

Lecture 5.3

高级代码优化技术

徐 辉

xuh@fudan.edu.cn



大纲

- 一、程序分析难题
- 二、指针分析
- 三、多版本和自适应优化
- 四、并行优化

程序分析难题

- 一般来讲，程序属性（program properties）是不可计算的（undecidability）
 - 如常量分析等数据流分析问题
- 一个算法可以无法同时做到
 - 可靠（sound）：不会漏报
 - 完备（complete）：不会误报
 - 结束（terminate）：不因某些输入导致算法无法终止
- 如何设计程序分析算法是一门艺术

莱斯定理: Rice's Theorem

CLASSES OF RECURSIVELY ENUMERABLE SETS
AND THEIR DECISION PROBLEMS⁽¹⁾

BY
H. G. RICE

*“Any nontrivial property about the
Language recognized by a Turing
Machine is undecidable.”*

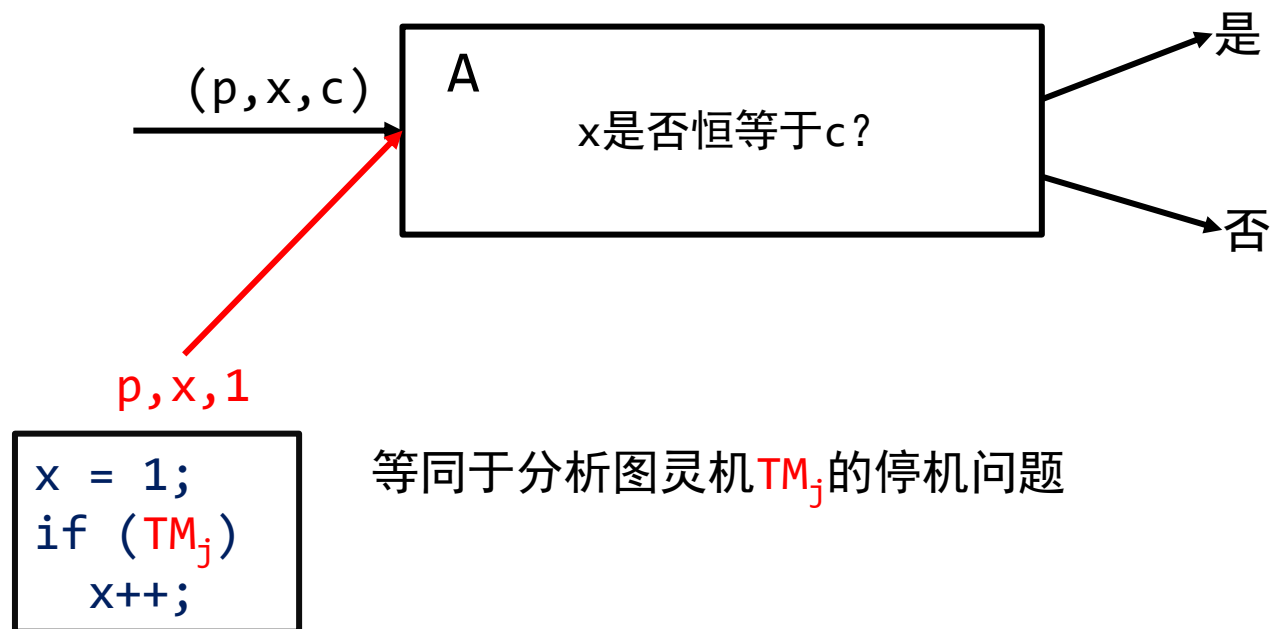
- Henry Gordon Rice, 1953

回顾Chomsky Hierarchy

| Class | Languages | Automaton | Rules | Word Problem | Example |
|--------|------------------------|--------------------|---|-----------------|-------------------------|
| type-0 | recursively enumerable | Turing machine | no restriction | undecidable | Post's corresp. problem |
| type-1 | context sensitive | linear-bounded TM | $\alpha \rightarrow \gamma$ $ \alpha \leq \gamma $ | PSPACE-complete | $a^n b^n c^n$ |
| type-2 | context free | pushdown automaton | $A \rightarrow \gamma$ | cubic | $a^n b^n$ |
| type-3 | regular | NFA / DFA | $A \rightarrow a$ or $A \rightarrow aB$ | linear time | $a^* b^*$ |

证明方法1：规约到图灵机停机问题

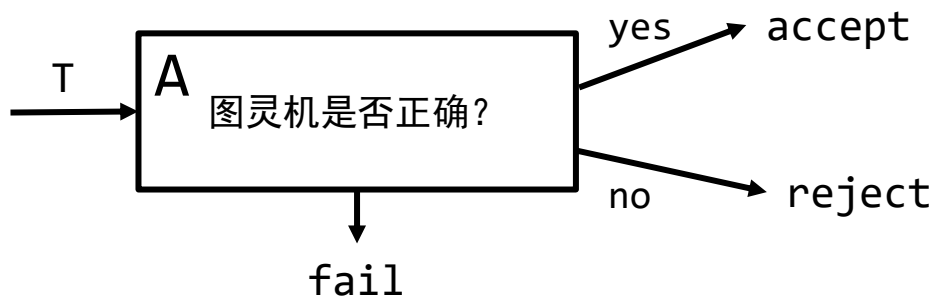
- 假设算法A可以分析任意程序p中的变量x的值是否为常量c
- 设计一个程序p证明算法A不存在？



证明方法2: By Contradiction

- 假设算法A可以分析确定型图灵机T是否正确，输出accept或reject，如果无法分析则fail。

- 正确: 1
- 错误: 0
- 失败: -1



- 证明思路: 将A输入A
 - 构造矛盾: A', 如果A输出1, 则A'输出0 (手动构造)
 - 将A'输入A进行分析

程序分析难题

- 指针分析
- 并发分析
- ...

大纲

- 一、程序分析难题
- 二、指针分析
- 三、多版本优化
- 四、并行优化

指针分析问题

- 指针别名 (Pointer alias) 分析: 判断两个指针是否在某时刻指向同一内存地址。
- 指针指向 (Points-to) 分析: 分析指针指向的内存地址, 分析结果可用于指针别名分析。

//C语言代码

```
int a = 0;  
int* p = &a;  
*p = 1;  
int c = a + 1;
```

//C语言代码

```
int a = 0;  
int b = 1;  
int* p = &b;  
p = p + 1;  
*p = 1;  
int c = a + 1;
```

可能造成Alias的操作

- 指针
- 函数调用传参（指针、引用）
- 数组索引

//C语言代码

```
int *p, i;  
p = &i;
```

//C语言代码

```
void foo(struct Node* a, struct Node* a);  
foo(x,x); // a和b在函数foo中是alias
```

//C语言代码

```
int i,j,a[100];  
i = j; // a[i] and a[j] alias
```

别名确定性

- May alias: 可能存在别名关系
- Must alias: 一定存在别名关系。
- Not may alias \Rightarrow must not alias
- May alias和Must alias有哪些应用场景?

