并行编译与优化
Parallel Compiler and Optimization

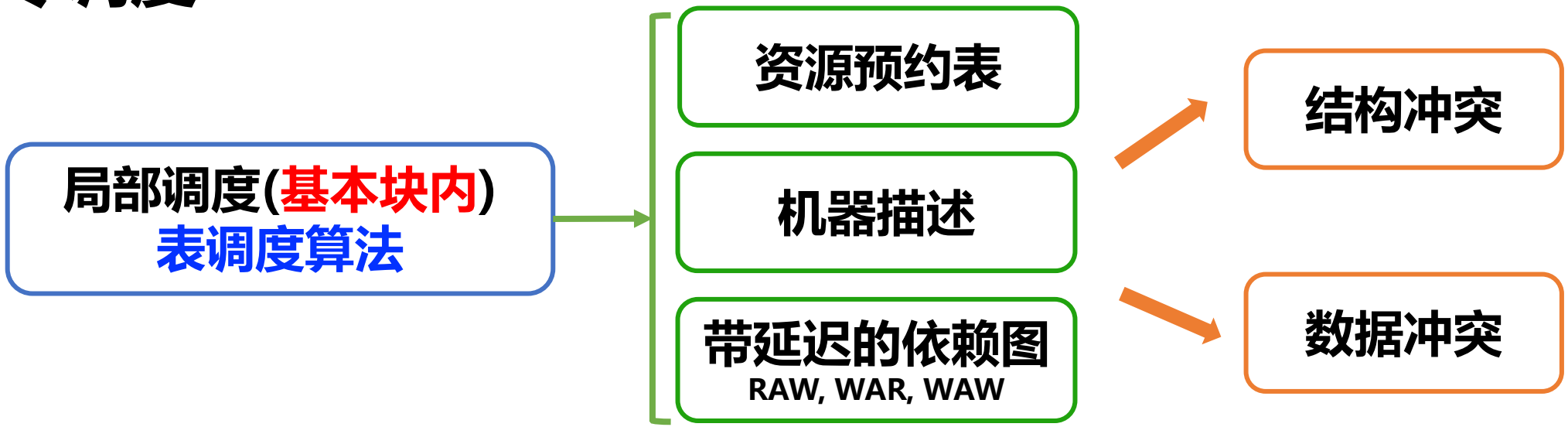
计算机研究所编译系统室

Lecture 13: Instruction Scheduling Part2

第十三课：指令调度（二）

回顾

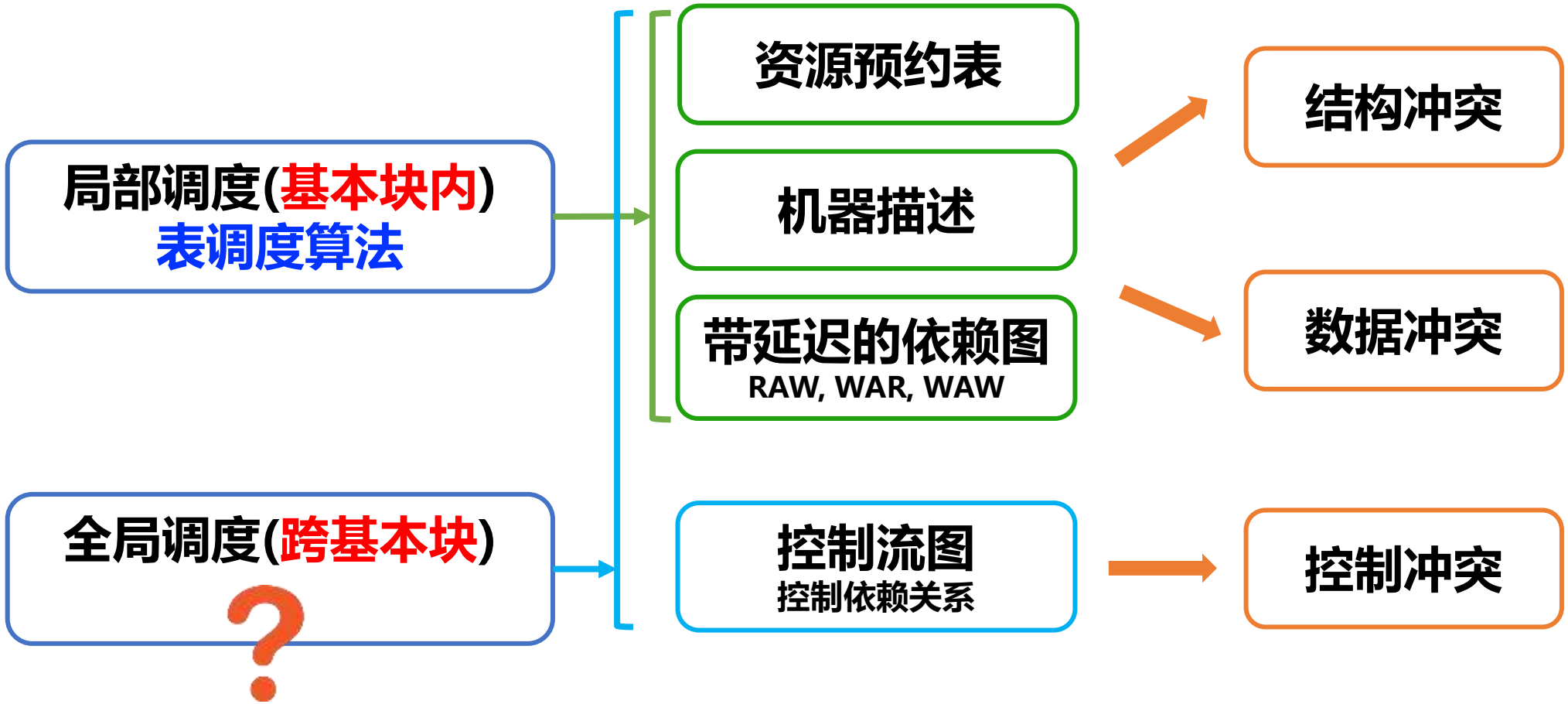
- 指令调度



- 基本块内的指令条数可能不够多，无法挖掘足够的指令级并行
  - 通过循环展开、循环合并等优化技术扩大基本块
  - 实现跨基本块的调度，即全局调度

回顾

- 指令调度



- 全局调度，不仅需要考虑资源约束和数据依赖，还要考虑控制依赖

学习内容

12.1 指令调度概述

12.2 指令调度的先决条件

12.3 局部调度: 基本块的表调度方法

12.4 优化指令调度的技术

12.5 控制依赖与投机执行

12.6 全局调度: 跨基本块的代码移动方法

12.7 轨迹调度

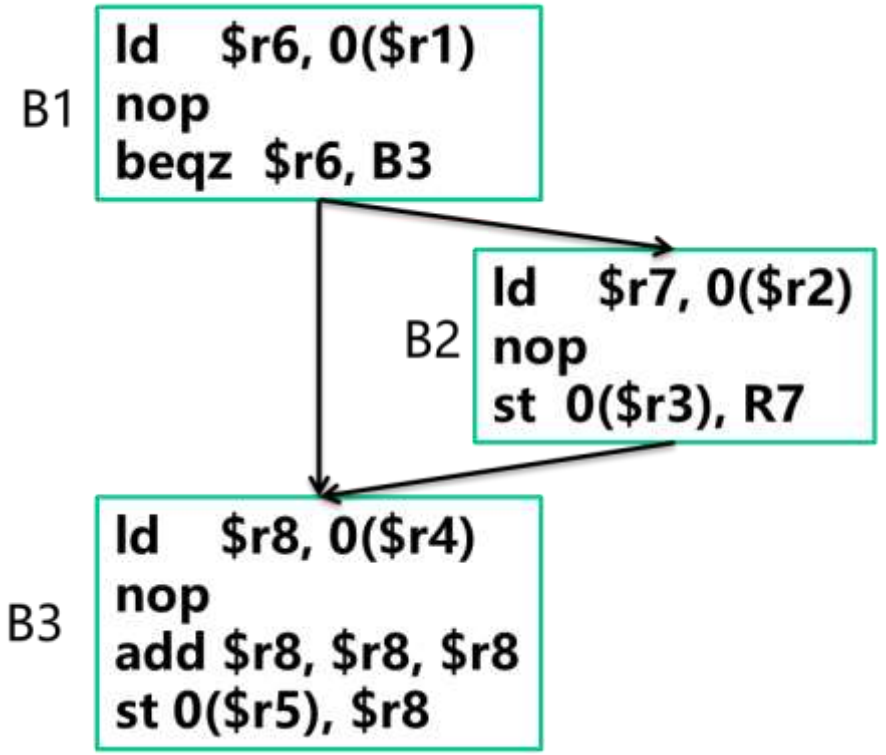
12.8 软件流水

学习目标

- 掌握跨基本块的代码移动方法，对给定控制流图和指令移动的源与目标结点，能够指出需要完成的全局调度操作
- 熟悉轨迹调度方法，对给定控制流图，能够计算出轨迹和轨迹调度的顺序
- 理解控制依赖和投机执行的概念，以及软件流水调度方法

12.5.1 控制依赖

- 如果一条语句S2的执行与否依赖于S1的输出，那么S2控制依赖于S1，记作S1 δ^c S2
- 当基本块B2的执行与否依赖于基本块B1的输出，称基本块B2控制依赖于基本块B1，记作B1 δ^c B2

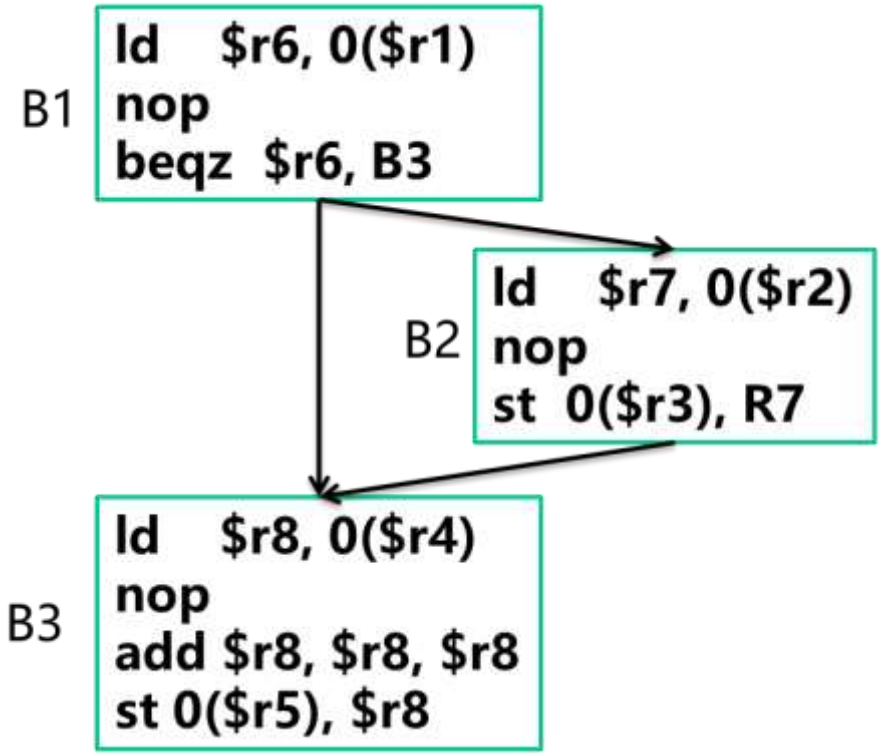


B1 δ^c B2

B1 δ^c B3 ??

