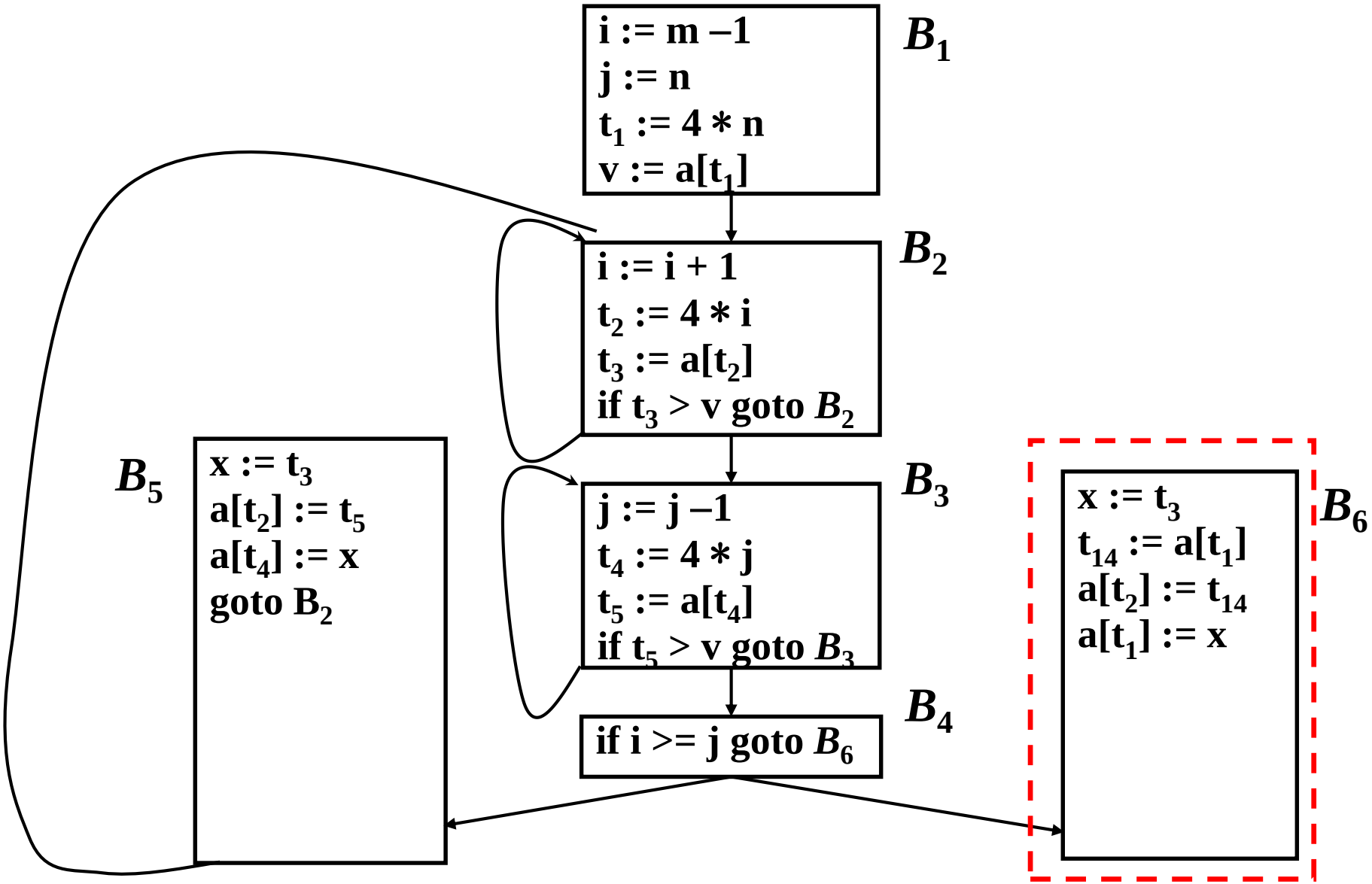
优化举例-公共子表达式删除



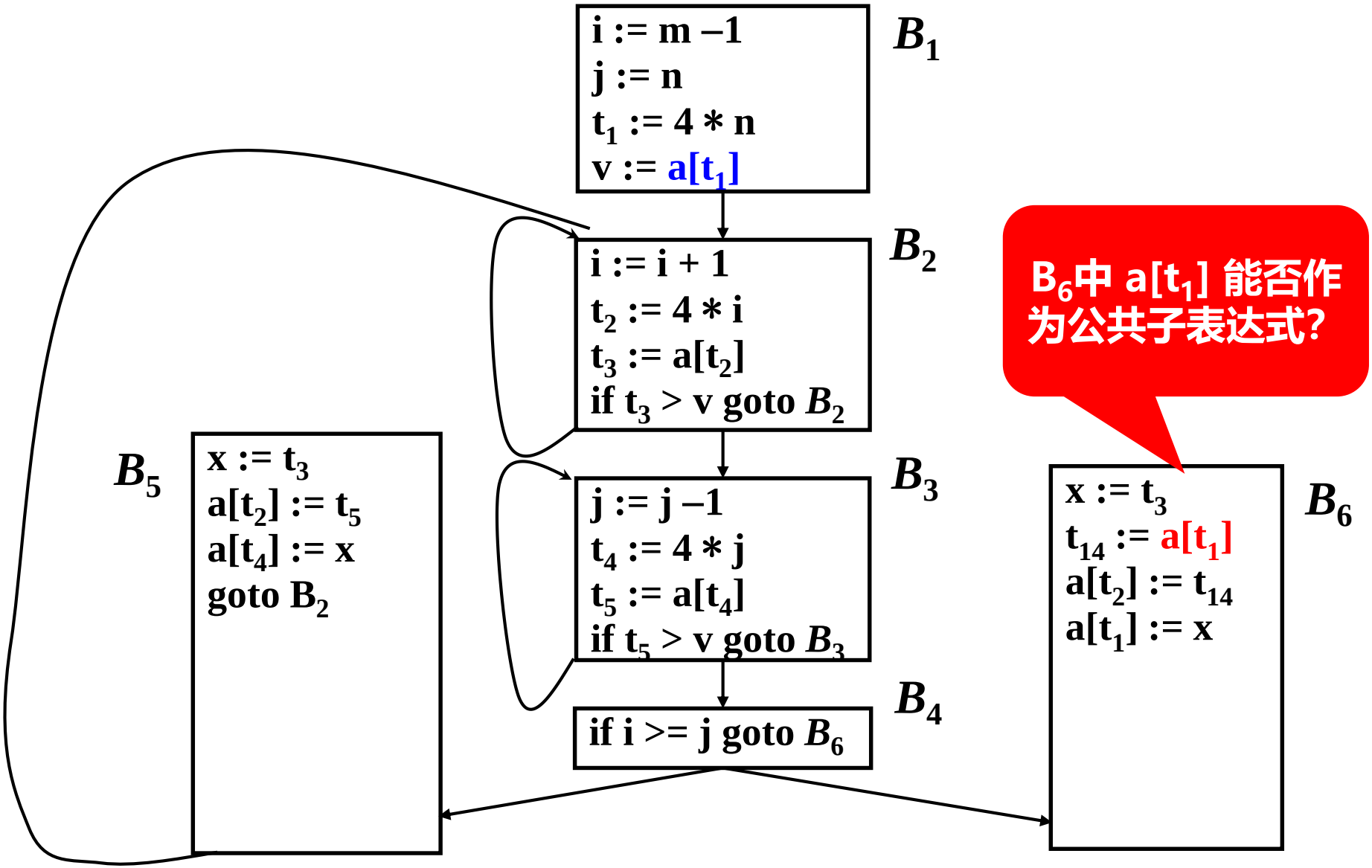
优化举例-公共子表达式删除



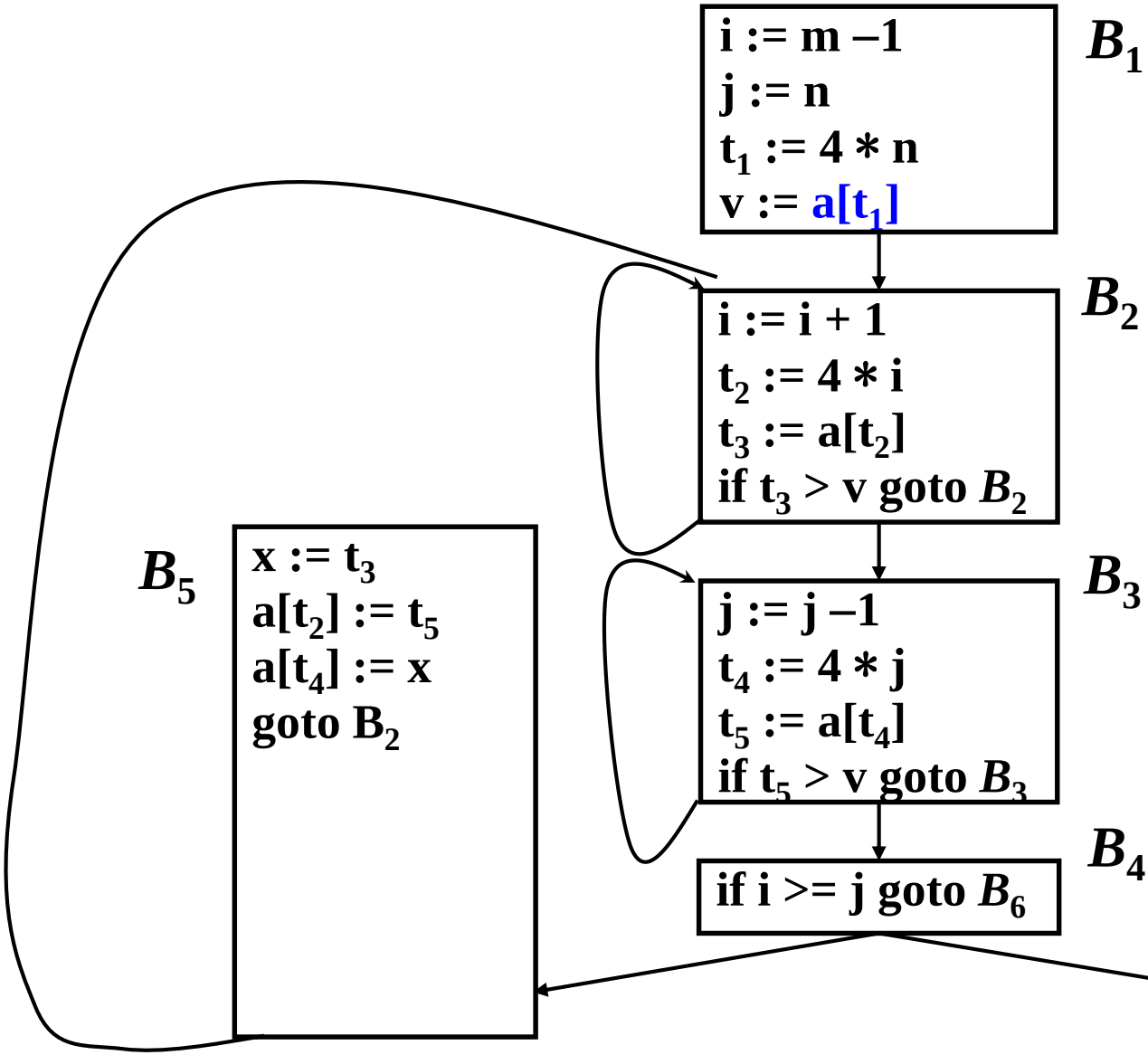
优化举例-公共子表达式删除



优化举例-公共子表达式删除



优化举例-公共子表达式删除



把$a[t_1]$作为公共子表达式是不稳妥的

因为B_5有对下标变量$a[t_2]$和$a[t_4]$的赋值

本章主要内容

- 基本块与流图
- 优化策略
  - 公共子表达式删除(common subexpression elimination)
  - 复制传播(copy propagation)
  - 常量合并(constant folding)
  - 死代码删除(dead code elimination)
  - 代码移动(code motion)
  - 强度削弱(strength reduction)
  - 删除归纳变量(induction variable elimination)