指针分析的用途

- 编译器优化（保守策略）
 - 常量传播（constant propagation）
 - 下列代码中a是常量吗？
 - 如果*p和a一定不是alias，则是
 - 如果*p和a一定是alias，则否
 - 如果*p和a可能是alias，则否
 - 适合May alias 分析
 - 死代码删除中的条件语句判定
 - ...

```
//C语言代码  
int a = 0;  
int b = 1;  
int* p = &b;  
p = p + 1;  
*p = 1;  
int c = a + 1;
```

为什么指针分析很难

//一段无意义C语言代码

```
struct Node {
```

```
    int data;
```

```
    Node* next, prev;
```

```
}
```

```
foo(Node* in1, Node* in2){
```

```
    Node* new = new Node();
```

```
    if ((l1||l2||l3)&&(l1||l2||l3)) {
```

```
        new.next = ...;
```

```
        new.prev = ...;
```

```
    } else {
```

```
        new.next = ...;
```

```
        new.prev = ...;
```

```
    }
```

```
    new = new + 1;
```

```
    if (...) {
```

```
        foo(in2, new);
```

```
    }
```

```
    ...}
```

多级指针、数据结构建模

跨函数分析

3SAT条件

裸指针运算

递归调用

指针分析方法

- Flow-sensitivity: 是否考虑代码执行顺序
 - Flow sensitive: 计算每一个程序点的指针指向
 - Flow insensitive: 计算任意程序点可能的指向
- Path-sensitivity: 是否考虑控制流
 - Path sensitive: 分析过程只考虑单条特定控制流
 - Path insensitive: 分析过程不区分控制流
- Context-sensitivity: 是否考虑函数调用
 - Context sensitive: 支持跨函数调用分析
 - Context insensitive: 以函数为分析边界

指针分析算法

- Andersen-style Analyses
- Steensgaard-style Analyses

指针分析表示方法

- 别名对: alias pairs
 - 如 $*p$ 和 $*q$ 、 x 和 $*p$ 、 x 和 $*q$
- 等价集合: equivalence sets
 - 如 $\{*p, x, *q\}$
- 指针指向: point-to
 - $p \rightarrow x, q \rightarrow x$

```
int x;  
p = &x;  
q = p;
```

Andersen-style指针分析思路

- 将指针赋值视作子集约束
- 通过约束表示和传递指针指向信息
- Flow-insensitive
 - 不考虑语句顺序
- Context-insensitive
 - 与函数如何被调用无关
- 主要步骤
 - 1) 将指针指向关系映射为子集约束；
 - 2) 初始化约束图；
 - 3) 计算传递闭包更新约束关系。

提取约束关系

| 约束类型 | 赋值语句 | 约束 | 含义 |
|---------|-----------|---------------------|---|
| Base | $a = \&b$ | $a \supseteq \{b\}$ | $loc(b) \in pts(a)$ |
| Simple | $a = b$ | $a \supseteq b$ | $pts(a) \supseteq pts(b)$ |
| Complex | $a = *b$ | $a \supseteq^* b$ | $\forall v \in pts(b), pts(a) \supseteq pts(v)$ |
| Complex | $*a = b$ | $*a \supseteq b$ | $\forall v \in pts(a), pts(v) \supseteq pts(b)$ |

```
p = &a  
q = &b  
*p = q;  
r = &c;  
s = p;  
t = *p;  
*s = r;
```

```
p  $\supseteq$  {a}  
q  $\supseteq$  {b}  
*p  $\supseteq$  q  
r  $\supseteq$  {c}  
s  $\supseteq$  p  
t  $\supseteq$  *p  
*s  $\supseteq$  r
```

初始化约束图

- 约束图

- 点表示变量的指针指向;
- 边表示特定约束关系;

- 初始化

- 包含关系: 箭头
- 指针指向: {}

| 赋值语句 | 约束 | 含义 | 边 |
|-----------|---------------------|---|-------------------|
| $a = \&b$ | $a \supseteq \{b\}$ | $loc(b) \in pts(a)$ | no edge |
| $a = b$ | $a \supseteq b$ | $pts(a) \supseteq pts(b)$ | $b \rightarrow a$ |
| $a = *b$ | $a \supseteq *b$ | $\forall v \in pts(b), pts(a) \supseteq pts(v)$ | no edge |
| $*a = b$ | $*a \supseteq b$ | $\forall v \in pts(a), pts(v) \supseteq pts(b)$ | no edge |

$p \supseteq \{a\}$

$q \supseteq \{b\}$

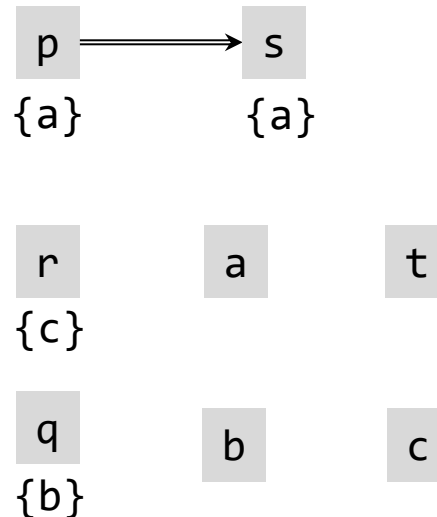
$*p \supseteq q$

$r \supseteq \{c\}$

$s \supseteq p$

$t \supseteq *p$

$*s \supseteq r$



更新约束关系：Worklist算法

假设约束图已经初始化

Let $W = \{ v \mid \text{pts}(v) \neq \emptyset \}$ (所有指向集合非空的节点)

While W not empty

$v \leftarrow \text{select from } W$

 for each $a \in \text{pts}(v)$ do

 for each constraint $p \supseteq *v$

 add edge $a \rightarrow p$, and add a to W if edge is new

 for each constraint $*v \supseteq q$

 add edge $q \rightarrow a$, and add q to W if edge is new

 for each edge $v \rightarrow q$ do

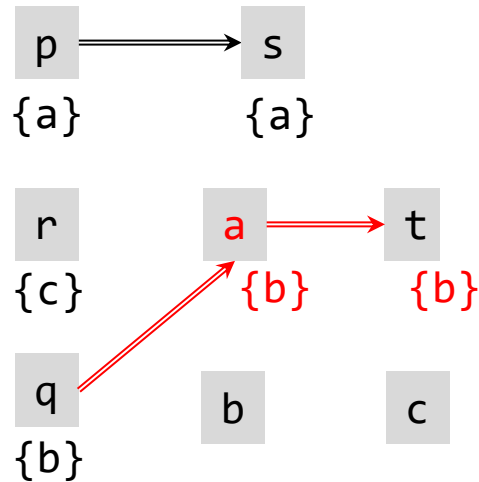
$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(v)$, and add q to W if $\text{pts}(q)$ changed

