北京大学计算机学院/信息科学技术学院

2024-2025年秋季学期　　《编译原理》

第九章 代码优化（续）
——机器无关优化

Code Optimization
Machine-Independent Optimization

Review：代码优化的范围

intra-procedural optimization

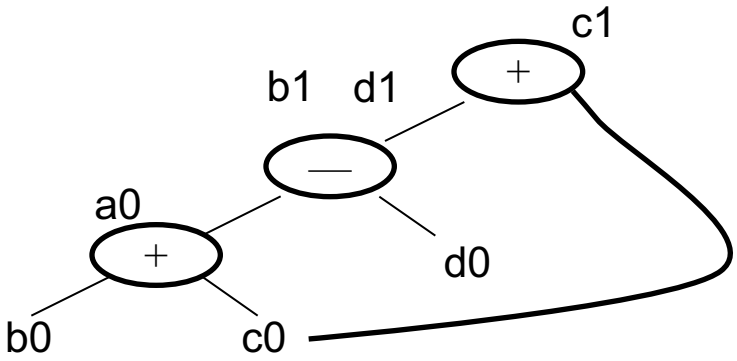
- 局部优化（local optimization）
  - 只对基本块内的语句进行分析，在基本块内实施的优化。
- 区域性优化（regional optimization）
  - 对若干个基本块构成的区域进行分析，比如对循环的优化
- 全局优化（global optimization）
  - 对整个过程所有基本块的信息及它们之间的关系进行分析，在此基础上在整个过程范围内实施优化。
- 过程间优化（inter-procedural optimization）
  - 对整个程序所有过程及其调用信息进行分析，对整个程序进行整体优化。
- 窥孔优化（peephole optimization）？

构建有向无环图（DAG）

- 基于基本块代码序列进行一个前向的扫描，如语句 x = y *op* z;
  - 查找已存在的y, z节点，若没有则创建一个新的y, z节点。
  - 查找已存在的叶子节点为y和z且操作为op的节点，若没有则为操作op创建一个新节点，其叶子结点为y和z (可能需要哈希处理)
  - 将x添加到新节点的标签列表中
  - 从之前出现标签x的节点列表中删除标签x（方便后续的引用）
- 对于语句x=y；将x添加到当前包含y的节点的标签列表中。

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

#	TYPE	Left	Right	Value
0	NAME			B0
1	NAME			C0
2	ADD	B0	C0	A0
3	NAME			D0
4	SUB	A0	D0	B1, D1
5	ADD	B1	C0	C1



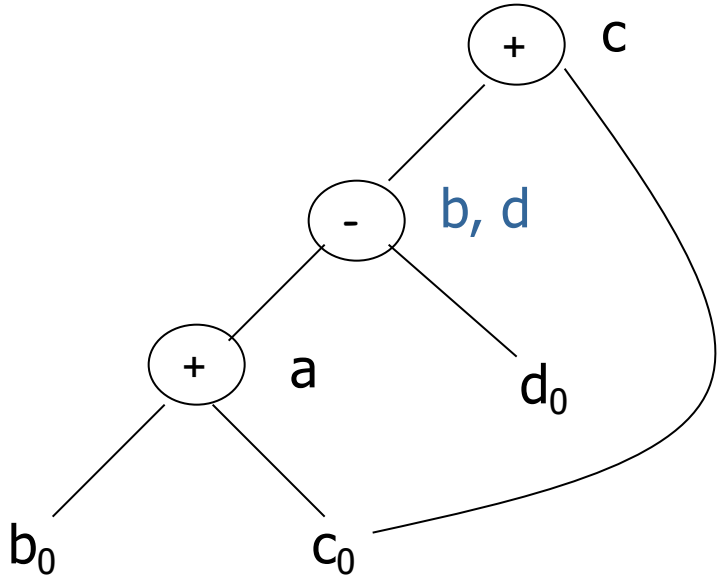
查找公共子表达式

- 假设变量b在代码片段的出口位置非活跃

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

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

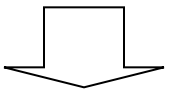
```
a = b + c
d = a - d
b = d
c = d + c
```



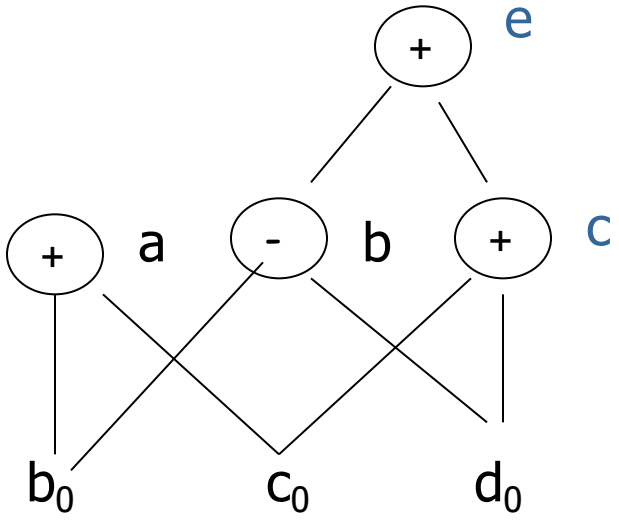
基本块内的死代码消除

- 删除掉DAG图中没有附加任何活跃变量的根节点
- 重复应用此转换规则，将从DAG中删除与死代码对应的所有节点。

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



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



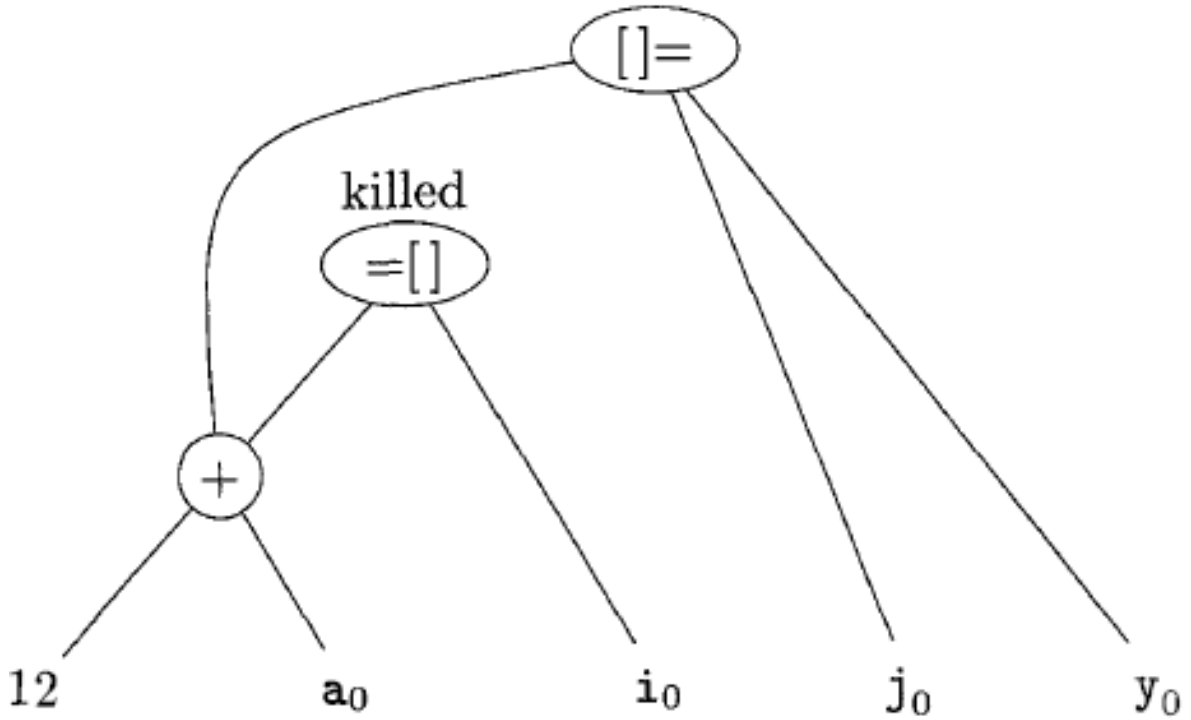
在出口位置

a，b　活跃

c，e　非活跃

数组引用的影响

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



此处a是一个数组。b指向是数组a中的一个位置偏移。

变量x被 b[j]=y 可能重新定值（“杀死”，kill）。

指针和函数调用

- **对下列赋值语句的问题分析：**

  x=*p *q=y

  - 若不知道p或q所指向的地址是什么。必须保守认为x=*p是对每个变量的引用，而*q=y是对每个变量的可能赋值。
  - 使用全局的指针分析，可以对变量集合范围进行限定。

- **过程调用的行为和指针参与的赋值很相似：**
  - 假设一个过程使用并更改它可以访问的任何数据。
  - 如果变量x在过程P的作用域中，则对P的调用既构成了对x所关联的DAG节点的使用，也会kill掉所有这些DAG节点。

窥孔优化示例（1）

- 冗余Load/Store的消除

```
r2 = r1 + 5                 r2 = r1 + 5
i  = r2            =>       i  = r2
r3 = i                      r4 = r2 x 3
r4 = r3 x 3
```

- 无用指令的消除（dead code elimination）

```
r1 = r0 + 0        =>
r2 = r0 x 1
```

- 强度减弱（strength reduction）

```
r1 = r2 x 2                 r1 = r2 + r2 / r1 = r2 << 1
r1 = r2 / 2        =>       r1 = r2 >> 1
r1 = r2 x 0                 r1 = 0
```

窥孔优化示例（2）

- 复写传播（copy propagation）

```
r2 = r1                 r2 = r1
r3 = r1 + r2 =>         r3 = r1 + r1 =>     r3 = r1 + r1
r2 = 5                  r2 = 5              r2 = r5
```

- 常数折叠（constant folding）

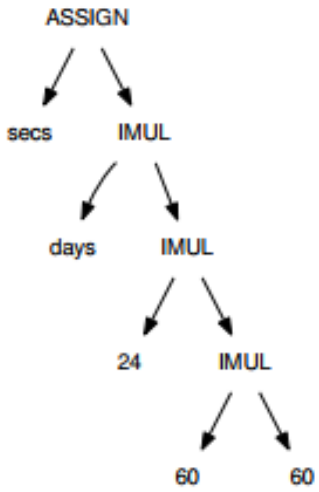
```
r2 = 2 x 3              =>      r2 = 6
```

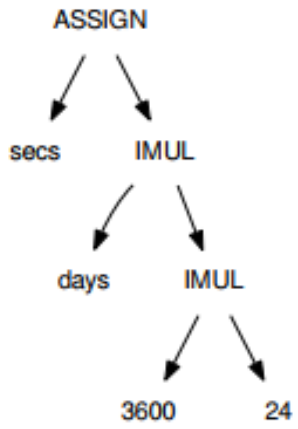
- 常数传播（constant propagation）

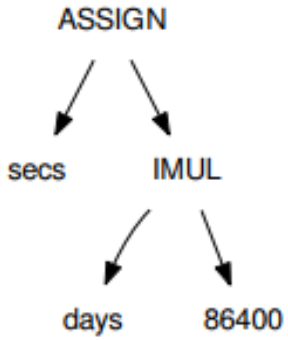
```
r2 = 4                  r2 = 4
r3 = r1 + r2  =>        r3 = r1 + 4  =>     r3 = r1 + 4
r2 = ...                r2 = ...            r2 = ...

r1 = 3                  r1 = 3              r1 = 3
r2 = 2 * r2   =>        r2 = 2 * 3   =>     r3 = 6
```

实现示例：常数折叠

```
if node n is a leaf:
        return;
else:
        n.left = ConstantFold(n.left);
        n.right = ConstantFold(n.right);
        if n.left and n.right are constants:
                n.value = n.operator(n.left,n.right);
                n.kind = constant;
                delete n.left and n.right;
```





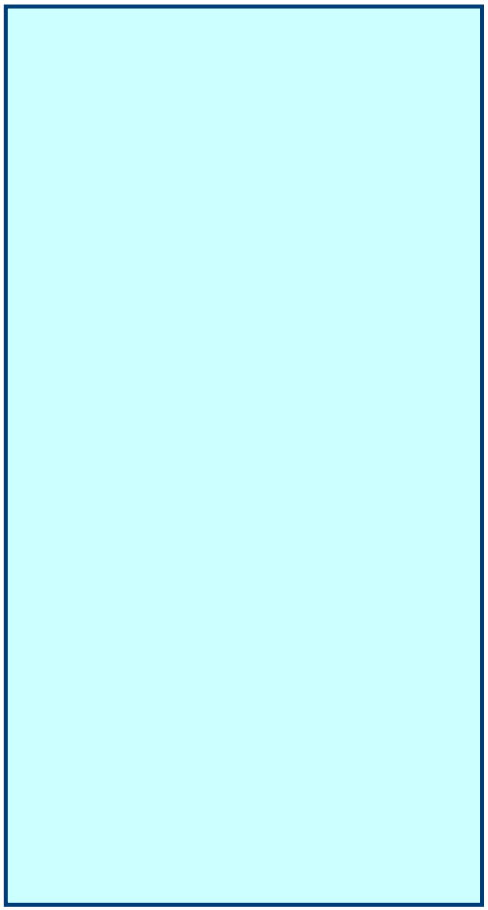


实现示例：滑动窗口（宽度为3）

IR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow MEM(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow MEM(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$MEM(r_{20}) \leftarrow r_{18}$

~~$r_{10} \leftarrow 2$~~
$r_{13} \leftarrow MEM(r_0 + @y)$
$r_{14} \leftarrow 2 \times r_{13}$



实现示例：滑动窗口（宽度为3）

IR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow MEM(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow MEM(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$MEM(r_{20}) \leftarrow r_{18}$

$r_{14} \leftarrow 2 \times r_{13}$
$r_{17} \leftarrow MEM(r_0+@x)$
$r_{18} \leftarrow r_{17} - r_{14}$

$r_{13} \leftarrow MEM(r_0 + @y)$
$r_{14} \leftarrow 2 \times r_{13}$

实现示例：滑动窗口（宽度为3）

IR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow MEM(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow MEM(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$MEM(r_{20}) \leftarrow r_{18}$

$r_{18} \leftarrow r_{17} - r_{14}$
$MEM(r_0 + @w) \leftarrow r_{18}$

$r_{13} \leftarrow MEM(r_0 + @y)$
$r_{14} \leftarrow 2 \times r_{13}$
$r_{17} \leftarrow MEM(r_0 + @x)$
$r_{18} \leftarrow r_{17} - r_{14}$
$MEM(r_0 + @w) \leftarrow r_{18}$

迭代数据流分析框架

Reaching Definitions

```
for all nodes n in N
    OUT[n] = emptyset;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = emptyset;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
    for all nodes s in successors(n)
            Changed = Changed U { s };
```

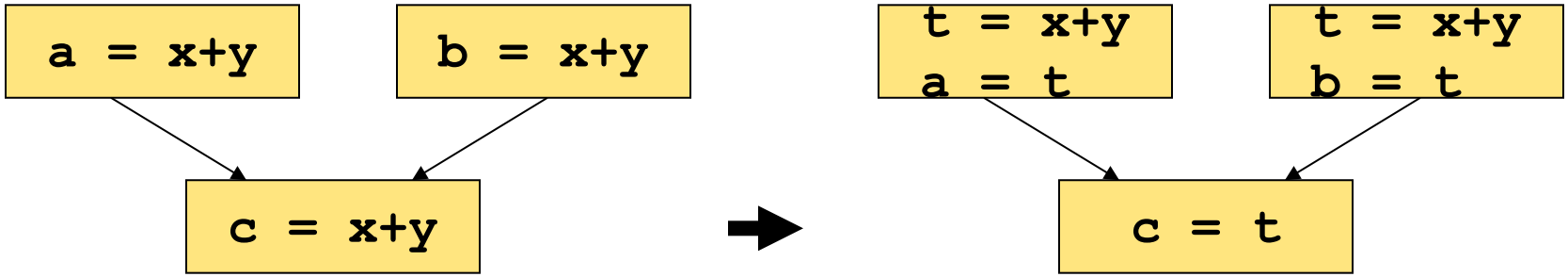
Available Expressions

```
for all nodes n in N
    OUT[n] = E;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = E;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] ∩ OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

Live Variables

```
for all nodes n in N - { Exit }
    IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { Exit };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    OUT[n] = emptyset;
    for all nodes s in successors(n)
        OUT[n] = OUT[n] U IN[p];