打包一起讲

死代码删除

❑死代码是指计算的结果永远不被引用的语句

例： 为便于调试，可能在程序中加打印语句，测试后改成右边的形式

debug = true;	debug = false;
. . .	. . .
if (debug) print …	if (debug) print …

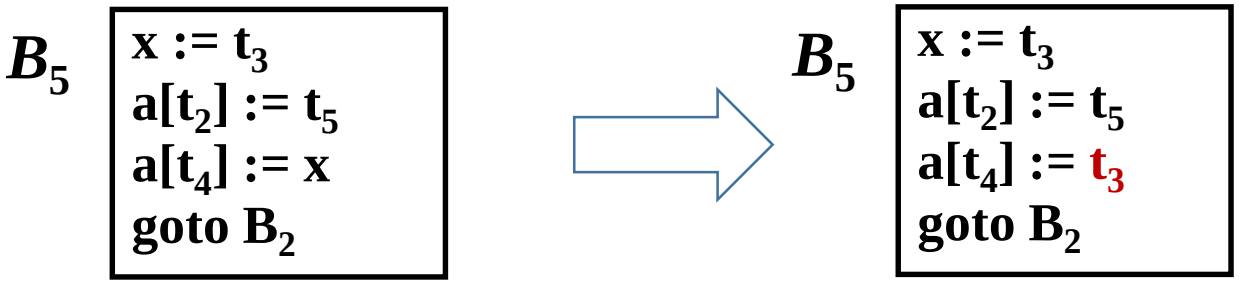
死代码删除

- 死代码是指计算的结果永远不被引用的语句
- 一些优化变换可能会引起死代码
  - 如：复制传播、常量合并

例： 为便于调试，可能在程序中加打印语句，测试后改成右边的形式

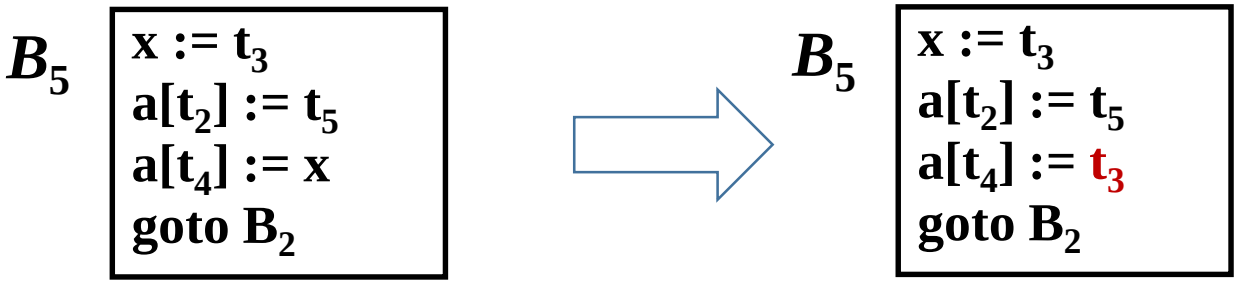
debug = true;	debug = false;
. . .	. . .
if (debug) print …	if (debug) print …