更新约束关系

```

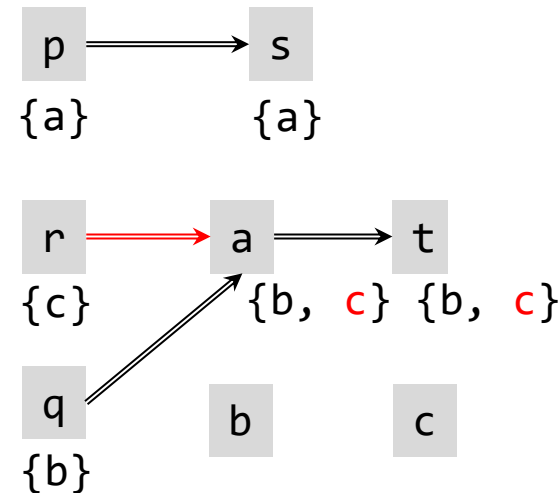
p ⊇ {a}      for each a ∈ pts(v) do
q ⊇ {b}      for each constraint p ⊇ *v
               add edge a→p, and add a to W if edge is new
*p ⊇ q       for each constraint *v ⊇ q
               add edge q→a, and add q to W if edge is new
r ⊇ {c}      for each edge v→q do
s ⊇ p         pts(q) = pts(q) ∪ pts(v), and add q to W if pts(q) changed
t ⊇ *p
*s ⊇ r
    
```

Step 1: Worklist: {**p**, s, r, q}



Result Worklist: {~~p~~, s, r, q, **a**, **t**}

Step 2: Worklist: {**s**, r, q, a, t}



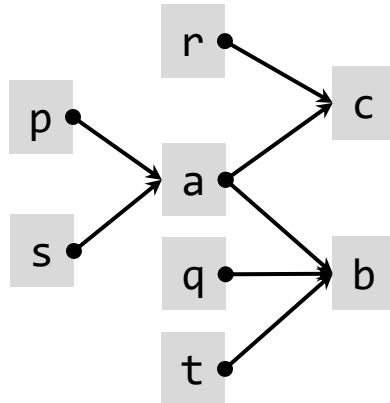
Result Worklist: {~~s~~, r, q, a, t}

精确性和算法复杂度

- 分析粒度较粗，相比flow sensitive存在误报。
 - *t和c不应为alias
- 复杂度 $O(n^3)$ ， n 是约束图的节点数。

```
p = &a  
q = &b  
*p = q;  
r = &c;  
s = p;  
t = *p;  
*s = r;
```

Flow Sensitive
分析结果



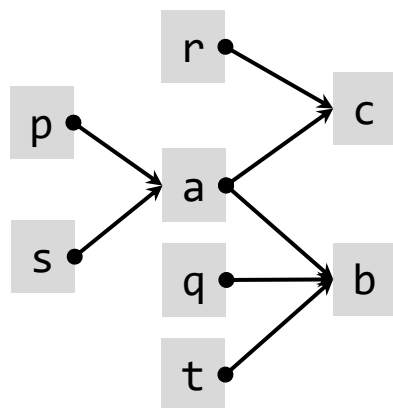
Andersen算法结果

```
pts(p) = {a}  
pts(q) = {b}  
pts(r) = {c}  
pts(s) = {a}  
pts(t) = {b, c}  
pts(a) = {b, c}  
pts(b) = ∅  
pts(c) = ∅
```

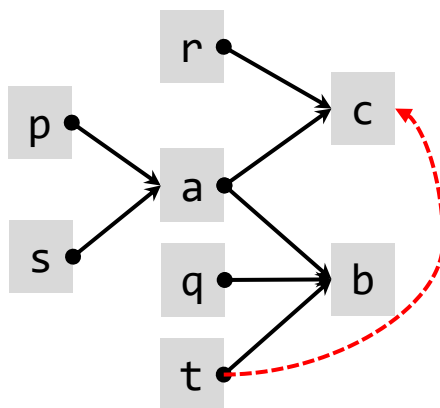
如何进一步优化性能？

- Andersen-style: 每个变量可以是任意变量的alias
 - $O(n^2)$ 空间
- 降低每个变量使用的空间（边数）？可以牺牲一些精度。
- 只有一条出边：将每个变量关联到一个abstract location
 - 如果 $*x$ 和 $*y$ 如果是alias, x 和 y 指向同一个abstract location
 - 接近线性复杂度 $O(n * \alpha(n))$

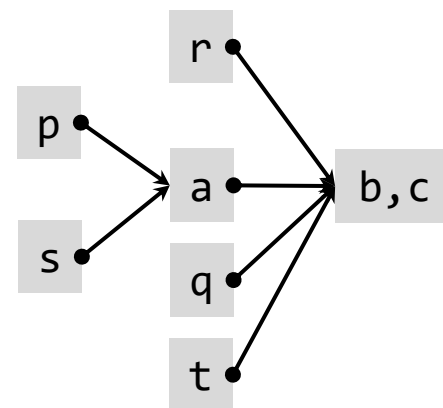
Flow Sensitive
分析结果



Andersen-style
分析结果



Steensgaard-style
分析结果



精度损失: $a = \&b$, $a = \&c$, b 和 c 不应是alias

Steensgaard-Style分析思路

- 使用等价约束（equality constraints）而非子集约束；
- 基于并查集的方法，如果 $x=y$ ，则 x 和 y 联通
- 接近线性复杂度 $O(n * \alpha(n))$ ，粒度比Andersen-style更粗

| 约束类型 | 赋值语句 | 约束 | 含义 | 注释 |
|---------|-----------|---------------------|---|-------------|
| Base | $a = \&b$ | $a \subseteq \{b\}$ | $loc(b) \subseteq pts(a)$ | Steensgaard |
| | | $a = \{b\}$ | $loc(b) = pts(a)$ | 简化版（便于理解） |
| Simple | $a = b$ | $a = b$ | $pts(a) = pts(b)$ | |
| Complex | $a = *b$ | $a = * b$ | $\forall v \in pts(b), pts(a) = pts(v)$ | |
| Complex | $*a = b$ | $* a = b$ | $\forall v \in pts(a), pts(v) = pts(b)$ | |

简化版的问题： $a = \&c$ ， $b = \&c$ ， a 和 b 不应是alias

并查集算法（简化版）

- 维护不存在相交关系的集合，支持查找和联合两种操作
 - Find(x): 返回包含变量x的集合
 - Union(x, y): 联合包含x和y的两个集合

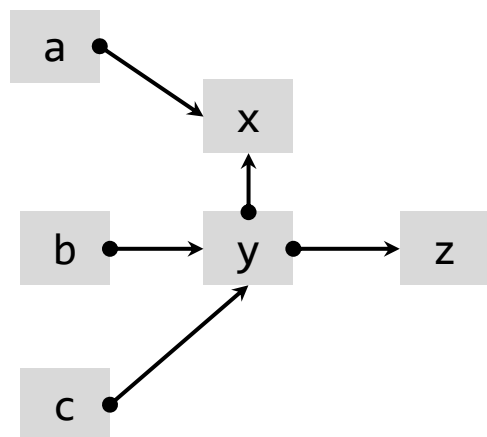
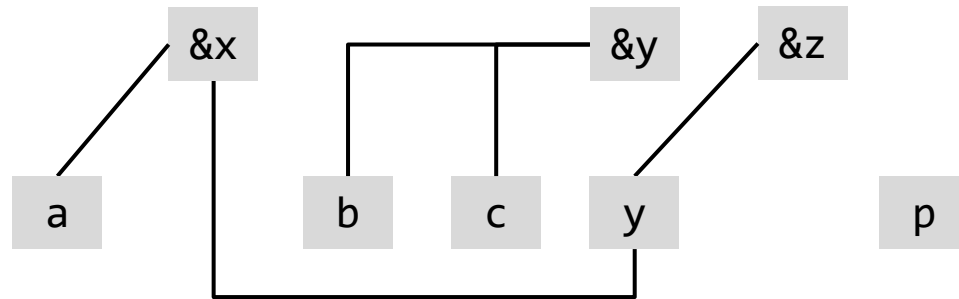
```
while(getPair()!=NULL){  
    [p,q] = readPair(p,q);  
    pset = find(p);  
    qset = find(q);  
    if(pset == qset)  
        continue;  
    else union(p,q);  
}
```