12.5.2 控制等价

- 如果基本块B2执行**当且仅当**基本块B1也执行，则称B1和B2**控制等价**



B1 δ B2

B1 δ B3 ??

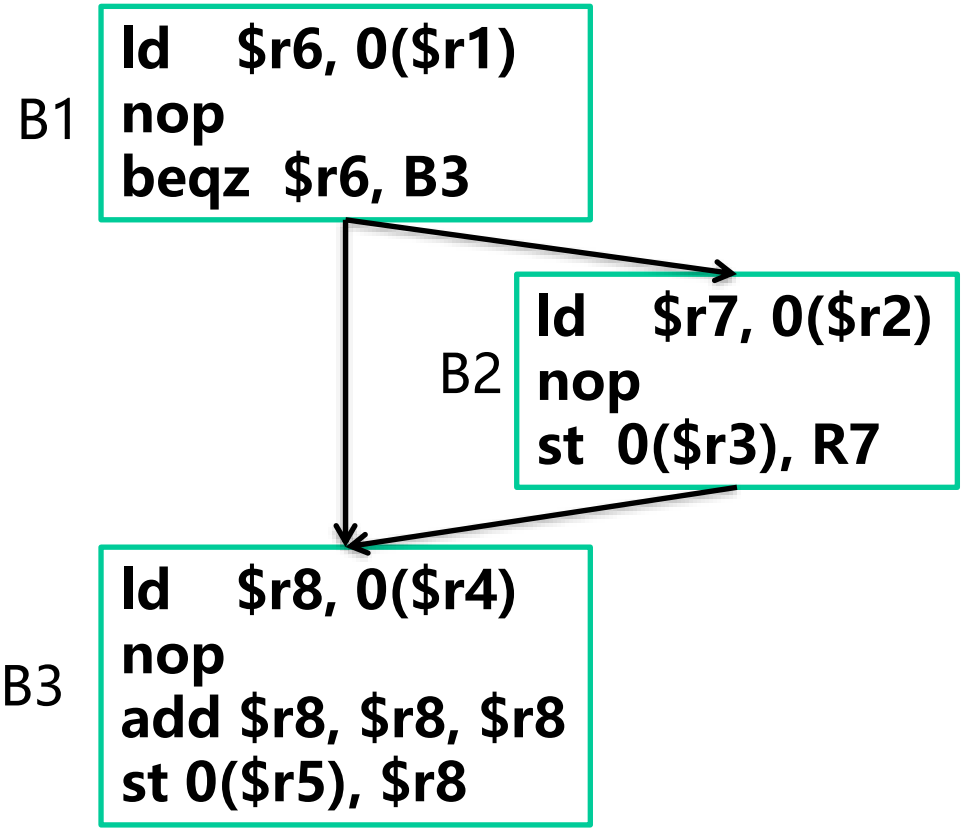
B1与B3控制等价

12.5.2 控制等价

- 必经结点
  - CFG中，如果从入口结点到结点b的每一条路径都经过结点a，则称a是b的必经结点，记作a dom b
- 后必经结点
  - CFG中，如果从结点a到出口结点的每一条路径都经过结点b，则称b是a的后必经结点，记作b pdom a
- 如果a dom b且b pdom a，则a和b控制等价

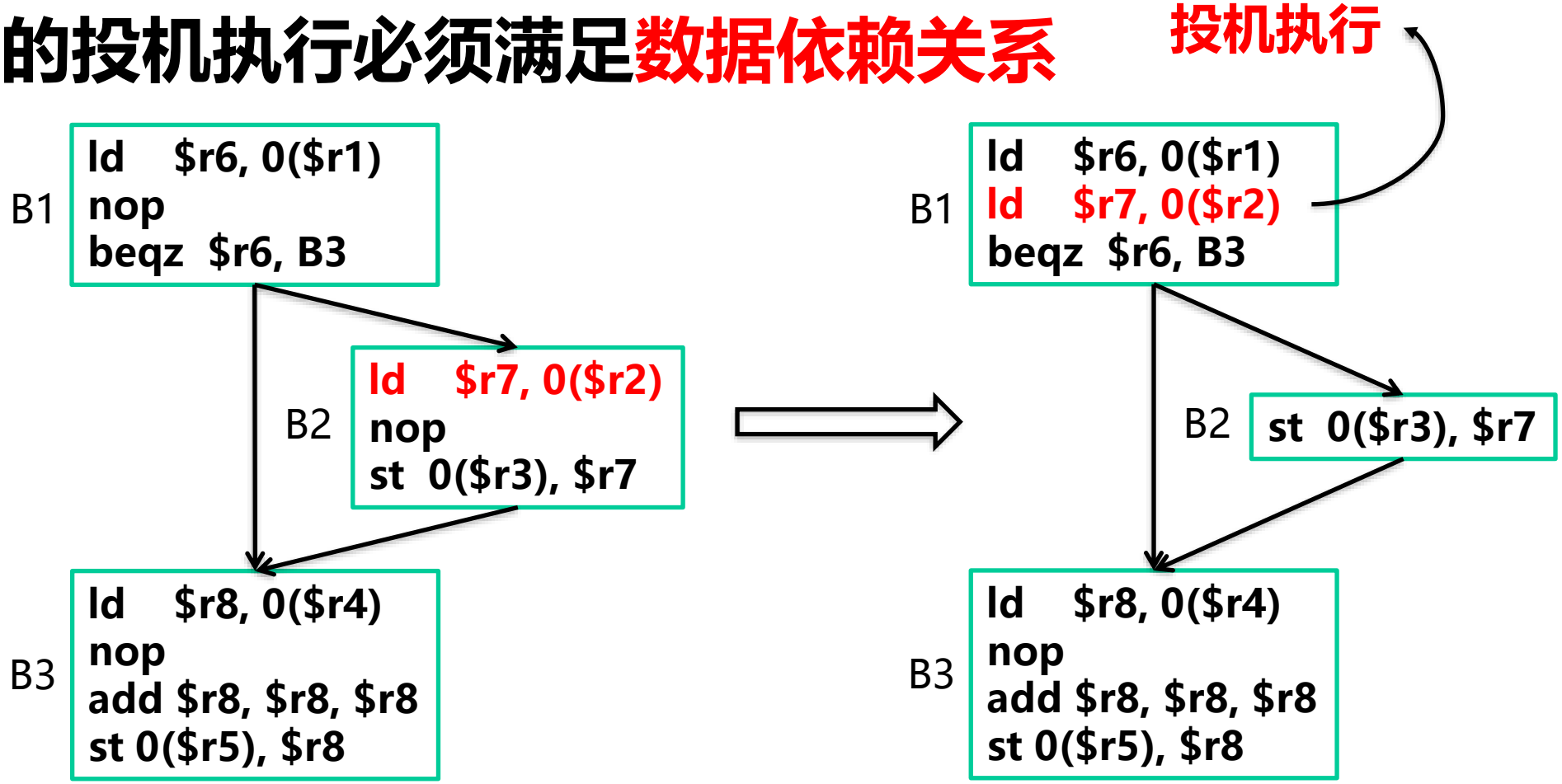
12.5.2 控制等价

- B1与B3控制等价
  - B1 dom B3, B3 pdom B1
- B1与B2非控制等价
  - B2控制依赖于B1 (B1 δ^c B2)
  - B1 dom B2, B2 does not pdom B1
- B2与B3非控制等价
  - B3 pdom B2, B2 dose not dom B3



12.5.3 投机执行

- 如果指令S1在它所控制依赖的所有指令都执行完之前，被执行，则S1的执行是**投机执行**(speculative execution)
- 安全的投机执行必须满足**数据依赖关系**



12.5.3 投机执行

- 全局指令调度是一种代码优化
  - 源程序中执行的所有指令需在调度后(即优化后)的程序中执行
  - 优化后的程序可能投机执行一些额外的指令
  - 这些额外指令不能产生有害的副作用
  - 投机执行不一定总是好的，也可能导致性能变慢