❑定义：在复制语句x = y之后尽可能用y代替x



复制传播

- 定义：在复制语句x = y之后尽可能用y代替x

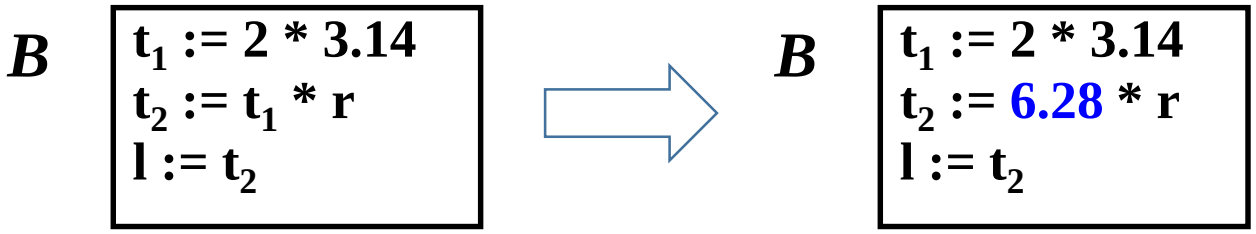


- 常用的公共子表达式删除和其他一些优化会引入一些复制语句
- 复制传播本身没有优化的意义，但可以给死代码删除创造机会

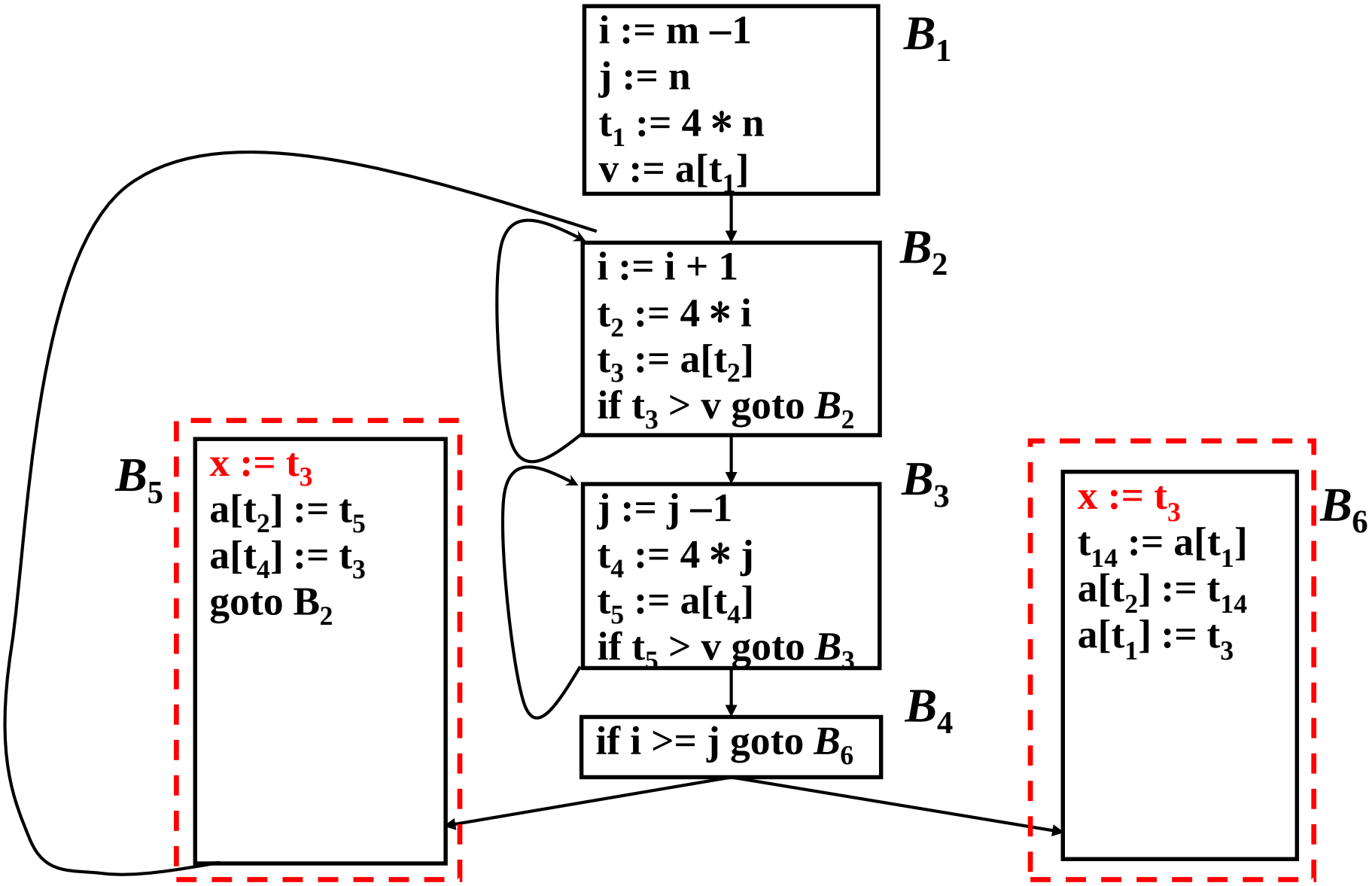
常量合并

❑如果在编译时刻推导出一个表达式的值是常量，就可以使用该常量来代替这个表达式。

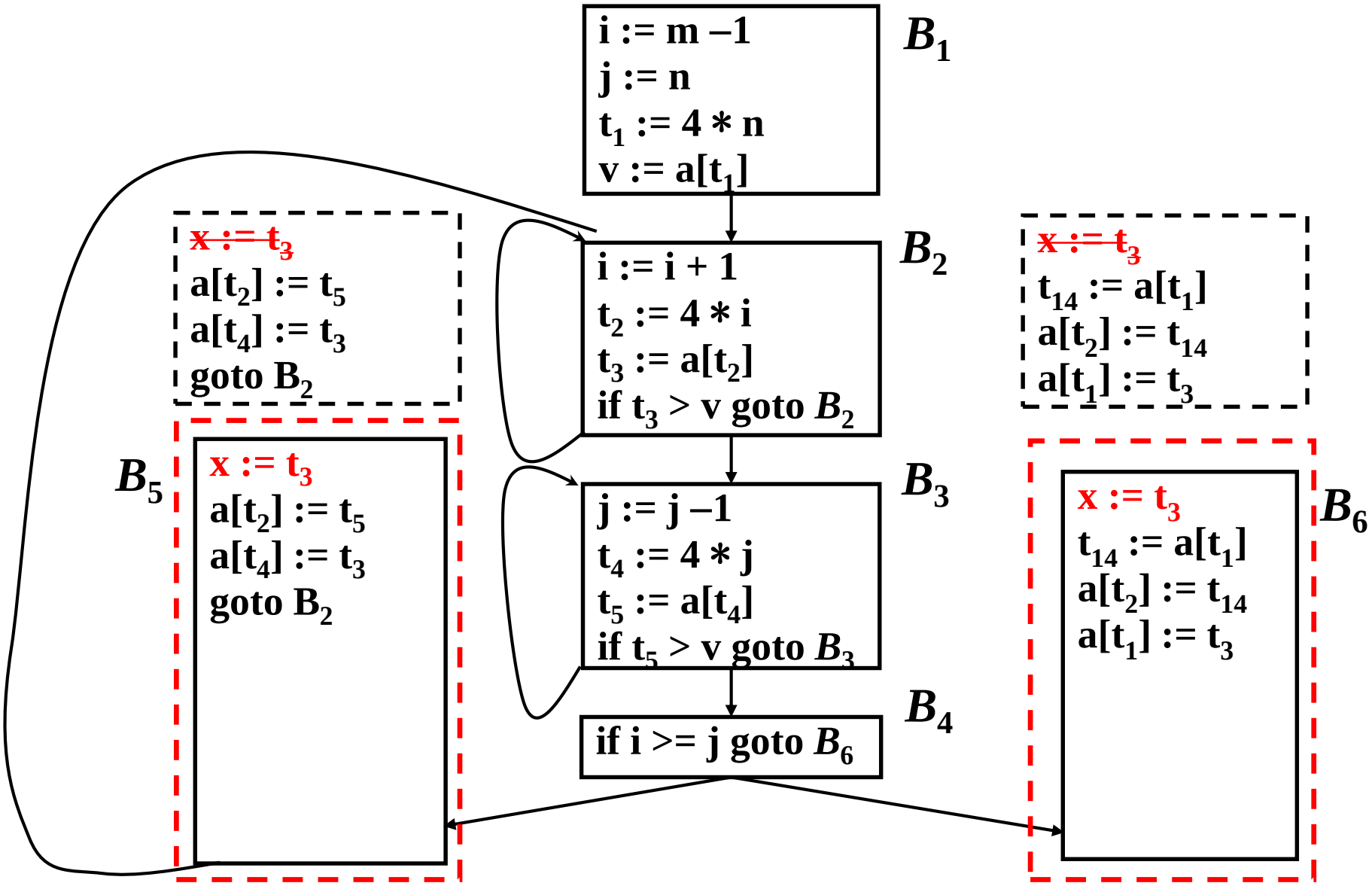
❖例：计算圆周长的表达式 $l = 2*3.14*r$



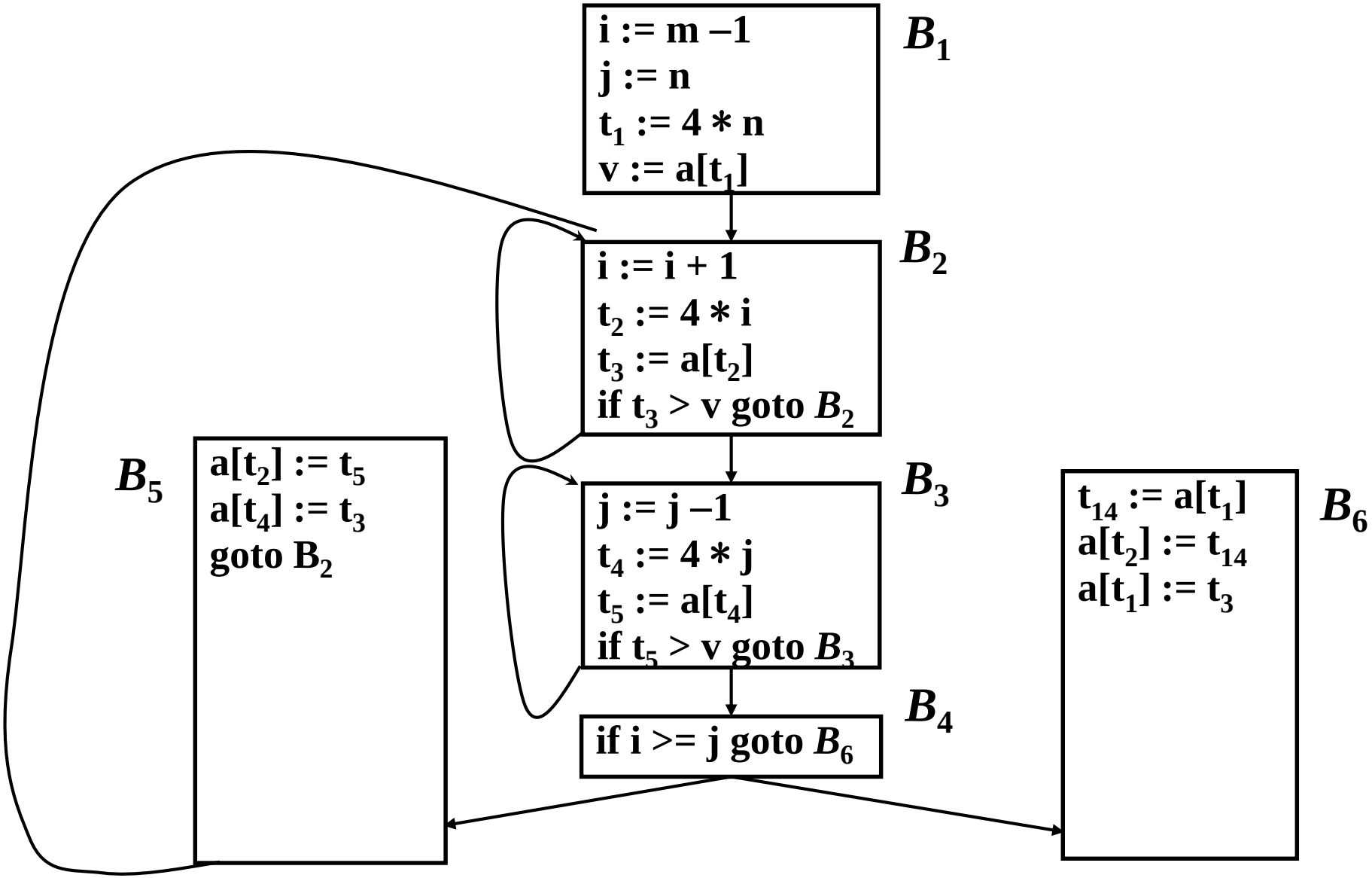
优化举例-死代码删除


i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 > v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x := t3
a[t2] := t5
a[t4] := t3
goto B2
B5
x := t3
t14 := a[t1]
a[t2] := t14
a[t1] := t3
B6

优化举例-死代码删除

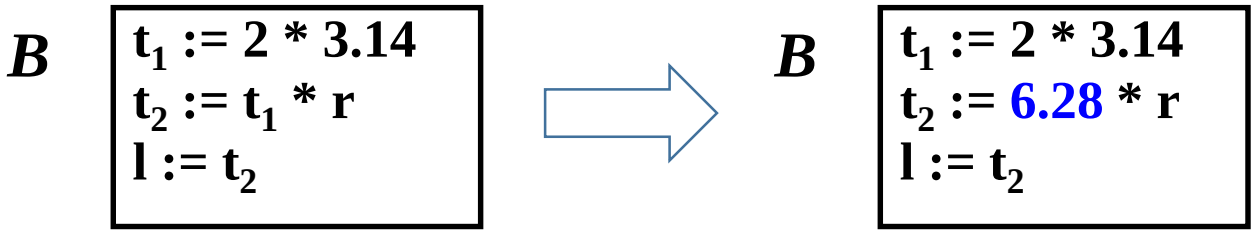

i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 > v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x := t3
a[t2] := t5
a[t4] := t3
goto B2
B5
x := t3
t14 := a[t1]
a[t2] := t14
a[t1] := t3
B6

优化举例-死代码删除


i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 > v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
a[t2] := t5