    IN[n] = use[n] U (out[n] - def[n]);

    if (IN[n] changed)
    for all nodes p in predecessors(n)
        Changed = Changed U { p };
```

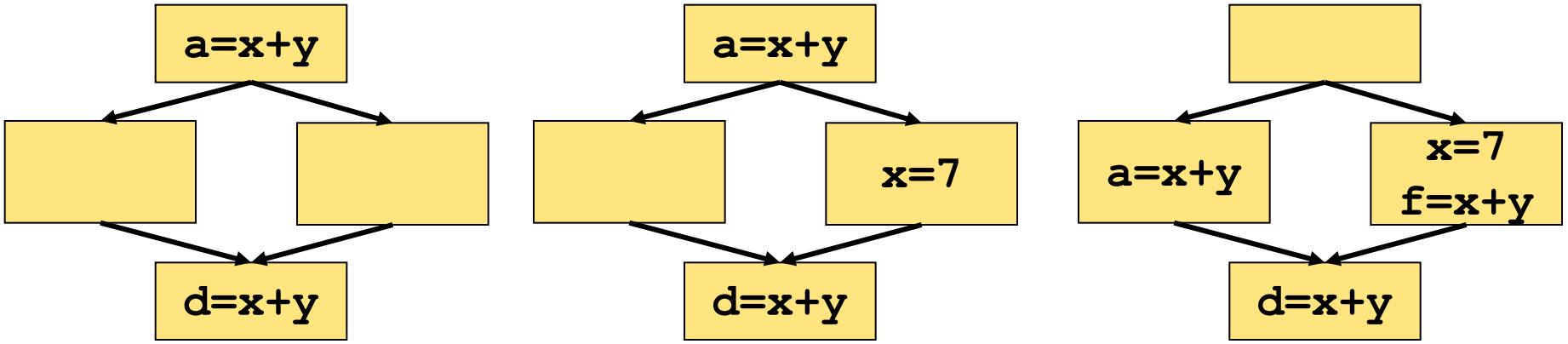
公共子表达式消除



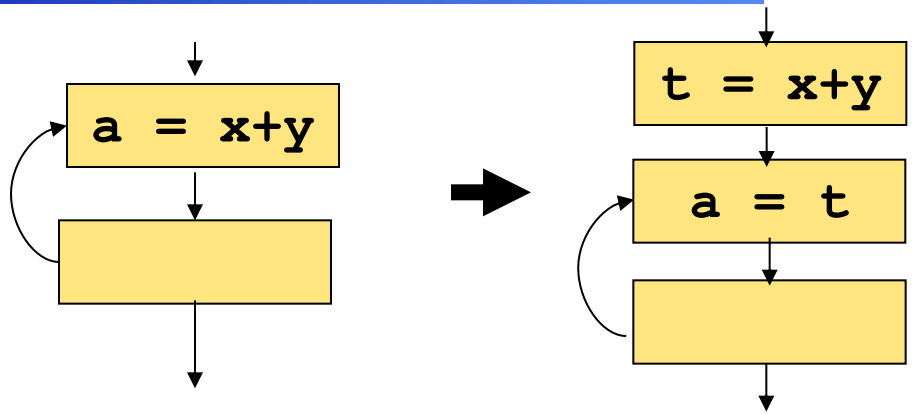
- 一个公共表达式在不同的路径上可以有不同的值。
- 在每一条通往程序点p的路径上：

  要么是计算了表达式x+y；

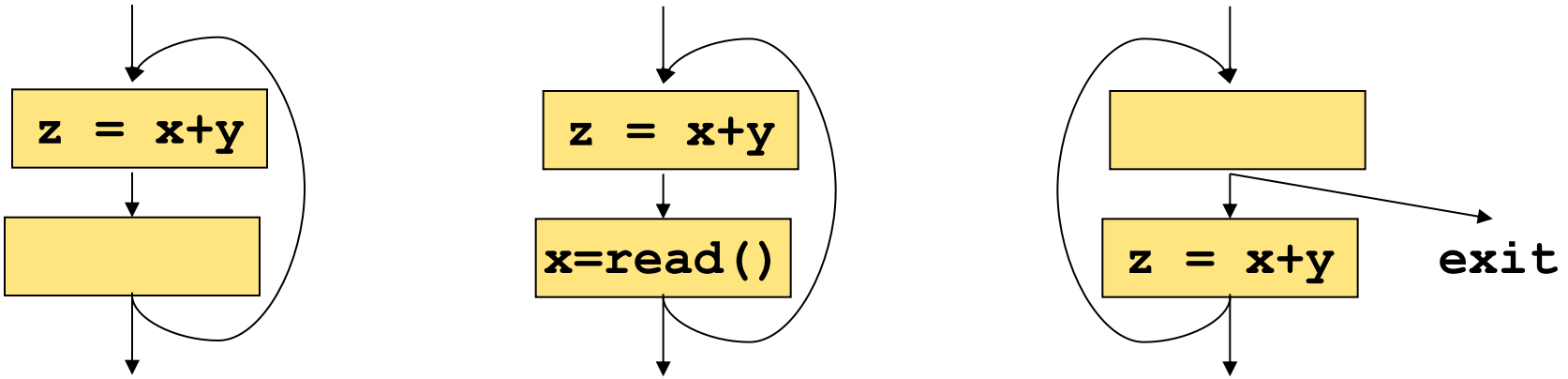
  要么是表达式的分量x或y没有被覆盖（重新定值）。



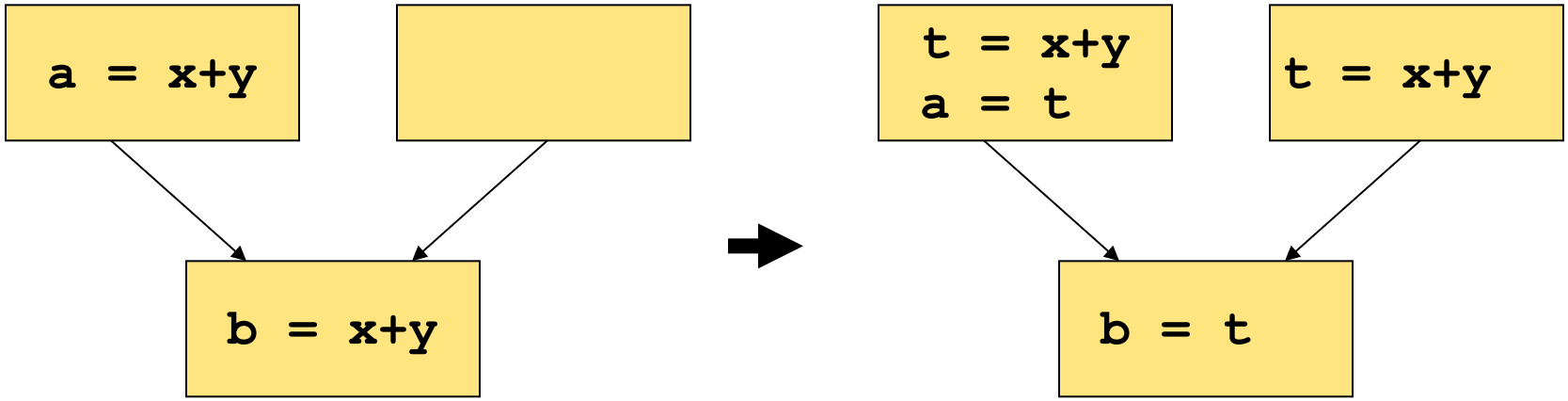
循环不变量外提



给定一个循环中的表达式（x+y），x+y的值在循环内是否会发生变化？代码是否至少执行一次？

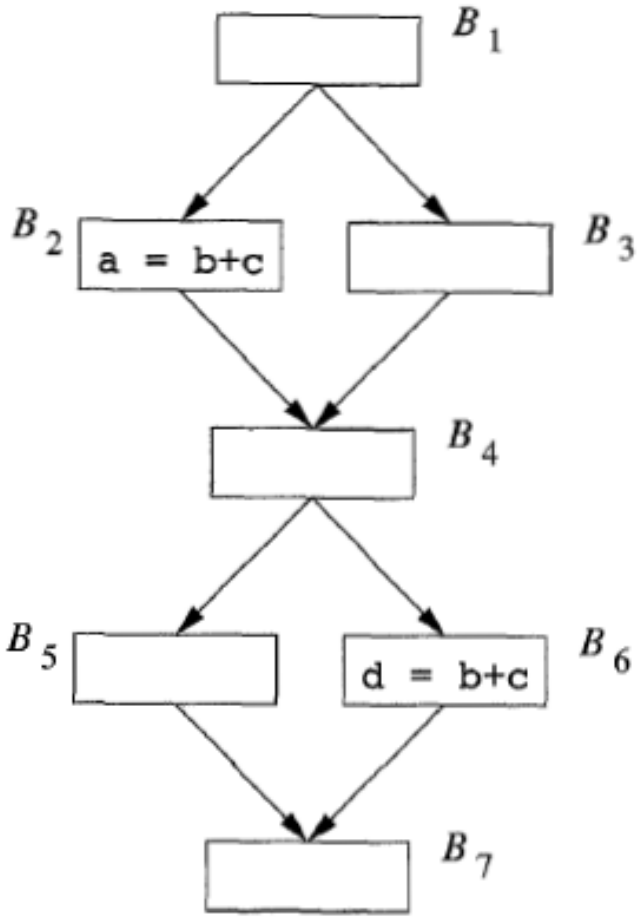


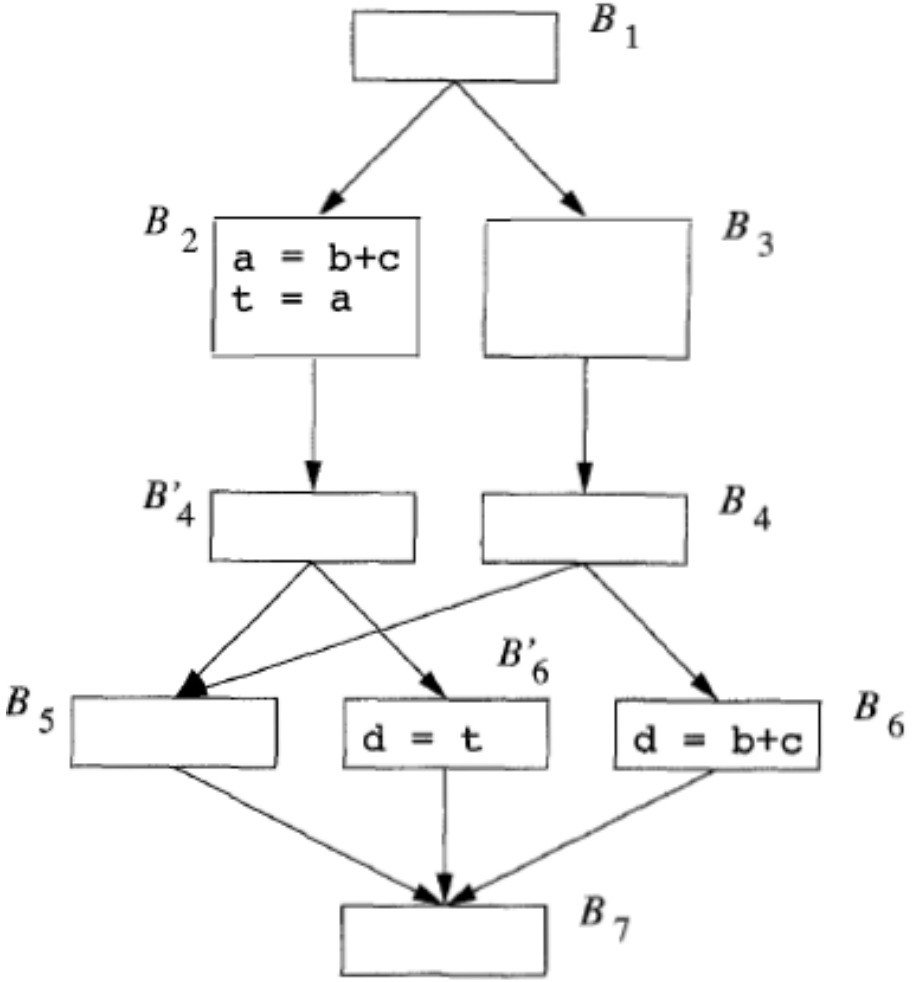
部分冗余



- 将x+y的计算放在合适位置，使得没有哪条路径会重复执行同一个表达式。
- 和循环不变代码移动一样，部分冗余消除需要放置一些新的表达式计算指令

使用代码复制消除部分冗余计算





关于部分冗余消除的进一步讨论

- 代码复制会导致流图节点数的指数级膨胀。
- 新的替代方案：将部分冗余补全为全局冗余，化归为全局冗余然后使用“可用表达式”的数据流分析方法解决。
- 部分冗余消除的延迟代码移动算法（Lazy Code Motion, LCM算法最早由Morel在1979年提出，后由Knoop等在1992年进行改进，使用了多遍数据流分析）
- 循环不变量外提可以是一种部分冗余消除。

数据流分析的其它选择

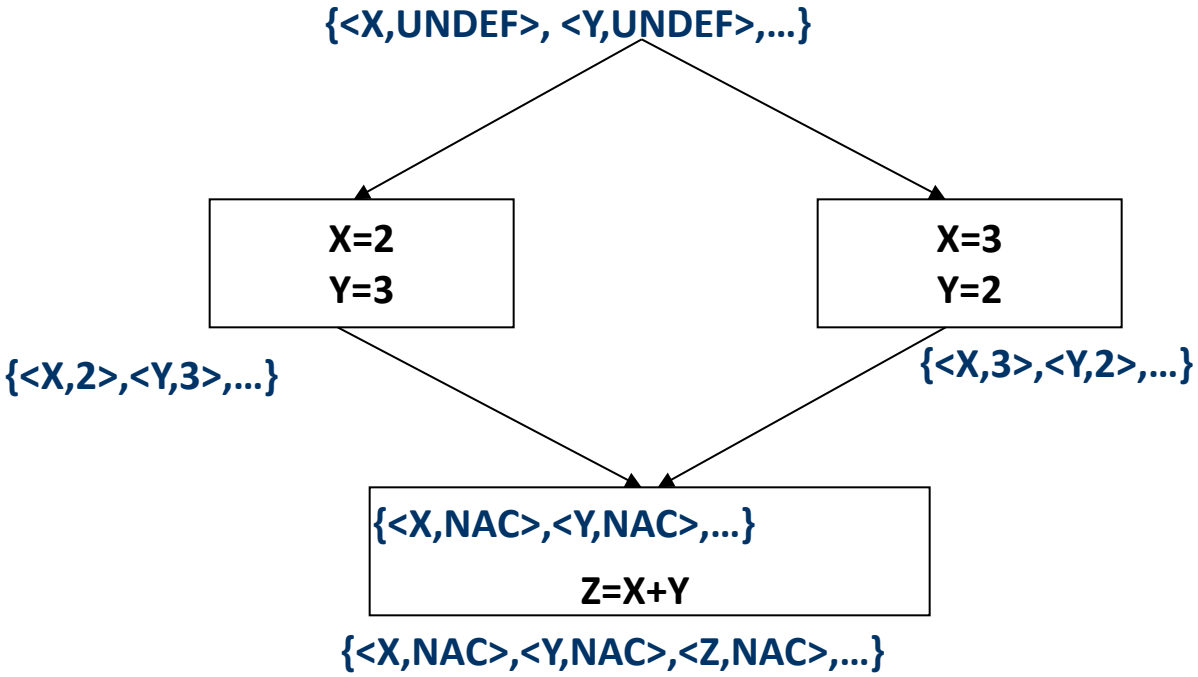
- 迭代数据流分析框架 [Kildall-1973]
- 强连通域分析[Allen-1969]
- 区间分析[Cocke & Allen 1970-1976]
- T1-T2 区间分析[Ullman-1973]
- 基于节点列表数据结构的分析方法[Kennedy-1975]
- 基于路径压缩的分析方法[Graham-Wegman -1976]
- 平衡路径压缩[Tarjan - 1975]
- 图语法方法[Farrow, Kennedy, Zucconi - 1976]
