

CS30017 编译

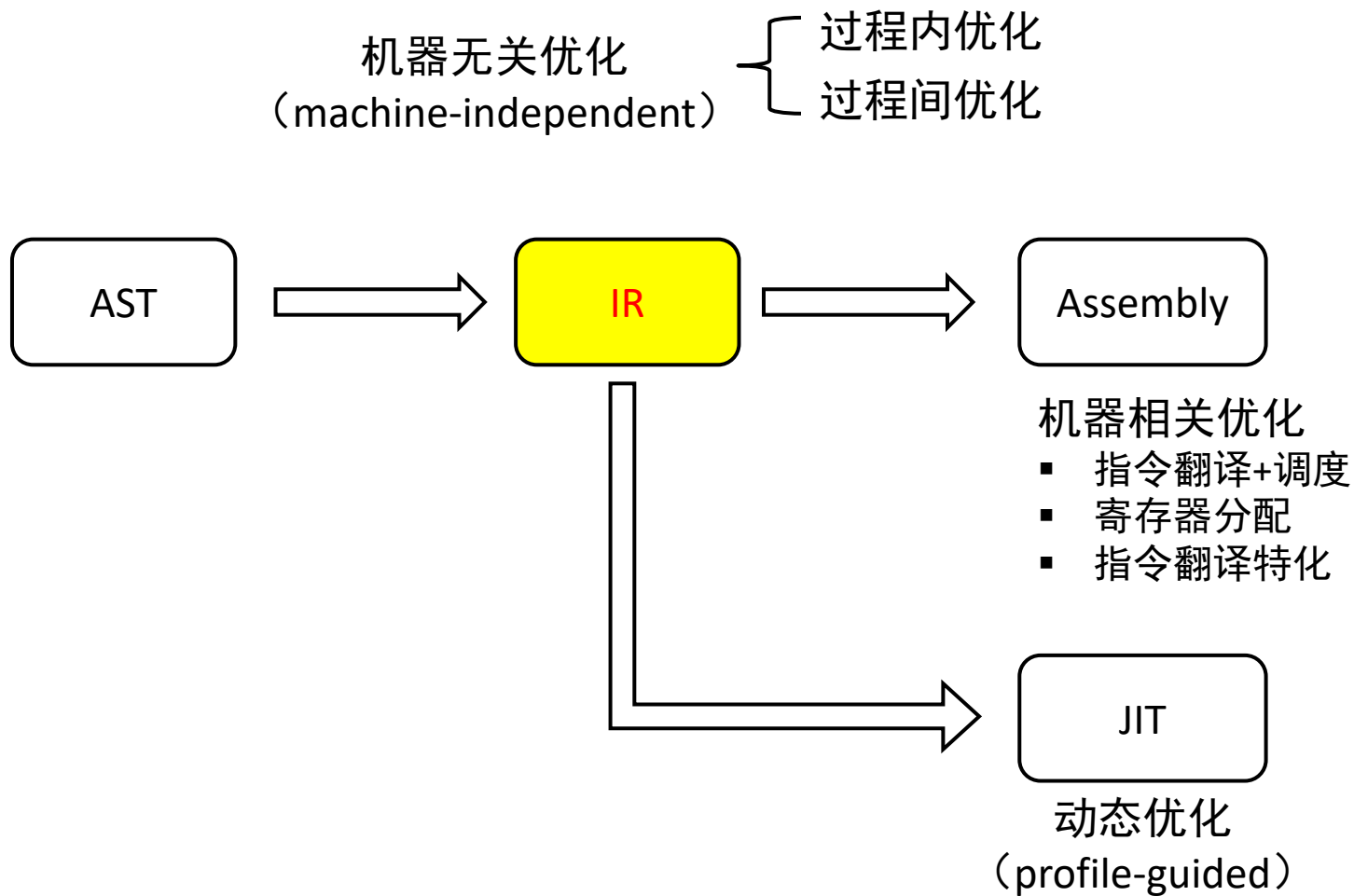
第九讲：IR过程内优化

徐辉

xuh@fudan.edu.cn



优化策略



过程内优化

一、常量传播优化

二、冗余代码优化

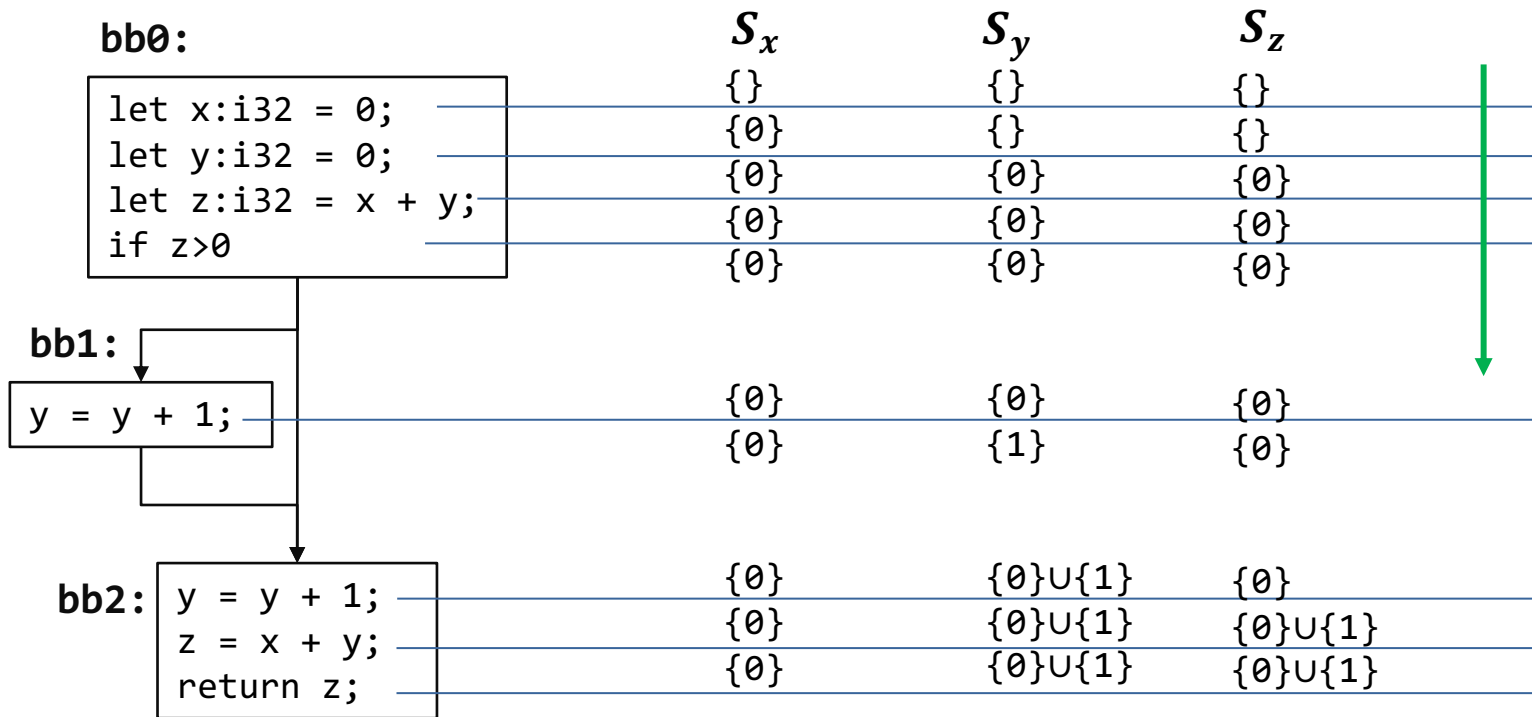
三、循环优化

四、更多优化思路

一、常量传播优化

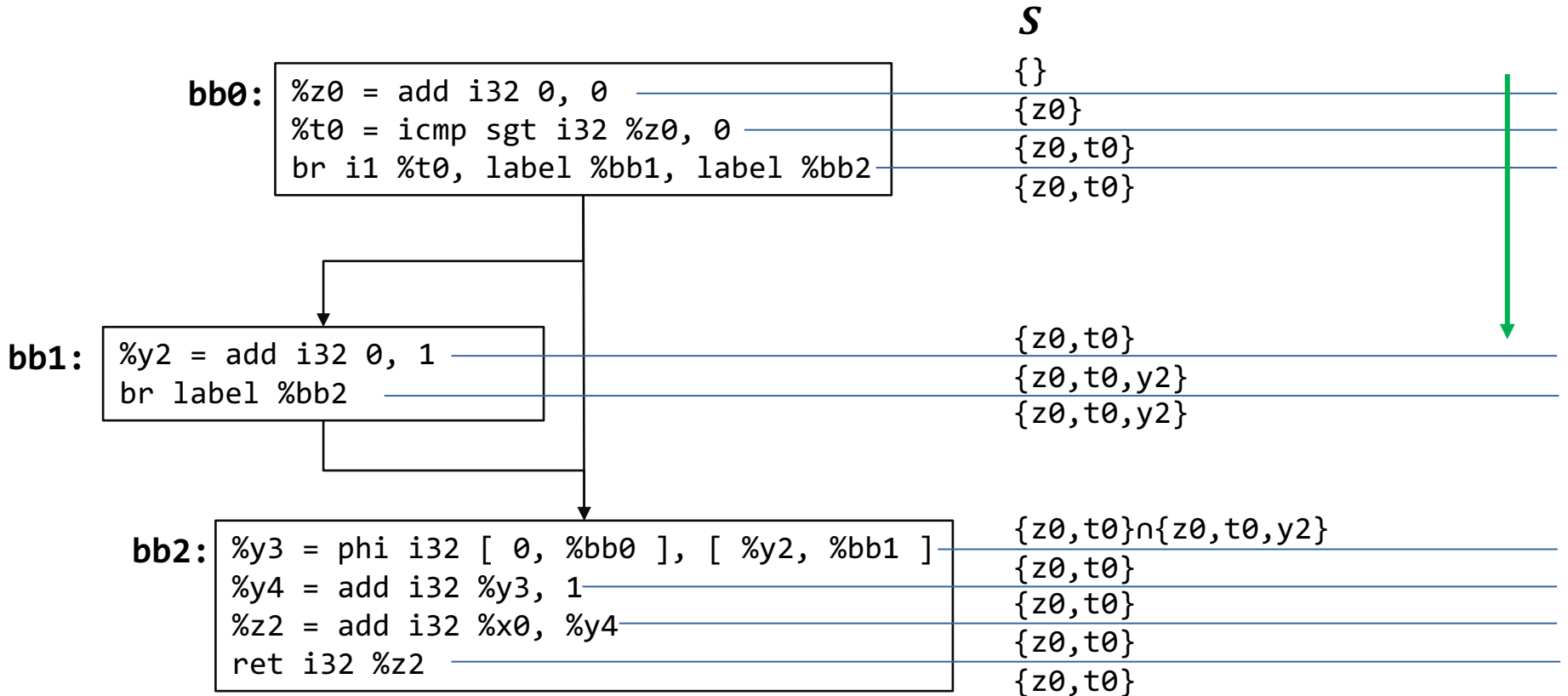
常量分析问题

- 分析变量/寄存器x在特定程序节点p是否为常量



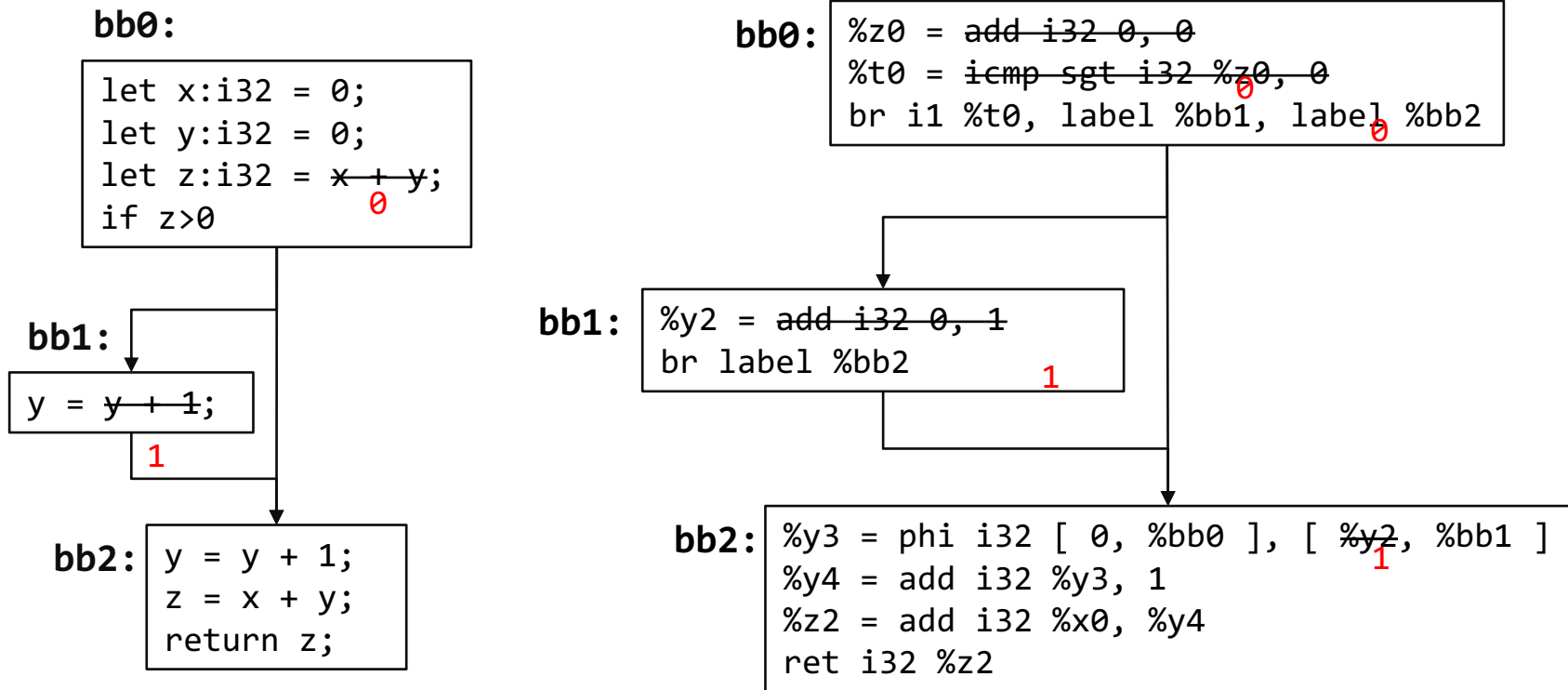
基于SSA的常量分析

- 分析哪些寄存器内容为常量

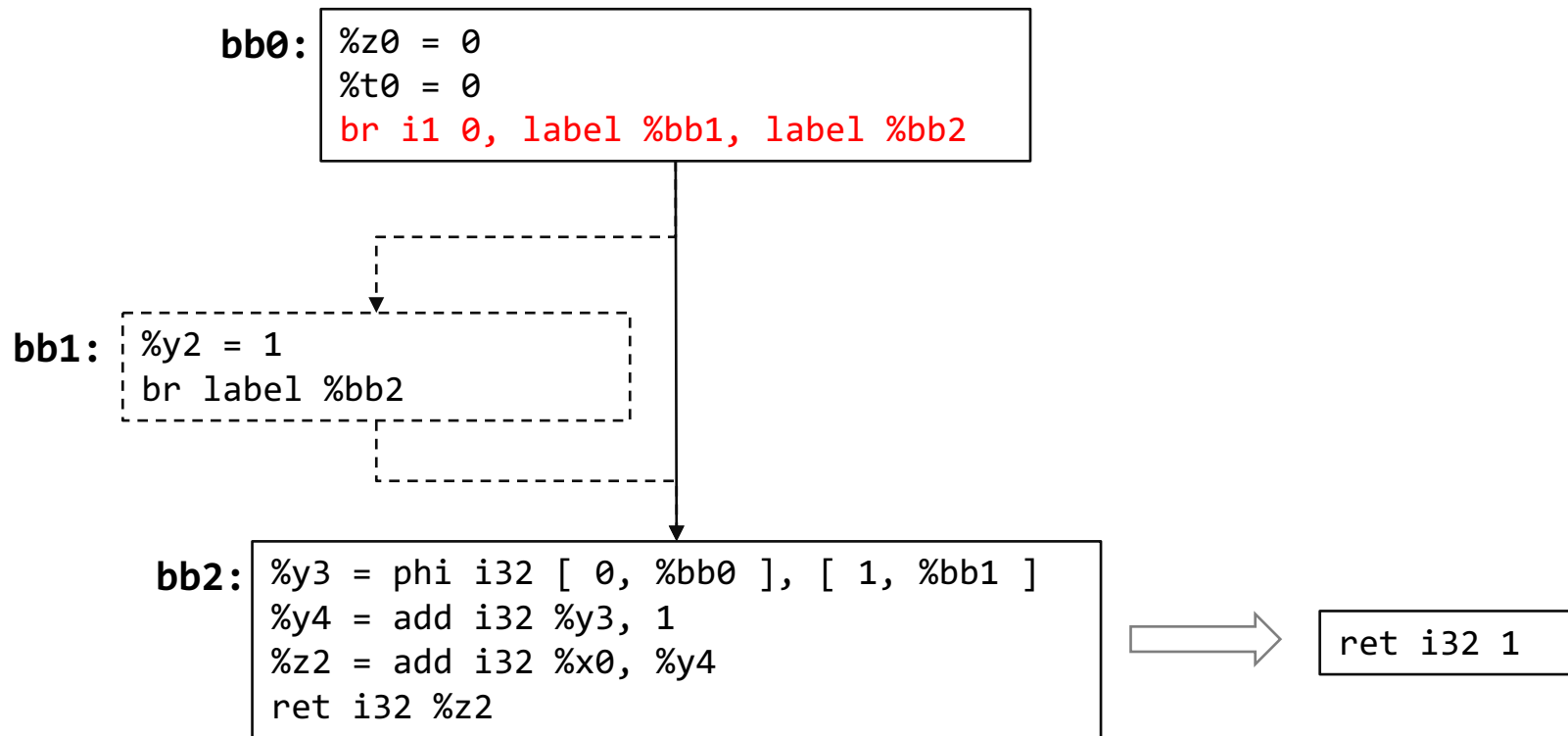


主要思想：在编译时完成常量相关的计算

- 常量传播：识别常量并将相应的变量替换为常量
- 常量折叠：编译时完成对常量表达式的计算



继续优化



删除不可达代码块

常量分析优化

指令合并

- 两条二元运算指令满足一定条件时可以合并：
 - 指令1：一个运算数为常量，另一个为变量
 - 指令2：一个运算数为常量，另一个为指令1的运算结果

```
y = x + 1  
y = y + 2  
z = y + 3
```



```
y = x + 1  
y = x + 3  
z = x + 6
```

```
%y0 = add i32 %x0, 1  
%y1 = add i32 %y0, 2  
%z0 = add i32 %y1, 3
```



```
%y0 = add i32 %x0, 1  
%y1 = add i32 %x0, 3  
%z0 = add i32 %x0, 6
```