a[t4] := t3
goto B2
B5
t14 := a[t1]
a[t2] := t14
a[t1] := t3
B6

常量合并

- 如果在**编译时刻**推导出**一个表达式的值是常量**，就可以使用该常量来代替这个表达式。
  - 例：计算圆周长的表达式 $l = 2*3.14*r$



- 常量合并本身没有优化的意义，**但可以给死代码删除创造机会**

本章主要内容

- 基本块与流图
- 优化策略
  - 公共子表达式删除(common subexpression elimination)
  - 复制传播(copy propagation)
  - 常量合并(constant folding)
  - 死代码删除(dead code elimination)
  - 代码移动(code motion)
  - 强度削弱(strength reduction)
  - 删除归纳变量(induction variable elimination)

涉及循环优化打包一起讲

代码移动

- 循环不变计算（loop-invariant computation）是指不管循环执行多少次都得到相同结果的表达式
- 代码移动是循环优化的一种，在进入循环前就对循环不变计算进行求值

代码移动

- 循环不变计算（loop-invariant computation）是指不管循环执行多少次都得到相同结果的表达式
- 代码移动是循环优化的一种，在进入循环前就对循环不变计算进行求值

例：while (i <= limit – 2) …

代码移动后变换成

```
t = limit – 2;
while (i <= t ) …
```

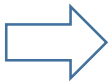
代码移动

- 循环不变计算（loop-invariant computation）是指不管循环执行多少次都得到相同结果的表达式
- 代码移动是循环优化的一种，在进入循环前就对循环不变计算进行求值。
- 对于多重嵌套循环， loop-invariant computation是相对于某一个循环的，可能对于更加外层的循环，它就不成立了。
- 因此，处理循环时，按照由里到外的方式