- 高层次语法导向分析方法 [Rosen - 1977]
- Slotwise Analysis [Dhamdhere & Rosen 1992]

数据流分析方法的比较和选择

Method	Speed	Simple?	Structure	Both Way?	Graph Class
Iterative	$O(n^2)$	Simple	No	Yes	All
Interval	$O(n^2)$	Middle	Yes	Yes	Reducible
Balance Tree	O(nlogn)	Complicated	Yes	No	Reducible
Path Comp.	O(nlogn)	Middle	Semi	Yes	Reducible
Node List	O(nlogn)	Middle	No	Yes	Reducible
Balance Path	$O(n\alpha(n,n))$	Complicated	No	?	Reducible
Grammar	n	Middle	Yes	Yes	L(Grammar)
High Level	n	Simple	Yes	Yes	Parse Trees

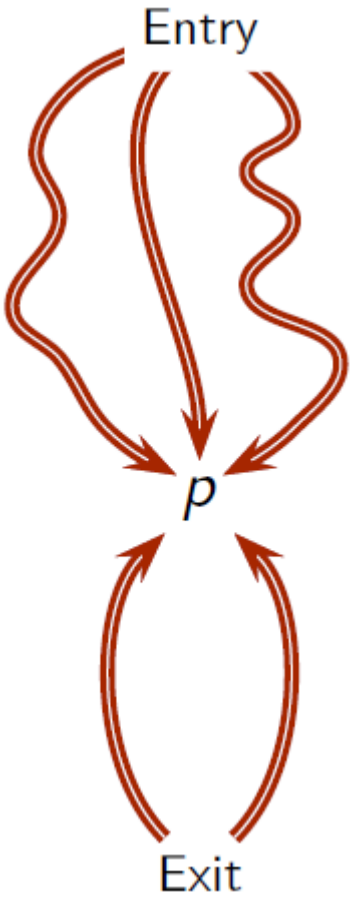
数据流分析方法的局限性

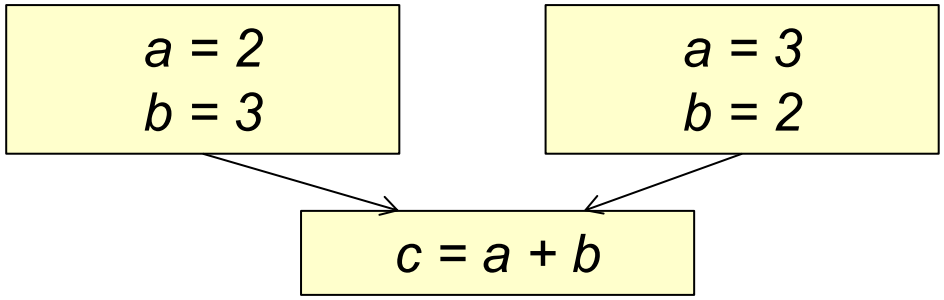
- 数据流分析方法的能力取决于传递方程，传递方程的性质决定了数据流解的精确性。



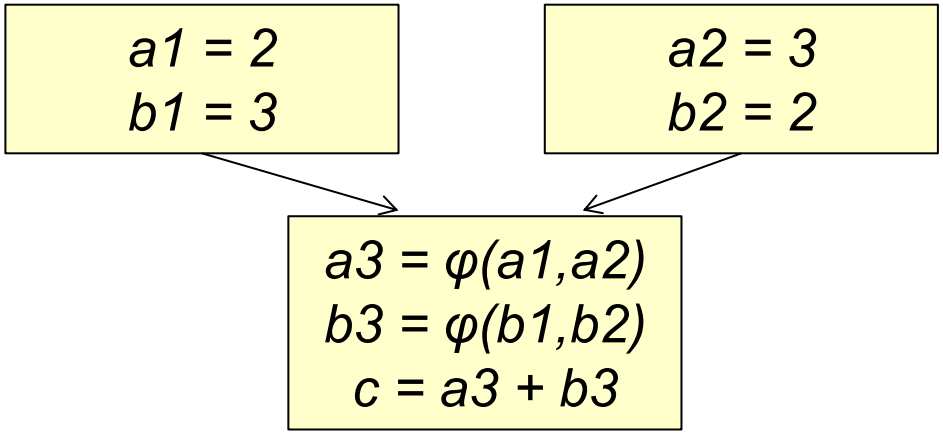
- 常数传播不可分配，这使其优化能力有限

迭代数据流分析的局限性



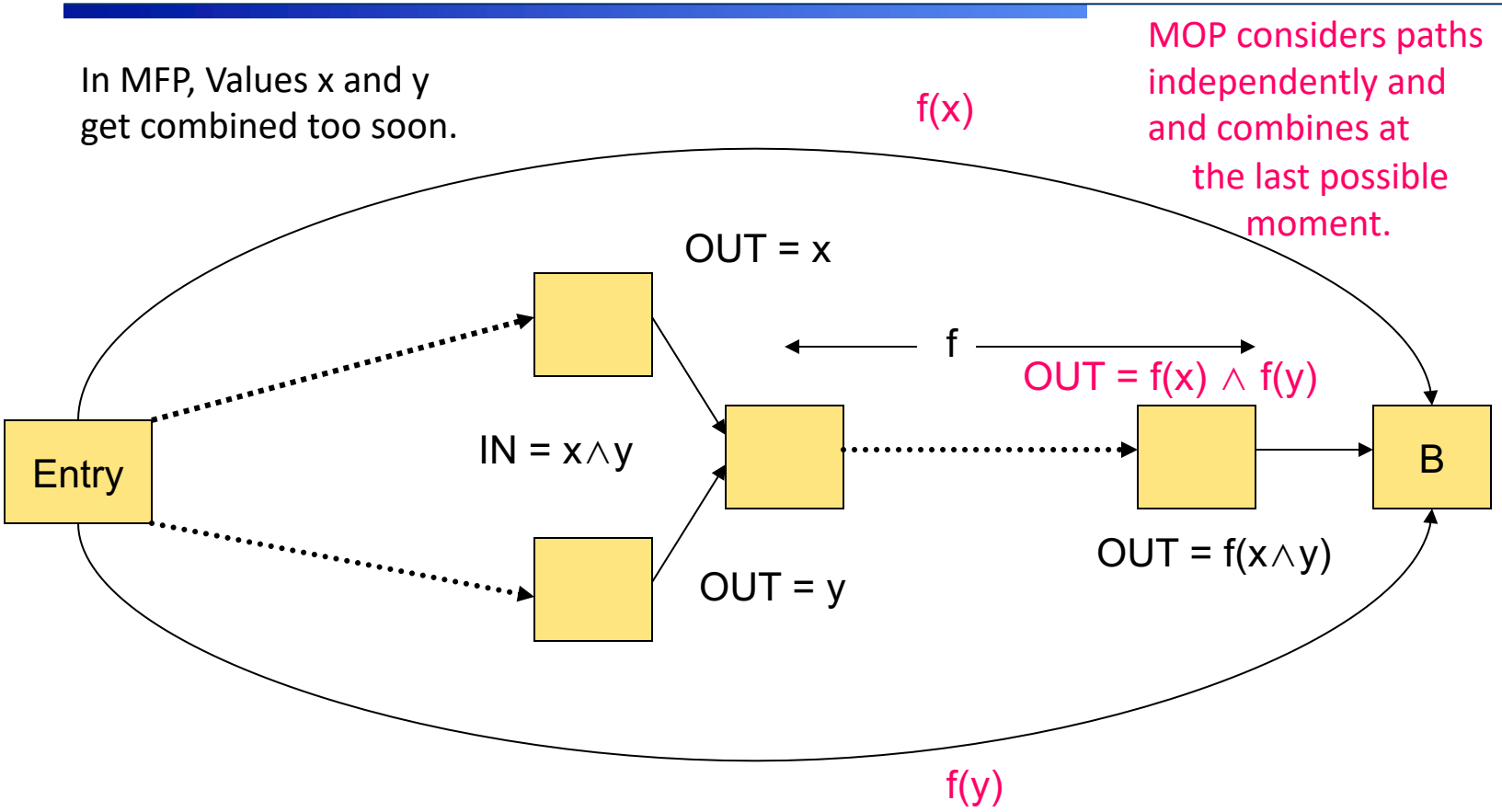


Variable c == constant 5?



$$c = \varphi(a1,a2) + \varphi(b1,b2) = \varphi(a1+b1,a2+b2)$$
$$= \varphi(2+3,3+2) = \varphi(5,5) = 5$$

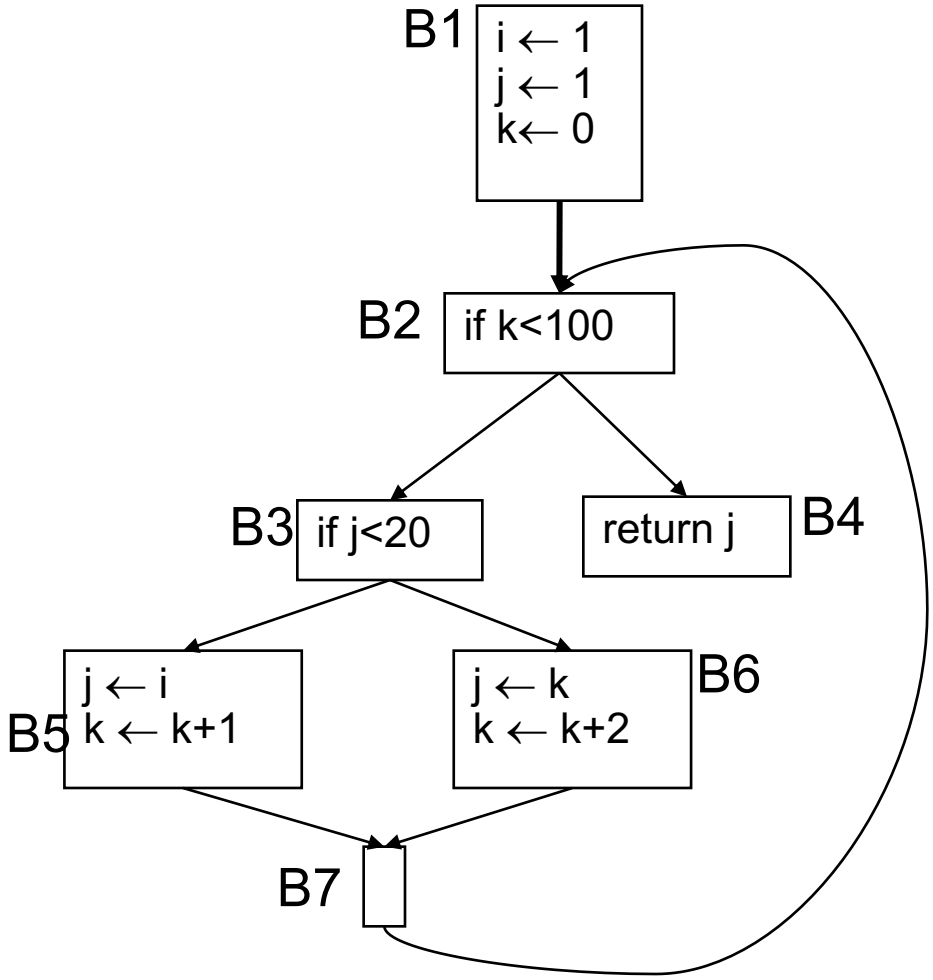
迭代数据流分析的局限性



Since $f(x \wedge y) \leq f(x) \wedge f(y)$, it is as if we added nonexistent paths, but we're safe.

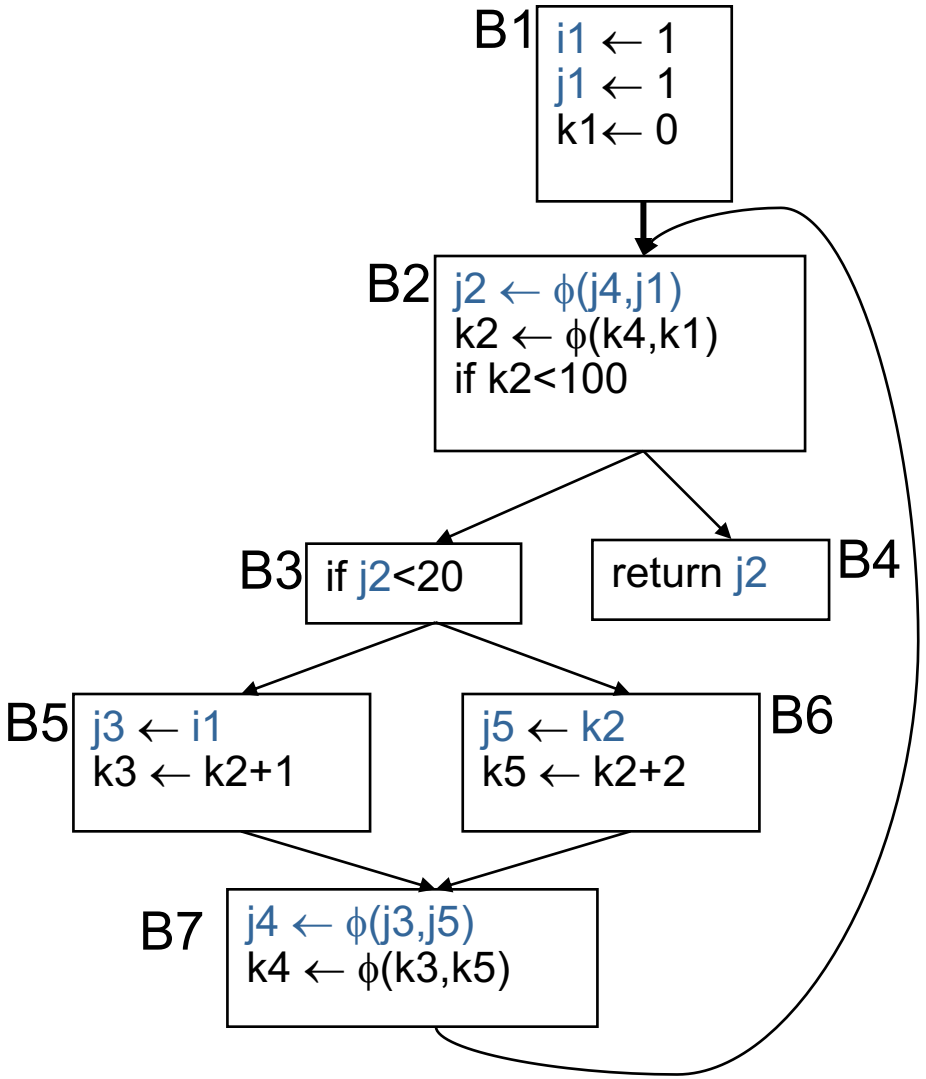
一个基于SSA开展优化的例子