学习内容

12.1 指令调度概述

12.2 指令调度的先决条件

12.3 局部调度: 基本块的表调度方法

12.4 优化指令调度的技术

12.5 控制依赖与投机执行

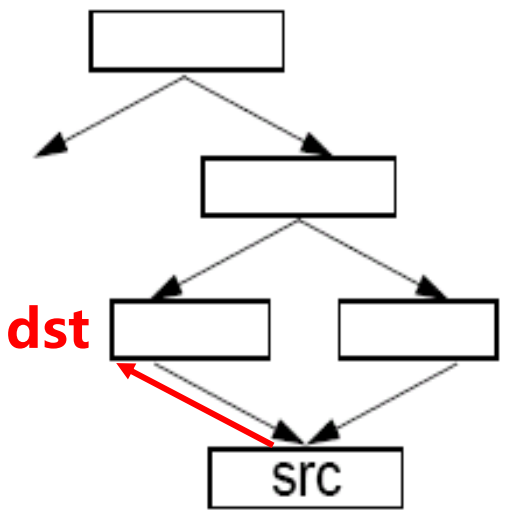
12.6 全局调度: 跨基本块的代码移动方法

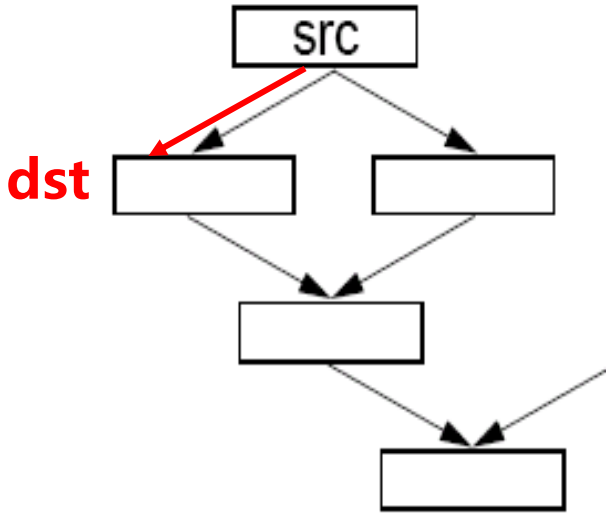
12.7 轨迹调度

12.8 软件流水

12.6.1 跨基本块的代码移动

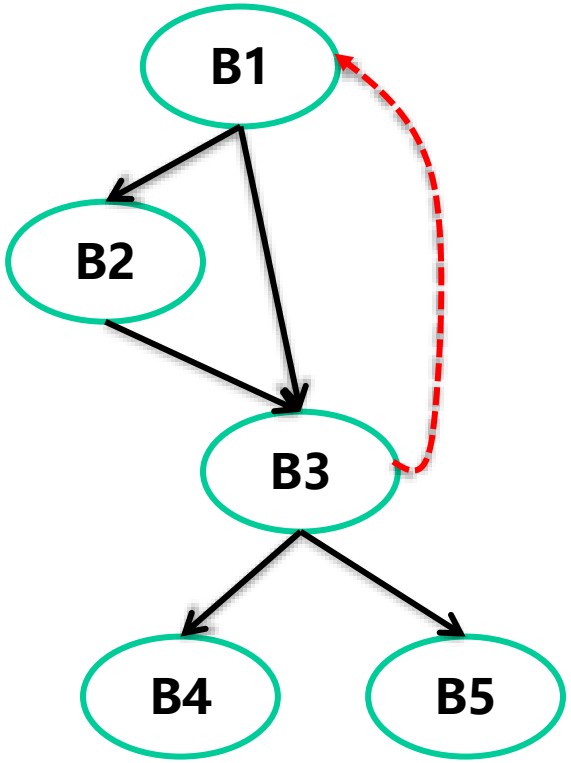
- 向上代码移动
  - 指令从src基本块逆着控制流路径向上移动到dst基本块
- 向下代码移动
  - 指令从src基本块沿着控制流路径向下移动到dst基本块
- 代码移动不能违反任何数据依赖关系





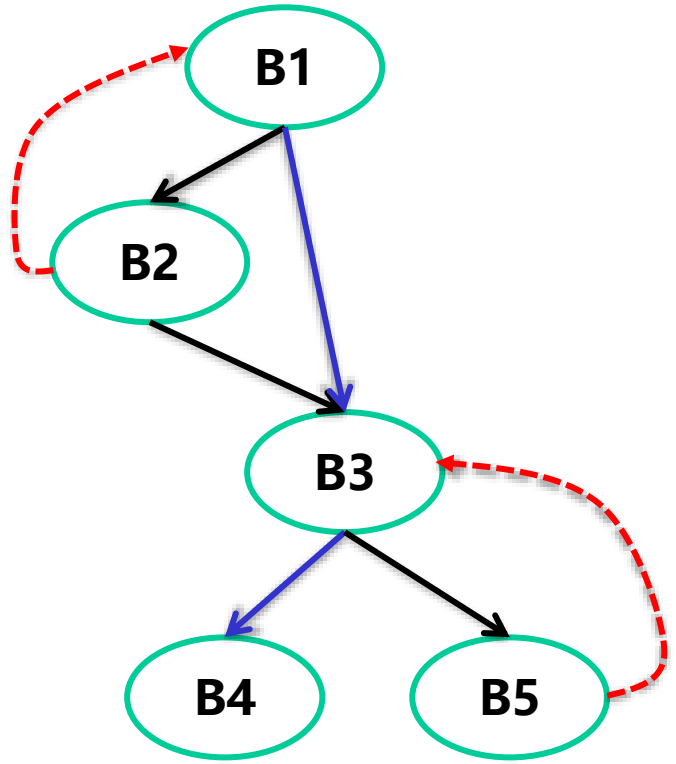
12.6.2 向上代码移动

- (1) src pdom dst + dst dom src
  - src 控制等价 dst
  - 被调度的指令原来执行一次，移动后也会执行一次，并且只会执行一次



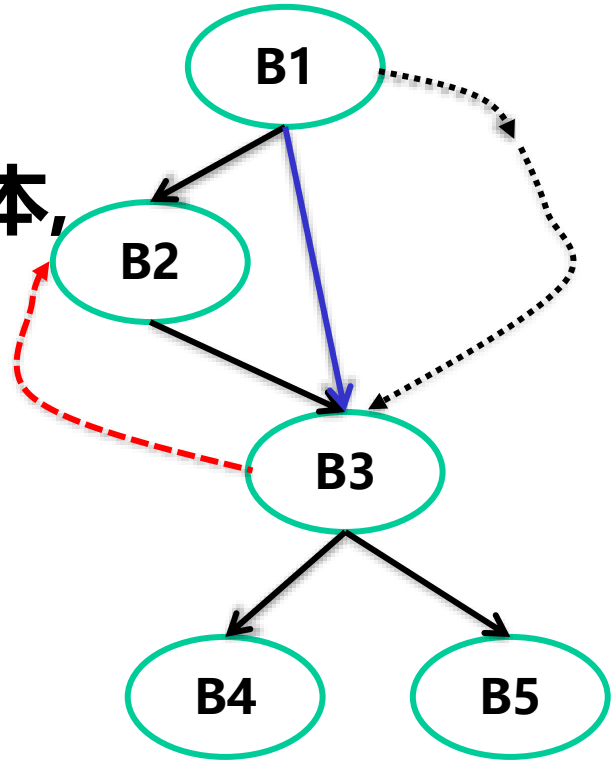
12.6.2 向上代码移动

- (2) src does not pdom dst + dst dom src
  - 存在从dst到exit的路径，不经过src
  - 代码移动是投机执行,可能导致执行原本不会执行的代码(额外的代码)
    - 例如: B2→B1,B5→B3
  - 需保证投机执行的代码不会产生有害的副作用
  - 只有当控制流到达src时，投机执行才有好处



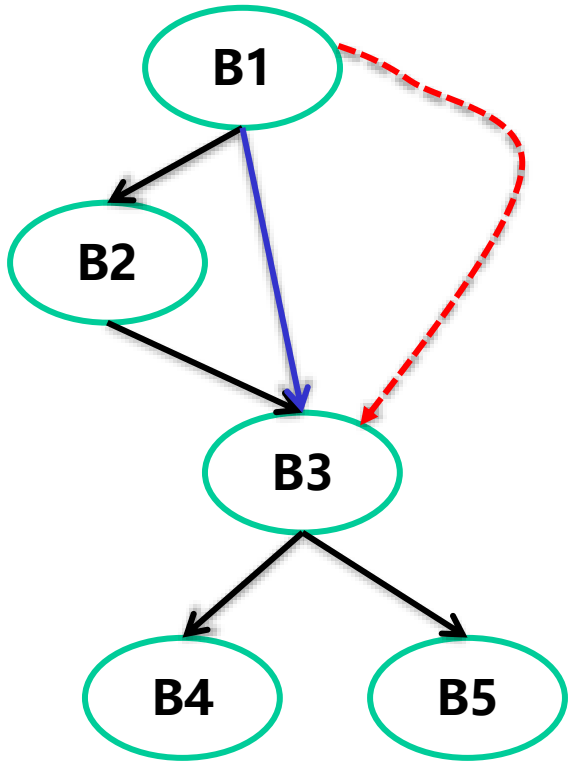
12.6.2 向上代码移动

- (3) src pdom dst + dst does not dom src
  - 存在从start到src的路径，不经过dst
  - 代码移动可能导致原本会执行的代码没有得到执行
    - 例如: B3→B2
  - 需要在不经过dst的路径上插入被移动的指令的副本，即插入补偿代码
  - 当被优化路径的执行频率高于其他未被优化的路径时，代码移动才有好处



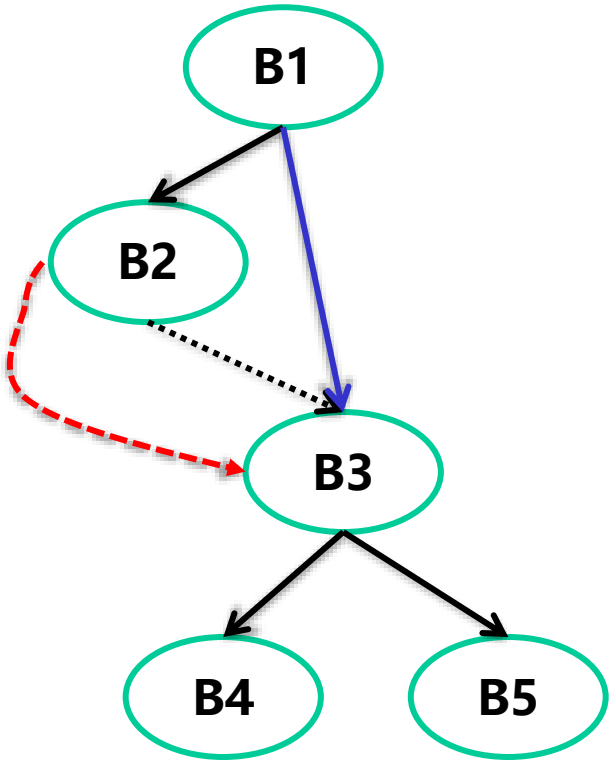
12.6.3 向下代码移动

- (1) src dom dst + dst pdom src
  - src <-控制等价-> dst
  - 被调度的指令原来执行一次，移动后也会执行一次，并且只会执行一次