思考：指令合并数据流分析算法实现

- 面向非SSA形式的源代码或Gimple IR
- 面向非完全 SSA形式LLVM IR：使用load/store
- 面向完全SSA形式LLVM IR

二、冗余代码优化

删除死代码：优化代码体积

- 代码块不可达：条件语句恒真或恒假
- 无用计算：缺少use的def
- ...

```
%y0 = add i32 %x0, 1  
%y1 = add i32 %y0, 2  
%z0 = add i32 %y1, 3
```



指令合并

```
%y0 = add i32 %x0, 1  
%y1 = add i32 %x0, 3  
%z0 = add i32 %x0, 6
```

全局值编号 (Global Value Numbering)

- 相同的运算 (运算符、运算数) 只算一次即可

```
%y0 = add i32 %x0, 1  
%y1 = add i32 %x0, 1  
%z0 = add i32 %x0, 1
```

```
%y0 = add i32 %x0, 1  
%y1 = %y0  
%z0 = %y0
```

非LLVM IR; 必须以指令开头



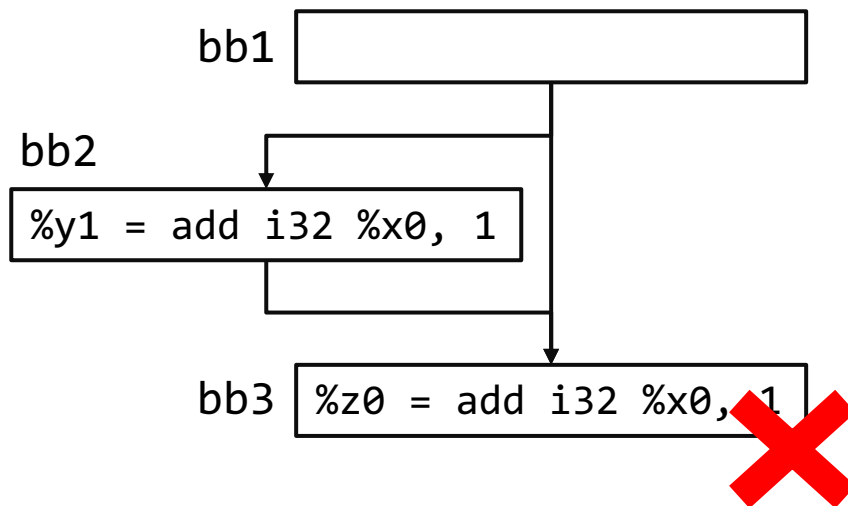
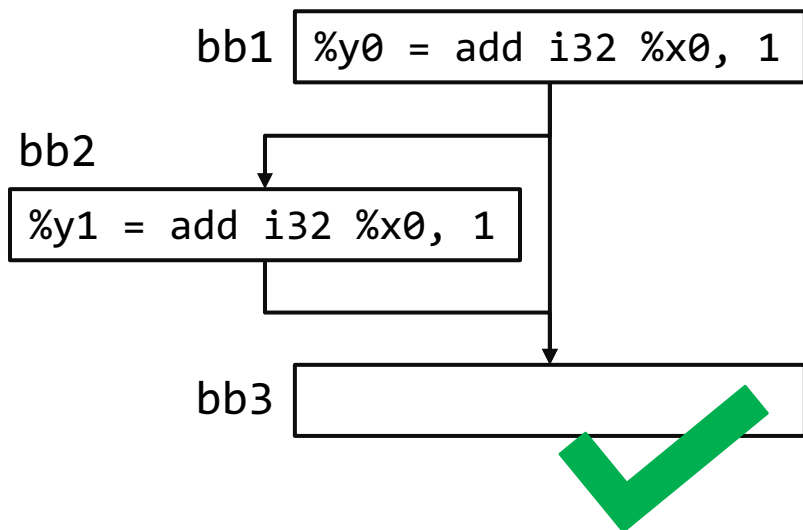
等价LLVM IR

```
%y0 = add i32 %x0, 1  
%x0 = bitcast i32 %y0 to i32  
%z0 = bitcast i32 %y0 to i32
```

或直接替换USE(%y1)为USE(%y0)

GVN应用：公共子表达式（可用表达式）

- 该表达式在存在支配关系的两条指令中重复出现

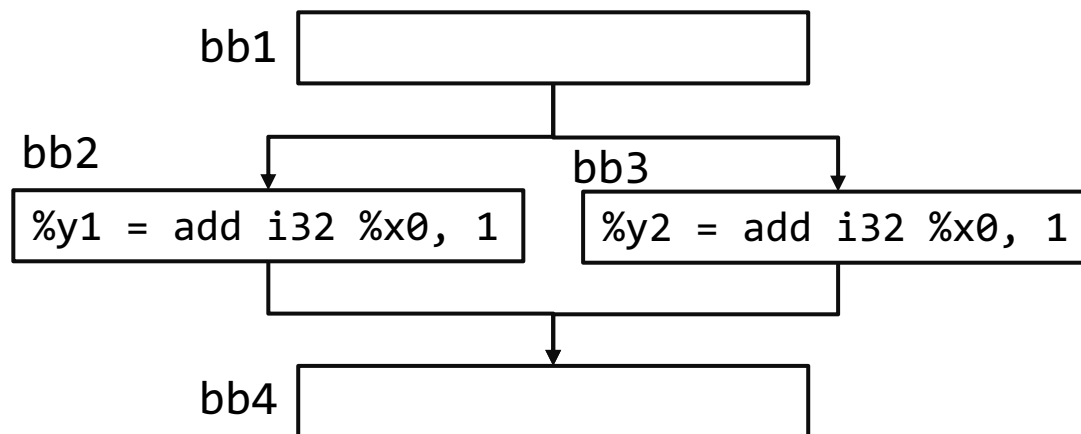


弱公共子表达式

逆向数据流分析 => 代码提升

GVN应用：繁忙表达式

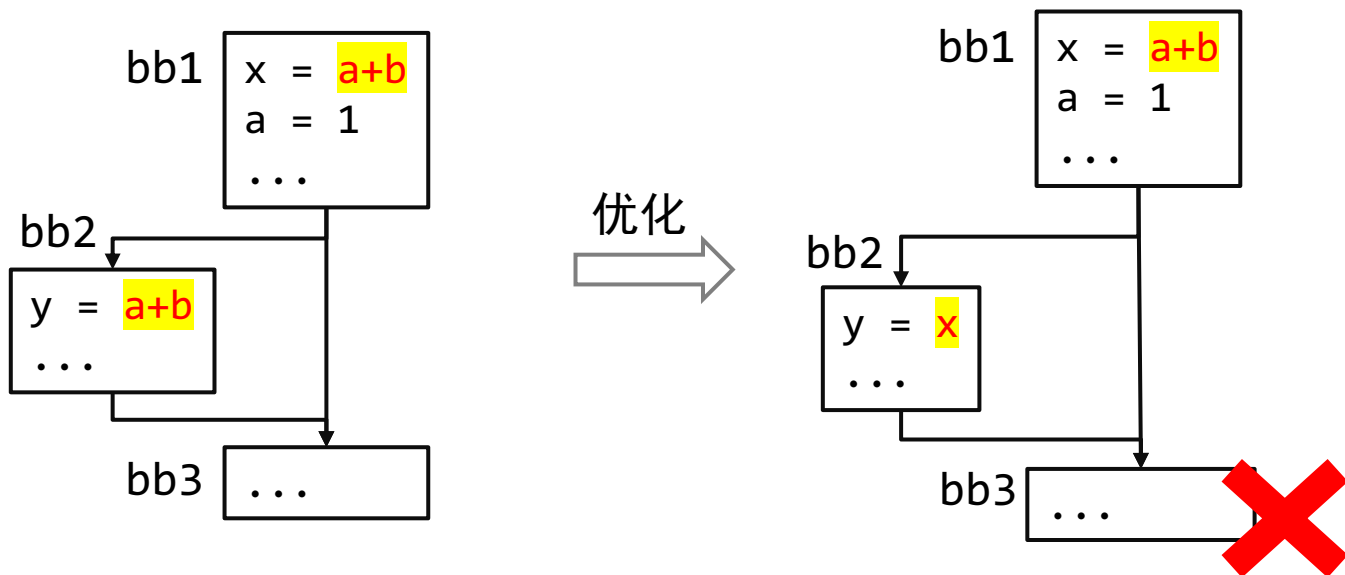
- 不同代码分支中都存在的表达式
- 可以优化代码体积



思考：基于GVN的优化算法实现

- 面向非SSA形式源代码或Gimple IR
- 面向非 SSA形式LLVM IR（使用load/store）
- 面向 SSA形式LLVM IR

公共子表达式分析：面向非SSA形式Gimple IR



• 正向遍历控制流图

- 如遇到指令： $x = a + b$
 - $Gen(n) = \{ \langle a + b \rangle \}$
 - $KILL(n) = \{ \langle \epsilon \rangle : \text{表达式} \epsilon \text{ 包含 } x \}$
- ...

