COMP130014.02 编译

第八讲：静态单赋值

徐 辉
xuh@fudan.edu.cn

大纲

- 一、IR代码精简：消除冗余Load/Store
- 二、纯寄存器表示
- 三、优化Phi指令

一、IR代码精简：消除冗余Load/Store

线性IR中的Load冗余

```
fn foo(x:int) -> int

bb0:
let y:int = 0;
let z:int = x + y;
if(z>0)

bb1:
y = y + 1;

bb2:
y = y + 1;
z = x + y;
ret z;
```

```
bb0:
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
store i32 %z0, i32* %z
%z1 = load i32, i32* %z
%t0 = icmp sgt i32 %z1, 0
br i1 %t0, label %bb1, label %bb2

bb1:
%y1 = load i32, i32* %y
%y2 = add i32 %y1, 1
store i32 %y2, i32* %y
br label %bb2

bb2:
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%x1 = load i32, i32* %x
%y5 = load i32, i32* %y
%z2 = add i32 %x1, %y5
store i32 %z2, i32* %z
%z3 = load i32, i32* %z
ret i32 %z3
```

优化思路：可用临时寄存器分析

bb0:
```
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
store i32 %z0, i32* %z
%z1 = load i32, i32* %z
%t0 = icmp sgt i32 %z1, 0
br i1 %t0, label %bb1, label %bb2
```

bb1:
```
%y1 = load i32, i32* %y
%y2 = add i32 %y1, 1
store i32 %y2, i32* %y
br label %bb2
```

bb2:
```
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%x1 = load i32, i32* %x
%y5 = load i32, i32* %y
%z2 = add i32 %x1, %y5
store i32 %z2, i32* %z
%z3 = load i32, i32* %z
ret i32 %z3
```

- 正向遍历控制流图
- Transfer函数定义：
  - `%t = load i32, i32* %x`
    - $S_x = S_x \cup \{t\}$
  - `store i32 %t, i32* %x`
    - $S_x = \{t\}$
- 遇到合并节点

$$\mathrm{IN}(n) = \bigcap_{n' \in \mathrm{predecessor}(n)} \mathrm{OUT}(n')$$

分析过程

bb0:

```
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
store i32 %z0, i32* %z
%z1 = load i32, i32* %z
%t0 = icmp sgt i32 %z1, 0
br i1 %t0, label %bb1, label %bb2
```

bb1:

```
%y1 = load i32, i32* %y
%y2 = add i32 %y1, 1
store i32 %y2, i32* %y
br label %bb2
```

bb2:

```
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%x1 = load i32, i32* %x
%y5 = load i32, i32* %y
%z2 = add i32 %x1, %y5
store i32 %z2, i32* %z
%z3 = load i32, i32* %z
ret i32 %z3
```

	S_x	S_y	S_z
bb0: `%x0 = load i32, i32* %x`	{x0}	{}	{}
`%y0 = load i32, i32* %y`	{x0}	{y0}	{}
`%z0 = add i32 %x0, %y0`	{x0}	{y0}	{}
`store i32 %z0, i32* %z`	{x0}	{y0}	{z0}
`%z1 = load i32, i32* %z`	{x0}	{y0}	{z0,z1}
bb1 entry	{x0}	{y0}	{z0,z1}
`%y1 = load i32, i32* %y`	{x0}	{y0,y1}	{z0,z1}
`%y2 = add i32 %y1, 1`	{x0}	{y0,y1}	{z0,z1}
`store i32 %y2, i32* %y`	{x0}	{y2}	{z0,z1}
bb2 entry	{x0}∩{x0}	{y0}∩{y2}	{z0,z1}∩{z0,z1}
`%y3 = load i32, i32* %y`	{x0}	{y3}	{z0,z1}
`%y4 = add i32 %y3, 1`	{x0}	{y3}	{z0,z1}
`store i32 %y4, i32* %y`	{x0}	{y4}	{z0,z1}
`%x1 = load i32, i32* %x`	{x0,x1}	{y4}	{z0,z1}
`%y5 = load i32, i32* %y`	{x0,x1}	{y4,y5}	{z0,z1}
`%z2 = add i32 %x1, %y5`	{x0,x1}	{y4,y5}	{z0,z1}
`store i32 %z2, i32* %z`	{x0,x1}	{y4,y5}	{z2}
`%z3 = load i32, i32* %z`	{x0,x1}	{y4,y5}	{z2,z3}