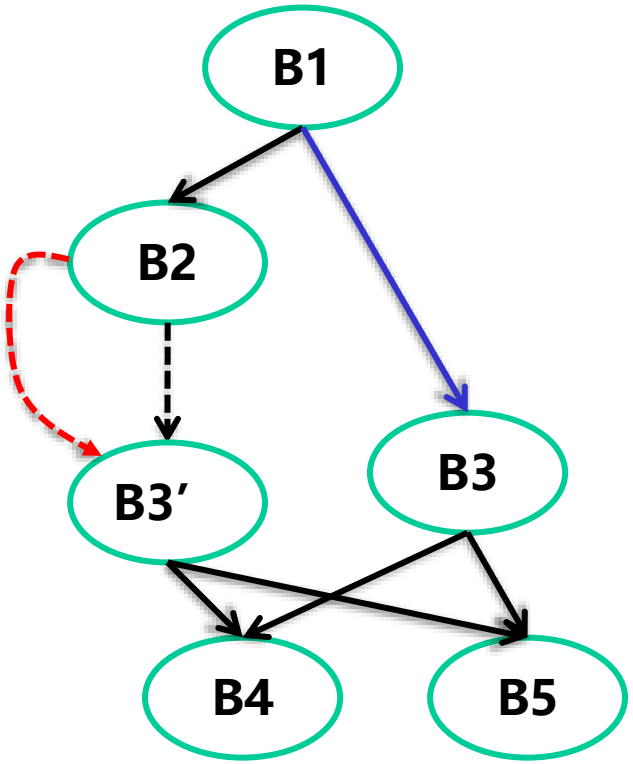
12.6.3 向下代码移动

- (2) src does not dom dst + dst pdom src
  - 存在从start到dst的路径，不经过src
  - 代码移动可能导致执行原本不会执行的代码(额外的代码)
    - 例如: B2→B3
  - 向下代码移动常用于写指令
    - 写指令具有副作用，可能覆盖原来的值



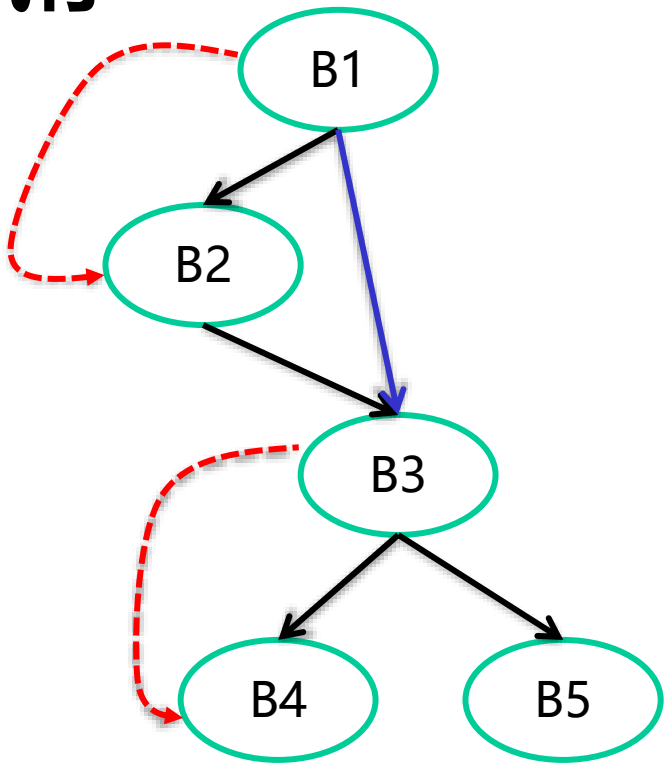
12.6.3 向下代码移动

- (2) src does not dom dst + dst pdom src
  - 存在从start到dst的路径，不经过src
  - 代码移动可能导致执行原本不会执行的代码(额外的代码)
    - 例如: B2→B3
  - 向下代码移动常用于写指令
    - 写指令具有副作用，可能覆盖原来的值
  - 通过代码复制绕过该问题
    - 复制从src到dst的路径上的基本块，并且只在dst的新拷贝中放置下移的写指令
  - 只有当控制流到达src时，代码移动才有好处



12.6.3 向下代码移动

- (3) src dom dst + dst does not pdom src
  - 存在从src到exit的路径，不经过dst
  - 代码移动可能导致原本会执行的代码没有得到执行
    - 例如: B1→B2， B3→B4
  - 需要在不经过dst的路径上插入补偿代码
  - 当被优化路径的执行频率高于其他未被优化的路径时，代码移动才有好处



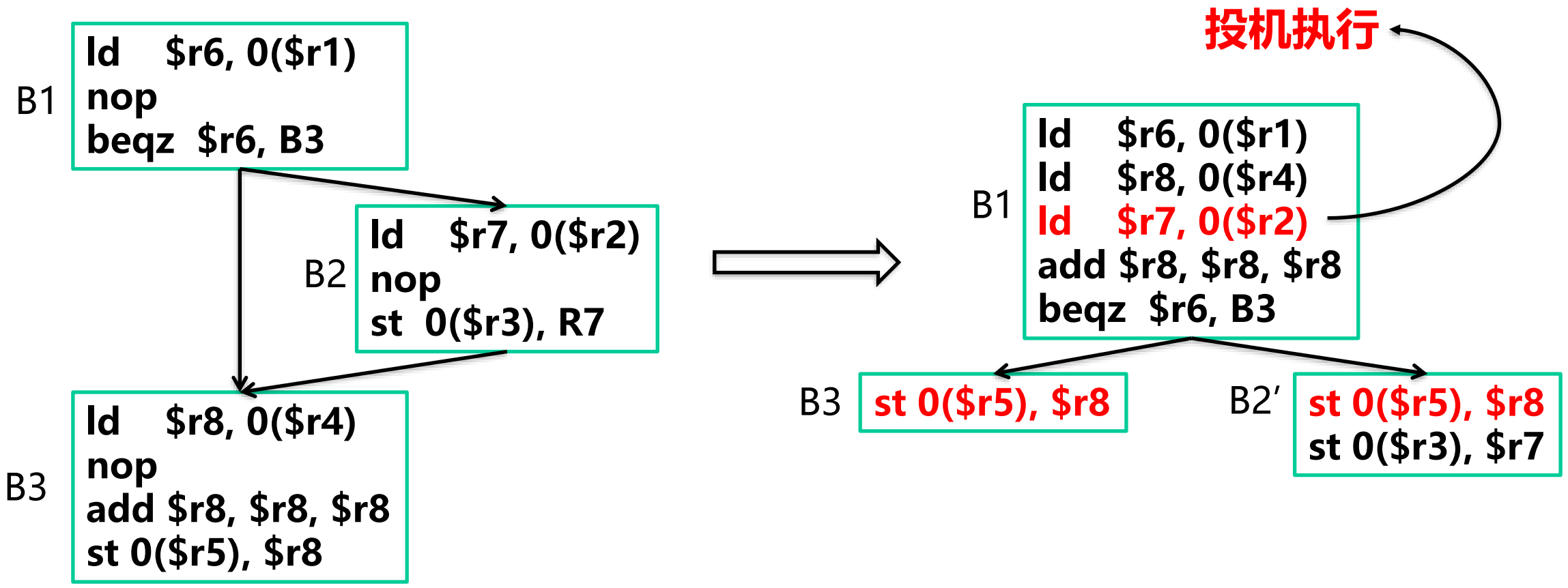
跨基本块的代码移动总结

- 向上和向下代码移动分别包含四种情况

•	Up: src pdom dst	dst dom src	Speculation	Compensation code
	Down: src dom dst	dst pdom src	Code duplication	
1	Yes	Yes	No	No
2	No	Yes	Yes	No
3	Yes	No	No	Yes
4	No	No	Yes	Yes

- 在控制等价的结点之间移动代码，最简单，且性价比最高
- 代码移动对某些路径有益，但会损害另一些路径的性能，因此全局代码移动的目标是使频繁执行的路径更快执行

跨基本块的代码移动示例



学习内容

12.1 指令调度概述

12.2 指令调度的先决条件

12.3 局部调度: 基本块的表调度方法

12.4 优化指令调度的技术

12.5 控制依赖与投机执行

12.6 全局调度: 跨基本块的代码移动方法

12.7 轨迹调度

12.8 软件流水

12.7.1 轨迹

- 轨迹(Trace)是控制流图中基本块构成的无环路径
- 构建轨迹的原则
  - 可以有分支，但不包含循环(无环，不包含向后边)
  - 轨迹是频繁执行的路径
  - 一个基本块只允许出现在一个轨迹中

12.7.2 轨迹调度的基本思想

- 一种常用的全局调度方法
  - 1981，Fisher提出
- 目标是使程序中频繁执行的轨迹执行得更快
- 基本思想
  - 基于控制流图构建轨迹
  - 将轨迹看作一个基本块，使用基本块调度方法调度轨迹中的指令
  - 从具有最高优先级的轨迹(即执行最频繁的轨迹)开始调度
  - 为保证程序正确性，需要处理轨迹入口和出口处的副作用