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

三、循环优化

循环中的不变代码

- 出现位置：循环条件、循环体中都可能出现

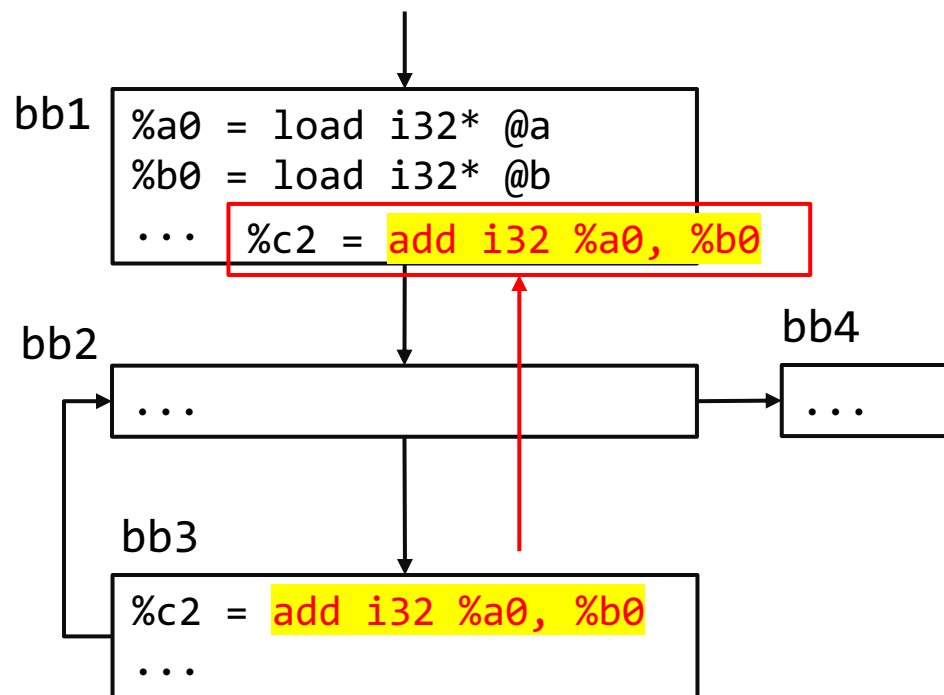
```
let a = ...;
let b = ...;
let s:list = ...;
for i in 1..100 {
  let t = (a + b)*i;
  s.push(t);
}
```

```
let a = ...;
let b = ...;
let s:list = ...;
for i in 1..s.len() {
  let t = (a + b)*i;
  s[i] = t;
}
```

```
let a = ...;
let b = ...;
let s:list = ...;
for i in 1..100 {
  let t = foo();
  s.push(t);
}
```

```
let a = ...;
let b = ...;
let s:list = ...;
for i in 1..s.len() {
  let t = s.pop();
  s[i] = t;
}
```

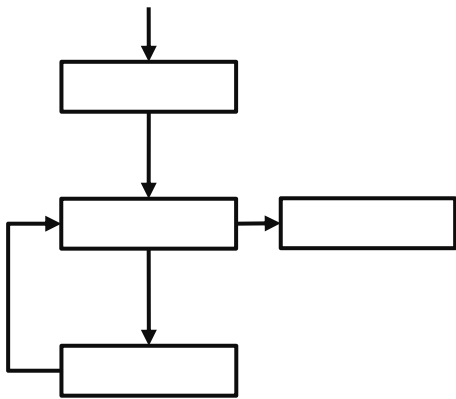
循环不变代码



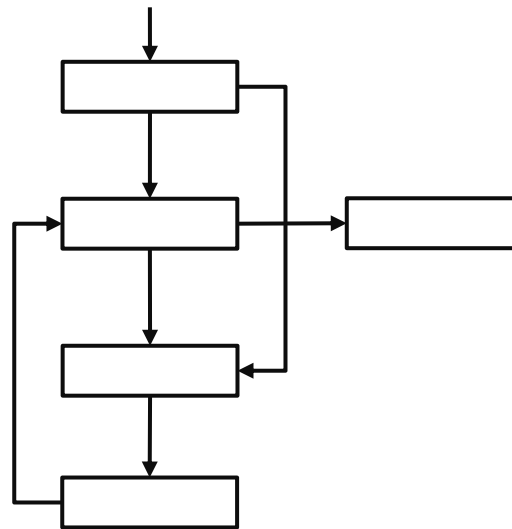
- 检测循环不变代码
 - 操作数定义自循环外部
 - 如何检测循环?
- 前移到循环外部
 - 支配节点

自然循环 (natural loop)

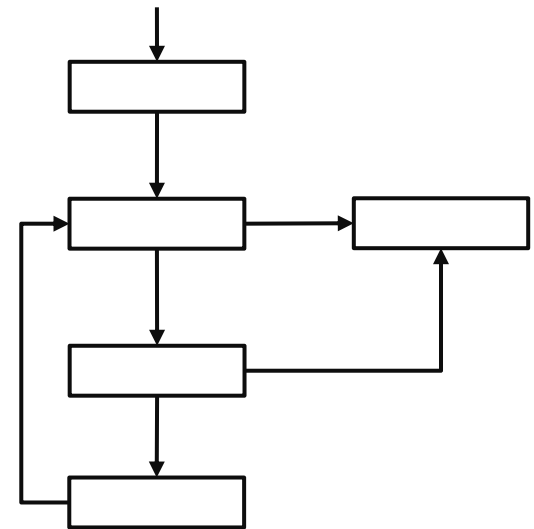
- 一个循环是自然循环的条件：
 - 有唯一的入口（支配所有节点）
 - 返回入口节点的返回边
- 一般正常的控制流语句形成的环：while、if-else、for
 - goto语句会造成非自然循环



自然循环



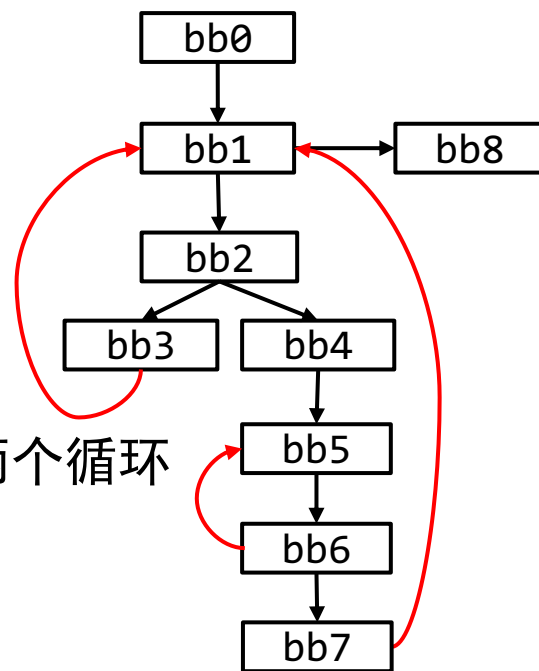
非自然循环



自然循环

自然循环的性质

- 两个自然循环之间的关系：
 - 嵌套：节点集合为包含关系
 - 分离：节点集合没有交集
 - 相切：节点集合存在交集，且交集节点的边同属两个循环
- 两个自然循环之间不可能相交
- 自然循环标识：每条返回边对应一个自然循环