分析结果

	Instruction	S_x	S_y	S_z
bb0:	%x = alloca i32			
	%y = alloca i32			
	%z = alloca i32			
	store i32 %0, i32* %x			
	store i32 0, i32* %y			
	%x0 = load i32, i32* %x	{x0}	{}	{}
	%y0 = load i32, i32* %y	{x0}	{y0}	{}
	%z0 = add i32 %x0, %y0	{x0}	{y0}	{}
	store i32 %z0, i32* %z	{x0}	{y0}	{z0}
	~~%z1 = load i32, i32* %z~~	{x0}	{y0}	{z0,z1}
	%t0 = icmp sgt i32 %z1, 0			
	br i1 %t0, label %bb1, label %bb2			
bb1:	~~%y1 = load i32, i32* %y~~	{x0}	{y0,y1}	{z0,z1}
	%y2 = add i32 %y1, 1	{x0}	{y0,y1}	{z0,z1}
	store i32 %y2, i32* %y	{x0}	{y2}	{z0,z1}
	br label %bb2			
bb2:		{x0}∩{x0}	{y0}∩{y2}	{z0,z1}∩{z0,z1}
	%y3 = load i32, i32* %y	{x0}	{y3}	{z0,z1}
	%y4 = add i32 %y3, 1	{x0}	{y3}	{z0,z1}
	store i32 %y4, i32* %y	{x0}	{y4}	{z0,z1}
	~~%x1 = load i32, i32* %x~~	{x0,x1}	{y4}	{z0,z1}
	~~%y5 = load i32, i32* %y~~	{x0,x1}	{y4,y5}	{z0,z1}
	%z2 = add i32 %x1, %y5	{x0,x1}	{y4,y5}	{z0,z1}
	store i32 %z2, i32* %z	{x0,x1}	{y4,y5}	{z2}
	~~%z3 = load i32, i32* %z~~	{x0,x1}	{y4,y5}	{z2,z3}
	ret i32 %z3			

优化结果

bb0:

```
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
store i32 %z0, i32* %z
%t0 = icmp sgt i32 %z0, 0
br i1 %t0, label %bb1, label %bb2
```

bb1:

```
%y2 = add i32 %y0, 1
store i32 %y2, i32* %y
br label %bb2
```

bb2:

```
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%z2 = add i32 %x0, %y4
store i32 %z2, i32* %z
ret i32 %z2
```

伪代码

```
For (each instruction n):
    IN[n] = {<v: ∅ >: v is a program variable}
    OUT[n] = {<v: ∅ >}
Repeat:
    For(each instruction n):
        For(each n's predecessor p)
            IN[n] = IN[n] ∩ OUT[p]
        OUT[n] = TRANSFER(n)
Until IN[n] and OUT[n] stops changing for all n
```

问题：算法是否一定会终止？

- 有循环的情况：
  - 每个程序节点的可用寄存器数目是单调递减的

```
fn fac(n: int) -> int {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    ret r;
}
```

```
define i32 @fac(i32 %0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, i32* %n
    store i32 1, i32* %r
    br label %bb1
bb1:
    %t1 = load i32, i32* %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, i32* %r
    %t4 = load i32, i32* %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, i32* %r
    %t6 = load i32, i32* %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, i32* %n
    br label %bb1
bb3:
    %t8 = load i32, i32* %r
    ret i32 %t8
}
```

线性IR中的Store冗余

```
fn foo(x:int) -> int

bb0:
let y:int = 0;
let z:int = x + y;
if(z>0)

bb1:
y = y + 1;

bb2:
y = y + 1;
z = x + y;
ret z;
```

```
bb0:
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
store i32 %z0, i32* %z
%t0 = icmp sgt i32 %z0, 0
br i1 %t0, label %bb1, label %bb2

bb1:
%y2 = add i32 %y0, 1
store i32 %y2, i32* %y
br label %bb2

bb2:
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%z2 = add i32 %x0, %y4
store i32 %z2, i32* %z
ret i32 %z2
```

优化思路：可用Store语句分析

bb0:

```
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
store i32 %z0, i32* %z
%t0 = icmp sgt i32 %z0, 0
br i1 %t0, label %bb1, label %bb2
```

bb1:

```
%y2 = add i32 %y0, 1
store i32 %y2, i32* %y
br label %bb2
```

bb2:

```
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%z2 = add i32 %x0, %y4
store i32 %z2, i32* %z
ret i32 %z2
```