```
{
    i=1;
    j=1;
    k=0;
    while(k<100)
    {
        if(j<20)
        {
            j=i;
            k=k+1;
        }
        else
        {
            j=k;
            k=k+2;
        }
    }
    return j;
}
```

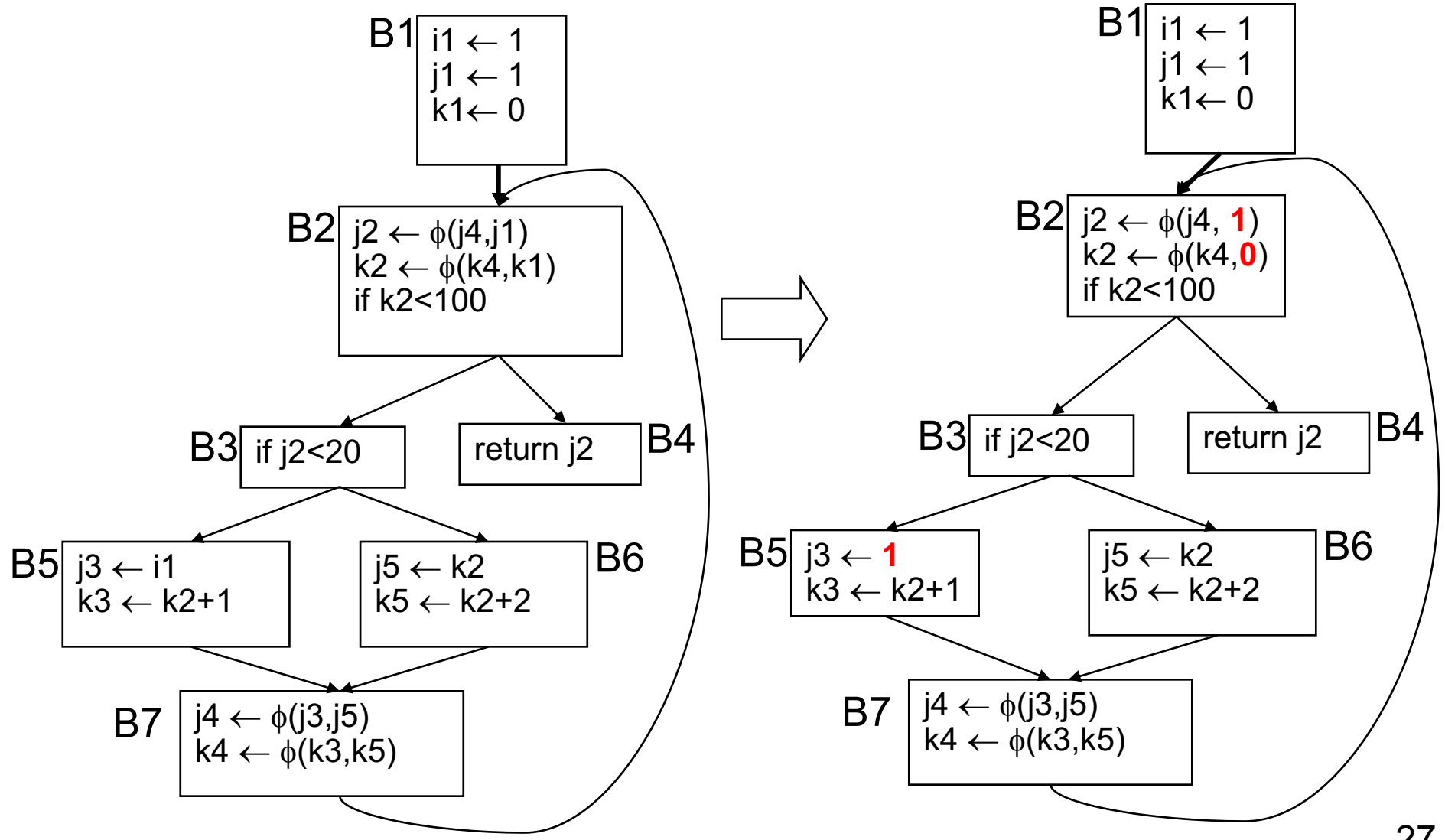


一个基于SSA开展优化的例子

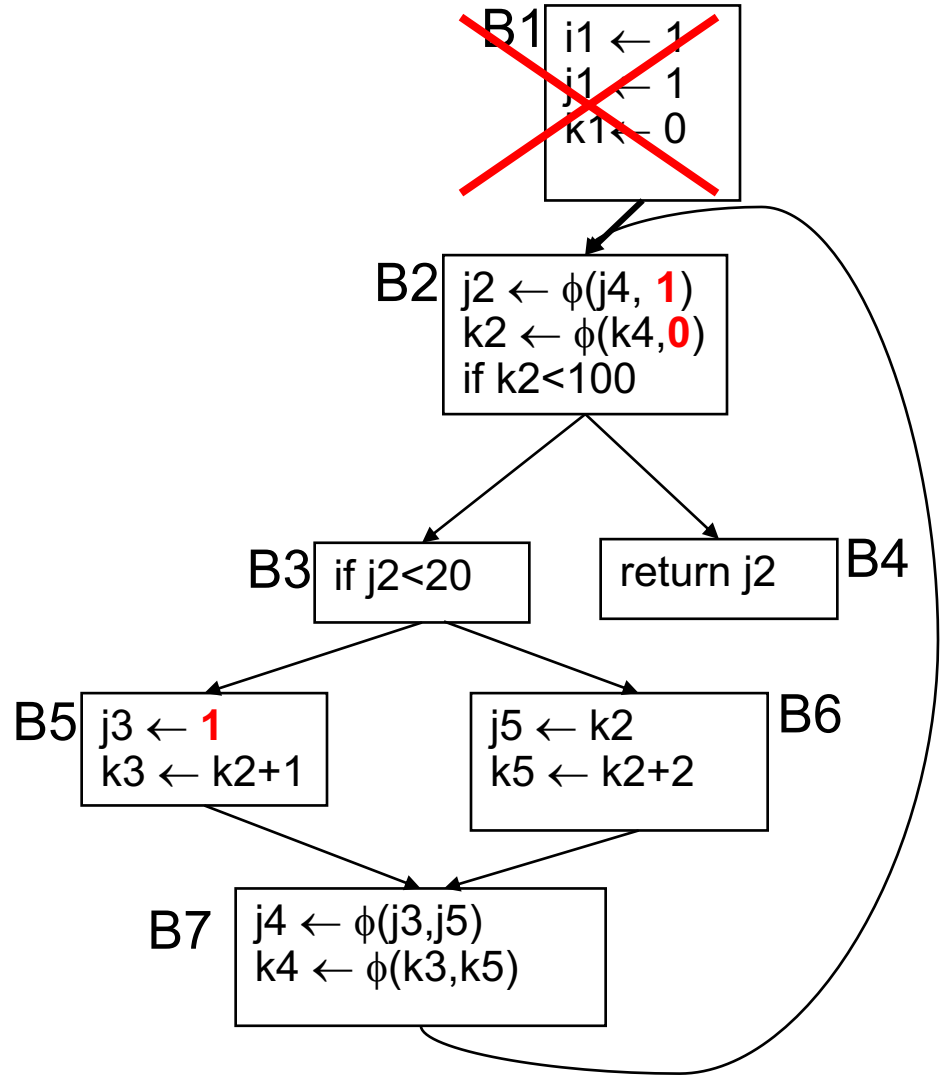
```
{
    i=1;
    j=1;
    k=0;
    while(k<100)
    {
        if(j<20)
        {
            j=i;
            k=k+1;
        }
        else
        {
            j=k;
            k=k+2;
        }
    }
    return j;
}
```

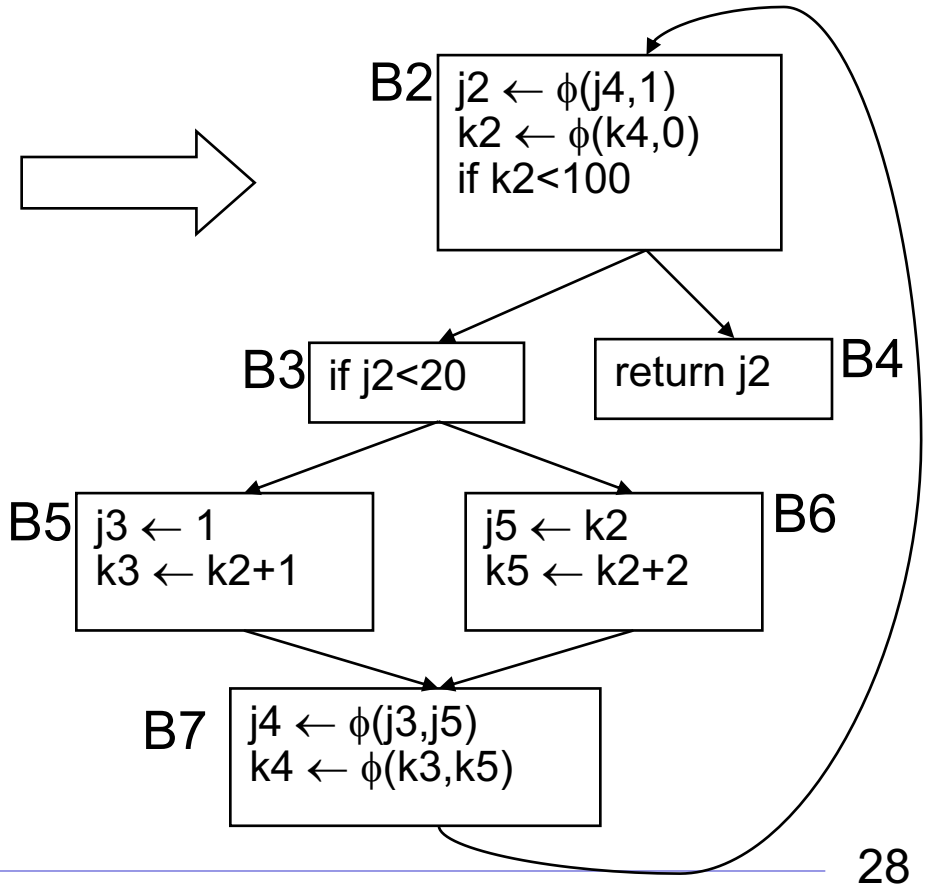


常数传播

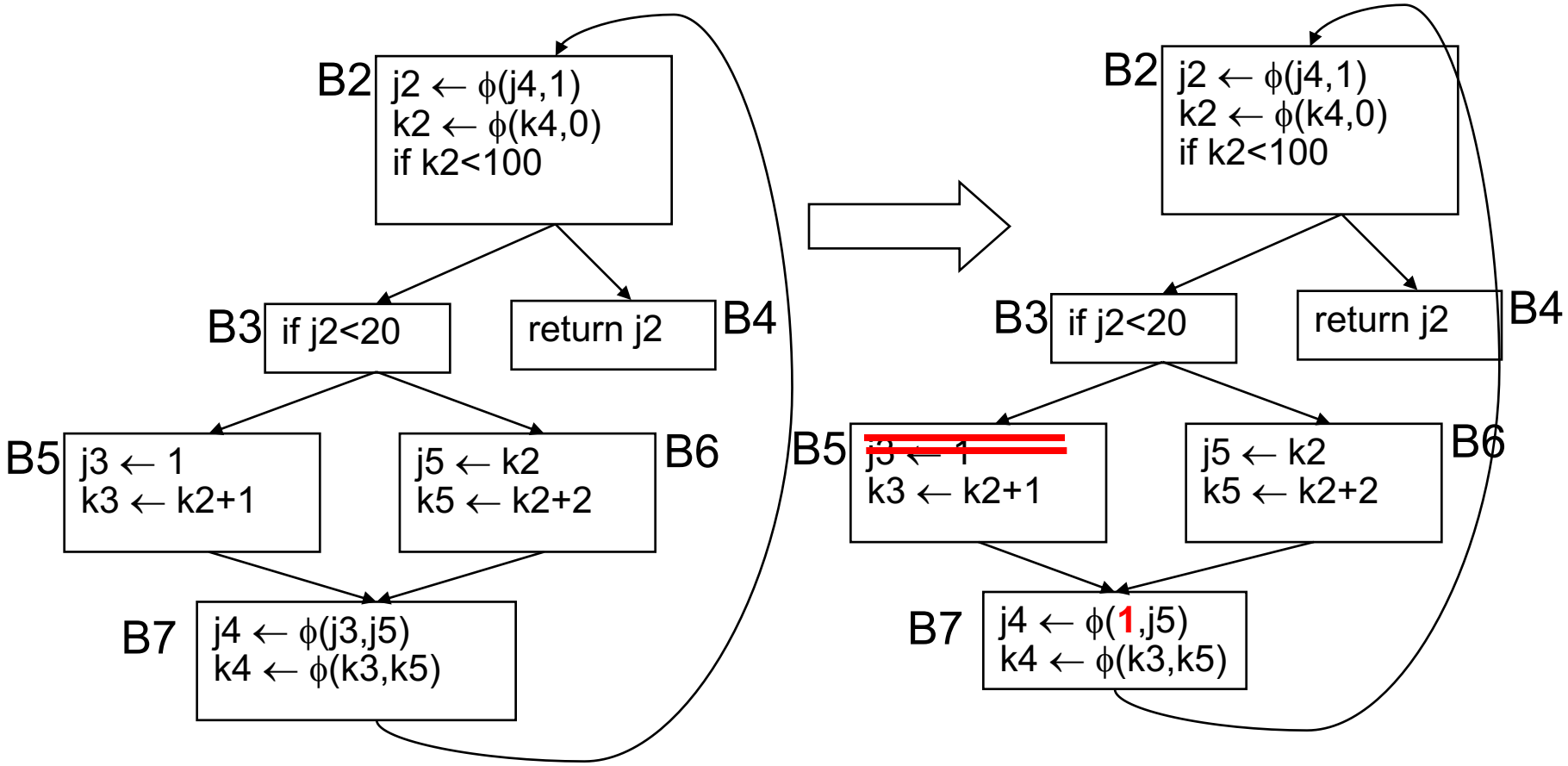

B1
i1 ← 1
j1 ← 1
k1← 0
B2
j2 ← ϕ(j4,j1)
k2 ← ϕ(k4,k1)
if k2<100
B3
if j2<20
B4
return j2
B5
j3 ← i1
k3 ← k2+1
B6
j5 ← k2
k5 ← k2+2
B7
j4 ← ϕ(j3,j5)
k4 ← ϕ(k3,k5)
B1
i1 ← 1
j1 ← 1
k1← 0
B2
j2 ← ϕ(j4, 1)
k2 ← ϕ(k4,0)
if k2<100
B3
if j2<20
B4
return j2
B5
j3 ← 1
k3 ← k2+1
B6
j5 ← k2
k5 ← k2+2
B7
j4 ← ϕ(j3,j5)
k4 ← ϕ(k3,k5)

死代码消除

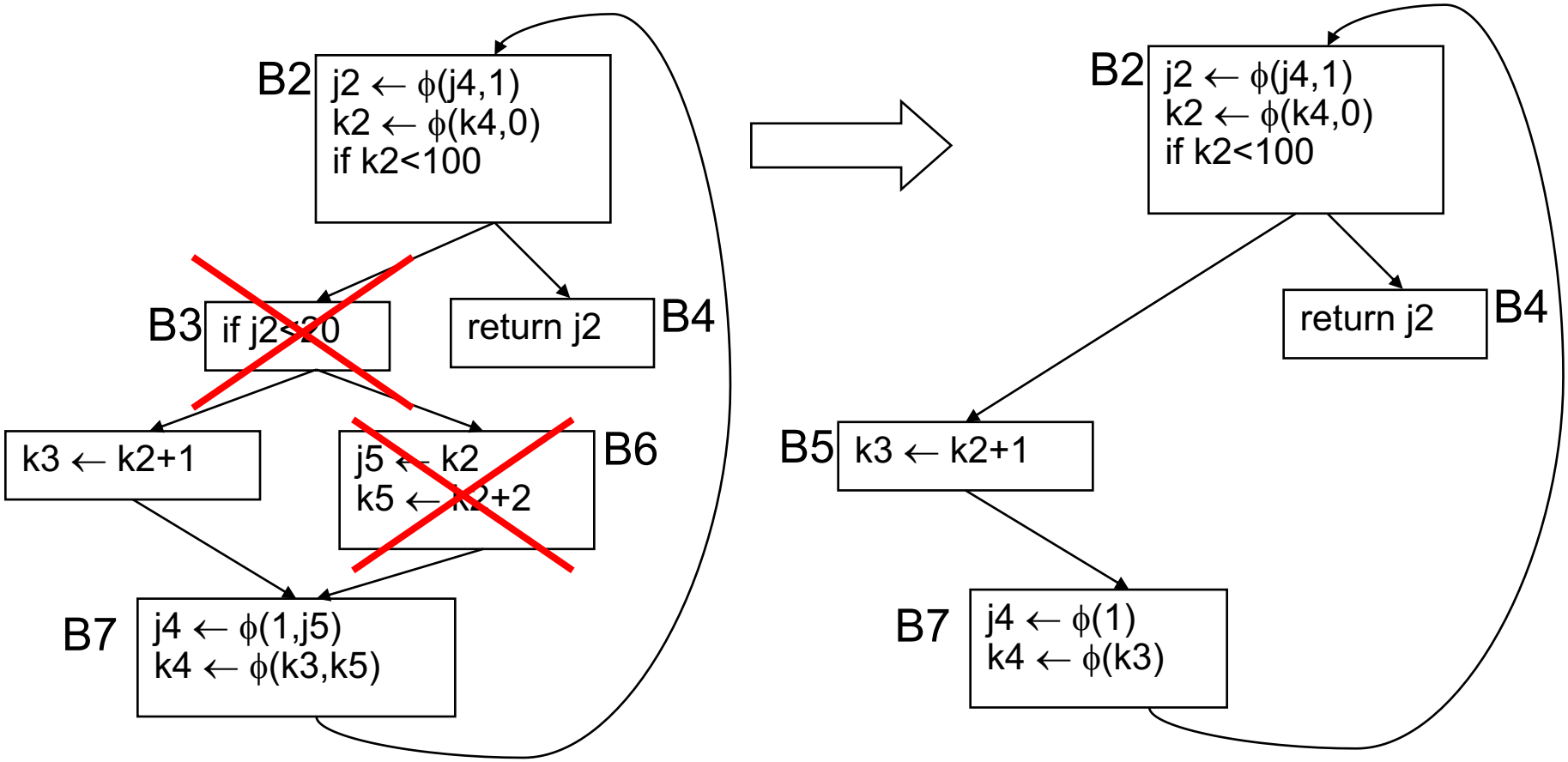

B1
i1 ← 1
j1 ← 1
k1 ← 0
B2
j2 ← ϕ(j4, 1)
k2 ← ϕ(k4,0)
if k2<100
B3
if j2<20
return j2
B4
B5
j3 ← 1
k3 ← k2+1
j5 ← k2
k5 ← k2+2
B6
B7
j4 ← ϕ(j3,j5)
k4 ← ϕ(k3,k5)


B2
j2 ← ϕ(j4,1)
k2 ← ϕ(k4,0)
if k2<100
B3
if j2<20
return j2
B4
B5
j3 ← 1
k3 ← k2+1
j5 ← k2
k5 ← k2+2
B6
B7
j4 ← ϕ(j3,j5)
k4 ← ϕ(k3,k5)

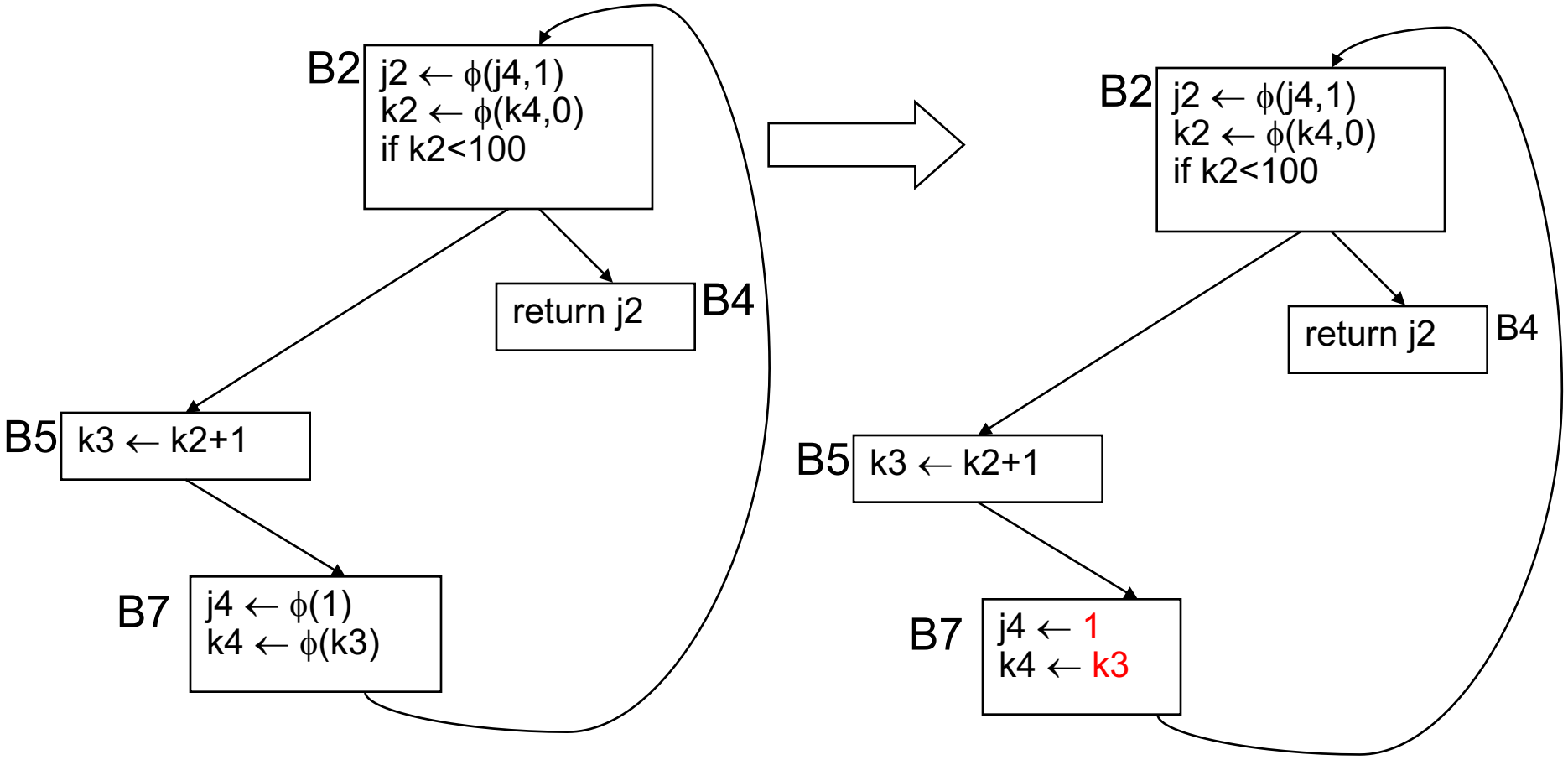
常数传播+死代码消除


B2
j2 ← ϕ(j4,1)
k2 ← ϕ(k4,0)
if k2<100
B3
if j2<20
return j2
B4
B5
j3 ← 1
k3 ← k2+1
j5 ← k2
k5 ← k2+2
B6
B7
j4 ← ϕ(j3,j5)
k4 ← ϕ(k3,k5)
B2
j2 ← ϕ(j4,1)
k2 ← ϕ(k4,0)
if k2<100
B3
if j2<20
return j2
B4
B5
j3 ← 1
k3 ← k2+1
j5 ← k2
k5 ← k2+2
B6
B7
j4 ← ϕ(1,j5)
k4 ← ϕ(k3,k5)

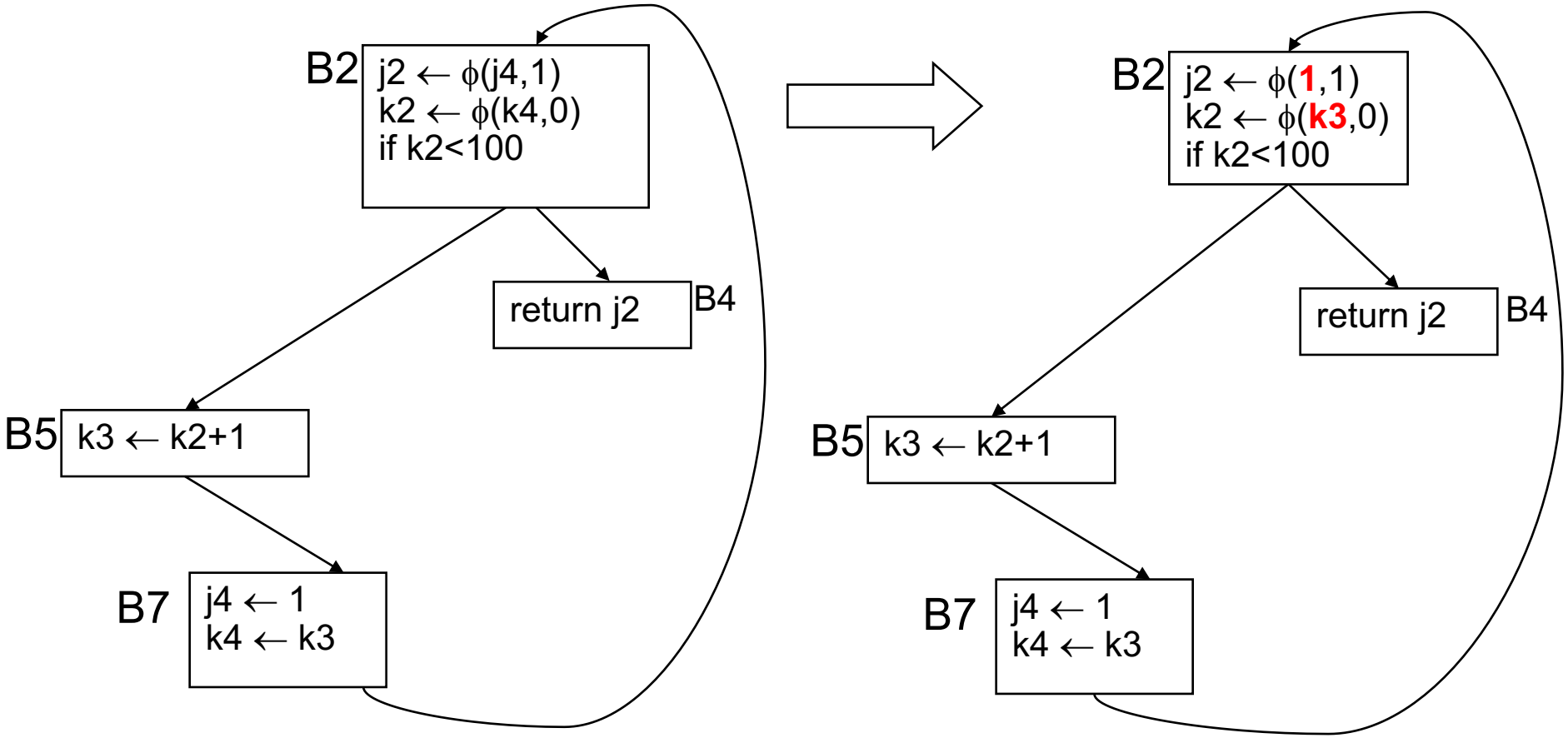
不可达代码消除


B2
j2 ← ϕ(j4,1)
k2 ← ϕ(k4,0)