示例

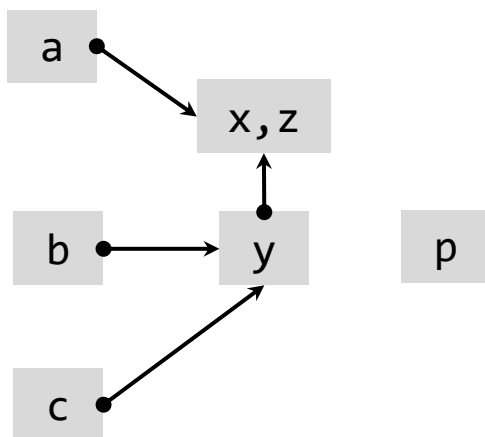
```
a = &x  
b = &y;  
if p  
    y = &z;  
else  
    y = &x;  
c = &y;
```



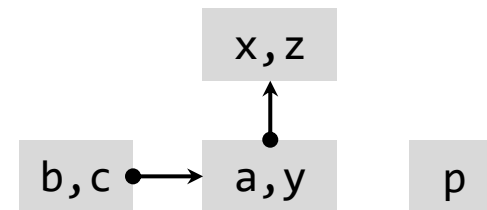
```
a, &x  
b, &y;  
p  
y, &z;  
y, &x;  
c, &y;
```



Andersen



Steensgaard



纯并查集（简化版）

Andersen vs Steensgaard

- 都是flow-insensitive、context-insensitive
- 不同点在于points-to集合的构造思路
 - Andersen-style:
 - 基于子集关系的
 - 每个节点对应一个变量
 - 每个节点有多条出边
 - 比较精准但效率不高
 - Steensgaard-style:
 - 基于等价关系的
 - 每个节点对应多个变量
 - 每个节点只有一条出边
 - 比较快但精准度有限

Flow-insensitive分析的缺陷

- 相对流敏感算法分析结果不够精准，可用性较差
 - 代码优化
 - 漏洞检测
- 如何进行流敏感、路径敏感的指针分析？
 - 维护每个program point的alias关系？

```
int a = 1;
int b = 2;
int c = 3;

int x = &a;
doSth();
x = &b;
if(x == &c) //恒为假
    doSth();
if(x == &a) //恒为假
    doSth(x)
```

```
int* a = malloc(10);
int* b = malloc(10);

int* x = a;
doSth();
x = b;
free(a) //a和x不是alias
free(x)
```

回到常量分析问题

- 下列哪个程序会使基于lattice的常量检测算法失准？

```
x = 1;  
if (x==2)  
    x = 3;  
x是否为常量?
```

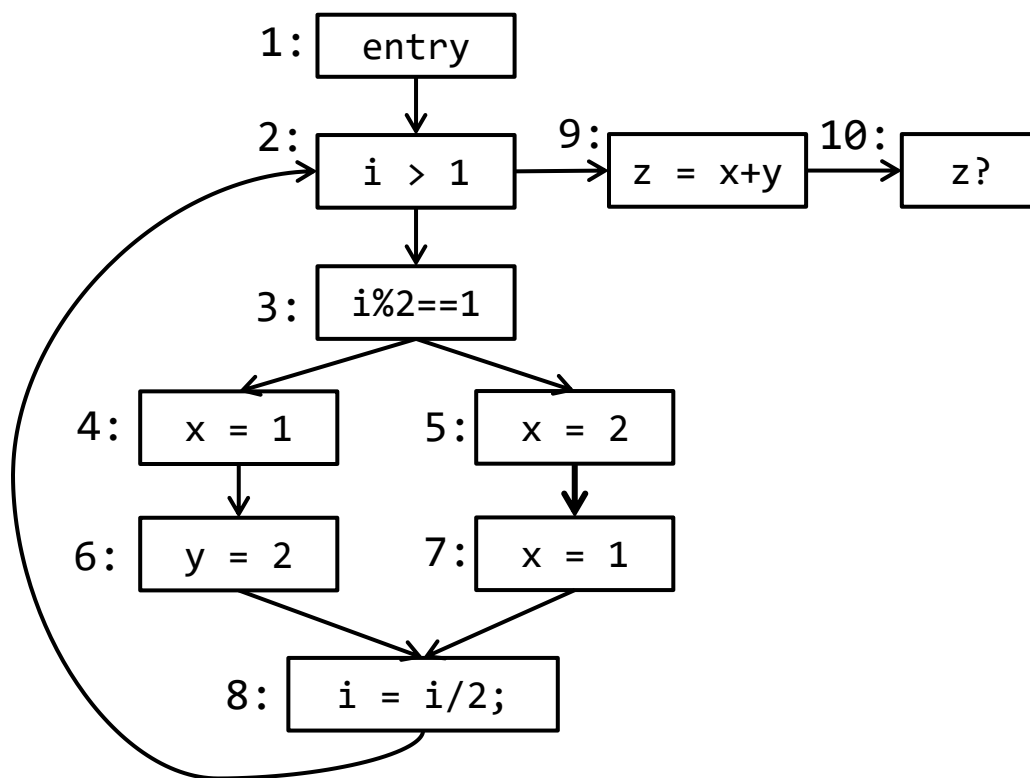
```
if (p)  
    x = 1;  
else  
    x = -1;  
y = x*x;  
y是否为常量?
```

```
if (p)  
    x = 1;  
else  
    x = 2;  
if (p)  
    y = x+2;  
else  
    y = x+1;  
y是否为常量?
```

Meet/Join Over All Passes

- 一个program point的属性信息是由每一条到达该点的路径分别计算得到的。
- 主要挑战：循环导致路径数是无限的

```
while (i>1) {  
  if (i%2==1){  
    x = 1;  
    y = 2;  
  } else {  
    x = 2;  
    y = 1;  
  }  
  i = i/2;  
}  
z = x+y;  
z是否为常量?
```



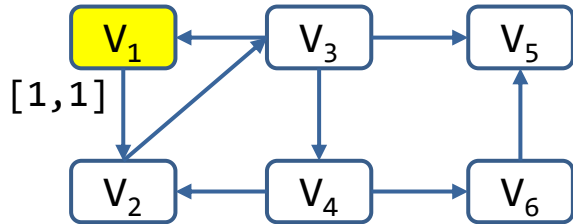
是否可以缩环？

- 是否可以基于强联通分量缩环？
 - 检测强联通分量：Tarjan算法

```
DFSVisit(v) {
    N[v] = c; //记录每个节点的到达时间
    L[v] = c; //记录下一跳的最早到达时间
    c++;
    push v onto the stack;
    for each w in OUT(v) {
        if N[w] == UNDEFINED {
            DFSVisit(w);
            L[v] = min(L[v], L[w]);
        } else if w is on the stack {
            L[v] = min(L[v], N[w]);
        }
    }
    if L[v] == N[v] { //找到强联通分量
        pop vertices off stack down to v;
    }
}
```