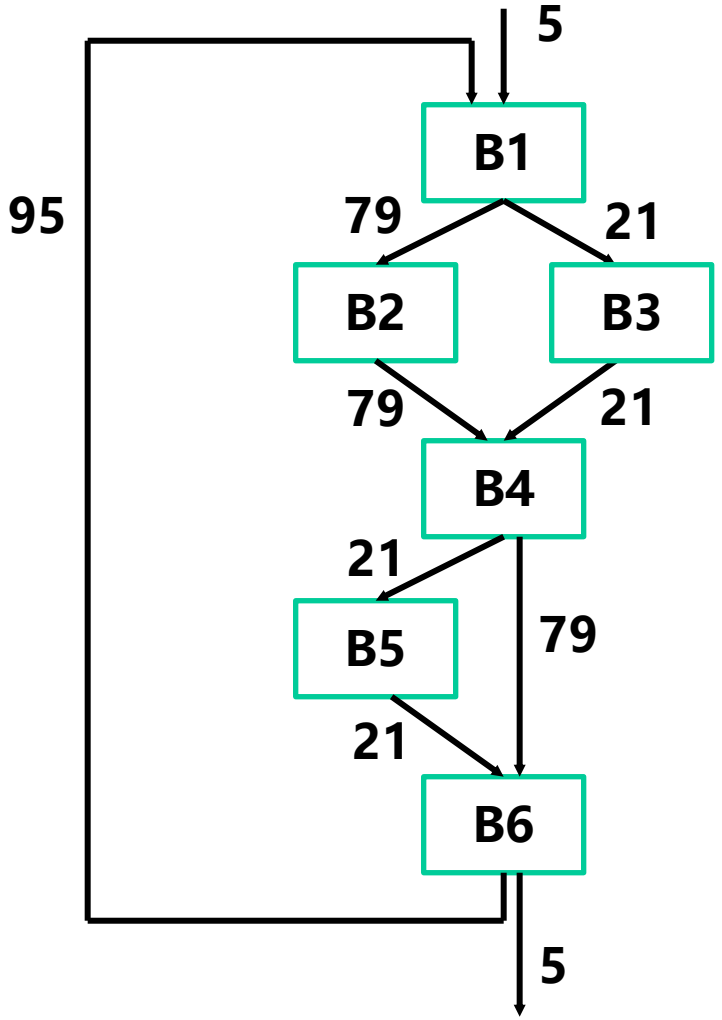
12.7.3 轨迹构建

- 加权控制流图: 控制流图的结点或边标记执行频率
  - 插桩动态获取
  - 静态分支预测

	condition	frequency
Pointer	!= NULL	85
operator	> 0	79
	!=	71
	fp_opcode	90
Successor	call	71
	loop	95

gcc/gcc/predict.def

```
B1: x = x + 3
    y = x + y
    if(x<0) goto B3

B2: z = x + z
    x = y
    goto B4

B3: z = x  - y
    y = x

B4: z = z * x
    if(z>0) goto B6

B5: x = z
    z = z – y

B6: y = z
    if(z>x) goto B1
…
```



12.7.3 轨迹构建

```
i = 0
标记所有基本块为未访问
while (还有未访问的基本块) {
    seed = 未访问基本块中执行频率最高的基本块
    trace[i] += seed;
    标记此基本块已被访问;
    current = seed;
    /* 向前扩展轨迹*/
    while (1) {
        next = best_successor_of(current);
        if (next == 0)  break;   // 当前轨迹已经不能进一步扩展
        trace[i] += next;
        标记找到的最佳后继next已被访问;
        current = next;
    }
    i++;  //寻找下一条轨迹
}
```

构建轨迹的关键是寻找当前结点的最佳后继，扩展轨迹。当没有找到最佳后继结点时，轨迹就不再扩展

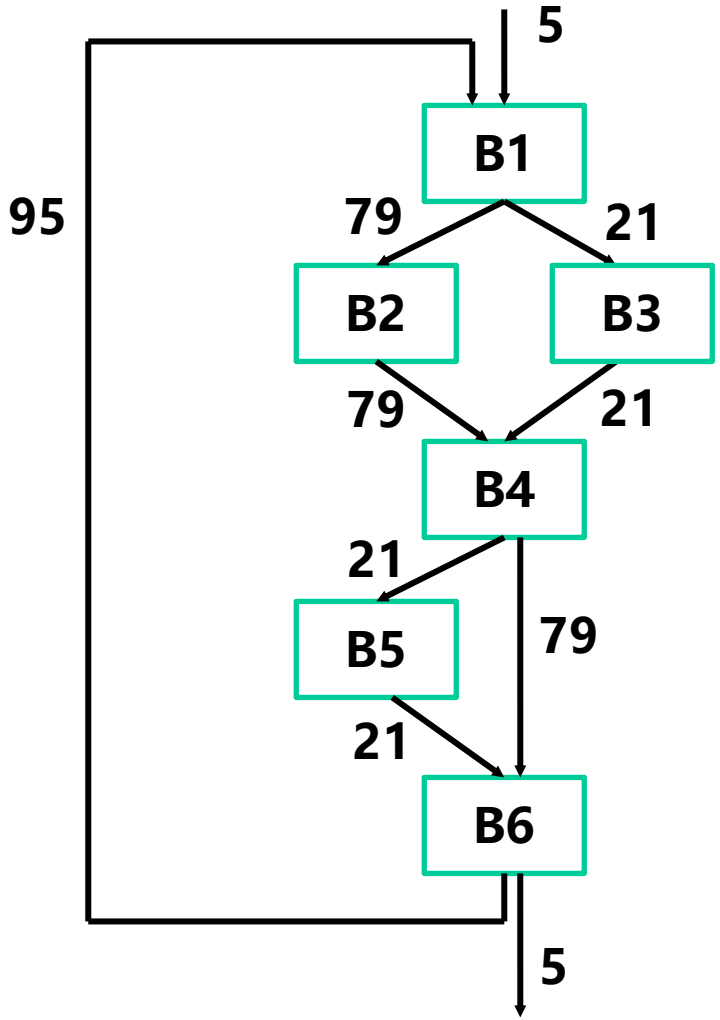
12.7.3 轨迹构建

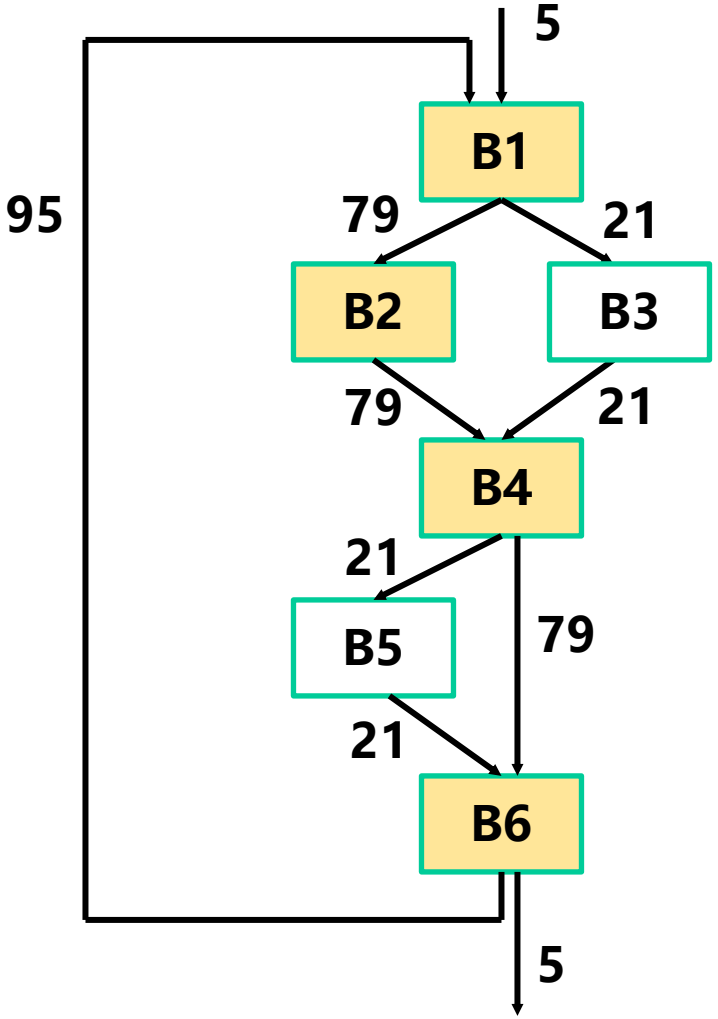
```
best_successor_of(BB){
    e = 离开基本块BB的具有最高执行概率的边;
    if (e是向后边)  //轨迹是无环路径
        return 0;

    if (probability(e) <= 阈值)  //非频繁执行的路径
        return 0;

    d = 边e的终点基本块;
    if (d已被访问)  //一个基本块只能出现在一条轨迹中
        return 0;

    return d;
}
```