if k2<100
B3
if j2<20
return j2
B4
k3 ← k2+1
B6
j5 ← k2
k5 ← k2+2
B7
j4 ← ϕ(1,j5)
k4 ← ϕ(k3,k5)
B2
j2 ← ϕ(j4,1)
k2 ← ϕ(k4,0)
if k2<100
return j2
B4
B5
k3 ← k2+1
B7
j4 ← ϕ(1)
k4 ← ϕ(k3)

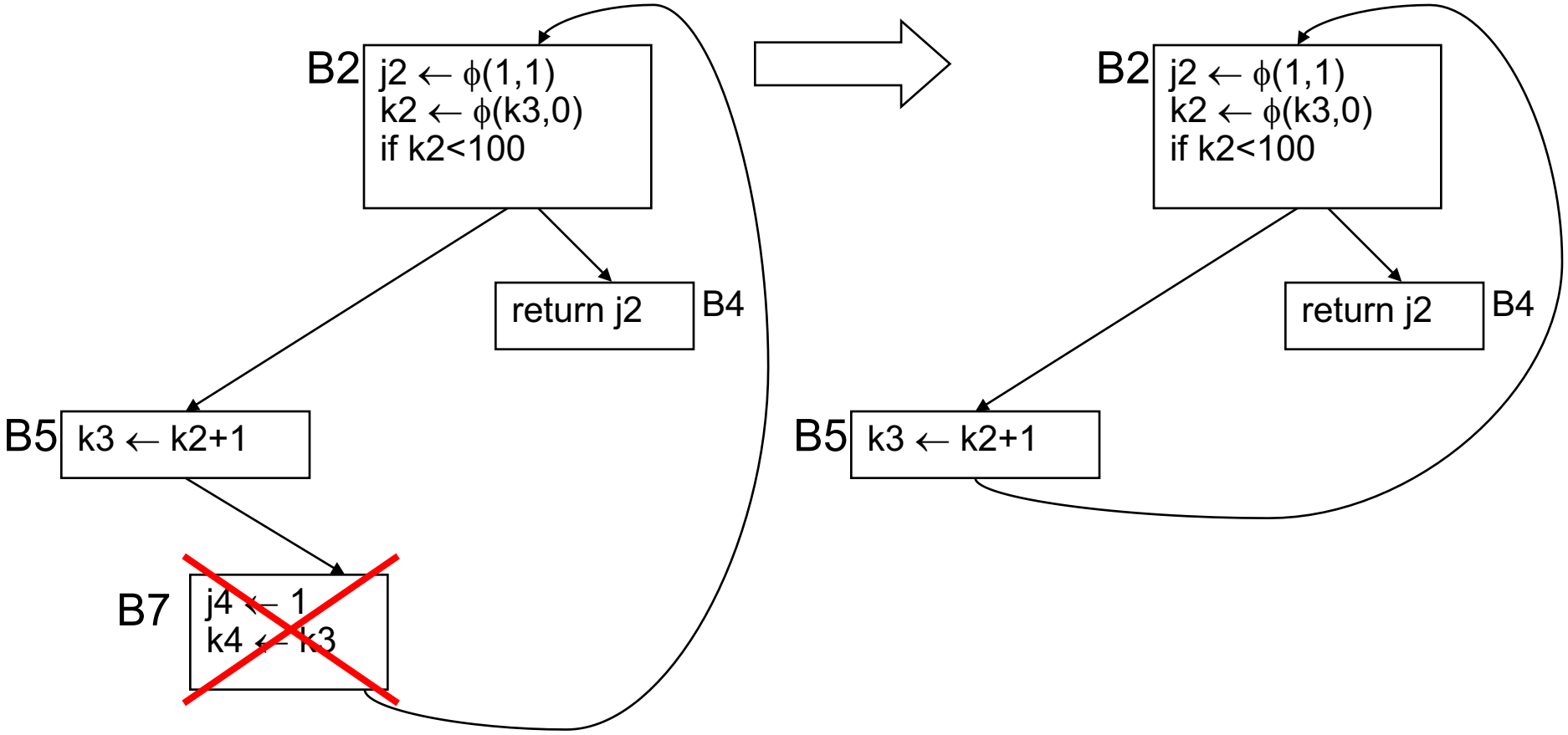
Φ函数化简


B2
j2 ← ϕ(j4,1)
k2 ← ϕ(k4,0)
if k2<100
return j2
B4
B5
k3 ← k2+1
B7
j4 ← ϕ(1)
k4 ← ϕ(k3)
B2
j2 ← ϕ(j4,1)
k2 ← ϕ(k4,0)
if k2<100
return j2
B4
B5
k3 ← k2+1
B7
j4 ← 1
k4 ← k3

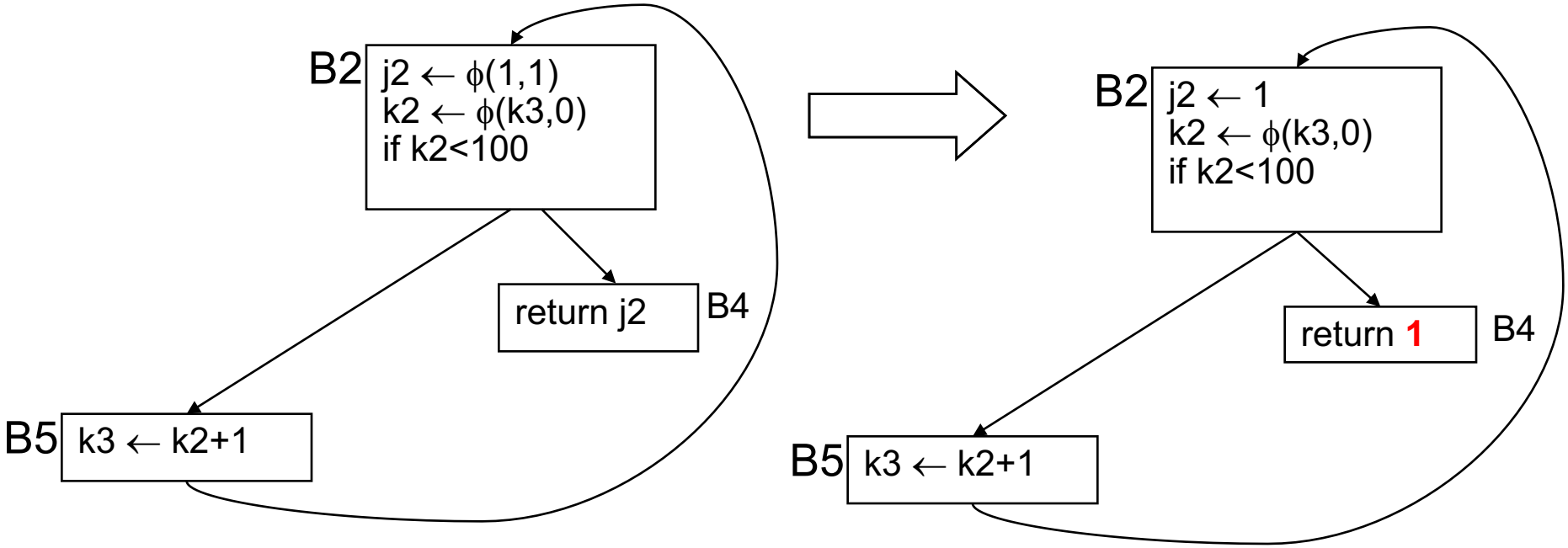
常数传播+复写传播


B2
j2 ← ϕ(j4,1)
k2 ← ϕ(k4,0)
if k2<100
return j2
B4
B5
k3 ← k2+1
B7
j4 ← 1
k4 ← k3
B2
j2 ← ϕ(1,1)
k2 ← ϕ(k3,0)
if k2<100
return j2
B4
B5
k3 ← k2+1
B7
j4 ← 1
k4 ← k3

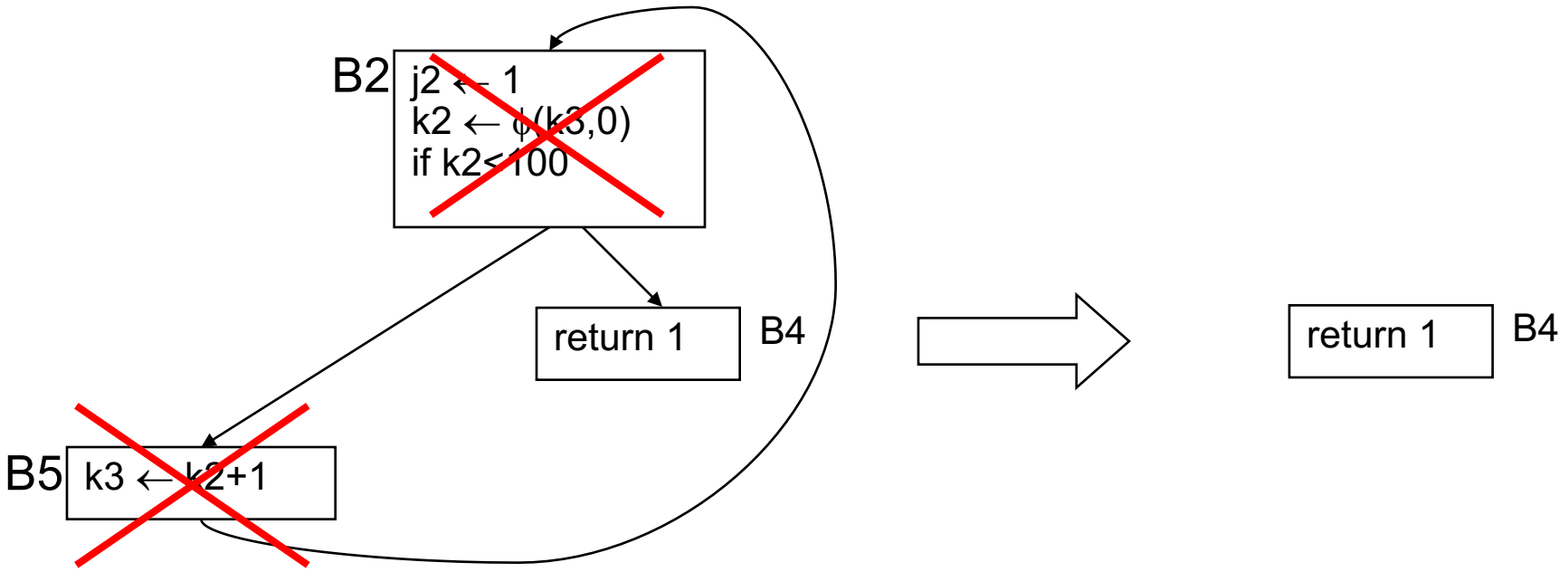
进一步死代码消除


B2
j2 ← ϕ(1,1)
k2 ← ϕ(k3,0)
if k2<100
return j2
B4
B5
k3 ← k2+1
B7
j4 ← 1
k4 ← k3
B2
j2 ← ϕ(1,1)
k2 ← ϕ(k3,0)
if k2<100
return j2
B4
B5
k3 ← k2+1

进一步Φ函数化简+常数传播



死代码消除得到最终优化结果



基于SSA形式优化的一些特点

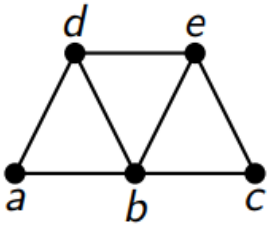
- 数据依赖信息内置于SSA形式，不需要单独的阶段来计算和记录依赖信息。
- 转换后的输出能够保留SSA形式，这使得对依赖信息的更新开销很小，方便编译器根据需要重复应用优化算法。
- 更容易获得高效算法：
  - 流图节点相对稀疏
  - 其复杂性取决于问题规模而非程序规模
  - 数据流信息可沿UD链/DU链线性传播
- 局部优化和全局优化没有明显界限
- 广泛应用于GCC、LLVM、JVM等工具中

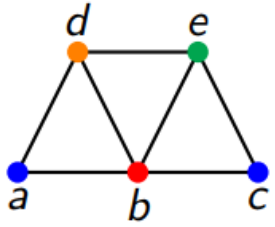
静态单赋值形式与弦图

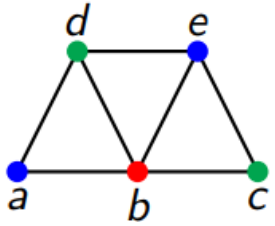
- 弦：连接环中不相邻的两个结点的边。
- 弦图：所有长度大于3的环均有弦的图。
- 一个无向图是弦图当且仅当它有一个完美消除序列算法。
- 如果编译器根据静态单赋值形式而不是变量活动范围建立干扰图，那么结果图是一个弦图。

[Brisk; Bouchez, Darte, Rastello; Hack, 2005]

- 弦图的k染色问题可以在O(|V|+|E|)时间内解决。
- SSA形式简化了寄存器分配的一部分工作，可为其冲突图计算最佳着色方案，可能使用更少的寄存器。







过程间分析和优化

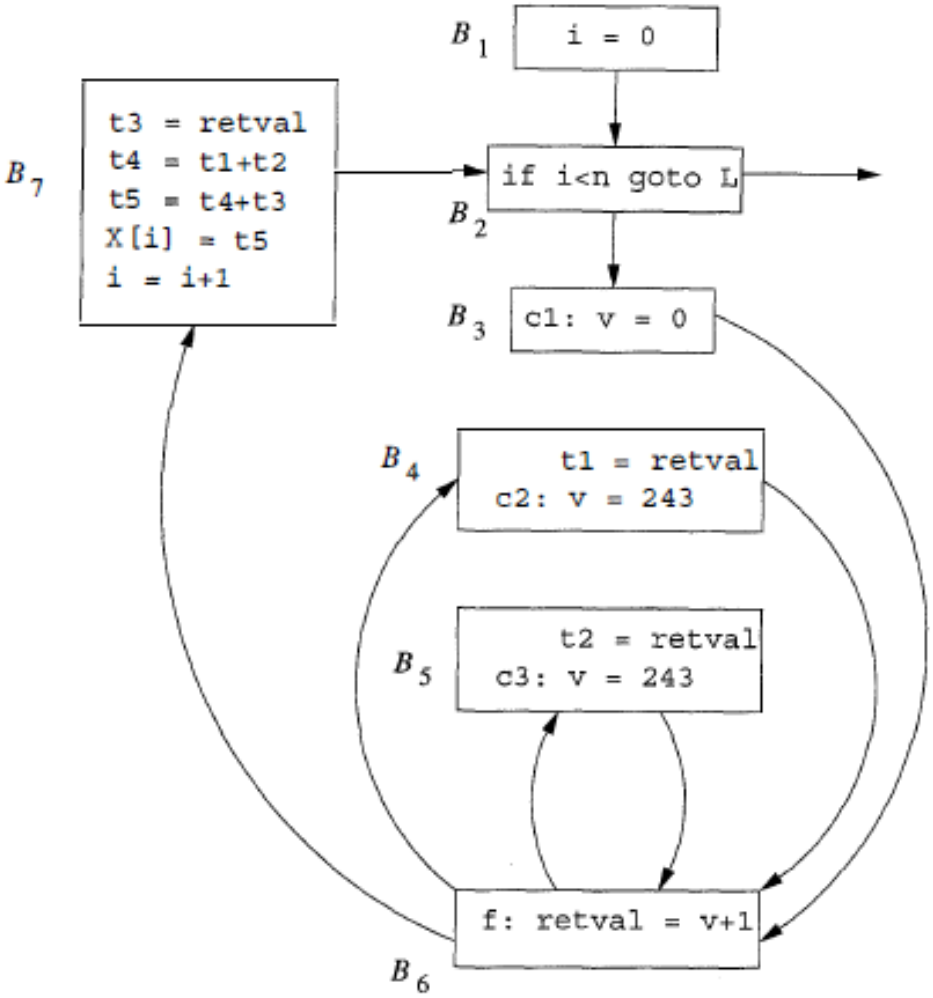
- 过程间分析
  - 跨越多个过程收集程序信息（通常贯穿整个程序）
  - 可以使用此信息对程序的分析和优化进行改进（例如过程内的公共子表达式消除）
  - 常见的有基于克隆的分析和基于摘要的分析。
- 过程间优化
  - 涉及多个过程之间的优化（如内联、克隆、过程间寄存器分配等）
  - 基于过程间分析的优化

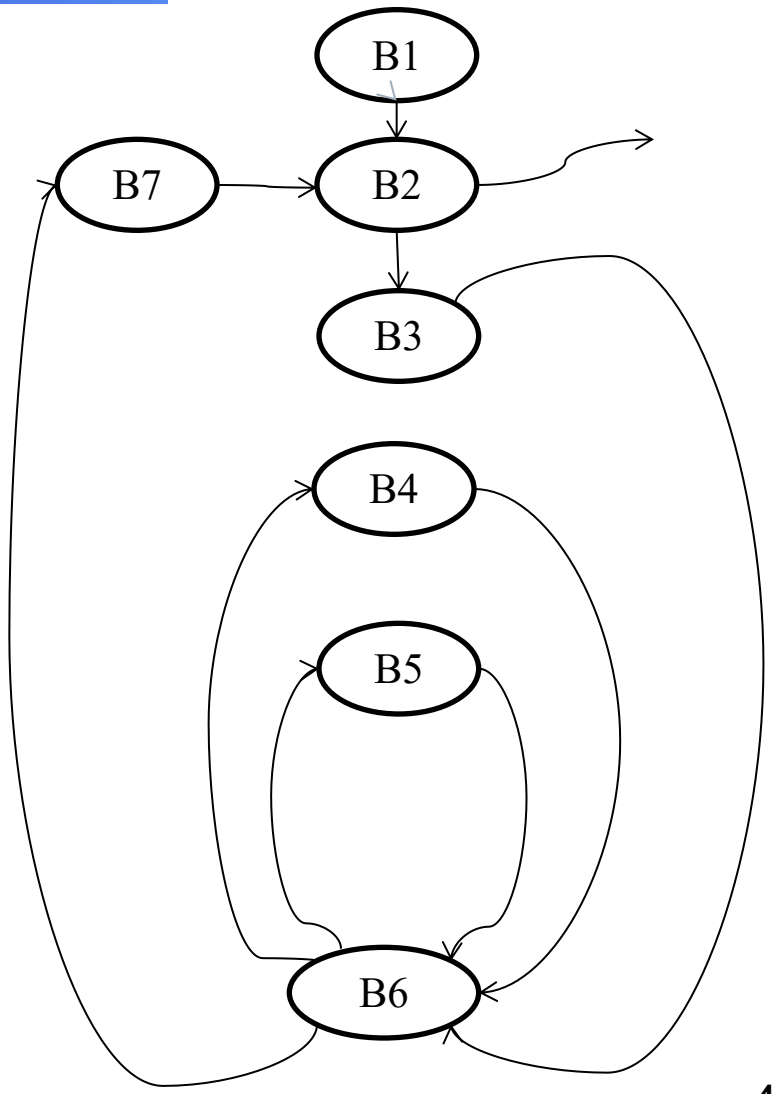
基于克隆的过程间分析