- 逆向遍历控制流图
- Transfer函数定义：
  - `store i32 %t, i32* %x`
    - $S = S \cup \{x\}$
  - `%t = load i32, i32* %x`
    - $S = S \backslash \{x\}$
  - `%t = alloc, i32* %x`
    - $S = S \backslash \{x\}$
- 遇到合并节点

$$\mathrm{OUT}(n) = \bigcap_{n' \in \mathrm{successor}(n)} \mathrm{IN}(n')$$

分析过程

bb0:

```
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
store i32 %z0, i32* %z
%t0 = icmp sgt i32 %z0, 0
br i1 %t0, label %bb1, label %bb2
```

bb1:

```
%y2 = add i32 %y0, 1
store i32 %y2, i32* %y
br label %bb2
```

bb2:

```
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%z2 = add i32 %x0, %y4
store i32 %z2, i32* %z
ret i32 %z2
```

S

```
{}
{z}
{z}
{z}
{z}
{z}
{z}
{z}
{z}
{y,z}∩{z}

{y,z}
{y,z}
{z}

{z}
{y,z}
{y,z}
{z}
{z}
{}
{}
```

分析结果

bb0:

{}
`%x = alloca i32`
{z}
`%y = alloca i32`
{z}
`%z = alloca i32`
{z}
`store i32 %0, i32* %x`
{z}
`store i32 0, i32* %y`
{z}
`%x0 = load i32, i32* %x`
{z}
`%y0 = load i32, i32* %y`
{z}
`%z0 = add i32 %x0, %y0`
{z}
~~`store i32 %z0, i32* %z`~~
{z}
`%t0 = icmp sgt i32 %z0, 0`
{z}
`br i1 %t0, label %bb1, label %bb2`
{z}

bb1:

{y,z}
`%y2 = add i32 %y0, 1`
{y,z}
`store i32 %y2, i32* %y`
{z}
`br label %bb2`

bb2:

{z}
`%y3 = load i32, i32* %y`
{y,z}
`%y4 = add i32 %y3, 1`
{y,z}
`store i32 %y4, i32* %y`
{z}
`%z2 = add i32 %x0, %y4`
{z}
`store i32 %z2, i32* %z`
{}
`ret i32 %z2`
{}

优化结果

bb0:

```
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
%t0 = icmp sgt i32 %z0, 0
br i1 %t0, label %bb1, label %bb2
```

bb1:

```
%y2 = add i32 %y0, 1
store i32 %y2, i32* %y
br label %bb2
```

bb2:

```
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%z2 = add i32 %x0, %y4
store i32 %z2, i32* %z
ret i32 %z2
```

二、纯寄存器表示

消除数据存取

bb0:
```
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
%t0 = icmp sgt i32 %z0, 0
br i1 %t0, label %bb1, label %bb2
```

bb1:
```
%y2 = add i32 %y0, 1
store i32 %y2, i32* %y
br label %bb2
```

bb2:
```
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%z2 = add i32 %x0, %y4
store i32 %z2, i32* %z
ret i32 %z2
```

分析方法：内存数值流分析

bb0:

```
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
%t0 = icmp sgt i32 %z0, 0
br i1 %t0, label %bb1, label %bb2
```

bb1:

```
%y2 = add i32 %y0, 1
store i32 %y2, i32* %y
br label %bb2
```

bb2:

```
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%z2 = add i32 %x0, %y4
store i32 %z2, i32* %z
ret i32 %z2
```

- 正向遍历控制流图
- Transfer函数定义：
  - store i32 %t, i32* %x
    - $S_{\mathrm{x}} = \{t\}$
- 遇到合并节点

$$\mathrm{IN}(n) = \bigcup_{n' \in \mathrm{predecessor}(n)} \mathrm{OUT}(n')$$

分析过程

bb0:
```
%x = alloca i32
%y = alloca i32
%z = alloca i32
store i32 %0, i32* %x
store i32 0, i32* %y
%x0 = load i32, i32* %x
%y0 = load i32, i32* %y
%z0 = add i32 %x0, %y0
%t0 = icmp sgt i32 %z0, 0
br i1 %t0, label %bb1, label %bb2
```

bb1:
```
%y2 = add i32 %y0, 1
store i32 %y2, i32* %y
br label %bb2
```

bb2:
```
%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
store i32 %y4, i32* %y
%z2 = add i32 %x0, %y4
store i32 %z2, i32* %z
ret i32 %z2
```