❑用较快的操作代替较慢的操作

- ❖如用加代替乘
- ❖课外阅读：Booth算法

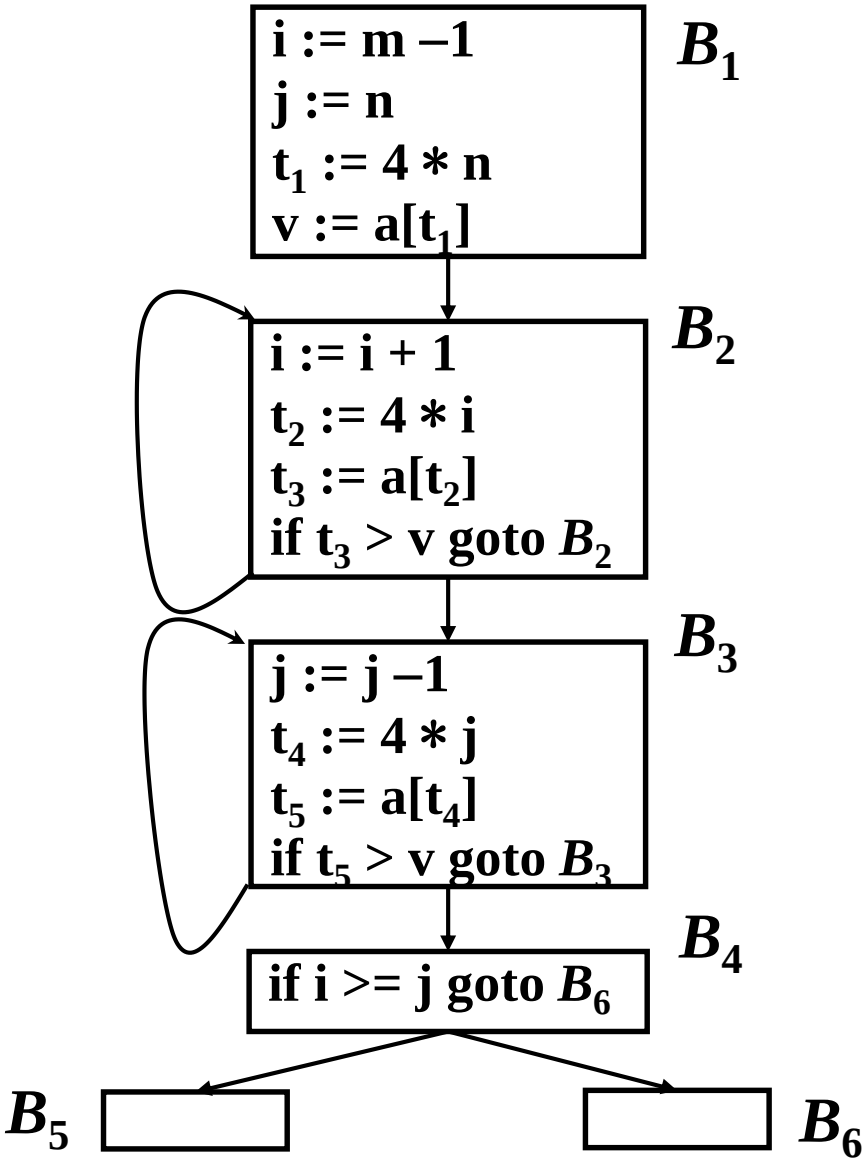
2 * x 或者 2.0 * x	x + x
x/2	x * 0.5
x^2	x * x
$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$	$((\ldots (a_n x + a_{n-1})x + a_{n-2})\ldots)x + a_1) + a_0$