当离开基本块的具有最高执行概率的边是向后边，或着其执行概率低于阈值，或者对应的终点基本块已被访问，则轨迹不能进一步扩展

12.7.3 轨迹构建

示例





假设阈值=75

① Seed = B1

② B1最佳后继是B2

轨迹：B1-B2

③ B2最佳后继是B4

轨迹：B1-B2-B4

④ B4最佳后继是B6

轨迹：B1-B2-B4-B6

⑤ B6具有最高执行概率的边是向后边

轨迹：B1-B2-B4-B6

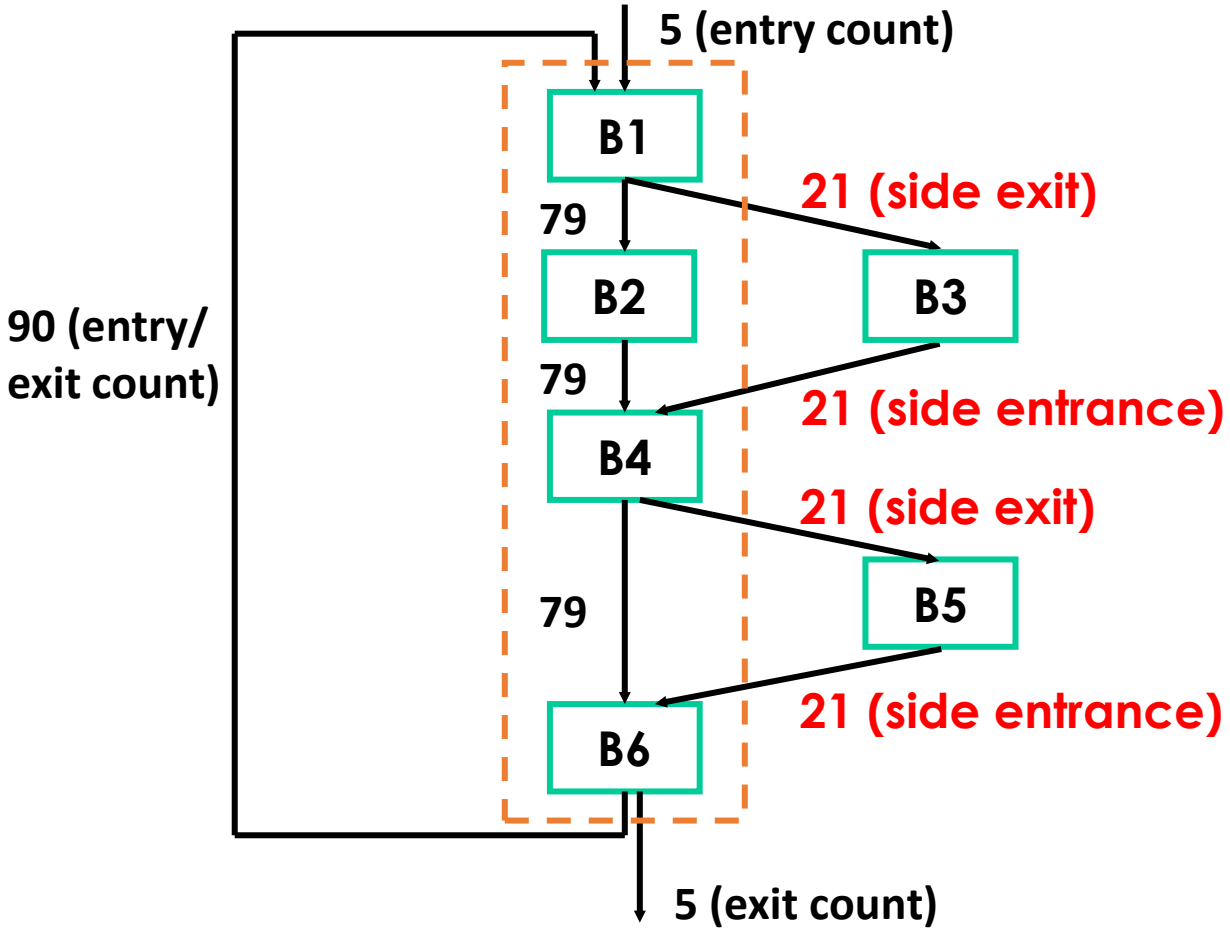
3条轨迹：{B1,B2,B4,B6}，{B3}，{B5}

调度(从执行最频繁的轨迹开始调度)

{B1,B2,B4,B6} → {B3} → {B5}

12.7.4 轨迹调度

- 将轨迹看作一个基本块进行调度，由于轨迹并不是完全顺序执行的指令序列，需要处理轨迹入口和出口处的副作用



- 离开轨迹(side exit)

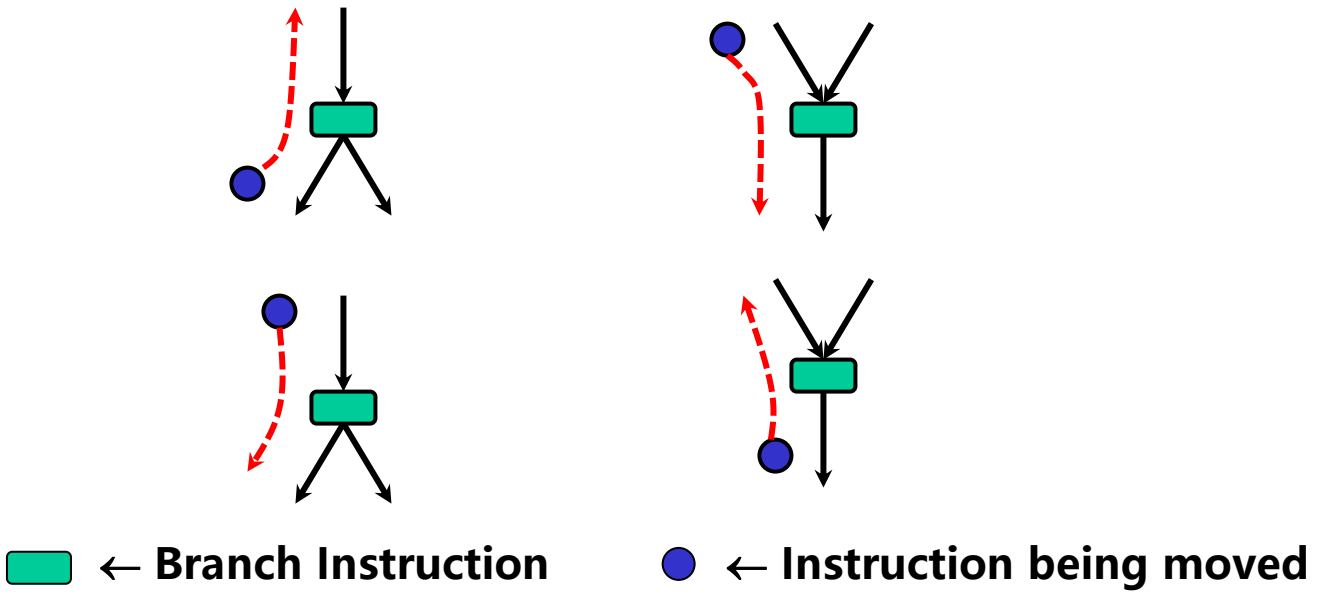
  B1→B3, B4→B5

- 进入轨迹(side entrance)

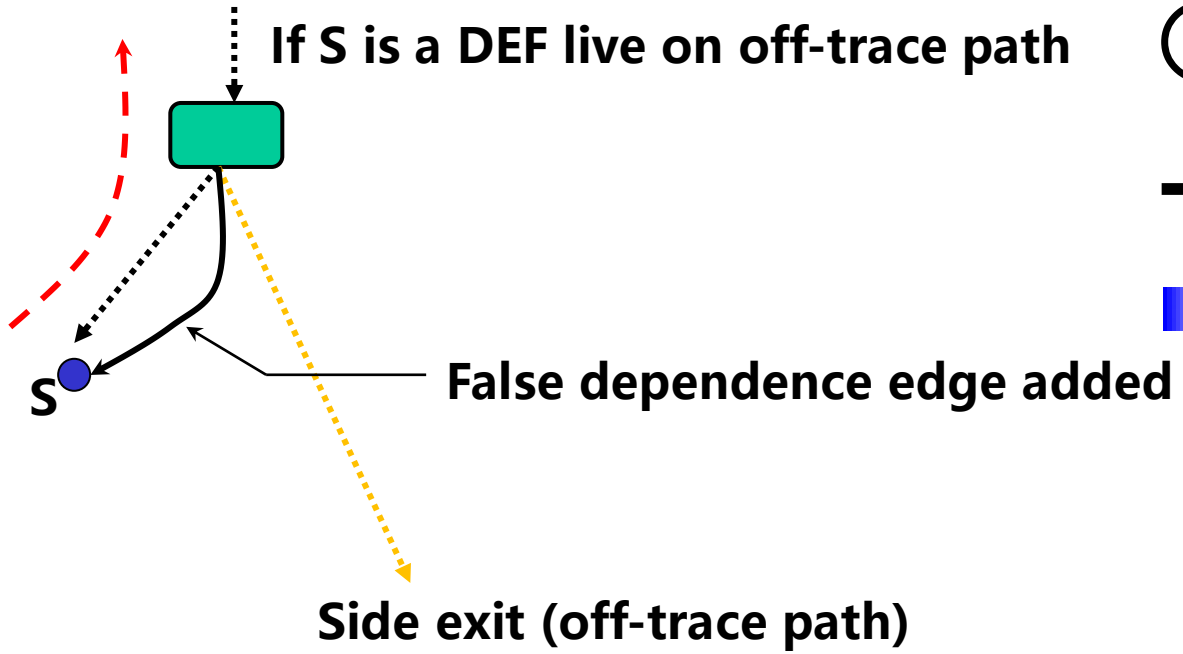
  B3→B4, B5→B6

12.7.4 轨迹调度

- 考虑轨迹内一条指令跨越分支或汇合指令进行调度的四种情况
  - 指令从分支结点的后继结点向上移动到分支结点
  - 指令从汇合结点的前驱结点向下移动到汇合结点
  - 指令从分支结点向下移动到分支结点的后继结点
  - 指令从汇合结点向上移动到汇合结点的前驱结点



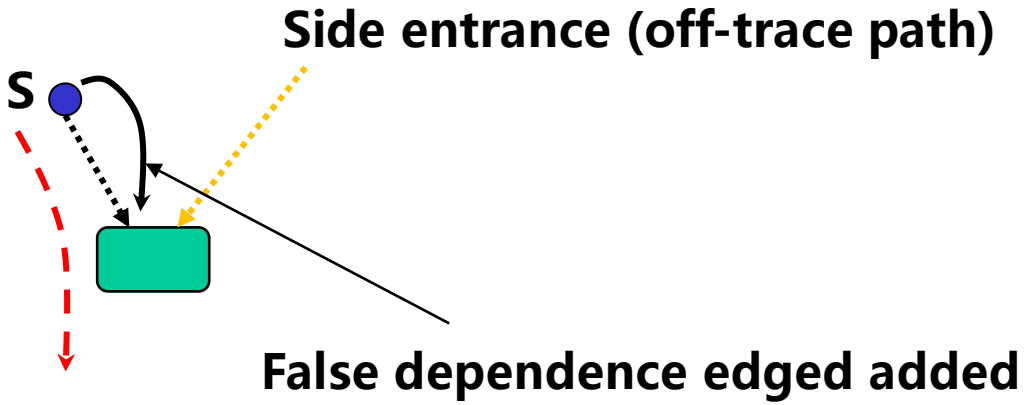
12.7.4 轨迹调度



①指令从分支结点的后继结点向上移动到分支结点

- **投机执行S指令**
  - 如果S是对某个变量a的定值，则a不能在离开轨迹路径上活跃
  - 如果a在离开轨迹路径上活跃，则需在依赖图中增加一条分支指令到S指令的额外的**伪依赖边**，以阻止这个向上代码移动