S_x	S_y	S_z
{}	{}	{}
{%0}	{}	{}
{%0}	{0}	{}
{%0}	{0}	{}
{%0}	{0}	{}
{%0}	{0}	{}
{%0}	{0}	{}
{%0}	{0}	{}
{%0}	{0}	{}
{%0}	{y2}	{}
{%0}	{y2}	{}
{%0}	{y2}	{}
{%0}∪{%0}	{0}∪{y2}	
{%0}	{0}∪{y2}	{}
{%0}	{0}∪{y2}	{}
{%0}	{y4}	{}
{%0}	{y4}	{}
{%0}	{y4}	{}
{}	{}	{z2}

分析结果

bb0:

~~%x = alloca i32~~
~~%y = alloca i32~~
~~%z = alloca i32~~
~~store i32 %0, i32* %x~~
~~store i32 0, i32* %y~~
~~%x0 = load i32, i32* %x~~
~~%y0 = load i32, i32* %y~~
%z0 = add i32 %x0, %y0
%t0 = icmp sgt i32 %z0, 0
br i1 %t0, label %bb1, label %bb2

S_x	S_y	S_z
{}	{}	{}
{%0}	{}	{}
{%0}	{0}	{}
{%0}	{0}	{}
{%0}	{0}	{}
{%0}	{0}	{}
{%0}	{0}	{}
{%0}	{0}	{}

bb1:

%y2 = add i32 %y0, 1
~~store i32 %y2, i32* %y~~
br label %bb2

S_x	S_y	S_z
{%0}	{0}	{}
{%0}	{y2}	{}
{%0}	{y2}	{}
{%0}	{y2}	{}

bb2:

%y3 = load i32, i32* %y
%y4 = add i32 %y3, 1
~~store i32 %y4, i32* %y~~
%z2 = add i32 %x0, %y4
~~store i32 %z2, i32* %z~~
ret i32 %z2

S_x	S_y	S_z
{%0}	{0,y2}	
{%0}	{0,y2}	{}
{%0}	{0,y2}	{}
{%0}	{y4}	{}
{%0}	{y4}	{}
{%0}	{y4}	{}
{}	{}	{z2}

纯寄存器表示

```
bb0:
  %z0 = add i32 %0, 0
  %t0 = icmp sgt i32 %z0, 0
  br i1 %t0, label %bb1, label %bb2

bb1:
  %y2 = add i32 0, 1
  br label %bb2

bb2:
  %y3 = phi i32 [ 0, %bb0 ], [ %y2, %bb1 ]
  %y4 = add i32 %y3, 1
  %z2 = add i32 %x0, %y4
  ret i32 %z2
```

练习：将下列代码转化为纯寄存器表示

```
define i32 @fac(i32 %0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, i32* %n
    store i32 1, i32* %r
    br label %bb1
bb1:
    %t1 = load i32, i32* %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, i32* %r
    %t4 = load i32, i32* %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, i32* %r
    %t6 = load i32, i32* %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, i32* %n
    br label %bb1
bb3:
    %t8 = load i32, i32* %r
    ret i32 %t8
}
```

数据流分析方法小结

	May Analysis (∪)	Must Analysis (∩)
Forward	纯寄存器表示	精简Load
Backward		精简Store

数据流分析方法小结：框架

$\boxed{\quad\quad}$ [n] = Transfer $\boxed{\quad\quad}$ [n]

$\boxed{\quad\quad}$ [n] = $\boxed{\quad\quad}$ $\boxed{\quad\quad}$ [n’]

n’ ∈ $\boxed{\quad\quad}$ (n)