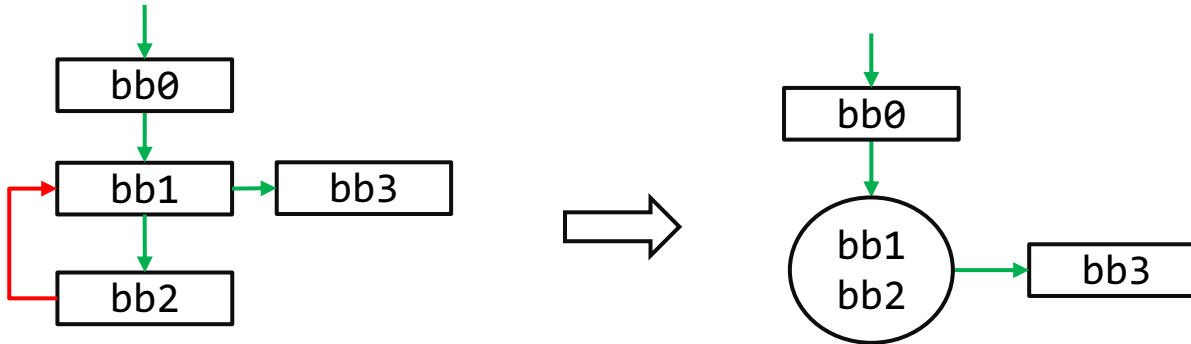
bb3 -> bb1 : 1-2-3



bb6 -> bb5 : 5-6

bb7 -> bb1 : 1-2-4-5-6-7

可规约控制流图：Reducible CFG

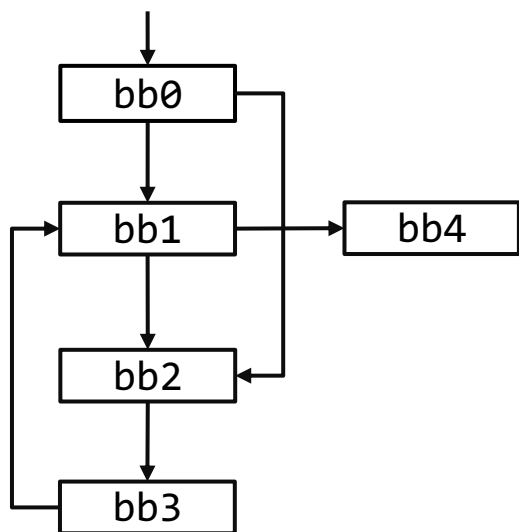
- 可规约CFG的所有循环都是自然循环
- 边可以分为前进边和返回边两个不交集=>可以缩环



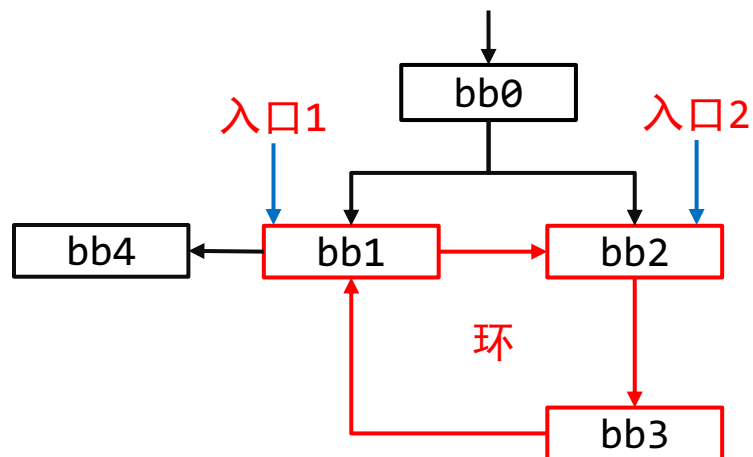
入边： 
出边： 

不可规约控制流图

- 无法确定循环入口和返回边



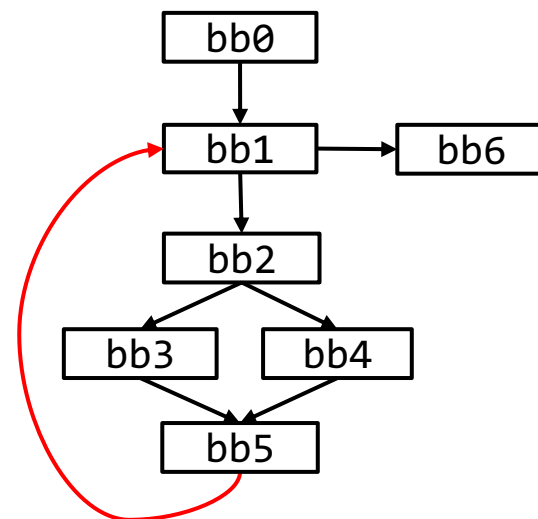
非自然循环



同态图
isomorphic

自然循环检测：基于支配关系

- 1) 遍历CFG=>支配关系矩阵
- 2) 比对图邻接表=>检测返回边
- 3) 识别每一条回边对应的环



	bb0	bb1	bb2	bb3	bb4	bb5	bb6
bb0	1	0	0	0	0	0	0
bb1	1	1	0	0	0	0	0
bb2	1	1	1	0	0	0	0
bb3	1	1	1	1	0	0	0
bb4	1	1	1	0	1	0	0
bb5	1	1	1	0	0	1	0
bb6	1	1	1	0	0	0	1

支配关系矩阵

	bb0	bb1	bb2	bb3	bb4	bb5	bb6
bb0	0	1	0	0	0	0	0
bb1	0	0	1	0	0	0	1
bb2	0	0	0	1	1	0	0
bb3	0	0	0	0	0	1	0
bb4	0	0	0	0	0	1	0
bb5	0	1	0	0	0	0	0
bb6	0	0	0	0	0	0	0

AND

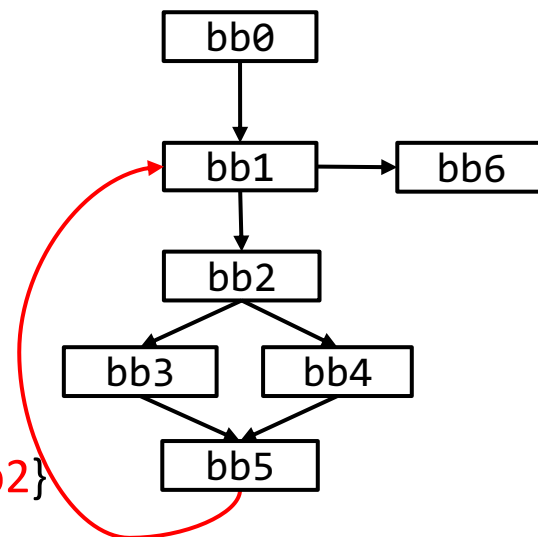
邻接表

bb1支配bb5, 存在边bb5->bb1

自然循环检测：基于支配关系

3) 识别每一条回边对应的环

- 初始化： $S = \{bb1, bb5\}$
- 到达bb5且被bb1支配： $S = \{bb1, bb5, bb3, bb4\}$
- 到达bb3且被bb1支配： $S = \{bb1, bb5, bb3, bb4, bb2\}$



	bb0	bb1	bb2	bb3	bb4	bb5	bb6
bb0	1	0	0	0	0	0	0
bb1	1	1	0	0	0	0	0
bb2	1	1	1	0	0	0	0
bb3	1	1	1	1	0	0	0
bb4	1	1	1	0	1	0	0
bb5	1	1	1	0	0	1	0
bb6	1	1	1	0	0	0	1

支配关系矩阵

	bb0	bb1	bb2	bb3	bb4	bb5	bb6
bb0	0	1	0	0	0	0	0
bb1	0	0	1	0	0	0	1
bb2	0	0	0	1	1	0	0
bb3	0	0	0	0	0	1	0
bb4	0	0	0	0	0	1	0
bb5	0	1	0	0	0	0	0
bb6	0	0	0	0	0	0	0