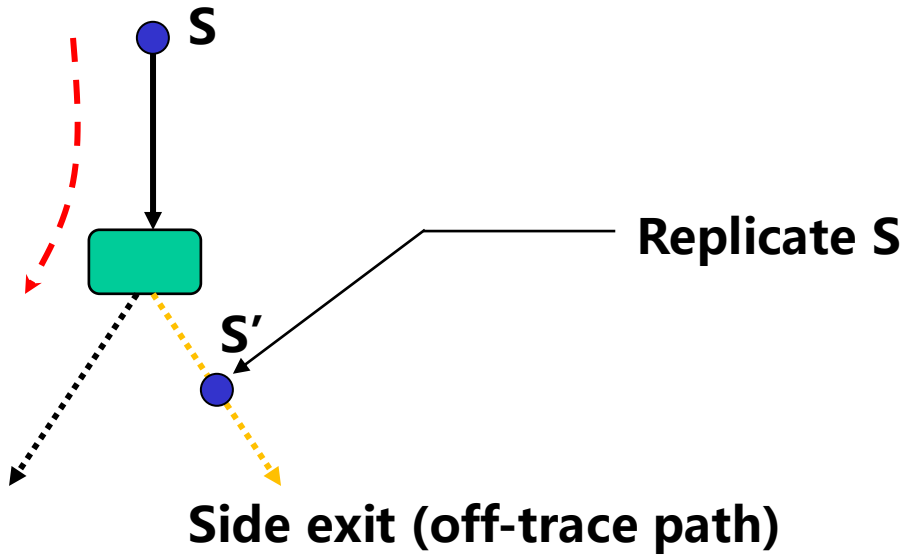
12.7.4 轨迹调度



②指令从汇合结点的前驱结点向下移动到汇合结点

- 可能**执行额外的计算**
  - 如果S是定值指令，则需在依赖图中增加一条S指令到汇合指令的**伪依赖边**，以阻止这个向下代码移动
  - 更一般的解决方法是允许代码移动，但是通过**代码复制**消除写指令下移的影响

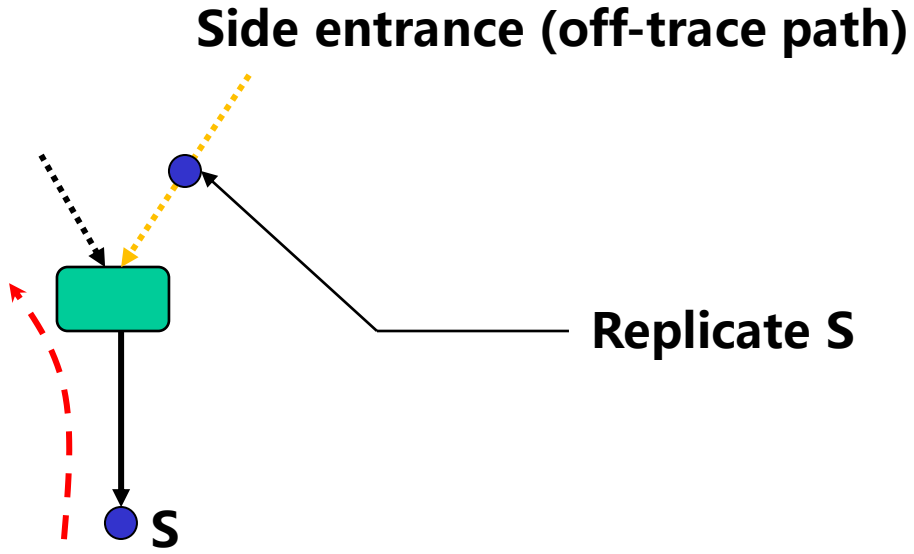
12.7.4 轨迹调度



③指令从分支结点向下移动到分支结点的后继结点

- S可能不会被执行
  - 为了避免代码移动造成的正确性问题，需在离开轨迹路径上复制S，即**插入补偿代码**
  - 插入补偿代码可能导致离开轨迹路径变慢，因此需保证被优化的轨迹具有较高执行频率，代码移动才有好处

12.7.4 轨迹调度



④指令从汇合结点向上移动到汇合结点的前驱结点

- S可能不会被执行
  - 为了避免代码移动造成的正确性问题，需在离开轨迹路径上复制S，即**插入补偿代码**
  - 插入补偿代码可能导致离开轨迹路径变慢，因此需保证被优化的轨迹具有较高执行频率，代码移动才有好处

12.7.4 轨迹调度

Fisher轨迹调度算法

①在程序中确定一个不包含循环的(最大的)区域r

②在r中构建一条轨迹t

③在t中增加额外的伪依赖边以限制投机执行

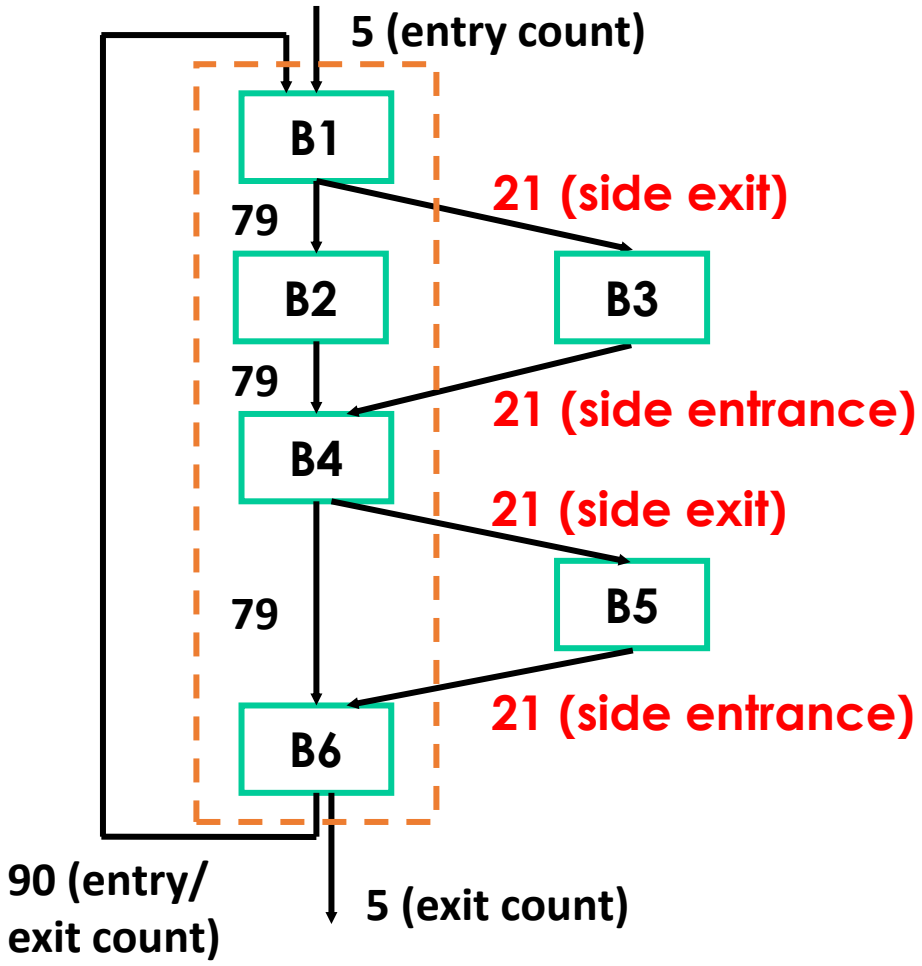
④将t中的指令按优先级排序并构建候选调度表L

⑤使用贪婪的表调度方法按优先级调度L中的指令

⑥按需在所有离开t的路径上加入补偿代码

12.7.4 轨迹调度

示例




```
B1: x = x + 3
    y = x + y
    if(x<0) goto B3

B2: z = x + z
    x = y
    goto B4

B3: z = x  - y
    y = x

B4: z = z * x
    if(z>0) goto B6

B5: x = z
    z = z – y

B6: y = z
    if(z>x) goto B1
…
```

```
B1: x = x + 3

    if(x<0) goto B3

B2: y = x + y
    z = x + z
    x = y
    goto B4

B3: y = x + y   //补偿代码
    z = x  - y
    y = x

B4: z = z * x
    if(z>0) goto B6

B5: x = z
    z = z – y

B6: y = z
    if(z>x) goto B1
…
```