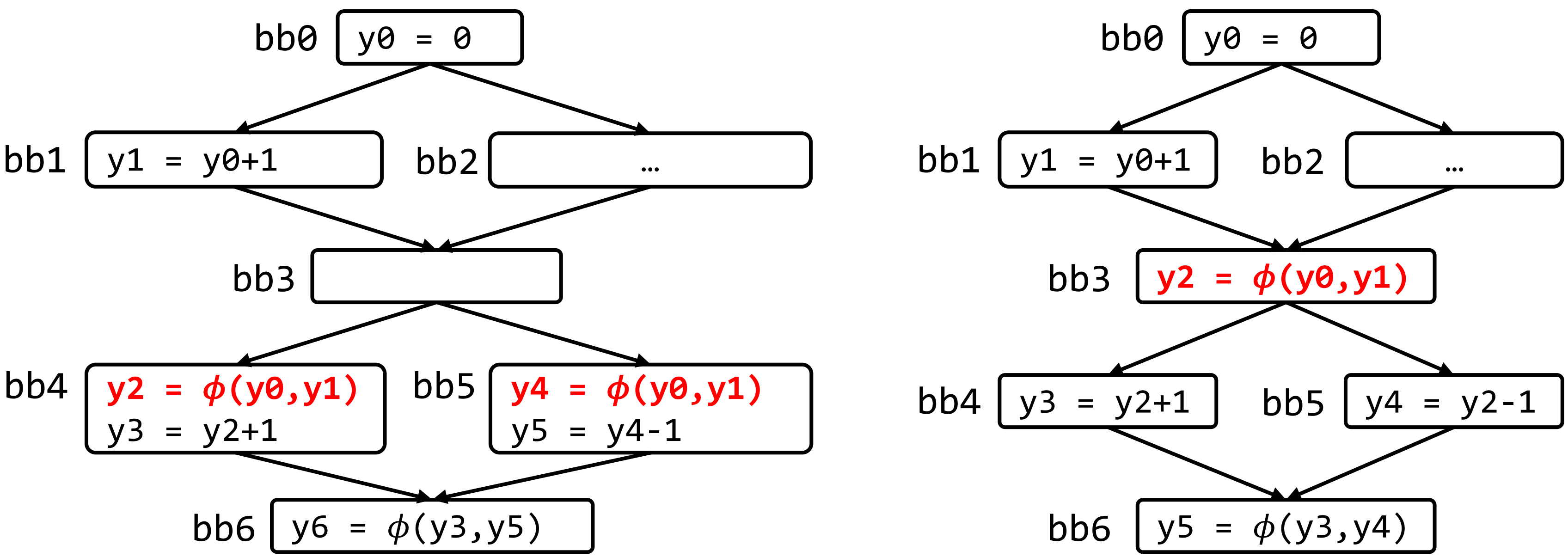
$\boxed{\quad\quad}$ $\boxed{\quad\quad}$ = IN or OUT

$\boxed{\quad\quad}$ = U (may) or ∩ (must)

$\boxed{\quad\quad}$ = predecessors or successors

三、Phi指令优化

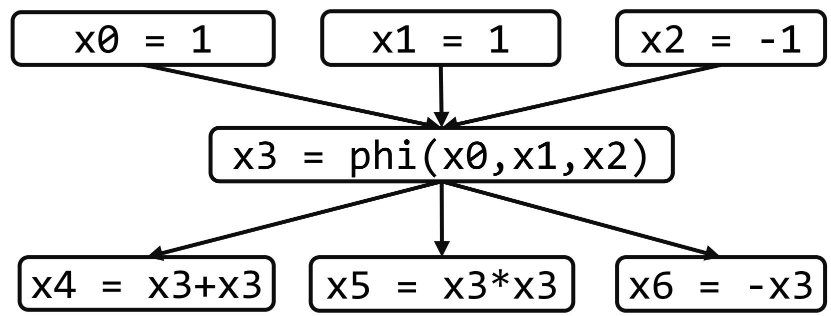
哪个Phi指令方案更优？



SSA简化def-use关系

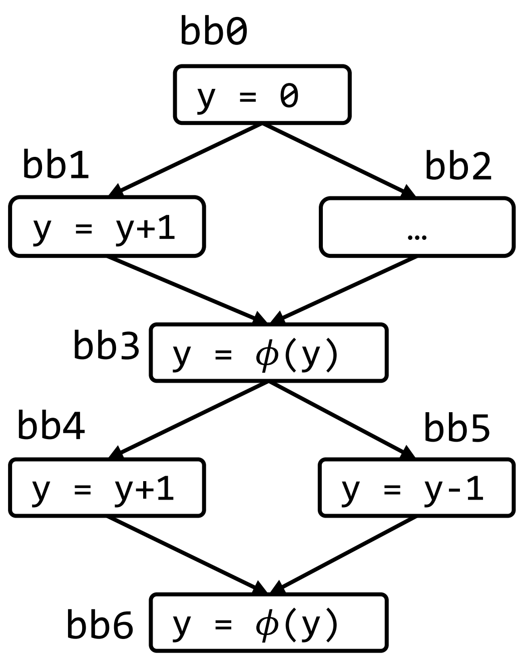
- 原始程序的def-use关系数量是$O(m \times n)$；
- SSA的def-use数量减少为$O(m + n)$。

```
match v1:
    0 => { x = 0; }
    1 => { x = 1; }
    _ => { x = -1; }
...
match v2:
    0 => { x = x + x; }
    1 => { x = x * x; }
    _ => { x = -x; }
```