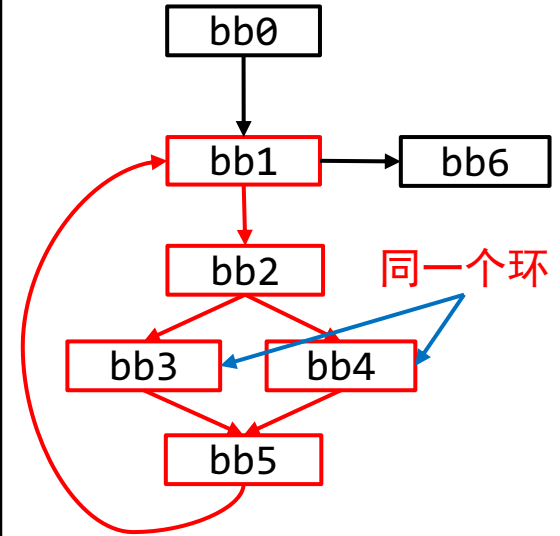
AND

临接表

bb1支配bb5，存在边bb5->bb1

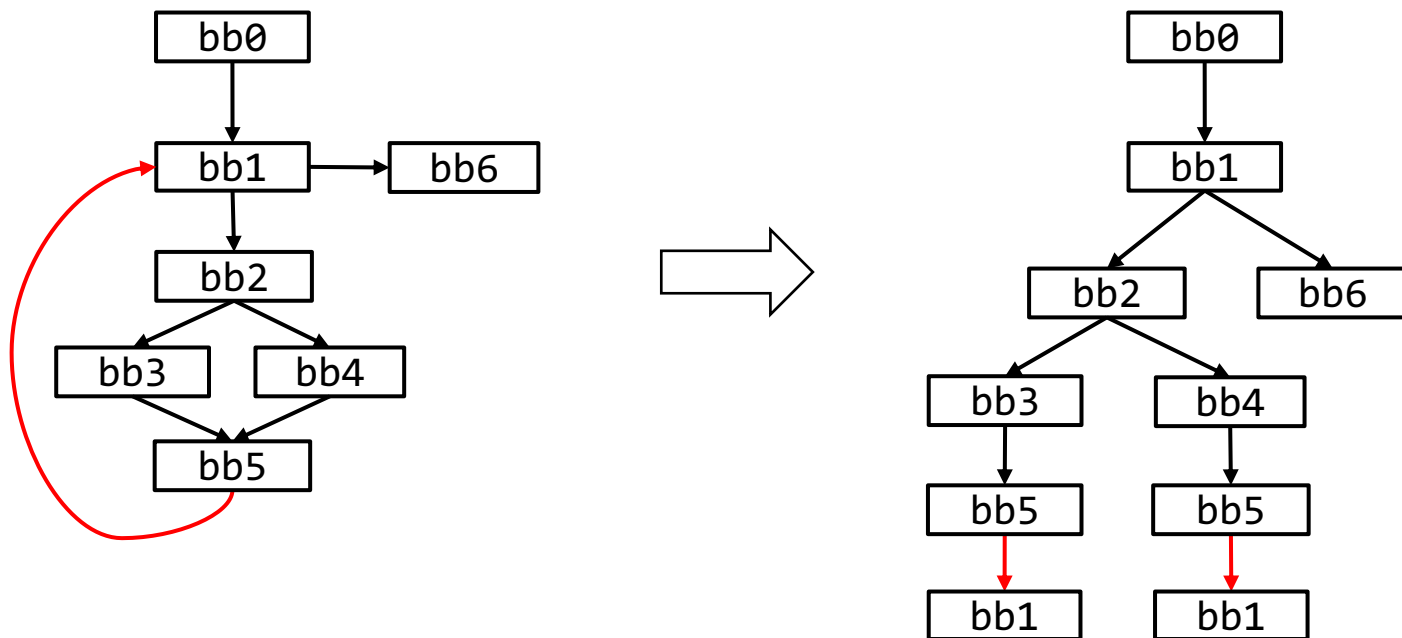
自然循环检测：深度优先搜索

```
stack s;
Visit(v) {
  s.push(v);
  for each w in OUT(v) {
    if s.contains(w) { //找到回边
      AddLoopback(w,v);
    } else {
      Visit(w);
    }
  }
  s.pop()
}
AddLoopback(v,w) {
  new = CreateLoop(top n items of s until w);
  old = Findloop(v, w)
  merge(old,new)
}
```

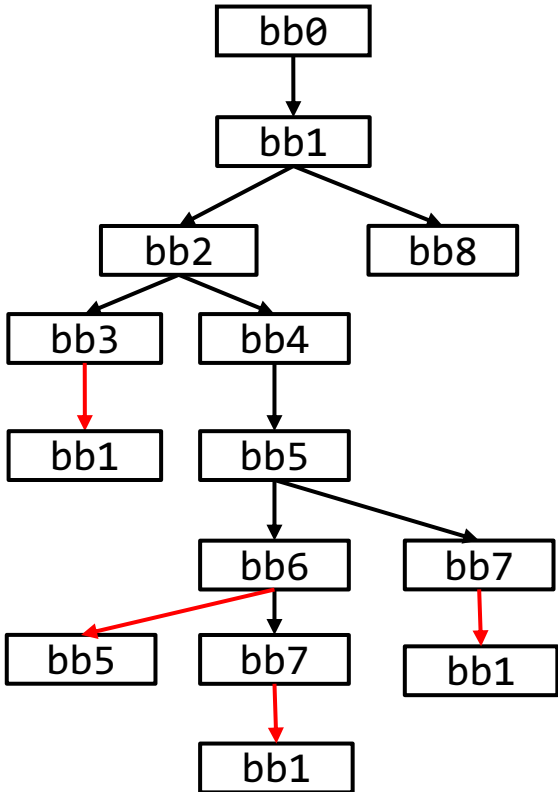
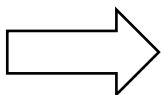
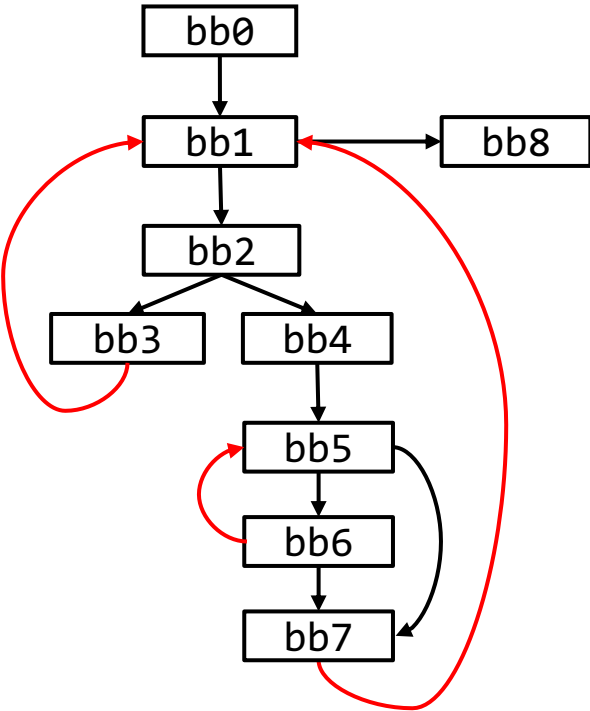