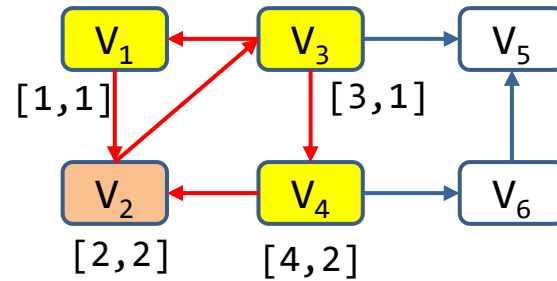

Tarjan算法找SCC: DFS

Stack:

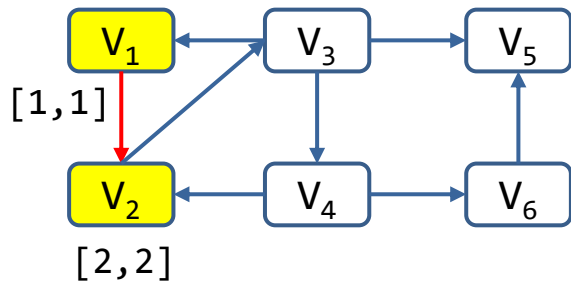


V1[1,1]

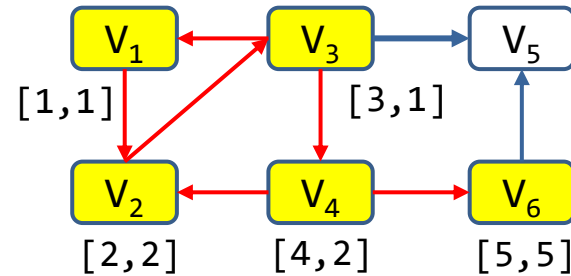
Stack:



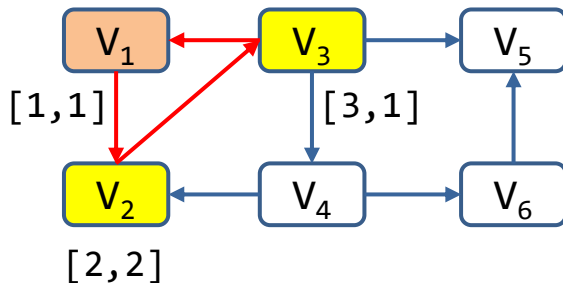
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]



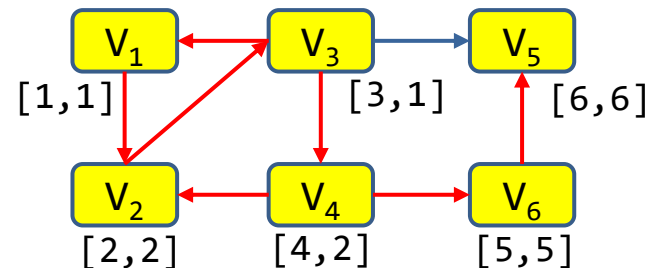
V2[2,2]
V1[1,1]



V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

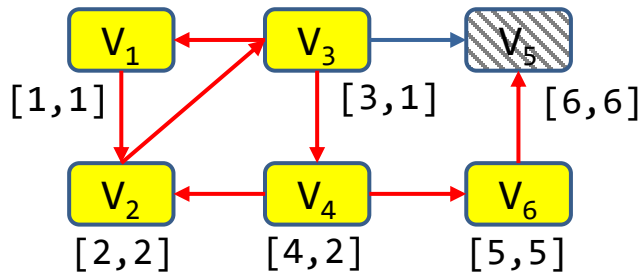


V3[3,1]
V2[2,2]
V1[1,1]



V5[6,6]
V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

通过Tarjan算法找SCC

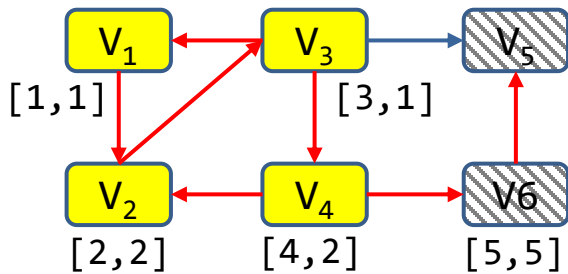


Stack:

$V_5[6, 6]$
 $V_6[5, 5]$
 $V_4[4, 2]$
 $V_3[3, 1]$
 $V_2[2, 2]$
 $V_1[1, 1]$

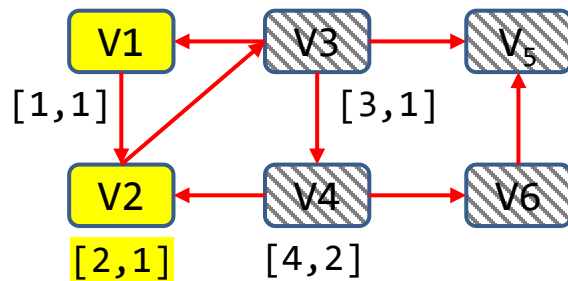
SCC:

{V5}



$V_6[5, 5]$
 $V_4[4, 2]$
 $V_3[3, 1]$
 $V_2[2, 2]$
 $V_1[1, 1]$

{V5}
 {V6}



$\min(L[v], L[w]);$

$V_2[2, 1]$
 $V_1[1, 1]$

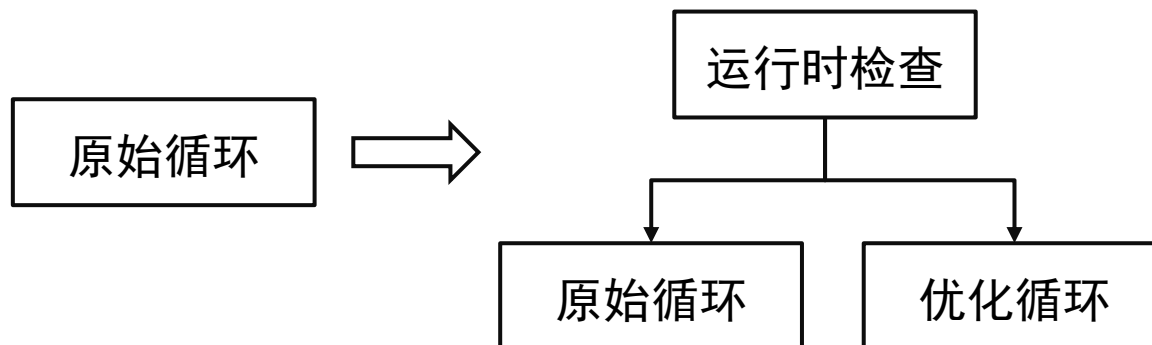
{V5}
 {V6}
 {4, 3, 2, 1}

大纲

- 一、程序分析难题
- 二、指针分析
- 三、多版本优化
- 四、并行优化

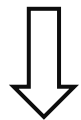
循环多版本

- 静态分析的局限性：如指针分析
- 乐观式优化：假设某些有利条件成立
- 进入循环前检查



举例

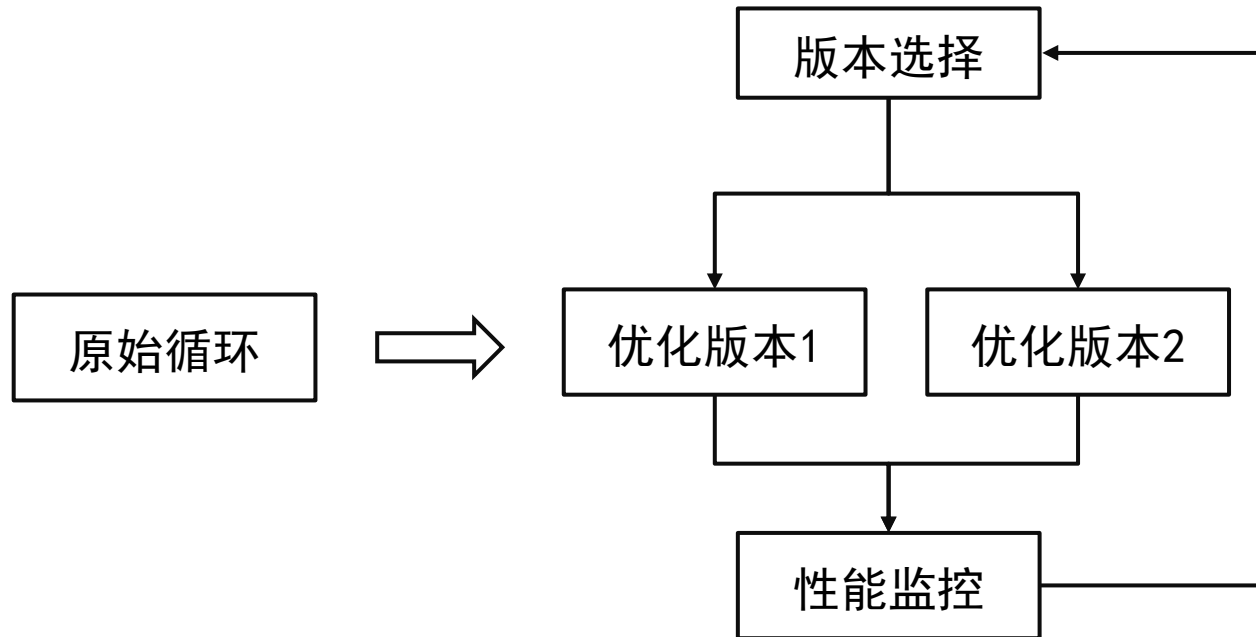
```
fn foo(a:int*, b:int*){  
    for(int i = 1; i < n; i++) {  
        for(int j = 1; j < n; j++) {  
            a[i] = a[i] + b[j];  
        }  
    }  
}
```



可以标量优化的前提：a和b不是alias

```
fn foo(a:int*, b:int*){  
    for(int i = 1; i < n; i++) {  
        t = a[i];  
        for(int j = 1; j < n; j++) {  
            t = t + b[j];  
        }  
        a[i] = t;  
    }  
}
```

自适应优化: Adaptive Optimization



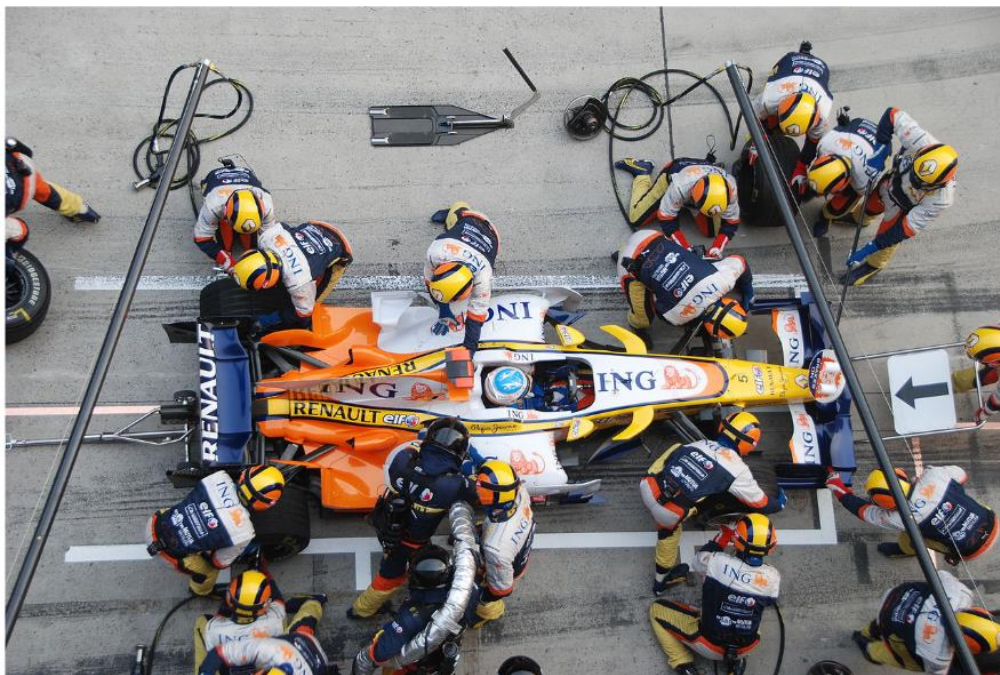
JIT Bailout

- JIT可以做一些激进优化并翻译为Assembly code
 - Javascript: 类型假设
 - JVM
- Bailout: 优化错误则自动回退到解释执行

```
//javascript代码
function sum(a) {
  var sum = 0;
  for (var i = 0; i < a.length; i++) {
    sum = sum + a[i];
  }
}
```

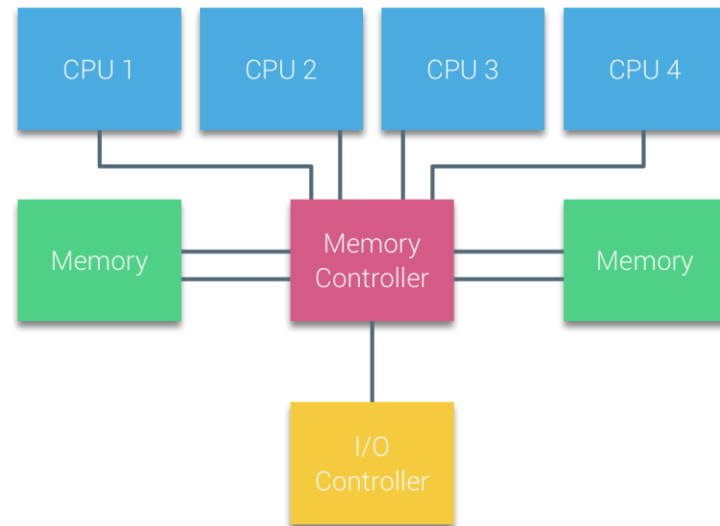
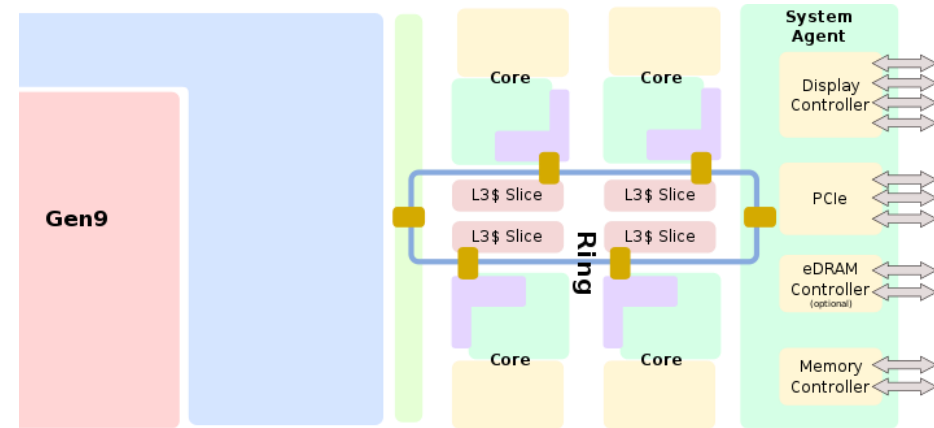
大纲