```
        for (i = 0; i < n; i++) {
c1:           t1 = g(0);
c2:           t2 = g(243);
c3:           t3 = g(243);
              X[i] = t1+t2+t3;
        }

        int g (int v) {
c4:          return f(v);
        }

        int f (int v) {
            return (v+1);
        }
```

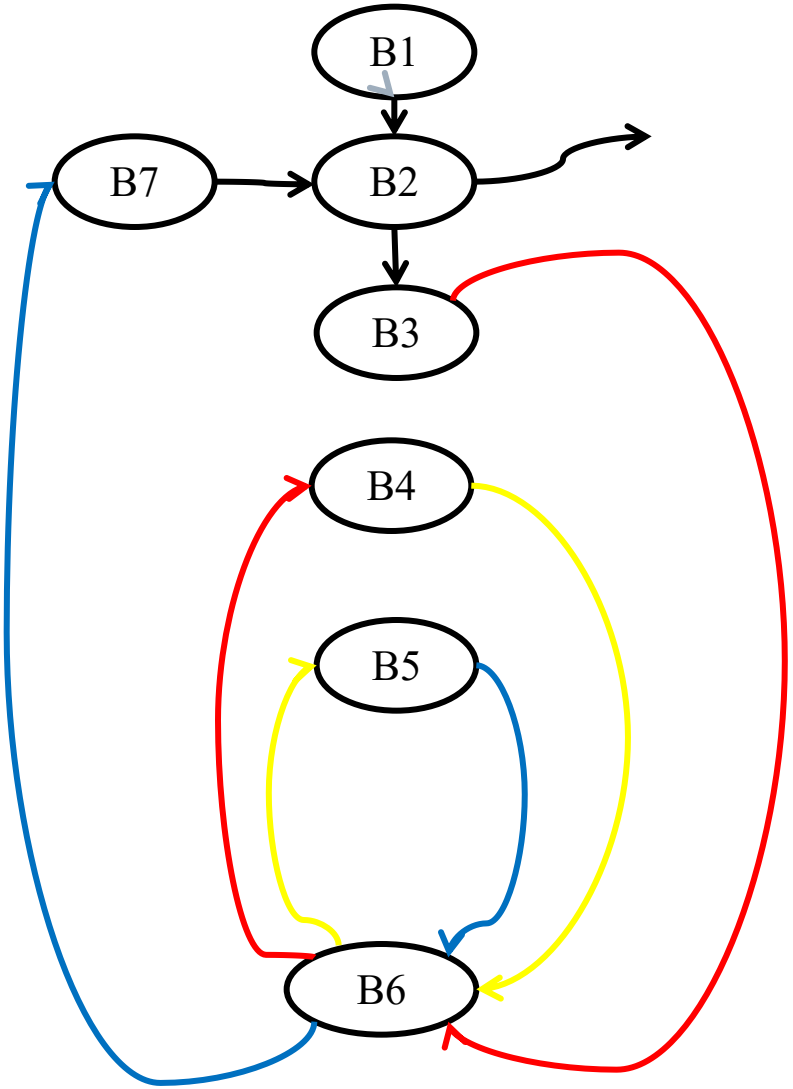
```
        for (i = 0; i < n; i++) {
c1:           t1 = g1(0);
c2:           t2 = g2(243);
c3:           t3 = g3(243);
              X[i] =  t1+t2+t3;
        }
        int g1 (int v) {
c4.1:         return f1(v);
        }
        int g2 (int v) {
c4.2:         return f2(v);
        }
        int g3 (int v) {
c4.3:         return f3(v);
        }