原理分析

- 展开控制流图，直到遇到重复节点为止

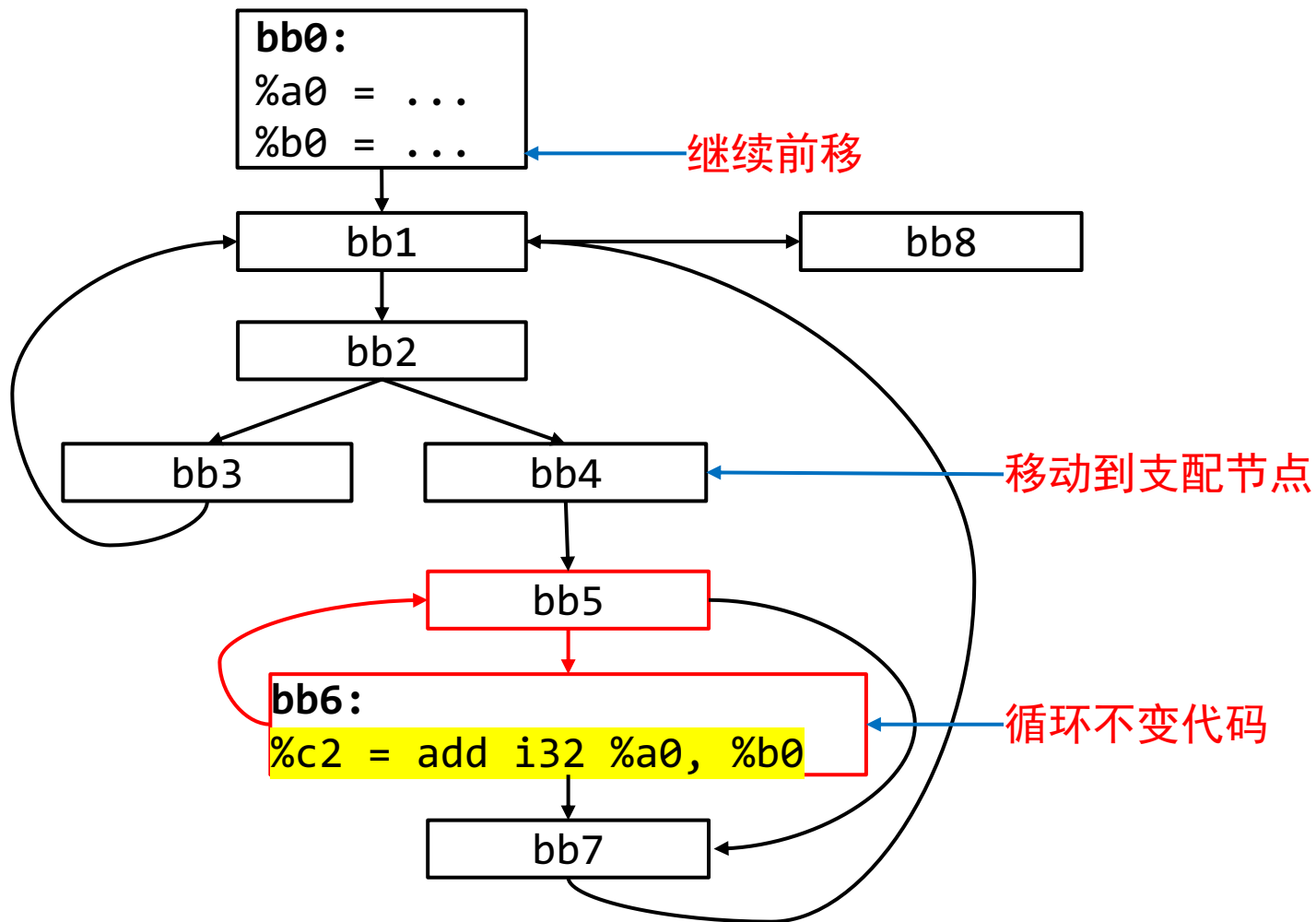


更多案例



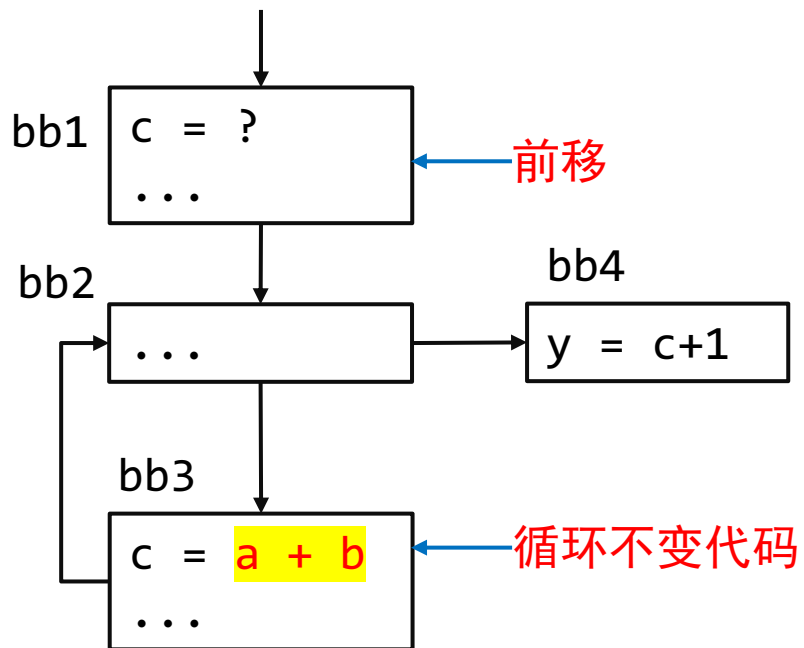
前移位置

- 单层循环：前移到最近的支配节点
- 多层循环：前移至不能移动为止

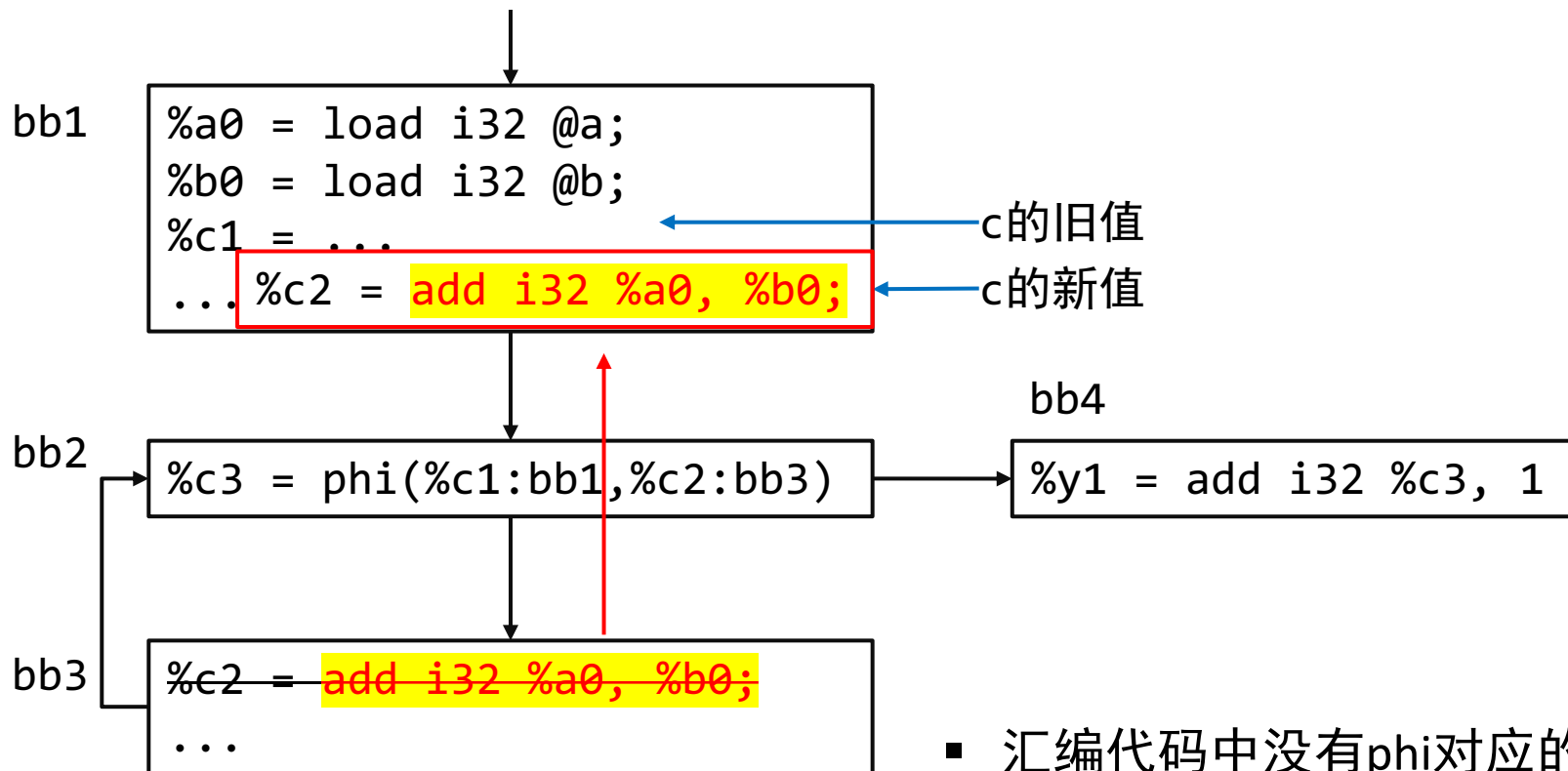


可能会有副作用？

- 如未进入循环，会错误修改x的值



SSA形式会有副作用吗？



- 汇编代码中没有phi对应的指令
- 翻译汇编前用store替代phi

归纳变量

- 变量 x 的值每轮循环增加固定值，则称 x 为归纳变量
 - 基本归纳变量 x
 - 依赖归纳变量 $y = ax + b$ ， a 和 b 为常量

```
for i in 1..100 {  
    y = 10 * i + 1;  
    s[i] = y;  
}
```



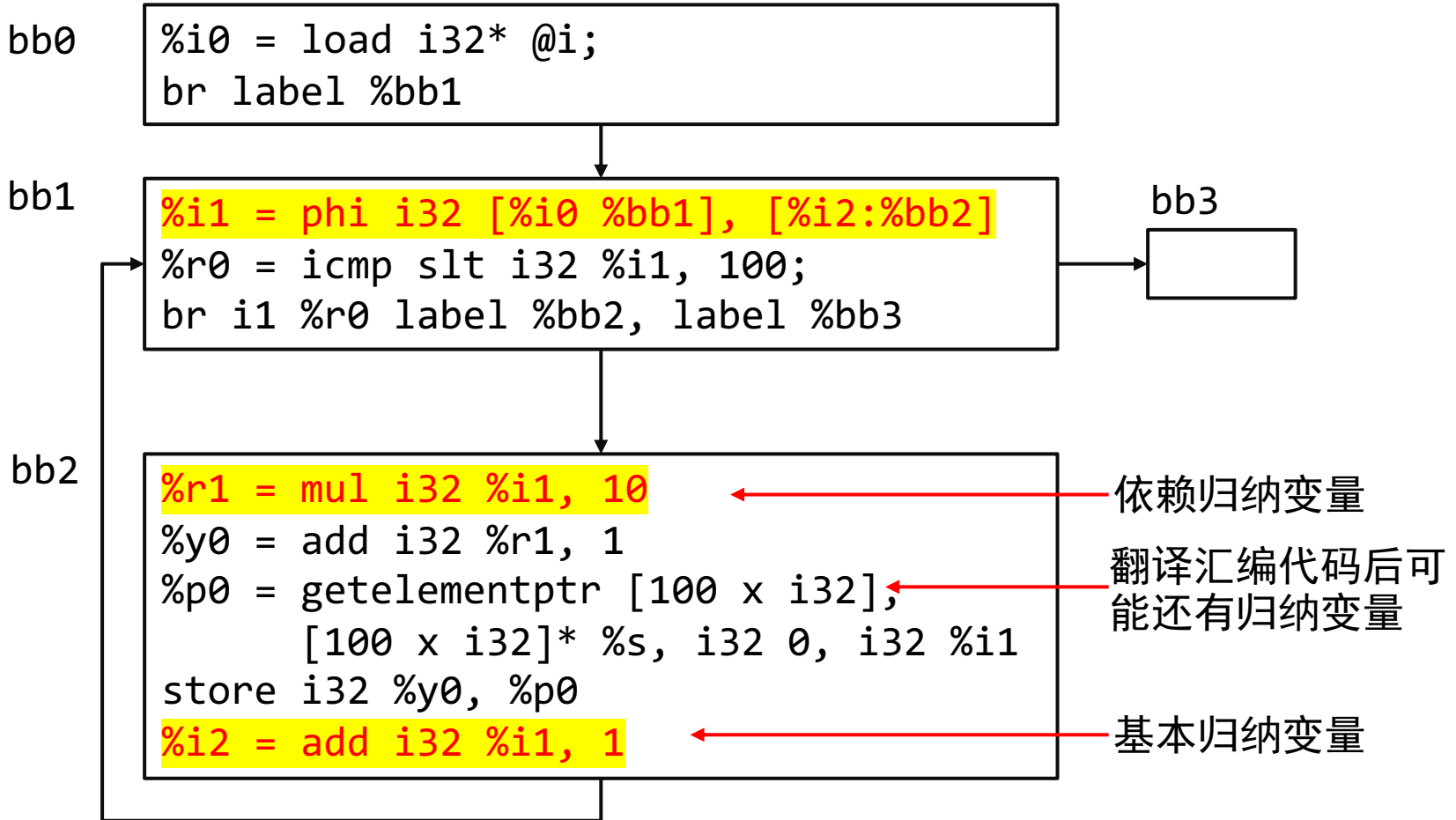
```
t1 = 1;  
for i in 1..100 {  
    t1 = t1 + 10;  
    s[i] = t1;  
}
```

```
let i:i32 = 1;  
while i < 100 {  
    y = 10 * i + 1;  
    s[i] = y;  
    i = i + 1;  
}
```



```
let i:i32 = 1;  
let t1 = 1;  
while i < 100 {  
    y = t1 + 10;  
    s[i] = y;  
    i = i + 1;  
}
```