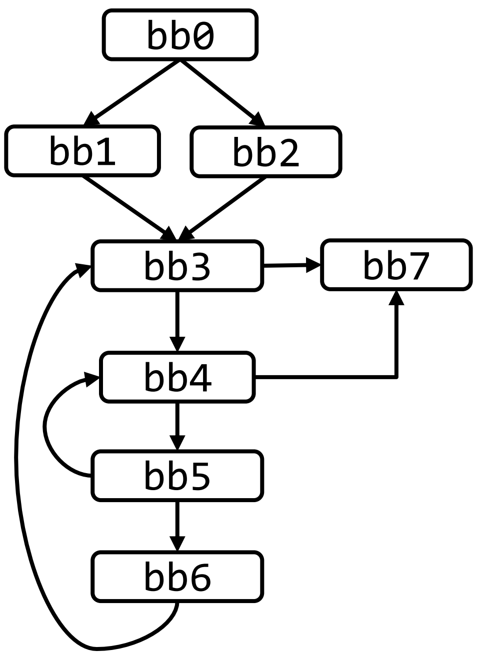
优化思路：基于支配边界优化Phi指令

- bb0支配bb2，bb1和bb2的支配边界都是bb3
- 如果bb1和bb2中都没有def(x)，bb3不需要phi(x)，可直接使用bb0中的def(x)
- 如果bb1中有def(y)，bb3中一定需要phi(y)



支配的基本概念

- 给定有向图$G(V,E)$与起点v_0，如果从v_0到某个点v_j均需要经过点v_i，则称v_i支配v_j或v_i是v_j 的一个支配点
  - $v_i \in Dom(v_j)$
- 如果$v_i \neq v_j$，则称v_i严格支配v_j



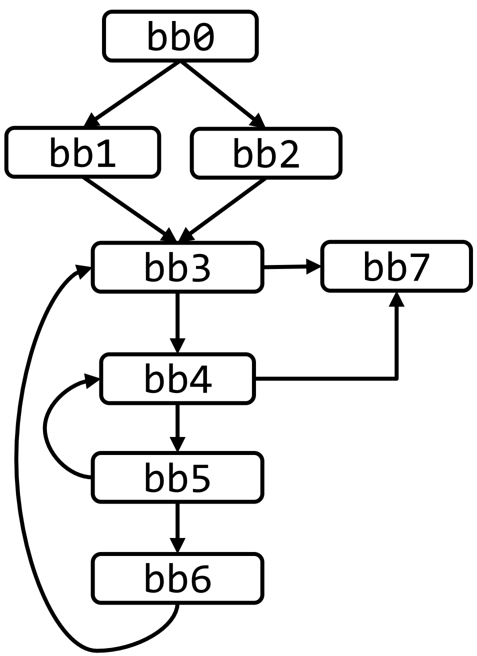
控制流图

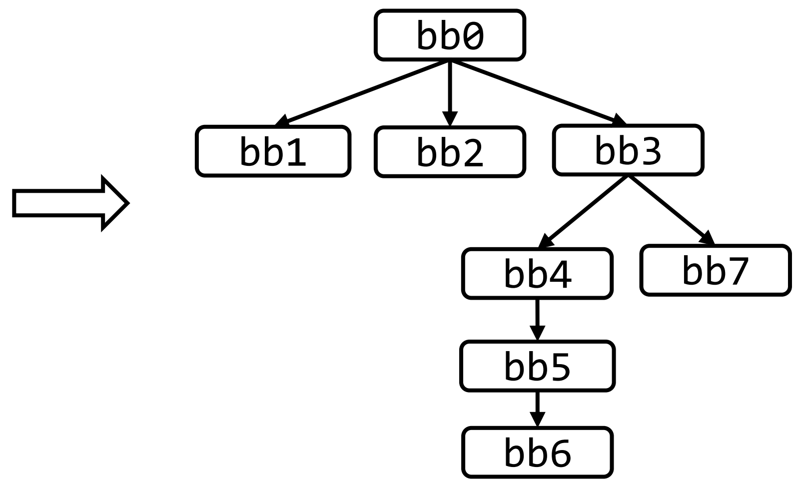
$$
\begin{aligned}
Dom(bb_0) &= \{bb_0\} \\
Dom(bb_1) &= \{bb_0, bb_1\} \\
Dom(bb_2) &= \{bb_0, bb_2\} \\
Dom(bb_3) &= \{bb_0, bb_3\} \\
Dom(bb_4) &= \{bb_0, bb3, bb_4\} \\
Dom(bb_5) &= \{bb_0, bb3, bb_4, bb_5\} \\
Dom(bb_6) &= \{bb_0, bb3, bb_4, bb5, bb_6\} \\
Dom(bb_7) &= \{bb_0, bb_3\}
\end{aligned}
$$