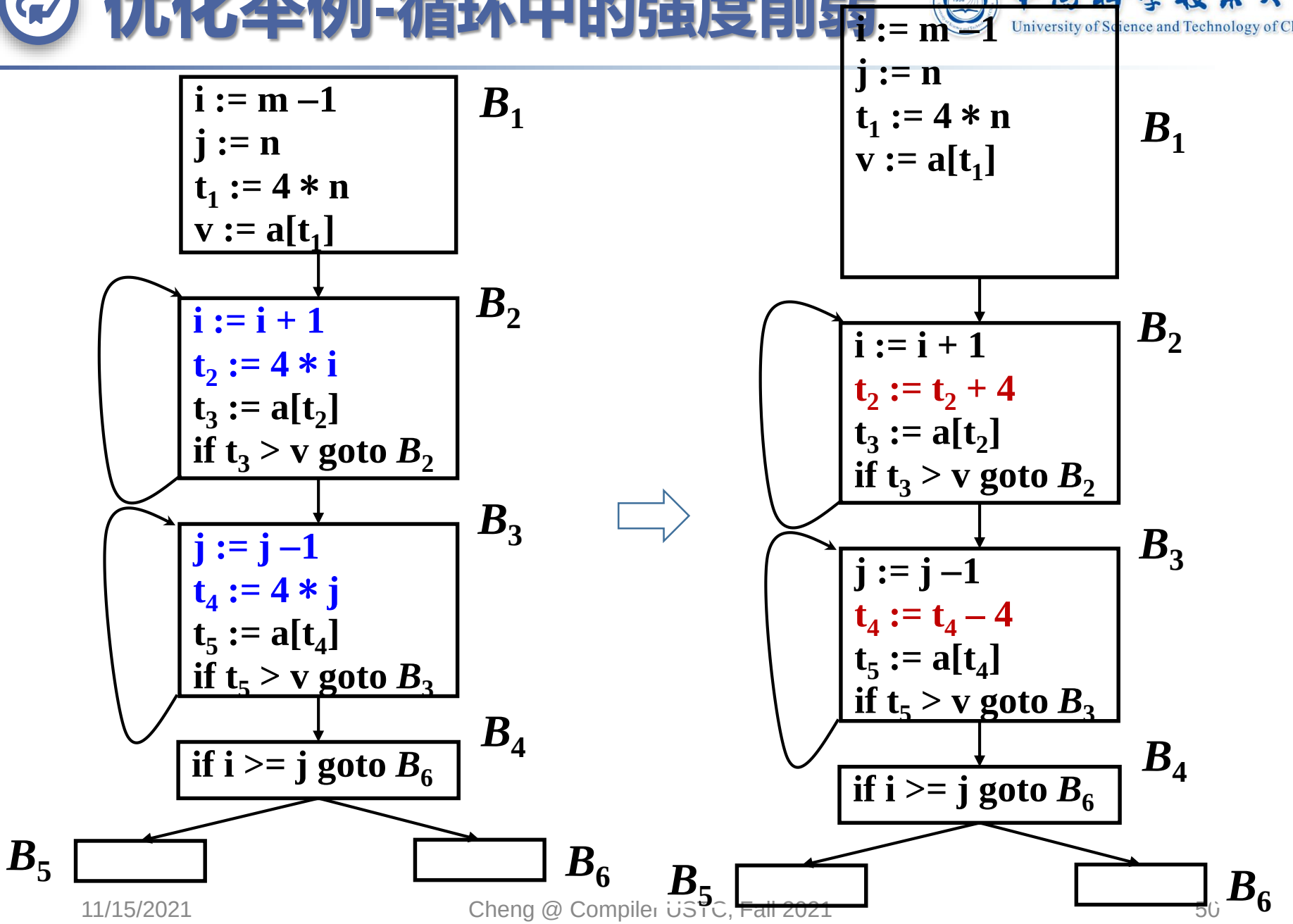
循环中的强度削弱

- 归纳变量：
  - 对于一个变量x，如果存在一个正的或者负的常数c使得每次x被赋值时它的值总增加c，那么x就成为归纳变量(Induction variable)
  - 从里向外分析循环
- 可能的优化：
  - 通过增量运算（加或减）来计算归纳变量
  - 将步调一致的一组归纳变量合并为一个