基于IR识别归纳变量



标量替换：Scalar Replacement

- 使用标量替换循环内部的频繁内存读写操作
- 在IR层自动替换R[i][j]的难点？ R[i][j]和A/B可能是alias

```
for i in 0..rowA {  
  for j in 0..colB {  
    for k in 0..colA {  
      R[i][j] = R[i][j] + A[i][k]*B[k][j];  
    }  
  }  
}
```

每次从内存或cache读写R[i][j]会影响性能



```
for i in 0..rowA {  
  for j in 0..colB {  
    t = R[i][j];  
    for k in 0..colA {  
      t = t + A[i][k]*B[k][j];  
    }  
    R[i][j] = t;  
  }  
}
```

使用临时变量替换，可直接使用寄存器中的值

四、更多优化思路

降低分支预测的代价

- Loop unswitching: 外提（减少）循环内条件判断
- Loop unroll: 将循环体复制多遍

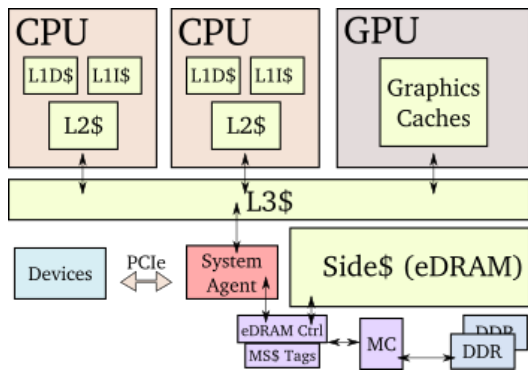
```
void testbrpred(int* a, int len, int x){
    unsigned long long cycle = rdtsc();
    while(len>-1){
        len-=1;
        if(a[len]>x) ;
        else ;
    }
    unsigned long long cycl = rdtsc()- cycle;
    printf("x = %d, cycles = %d\n", x, cycl);
}

int main(int argc, char** argv){
    int a[1000];
    srand(time(NULL));
    for(int i = 0; i < 1000; i++) a[i] = rand()%1000;
    testbrpred(a,1000,100);
    testbrpred(a,1000,300);
    testbrpred(a,1000,500);
    testbrpred(a,1000,700);
    testbrpred(a,1000,900);
}
```

```
x = 100, cycles = 23630
x = 300, cycles = 47175
x = 500, cycles = 63744
x = 700, cycles = 49642
x = 900, cycles = 26301
```

面向访存的优化：Cache

- Cache访问速度优于内存访问速度
- 最小单位是cache line
- 通过降低cache miss提升代码性能



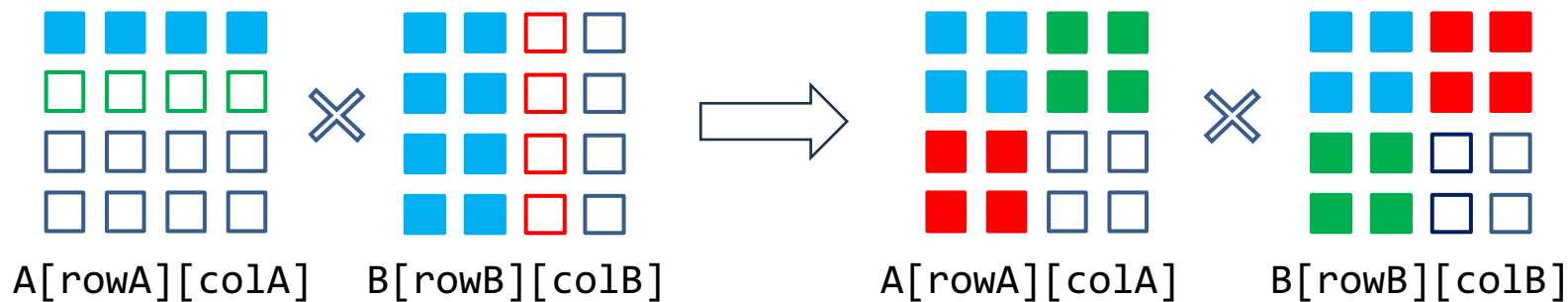
index	valid	tag	data
001	0x...		64 B
002	0x...		64 B
003	0x...		64 B
...	0x...		64 B

cache	size	line	speed
L1	32 KB + 32 KB	64 B	4-5 cycles
L2	256 KB	64 B	12 cycles
L3	up to 2 MB	64 B	30-50 cycles

矩阵乘法：（循环）分块

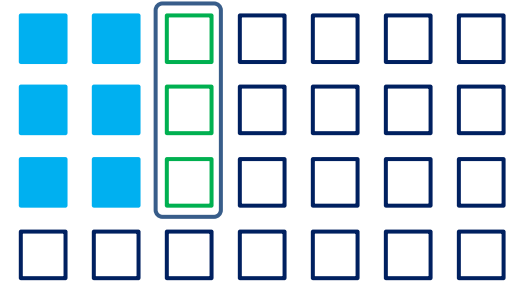
- B矩阵的每一列需要被加载多次：每轮外层循环加载一次
- 如果A矩阵列数足够多，其每一行元素也可能被多次加载

```
for i in 0..rowA {  
  for j in 0..colB {  
    for k in 0..colA {  
      R[i][j] = R[i][j] + A[i][k]*B[k][j];  
    }  
  }  
}
```

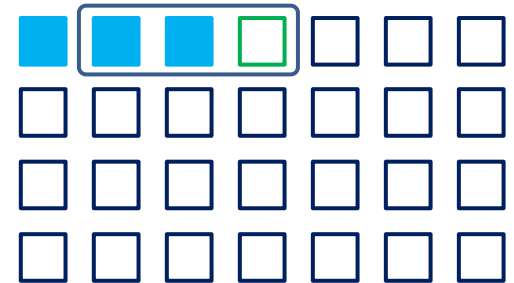


循环交换

```
for i in 1..m-2 {  
  for j in 0..n-1 {  
    R[i][j] = A[i-1][j] + A[i][j] + A[i+1][j];  
  }  
}
```



```
for j in 0..n-1 {  
  for i in 1..m-2 {  
    R[i][j] = A[i-1][j] + A[i][j] + A[i+1][j];  
  }  
}
```



循环合并和拆分

```
for i in 0..n-1 {  
    R1[i] = A[i] + B[i];  
}  
for i in 0..n-1 {  
    R2[i] = A[i] + B[i];  
}
```

合并

```
for i in 0..n-1 {  
    R1[i] = A[i] + B[i];  
    R2[i] = A[i] + B[i];  
}
```

```
for i in 0..n-1 {  
    R1[i] = A[i] + B[i];  
    R2[i] = C[i] + D[i];  
}
```

拆分

```
for i in 0..n-1 {  
    R1[i] = A[i] + B[i];  
}  
for i in 0..n-1 {  
    R2[i] = C[i] + D[i];  
}
```

可能对寄存器分配有利：减少冲突关系

练习

- 找出LLVM的过程内优化功能并测试分析其效果
 - 链接: <https://llvm.org/docs/Passes.html>

- Transform Passes
 - -adce: Aggressive Dead Code Elimination
 - -always-inline: Inliner for `always_inline` functions
 - -argpromotion: Promote 'by reference' arguments to scalars
 - -bb-vectorize: Basic-Block Vectorization
 - -block-placement: Profile Guided Basic Block Placement
 - -break-crit-edges: Break critical edges in CFG
 - -codegenprepare: Optimize for code generation
 - -constmerge: Merge Duplicate Global Constants
 - -dce: Dead Code Elimination

```
#: opt -dce -S in.ll -o out.ll
```