$$
Dom(v) = \begin{cases} \{v\}, & if\ v = v_0 \\ \{v\} \cup \left(\bigcap\limits_{p \in pred(v)} Dom(p) \right), & if\ v \neq v_0 \end{cases}
$$

支配树的基本概念

- 所有v_j的严格支配点中与v_j最接近的点成为v_j的最近支配点
  - $Idom(v_j) = v_i$，v_j的其它严格支配点均严格支配v_i
- 连接接所有的最近支配关系，形成一棵支配树
  - 根节点外的每一点均存在唯一的最近支配点

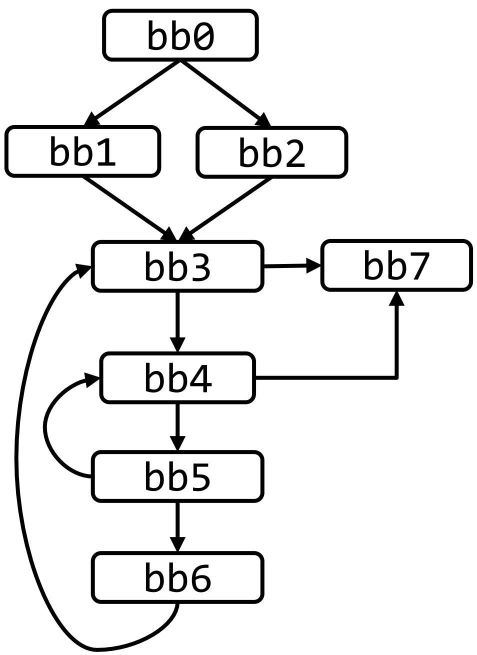




支配树

支配边界Dominance Frontier

- v_i的支配边界是所有满足条件的v_j的集合
  - v_i支配v_j的一个前序节点
  - v_i并不严格支配v_j

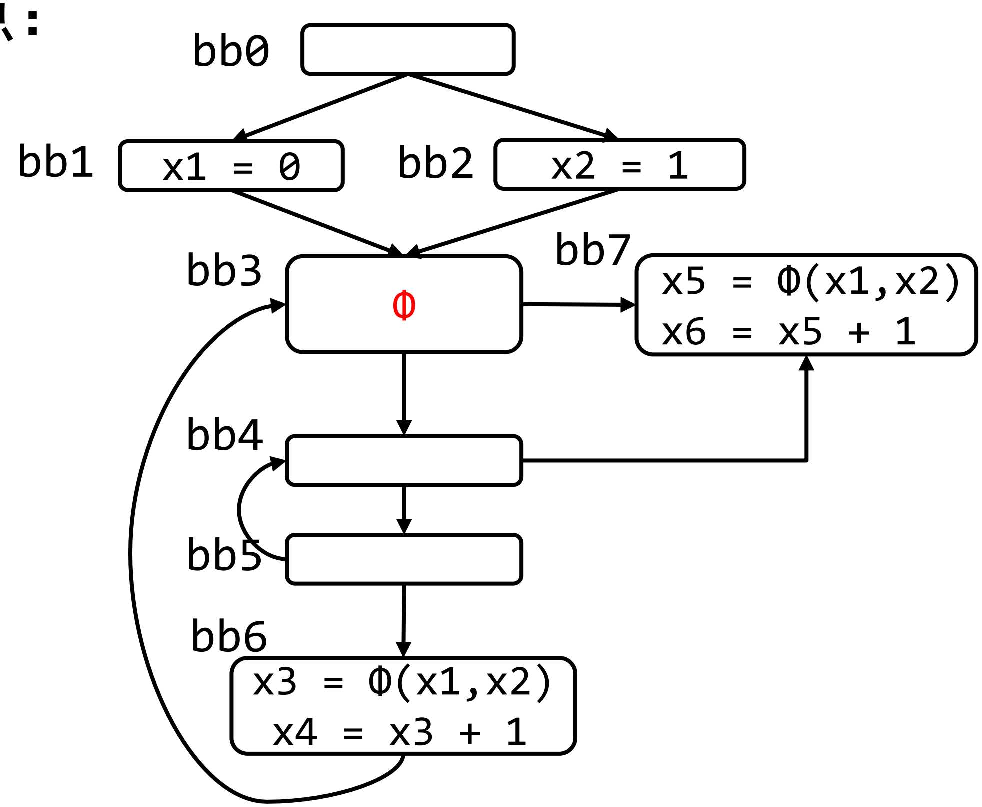


$$DF(bb_0) = \{\}$$
$$DF(bb_1) = \{bb_3\}$$
$$DF(bb_2) = \{bb_3\}$$
$$DF(bb_3) = \{bb_3\}$$
$$DF(bb_4) = \{bb_3, bb_4, bb_7\}$$