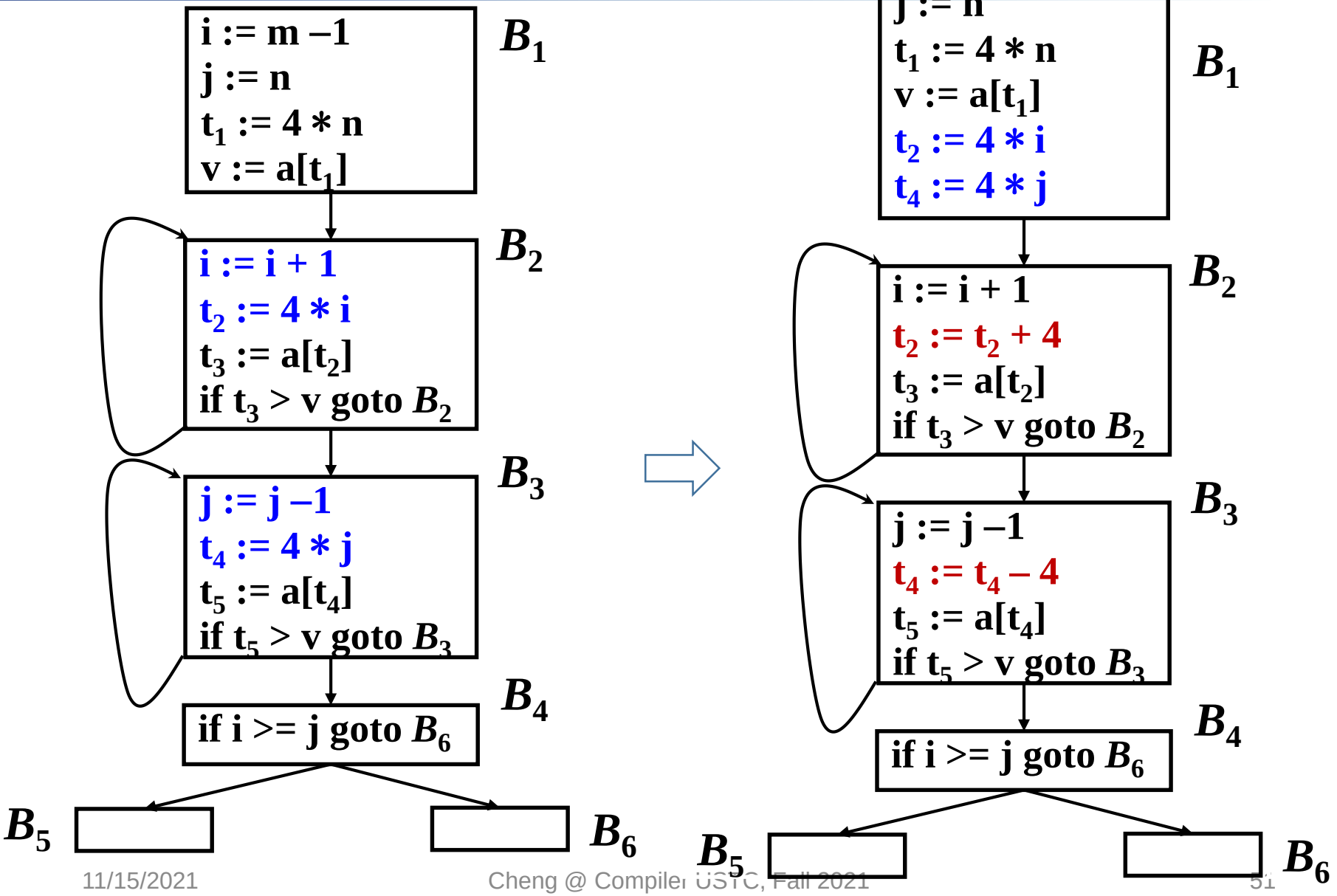
优化举例-循环中的强度削弱



优化举例-循环中的强度削弱



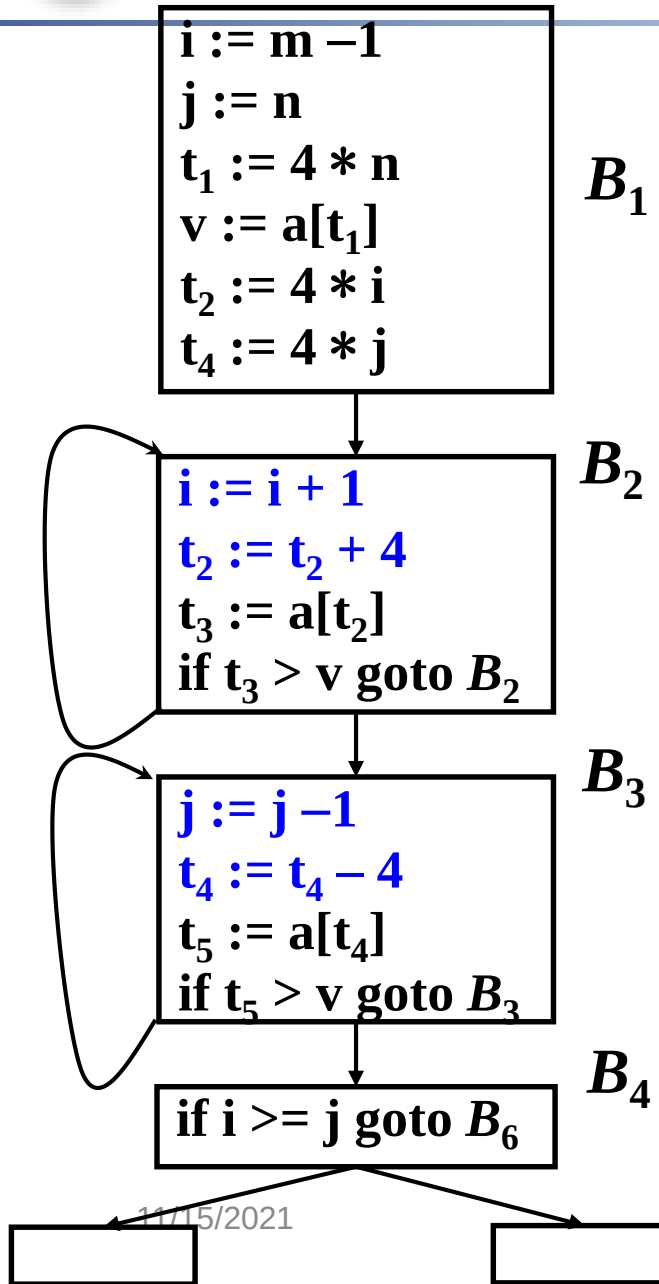
优化举例-循环中的强度削弱



删除归纳变量

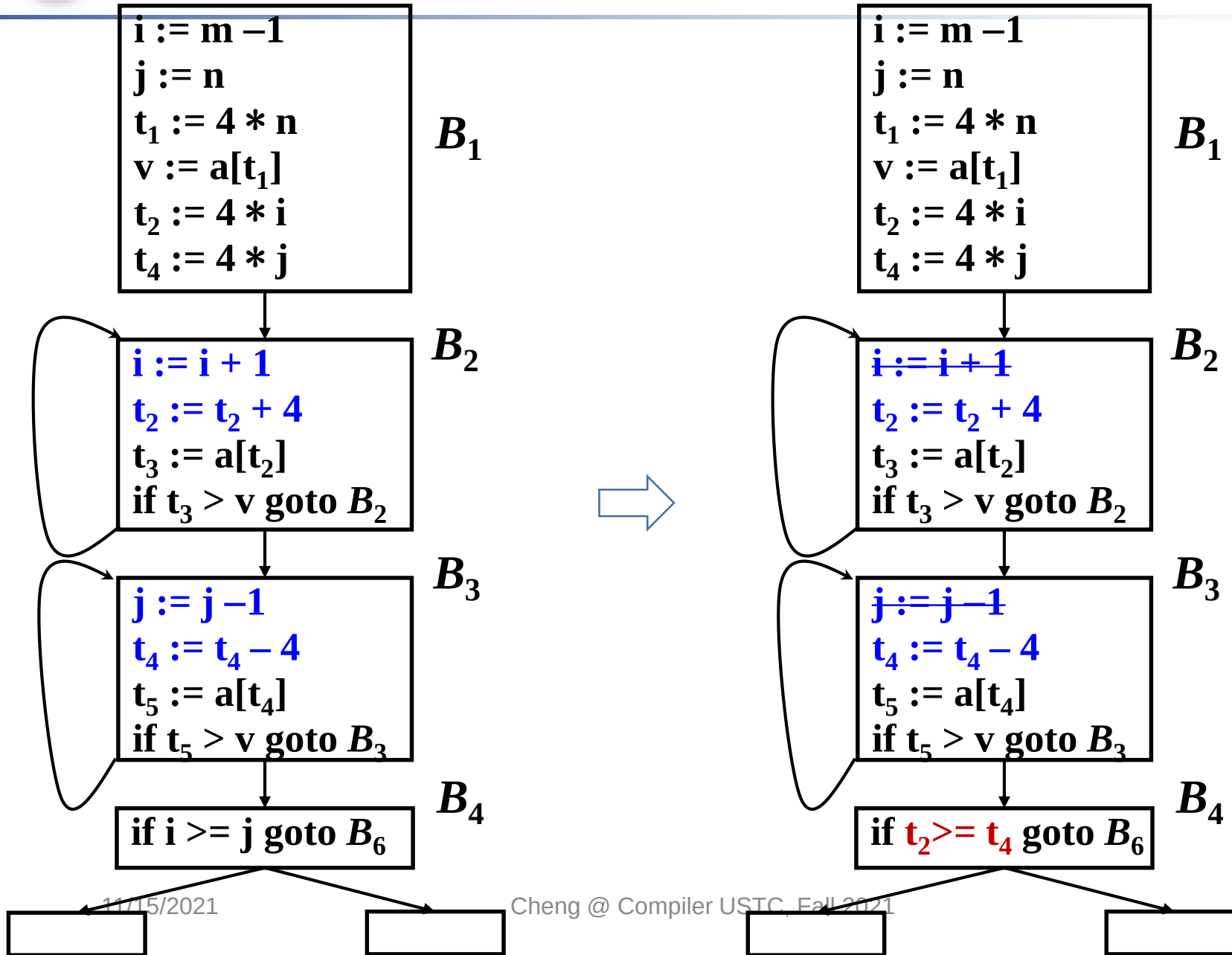
- 在程序循环的执行过程中，如果有一组归纳变量的值的变化保持步调一致，通常可以只保留一个。
- 强度削弱可以给删除归纳变量提供机会。

优化举例-删除归纳变量



- 在程序循环的执行过程中，如果有一组归纳变量的值的变化保持步调一致，通常可以只保留一个。
- 强度削弱可以给删除归纳变量提供机会。

优化举例-删除归纳变量


i := m –1
j := n
t1 := 4 * n
v := a[t1]
t2 := 4 * i
t4 := 4 * j
B1
i := i + 1
t2 := t2 + 4
t3 := a[t2]
if t3 > v goto B2
B2
j := j –1
t4 := t4 – 4
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
i := m –1
j := n
t1 := 4 * n
v := a[t1]
t2 := 4 * i
t4 := 4 * j
B1
i := i + 1
t2 := t2 + 4
t3 := a[t2]
if t3 > v goto B2
B2
j := j –1
t4 := t4 – 4
t5 := a[t4]
if t5 > v goto B3
B3
if t2>= t4 goto B6
B4

《编译原理与技术》
独立于机器的优化

At the end, if you fail, at least you did something interesting, rather than doing something boring and also failing. Or doing something boring and then forgetting how to do something interesting.

—— Barbara Liskov (Turing Award 2008)