        int f1 (int v) {
            return (v+1);
        }
        int f2 (int v) {
            return (v+1);
        }
        int f3 (int v) {
            return (v+1);
        }
```

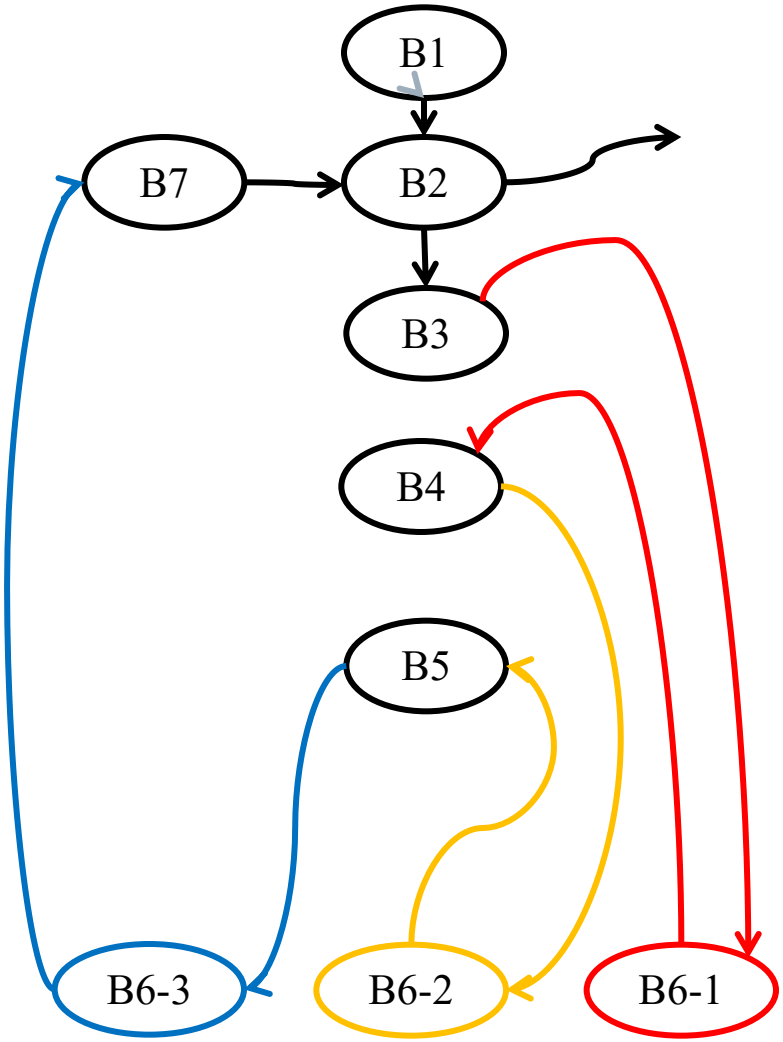
构建超级流图（Super Graph）



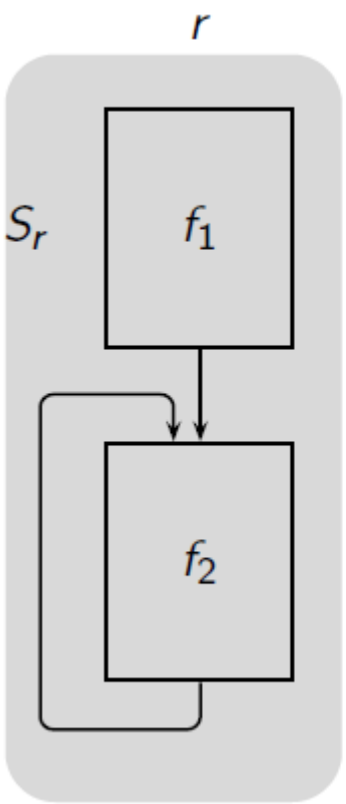


构建超级流图（Super Graph）





基于摘要的过程间分析



Iteration #3

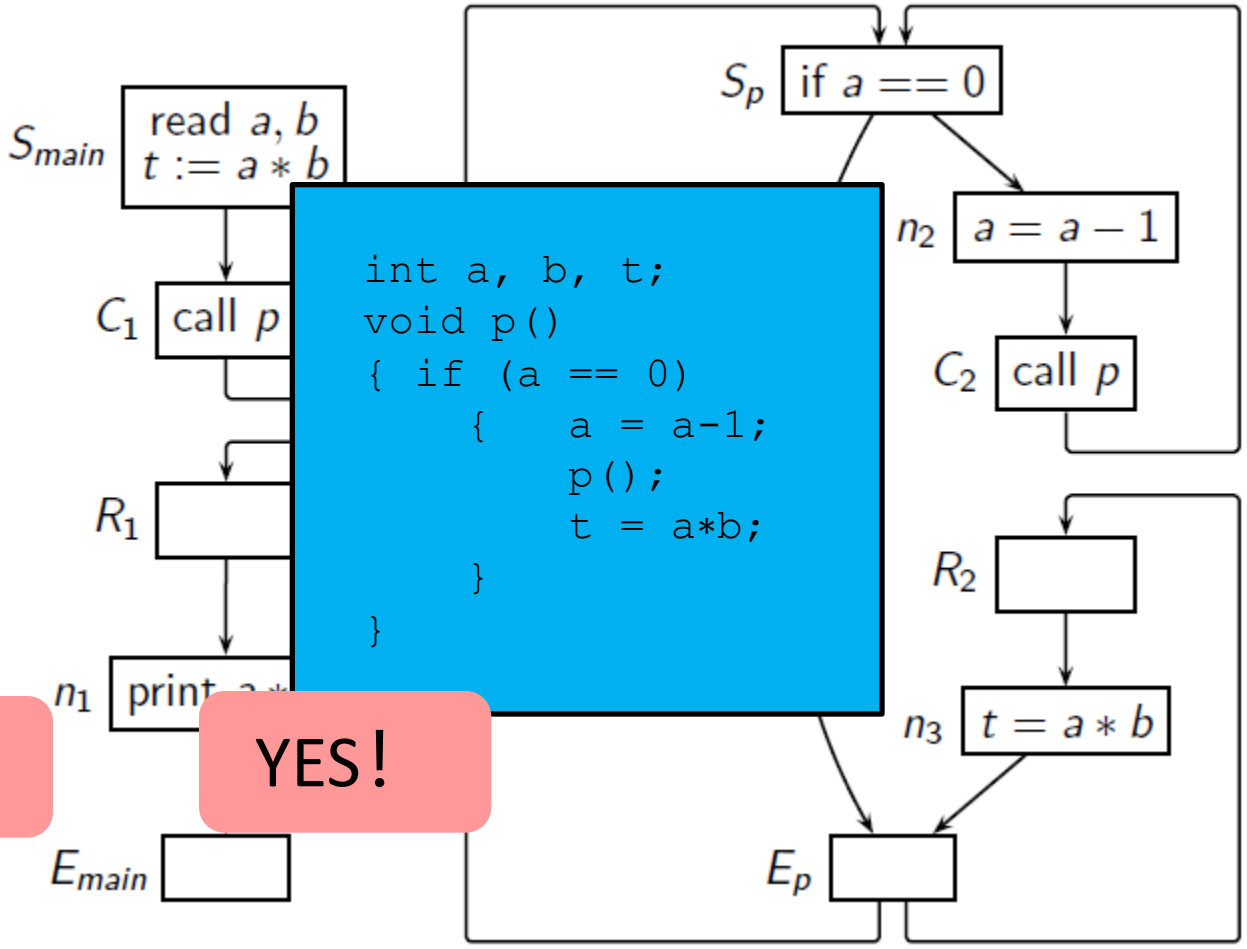
$$\Phi_r(u_1) = \phi_{id}$$

$$\Phi_r(u_2) = f_1$$

$$\Phi_r(u_3) = f_1 \sqcap f_2 \circ (f_1 \sqcap f_2 \circ f_1)$$

$$\Phi_r(u_4) = f_2 \circ (f_1 \sqcap f_2 \circ (f_1 \sqcap f_2 \circ f_1))$$

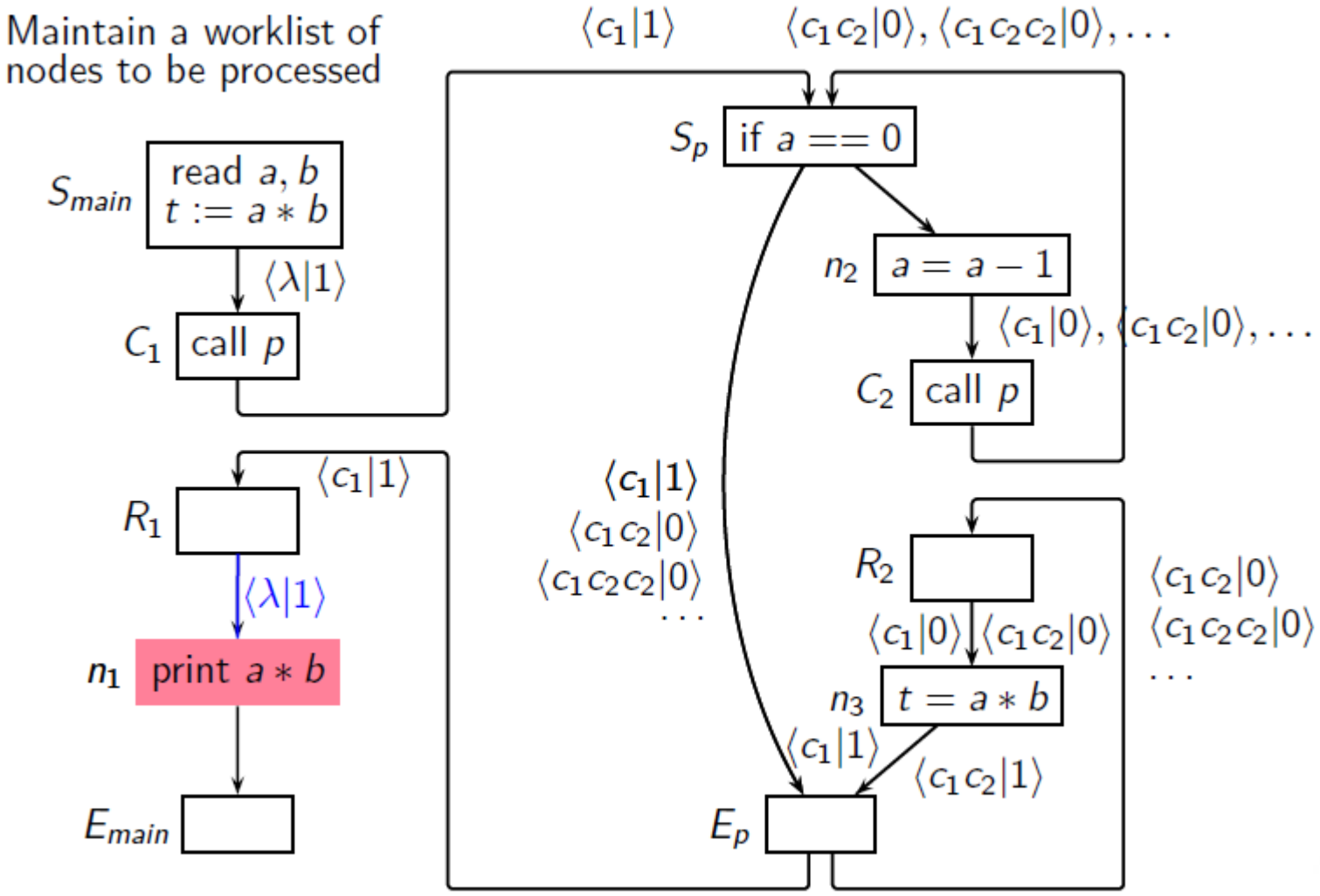
采用调用串的过程间分析



Is a * b available?

YES!

采用调用串的过程间分析



函数内联

- 函数内联是将代码中的函数调用直接替换为等价代码片段的一种变换方法。这非常适用于计算量不大的极简函数的优化。

```
int quadratic( int a, int b, int x )
{ return a*x*x + b*x + 30; }
for(i=0;i<1000;i++){
    y= quadratic(10,20,i*2);
}
for(i=0;i<1000;i++){y = 10*(i*2)*(i*2) + 20*(i*2) + 30;}
for(i=0;i<1000;i++){y = 40*i*i + 40*i + 30;}
```

- 自动权衡有难度，C/C++中有inline关键词支持

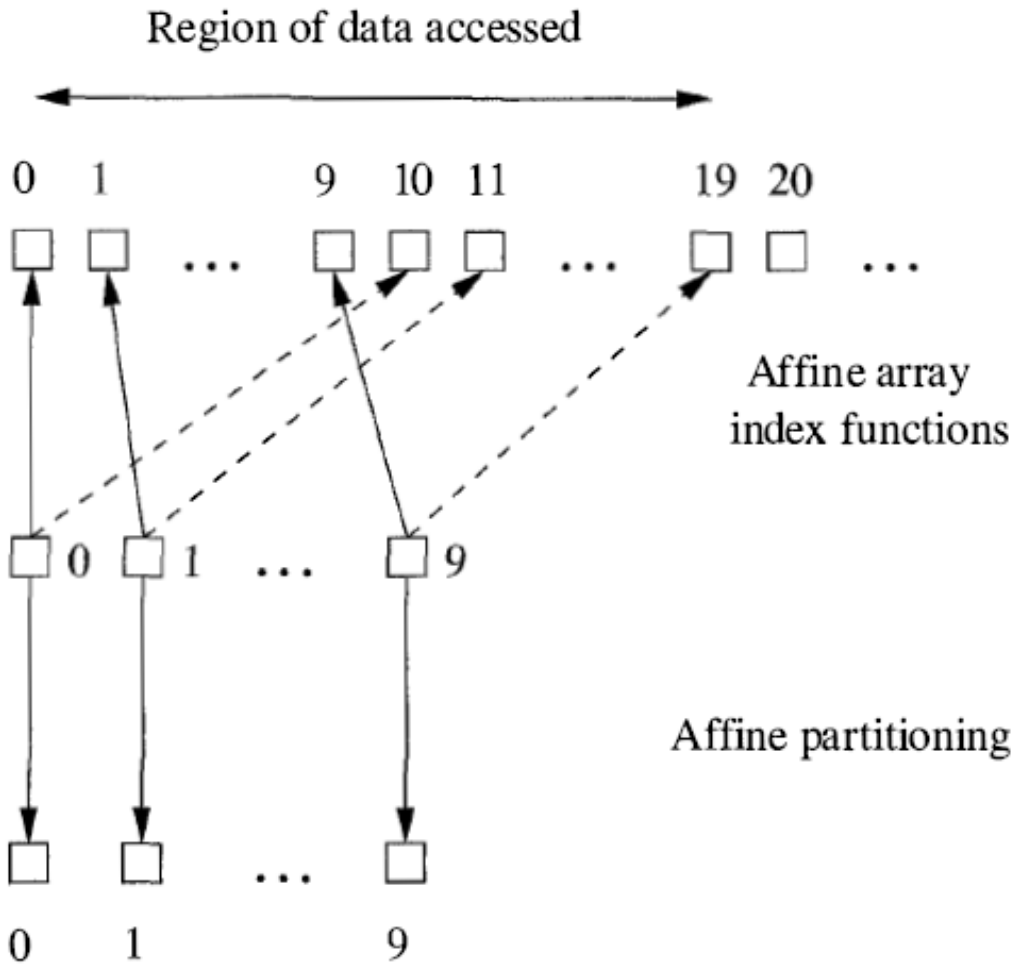
面向并行性和存储局部性的优化

```
float  z[100];
for (i=0;i<10;i++)
    z[i+10] = z[i];
```

数据空间

迭代空间

处理器空间



数据之间的依赖

- **True dependence**

  a =

  = a

  Data dependence

- **Output dependence**

  a =

  a =

- **Anti dependence**

  = a

  a =

- **Input dependence**

  = a

  = a

Name dependence / Pseudo dependence

数据依赖分析

- 仅仅对循环边界和数组下标为基于循环变量的仿射表达式的情况进行讨论。

```
for i = 1 to n
    for j = 2i to 100
        a[i + 2j + 3][4i + 2j][i * i] = …
                                … = a[1][2i + 1][j]
```

- 若下列约束条件可能成立，则假设读写操作之间存在数据相关性：

$$\exists\ i_r, j_r, i_w, j_w \quad 1 \le i_w, i_r \le n \quad 2i_w \le j_w \le 100 \quad 2i_r \le j_r \le 10$$
$$i_w + 2j_w + 3 = 1 \qquad 4i_w + 2j_w = 2i_r + 1$$

- 对数据访问的每个维度进行计算，若下标为非仿射表达式则忽略
  - 方程有整数解 => 可能存在数据相关
  - 方程无整数解 => 无数据相关

最大公约数检测

- 整数a_1，a_2，…，a_n的最大公约数GCD（Greatest Common Divisor）是能够整除所有这些整数的最大整数。

- 定理：线性丢番图方程

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = c$$

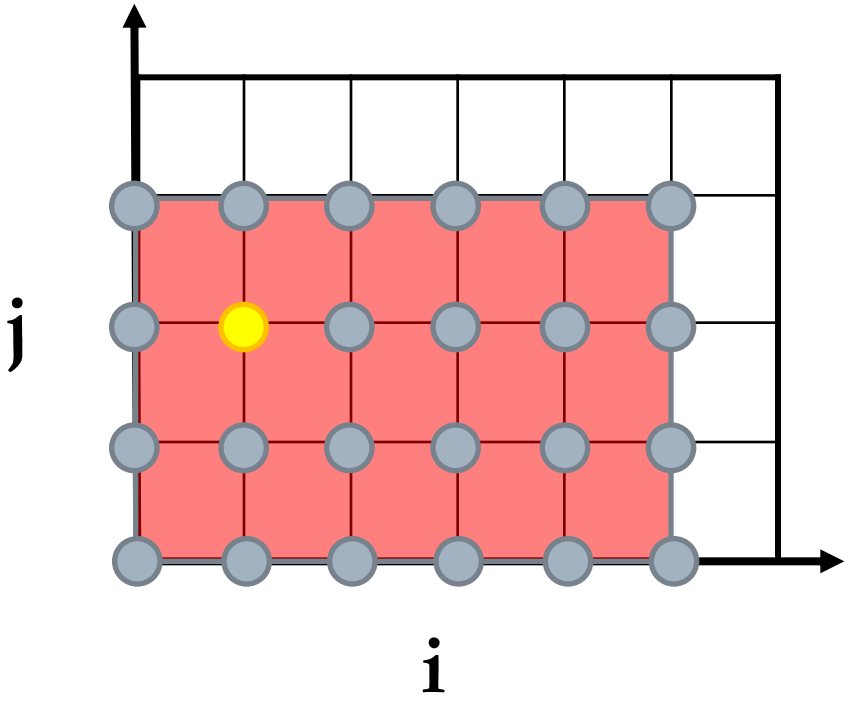
当且仅当$\gcd(a_1, a_2, \ldots, a_n)$能够整除c的时候存在整数解$x_1, x_2, \ldots, x_n$

循环的迭代空间（1）

右图是对下列二重循环的迭代空间的抽象示例。

```
for i= 0, 5
  for j = 0, 3
    a[i, j] =  3
```

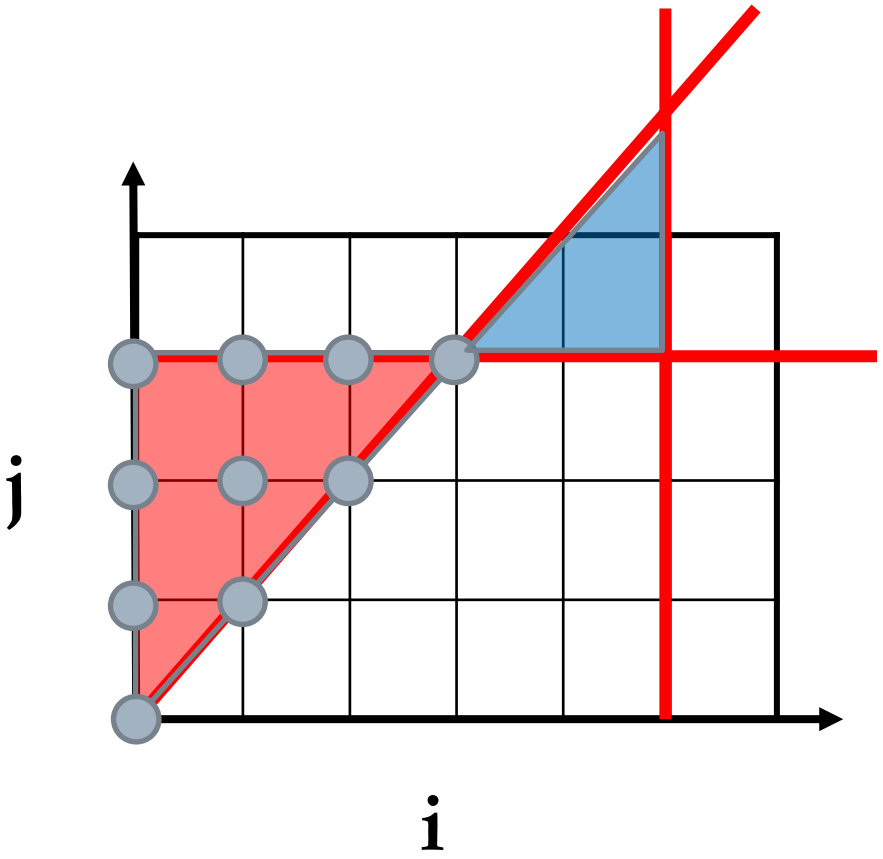
每次迭代可用线性空间中的坐标表示。如i=1, j=2的那次迭代在右图中使用黄色的点标记。



循环的迭代空间（2）

右图是对下列二重循环的迭代空间的抽象示例。