- 一、程序分析难题
- 二、指针分析
- 三、多版本优化
- 四、并行优化



并行计算

- 并行计算架构
 - 多核处理器 (multicore)
 - 多线程并行计算
 - 多CPU (multiprocessor)
 - UMA: cache coherence
 - NUMA
 - 分布式系统
- 关键问题:
 - 任务分解: 数据分块
 - 数据更新同步



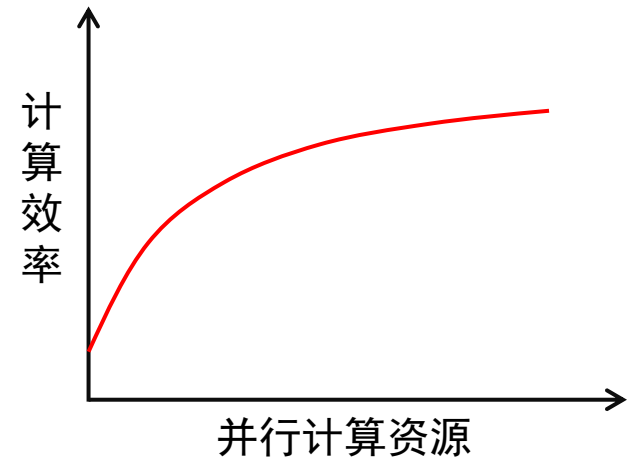
并行性能提升上线：Amdahl's Law

- 如果单线程运行一个任务需要 n 小时
- 其中可以并行的部分耗时 m
- 如果增加线程数至 s ，则所需时间为： $(n-m)+m/s$

- 效率提升：
$$\frac{n}{(n-m) + m/s}$$

- 另 $p=m/n$ ，则
$$\frac{1}{(1-p) + p/s}$$

$$\lim_{s \rightarrow \infty} \frac{1}{(1-p) + p/s} = \frac{1}{1-p}$$

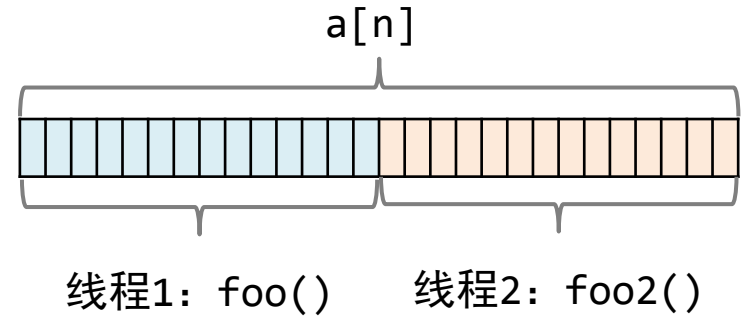


主要工具

- OpenMP：用于共享内存
 - 自动创建多线程
 - 线程同步、内存屏障
- MPI：Message Passing Interface
 - 用于分布式计算环境
 - 自动创建多个进程
 - 基于socket同步数据

传统多线程

```
int test(){
    int a[n];
    for (int i = 0; i < n; i++) {
        a[i] = 2 * i;
    }
    return 0;
}
```



```
int paratest(){
    pthread_t t1;
    pthread_t t2;
    pthread_create(&t1, NULL, foo1, NULL);
    pthread_create(&t2, NULL, foo2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

```
int a[n];
foo1(){
    for (int i=0; i<n/2; i++)
        a[i] = 2 * i;
}
foo2(){
    for (int i=n/2; i<n; i++)
        a[i] = 2 * i;
}
```

OpenMP应用举例

```
int test(){
    unsigned long long start = rdtsc();
    int a[100000];
    #pragma omp parallel for num_threads(2)
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }
    unsigned long long cycles = rdtsc() - start;
    printf("cycles = %d\n", cycles);
    return 0;
}
```

Assembly Code


```
push    rbp
mov     rbp, rsp
sub     rsp, 61A80h
mov     rax, offset _omp_outlined_
mov     rdi, offset unk_404058
mov     esi, 1
mov     rdx, rax
lea     rcx, [rbp+var_61A80]
mov     al, 0
call    __kmpc_fork_call
xor     eax, eax
add     rsp, 61A80h
pop     rbp
retn
```