$$DF(bb_5) = \{bb_3, bb_4\}$$
$$DF(bb_6) = \{bb_3\}$$
$$DF(bb_7) = \{\}$$

利用支配边界设置Phi指令

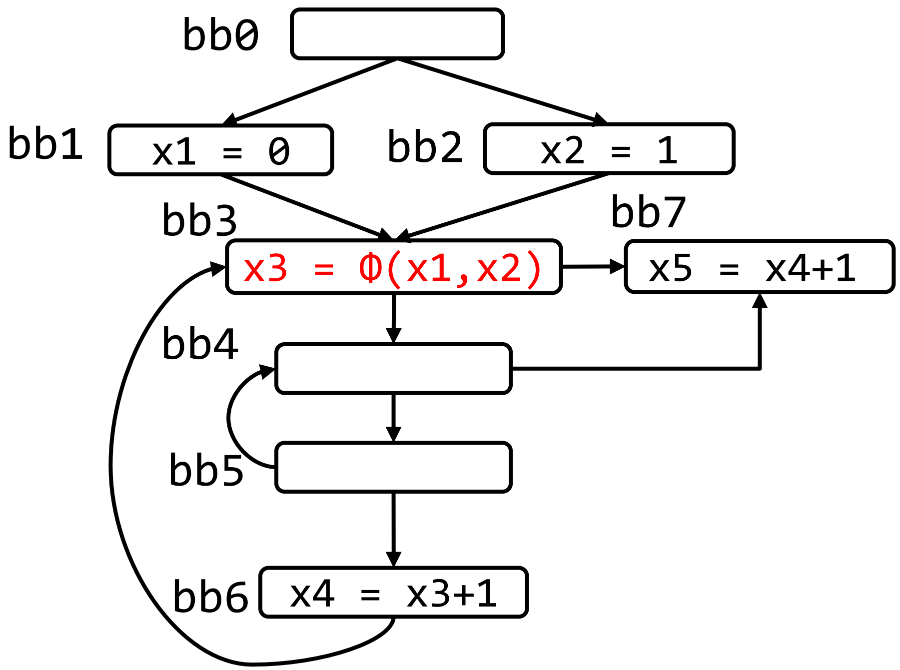
- 初始化：枚举所有变量的def-sites
  - def-sites(x) = {bb1,bb2,bb6,bb7}
- 为每个变量在bb_j增加phi节点：
  - $bb_i \in$ def-sites(x)
  - $bb_j \in DF(bb_i)$
- 在bb3增加phi指令的phi(x)

$$
\begin{aligned}
&DF(bb_0) = \{\} \\
&DF(bb_1) = \{bb_3\} \\
&DF(bb_2) = \{bb_3\} \\
&DF(bb_3) = \{bb_3\} \\
&DF(bb_4) = \{bb3, bb_4, bb_7\} \\
&DF(bb_5) = \{bb3, bb_4\} \\
&DF(bb_6) = \{bb_3\} \\
&DF(bb_7) = \{\}
\end{aligned}
$$



优化结果

- 重新编号
- 删除只有一个元素的phi指令



练习

- 分析右侧代码：
  1） 计算支配树
  2） 计算支配边界
  3） 修改为SSA