```
for i = 0, 5
  for j = i, 3
    a[i, j] = 0
```



i=0、1、2、3的迭代空间如红色三角区域所标记

i=4、5的情况如蓝色三角区域所标记，并不会执行

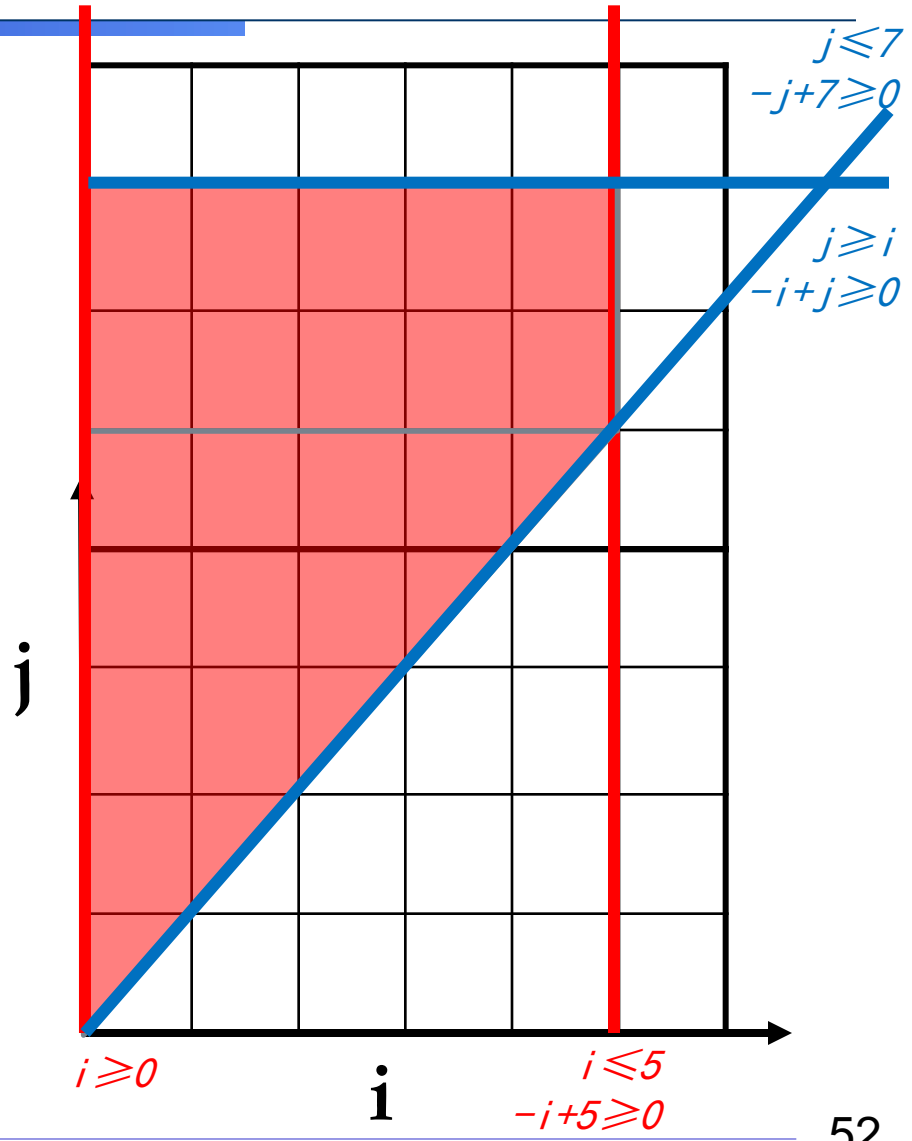
循环的迭代空间（3）

右图是对下列二重循环迭代空间的抽象示例

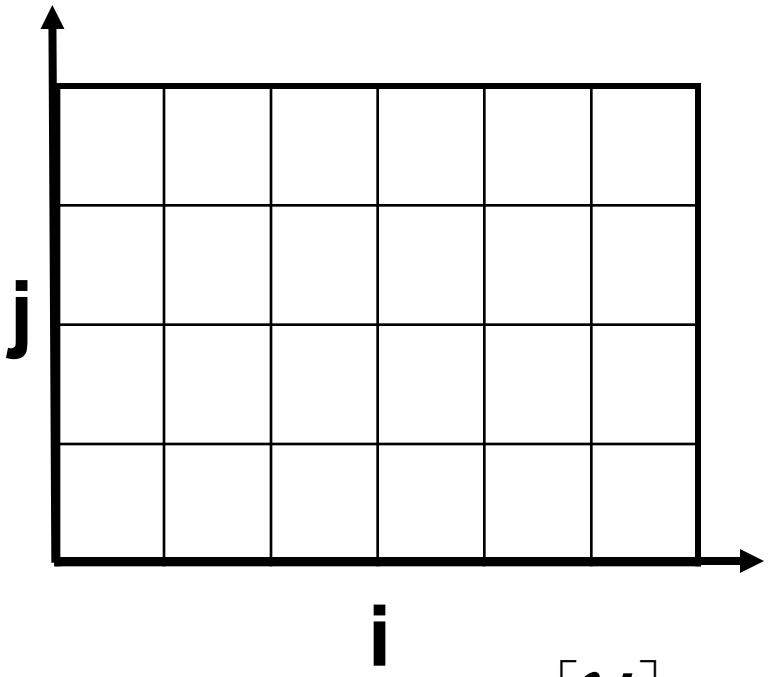
```
for i = 0, 5
  for j = i, 7
    a[i, j] = 0
```

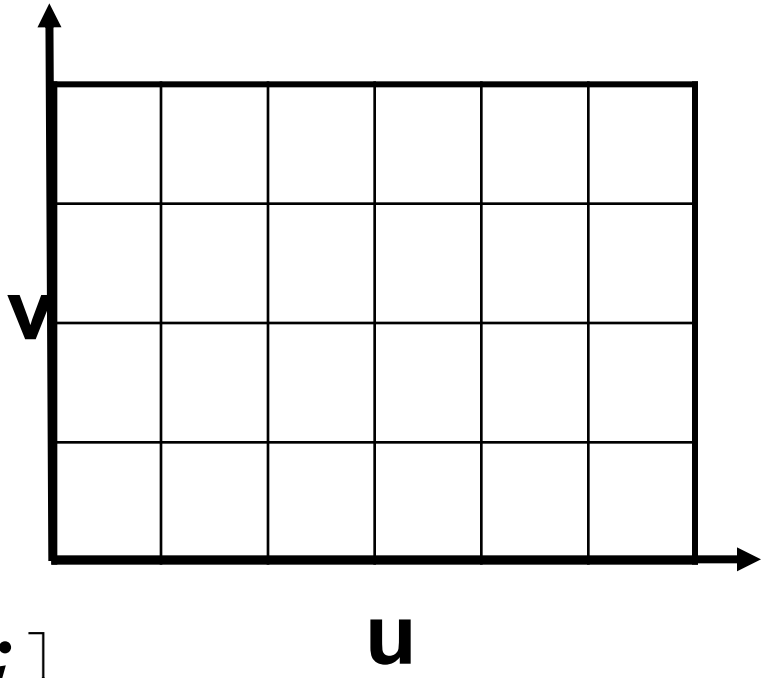
可表示为如下矩阵形式

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ 0 \\ 7 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$



线性空间变换





$$\begin{bmatrix} u \\ v \end{bmatrix} = B \begin{bmatrix} i \\ j \end{bmatrix} + b$$

循环变换，以循环交换为例

```
for i = 1, 100
  for j = 1, 200
    A[i, j] = A[i, j]  + 3
  end_for
end_for
```

```
for u = 1, 200
  for v = 1, 100
    A[v,u] = A[v,u]+ 3
  end_for
end_for
```

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

原循环的迭代空间描述

```
for i = 1, 100
  for j = 1, 200
    A[i, j] = A[i, j]  + 3
  end_for
end_for
```

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 100 \\ -1 \\ 200 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} -1 \\ 100 \\ -1 \\ 200 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

新循环的迭代空间描述

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} -1 \\ 100 \\ -1 \\ 200 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ 0 & -1 \\ 1 & 0 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} -1 \\ 100 \\ -1 \\ 200 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

```
for u = 1, 200
  for v = 1, 100
   A[v,u] = A[v,u]+ 3
  end_for
end_for
```

原循环中数组访问下标

```
for i = 1, 100
  for j = 1, 200
    A[i, j] = A[i, j]  + 3
  end_for
end_for
```

$$A\left[\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix}, \begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix}\right]$$

$$\begin{bmatrix}u\\ v\end{bmatrix}=\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix} \quad A\left[\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}^{-1}\begin{bmatrix}u\\ v\end{bmatrix}, \begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}^{-1}\begin{bmatrix}u\\ v\end{bmatrix}\right]$$

新循环中数组访问下标

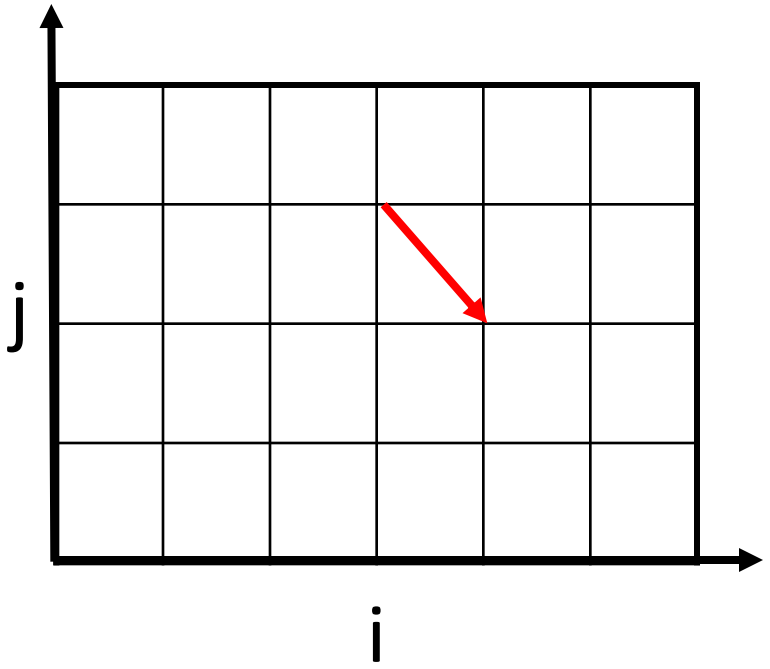
$$A\left[\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}^{-1}\begin{bmatrix}u\\ v\end{bmatrix},\begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}^{-1}\begin{bmatrix}u\\ v\end{bmatrix}\right]$$

```
for u = 1, 200
  for v = 1, 100
   A[v,u] = A[v,u]+ 3
 end_for
end_for
```

$$A\left[\begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}u\\ v\end{bmatrix},\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}u\\ v\end{bmatrix}\right]$$

依赖变量对循环变换的影响

```
for i = 2, 1000
  for j = 1, 1000
    A[i, j] = A[i-1, j+1]+3
  end_for
end_for
```



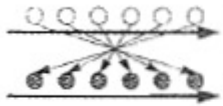

```
for u = 1, 1000
  for v = 2, 1000
    A[v, u] = A[v-1, u+1]+3
  end_for
end_for
```

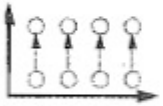
此处的依赖向量（dependence vector）为

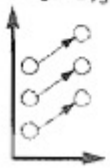
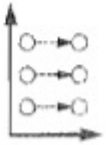
$$d_{old} = (1, -1)$$

循环变换

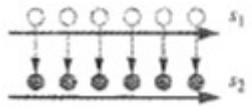
源代码	分划	转换后的代码

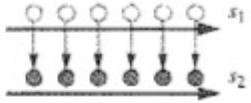
for (i=0; i<=N; i++)
  Y[N-i] = Z[i]; /*s1*/
for (j=0; j<=N; j++)
  X[j] = Y[j]; /*s2*/
 
	反置
	$s_1 : p = N - i$
	$(s_2 : p = j)$
		for (p=0; p<=N; p++){
		  Y[p] = Z[N-p];
		  X[p] = Y[p];
		}
		 

for (i=1; i<=N; i++)
  for (j=0; j<=M; j++)
  Z[i,j] =
  Z[i-1,j];
 
	交换
	$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$
		for (p=0; p<=M; p++)
		  for (q=1; q<=N; q++)
		  Z[q,p] = Z[q-1,p];
		 

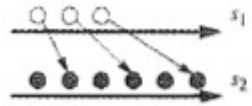
for (i=1; i<=N+M-1; i++)
  for (j=max(1,i+N);
  j<=min(i,M); j++)
  Z[i,j] =
  Z[i-1,j-1];
 
	倾斜
	$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
		for (p=1; p<=N; p++)
		  for (q=1; q<=M; q++)
		  Z[p,q-p] =
		  Z[p-1,q-p-1];
		 

源代码	分划	转换后的代码

for (i=1; i<=N; i++)
  Y[i] = Z[i]; /*s1*/
for (j=1; j<=N; j++)
  X[j] = Y[j]; /*s2*/
 
	融合
	$s_1 : p = i$
	$s_2 : p = j$
		for (p=1; p<=N; p++){
		  Y[p] = Z[p];
		  X[p] = Y[p];
		}
		 

for (p=1; p<=N; p++){
  Y[p] = Z[p];
  X[p] = Y[p];
}
 
	裂变
	$s_1 : i = p$
	$s_2 : j = p$
		for (i=1; i<=N; i++)
		  Y[i] = Z[i]; /*s1*/
		for (j=1; j<=N; j++)
		  X[j] = Y[j]; /*s2*/
		 

for (i=1; i<=N; i++) {
  Y[i] = Z[i]; /*s1*/
  X[i] = Y[i-1]; /*s2*/
}
 
	重新索引
	$s_1 : p = i$
	$s_2 : p = i - 1$
		if (N>=1) X[1]=Y[0];
		for (p=1; p<=N-1; p++){
		  Y[p]=Z[p];
		  X[p+1]=Y[p];
		}
		if (N>=1) Y[N]=Z[N];
		 

for (i=1; i<=N; i++)
  Y[2*i] = Z[2*i]; /*s1*/
for (j=1; j<=2*N; j++)
  X[j]=Y[j]; /*s2*/
 
	比例变换
	$s_1 : p = 2 * i$
	$(s_2 : p = j)$
		for (p=1; p<=2*N; p++){
		  if (p mod 2 == 0)
		  Y[p] = Z[p];
		  X[p] = Y[p];
		}
		 

循环变换示例——循环融合

```
for i = 1, 1000
  A[i] = B[i]  + 3
end_for

for j = 1, 1000
  C[j] = A[j]  + 5
end_for
```

```
for i = 1, 1000
  A[i] = B[i]  + 3
  C[i] = A[i]  + 5
end_for
```

- 经过变换后，数组元素之间具有更好的重用性

循环变换示例——循环裂变

```
for i = 1, 1000
  A[i] = A[i-1]  + 3
  C[i] = B[i]  + 5
end_for
```

```
for i = 1, 1000
  A[i] = A[i-1]  + 3
end_for

for i = 1, 1000
  C[i] = B[i]  + 5
end_for
```

- 原来的循环的不同迭代之间存在数据相关。
- 裂变之后，第二个循环可以被并行。

循环变换示例——大循环划分子块

当数组过大的时候，可能需要划分子块。例如下面的矩阵B在行优先存储的情况下局部性可能不够好，
循环交换并不能解决此问题。

```
for i = 1, 1000
  for j = 1, 1000
    A[i, j] = A[i, j]  +  B[j, i]
  end_for
end_for
```

```
for v = 1, 1000, 20
  for u = 1, 1000, 20
    for j = v, v+19
      for i = u, u+19
        A[i, j] = A[i, j]  +  B[j, i]
      end_for
    end_for
  end_for
end_for
```

对小子块的访问将具有良好的局部性。

循环变换示例——循环展开

```
for j = 1, 2*m
  for i = 1, 2*n
    A[i, j] = A[i-1, j]  +  A[i-1, j-1]
  end_for
end_for
```

```
for j = 1, 2*m, 2
  for i = 1, 2*n, 2
    A[i, j] = A[i-1,j] + A[i-1,j-1]
    A[i, j+1] = A[i-1,j+1] + A[i-1,j]
    A[i+1, j] = A[i, j] + A[i, j-1]
    A[i+1, j+1] = A[i, j+1] + A[i, j]
  end_for
end_for
```

- 数组之间的重用性更好
- 寄存器分块

基于仿射变换的若干应用

- 分布式内存计算机中数据的分配问题。（使用投影的方式决定哪个处理器使用数组的哪一部分）
- 多发射处理器中对关键资源的分配和使用。
- 对向量操作、SIMD指令的编译支持。
- 面向数据局部性的排布优化和预取优化。