```
void __kmpc_fork_call (
    ident_t * loc, //源代码信息
    kmp_int32  argc,
    kmpc_micro microtask,
    ...
)
```

循环中的数据依赖问题： 示例1

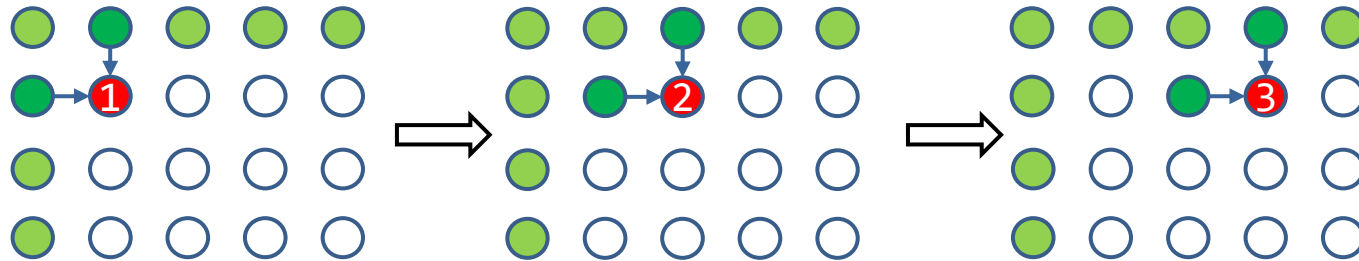
```
int test(){  
    int a[20];  
    a[0] = 1;  
    a[1] = 1;  
    #pragma omp parallel for num_threads(4)  
    for (int i = 0; i < 20; i++) {  
        a[i] = a[i-1] + a[i-2];  
    }  
    return 0;  
}
```

数据依赖，无法并行

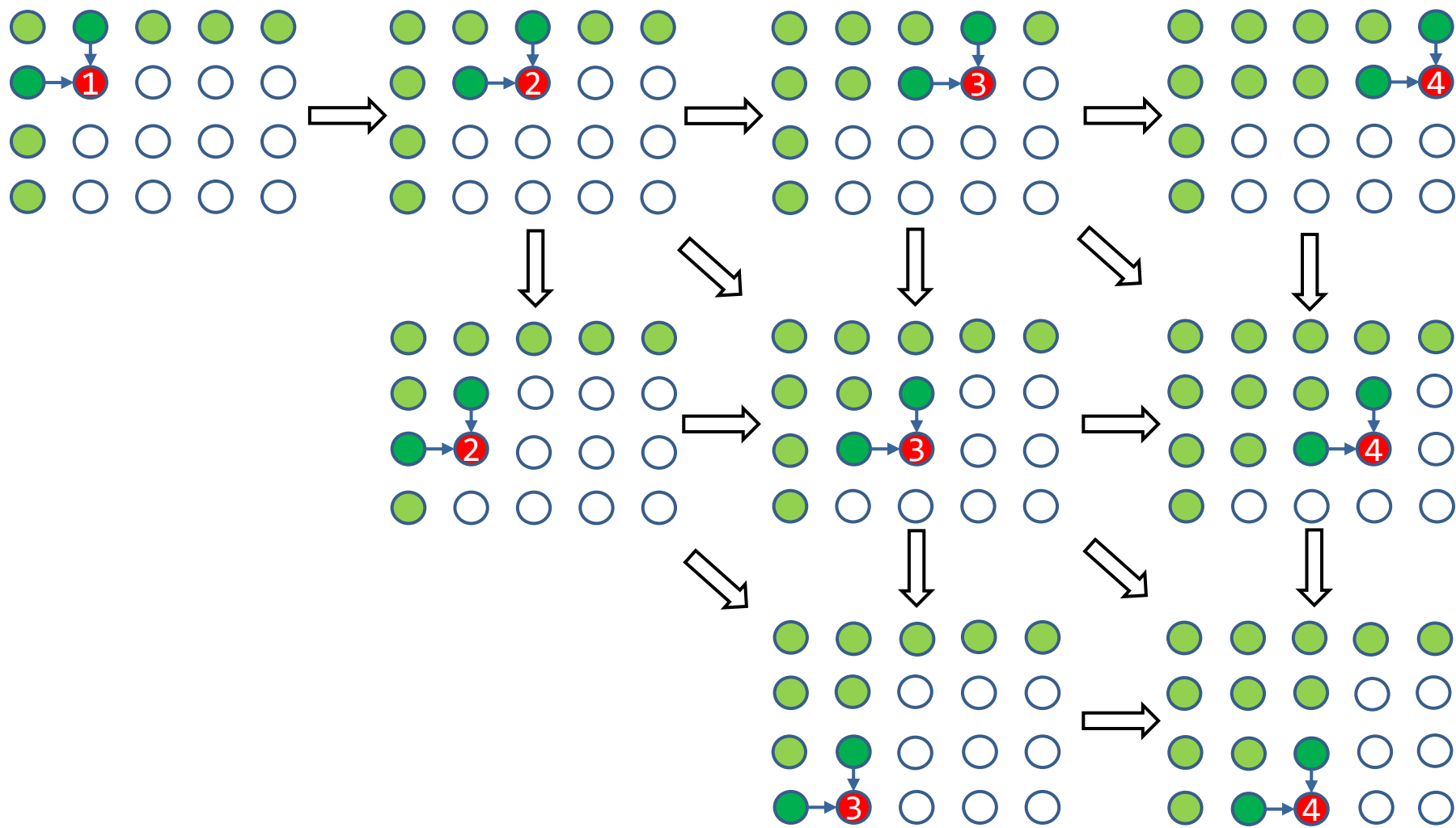


循环中的数据依赖问题： 示例2

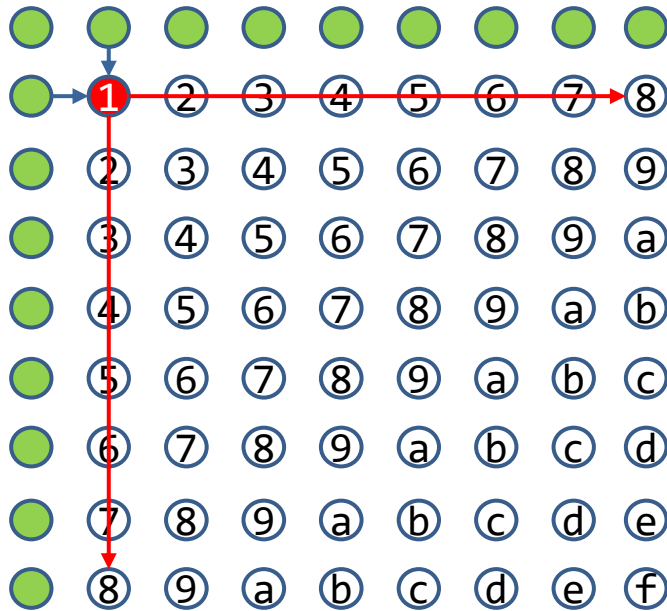
```
for(int i = 1; i < n; i++) {  
    for(int j = 1; j < n; j++) {  
        a[i][j] = a[i-1][j] + a[i][j-1];  
    }  
}
```



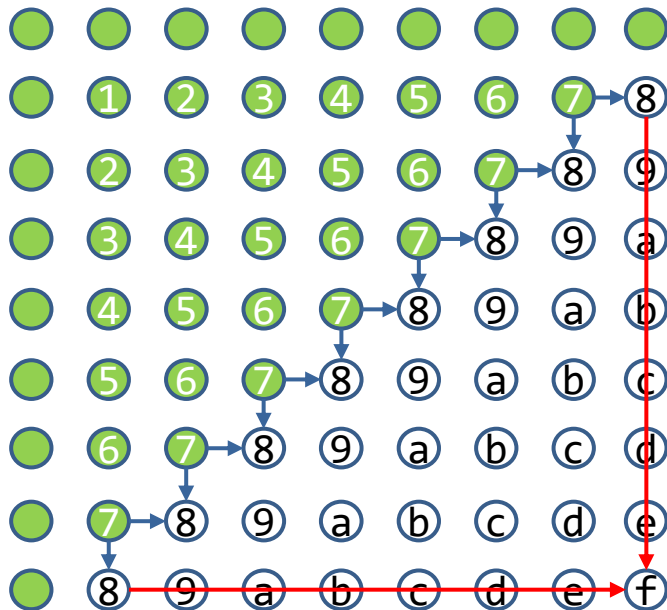
依赖分析



依赖优化: Polyhedral model



```
for(int i = 1; i < n-1; i++) {  
    for(int j = 1; j < i+1; j++) {  
        a[i-j+1][j] = a[i-j][j]  
            + a[i-j+1][j-1];  
    }  
}
```



```
for(int i = 1; i < n-1; i++) {  
    for(int j = n-1; j > i-1; j--) {  
        a[j-i+1][j] = a[j-i+1][j]  
            + a[j-i][j];  
    }  
}
```

总结

- 程序分析难题
- 指针分析
 - may alias、must alias
 - 流不敏感分析：Anderson style、Steensgaard
 - 路径敏感分析：SCC检测=>生成树
- 多版本和自适应优化
 - 循环多版本、JIT bailout
- 并行优化
 - OpenMP和数据依赖问题