学习内容

12.1 指令调度概述

12.2 指令调度的先决条件

12.3 局部调度: 基本块的表调度方法

12.4 优化指令调度的技术

12.5 控制依赖与投机执行

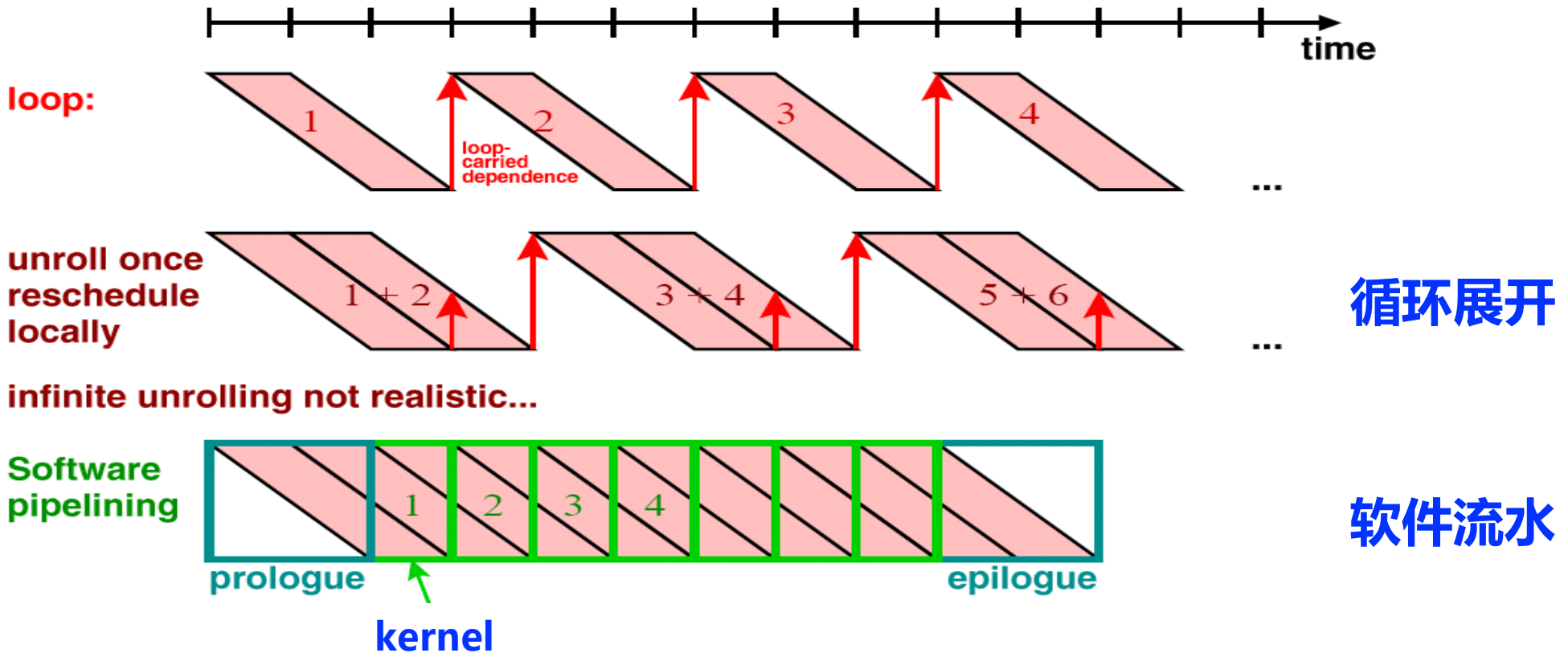
12.6 全局调度: 跨基本块的代码移动方法

12.7 轨迹调度

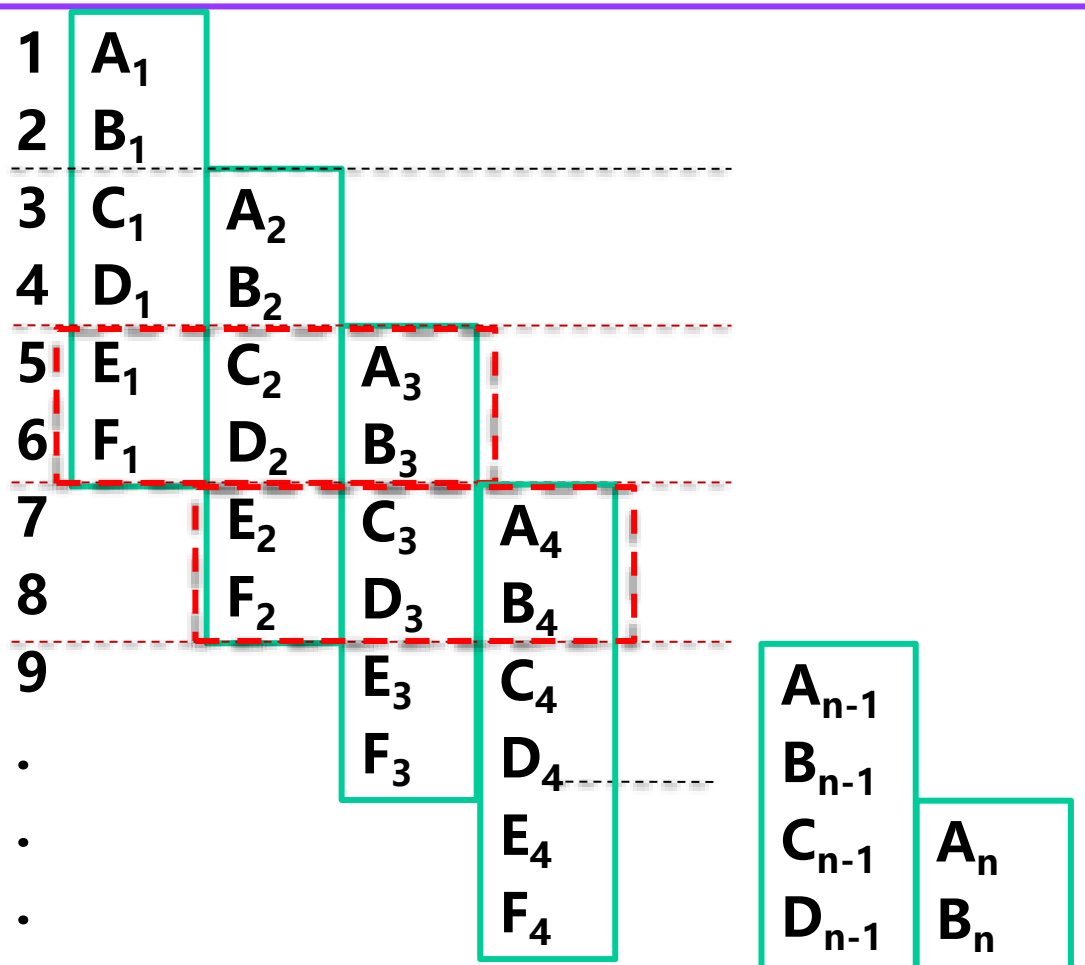
12.8 软件流水

12.8.1 软件流水的基本思想

- 针对循环的一种优化技术: 通过交错执行一个循环体的多个连续迭代中的指令来，以提高指令级并行，优化循环性能



12.8.1 软件流水的基本思想



	unit1	unit2	unit3
Prologue	A_1		
	B_1		
	C_1	A_2	
	D_1	B_2	
	for i=1, n-2		
Kernel	E_i	C_{i+1}	A_{i+2}
	F_i	D_{i+1}	B_{i+2}
Epliogue		E_{n-1}	C_n
		F_{n-1}	D_n
			E_n
			F_n

12.8.1 软件流水的基本思想

- 目标
  - 将不同连续迭代的指令组合起来，形成流水稳定状态(核心)
  - 该状态不会违背数据依赖，也不会有资源冲突
- 启动间隔(Initiation interval, //)
  - 连续两个核心迭代的第一条指令之间的时钟周期，等于核心长度
- 循环总执行时间

$$\text{length(prologue)} + \text{length(epilogue)} + // \times \#\text{iterations(kernel)}$$

- 软件流水寻找最小启动间隔(Minimal II, MII)

12.8.1 软件流水的基本思想

- 软件流水
  - 寻找一个常数的最小启动间隔MII
  - 计算一个统一调度方案S(n)，对每次迭代，S(n)相同
    - S(n)给定指令n相对于它所处的迭代的开始执行时刻
    - 第i次迭代的指令n，在第i*MII+S(n)个时刻开始执行 (i从0开始计数)
  - 通常情况下是NP-完全问题
- 软件流水调度方法
  - 模调度方法 (Lam迭代方法，Rau迭代方法，路径代数方法等)
  - 核心识别方法 (完美调度、Petri网模式等)

12.8.2 模调度方法

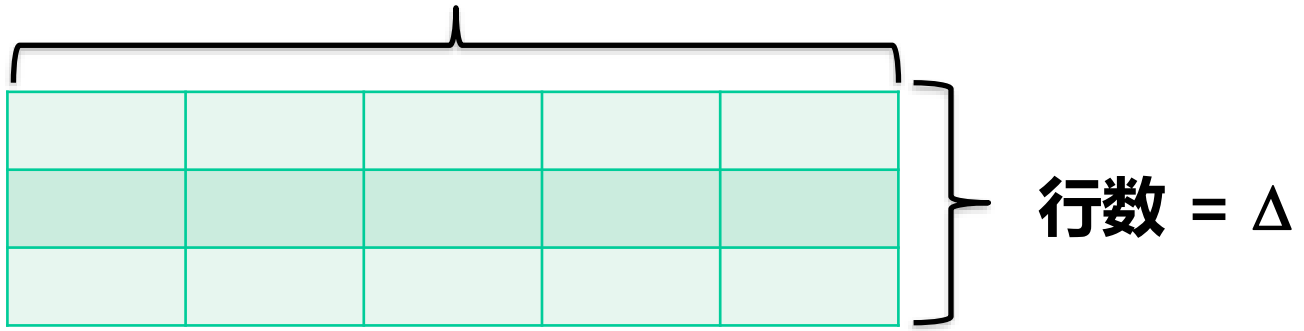
- 寻找服从资源约束和数据依赖约束的调度，尝试将循环体中的所有指令放在Δ个周期内
- 如果一条指令在时刻t违背了资源约束(不能在t时刻调度)，则该指令也不能在t+Δ时刻，或者任意t'时刻(满足t=t' mod Δ)调度
- 通过迭代回溯寻找满足约束的调度: 试着增加Δ，直到Δ的值能够形成一个调度

12.8.2 模调度方法

- 定义两张表
  - 每次迭代的资源预约表(RT)
  - 模资源预约表(RT_s)

$$RT_s[i] = \sum_{t|t \bmod \Delta = i} RT[t]$$

R = 目标机的资源向量, 每列对应一个资源



模调度没有资源冲突当且仅模资源预约表的每一行$RT_s[i]$所需的资源数没有超过R的限制，即$RT_s[i] \le R$ for all i=0,1,…,Δ

12.8.3 模调度方法

- 寻找最小启动间隔MII
  - 资源约束最小启动间隔(Resource MII, ResMII)

    由一次迭代的资源需求确定的MII
  - 并发约束最小启动间隔(Recurrence MII, RecMII)

    忽略资源约束情况下，由数据依赖环确定的MII

Low bound on *MII* = max(*ResMII*, *RecMII*)

12.8.3 模调度方法

- 计算ResMII
  - 假设目标机有一组资源R，每种资源r的数目为N_r，一次迭代使用r的时钟周期数为user(r)

$$ResMII = MAX(\lceil uses(r) / N_r \rceil) \quad \text{for all } r \text{ in } R$$

  - 此处假设了一个操作只在一种资源上运行，实际上要复杂一些，一个操作的执行可以有多种选择(即可以由不同的功能部件执行)

12.8.3 模调度方法

计算ResMII示例

```
1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2), r4
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop
```

- Resources
  - 4-issue, 2 ALUs, 1 MEM, 1 BR
- Latencies
  - add=1 cycle, mul =3 cycles
  - ld = 2 cycles，st = 1 cycle
  - br = 1 cycle

- ALU: used by 2, 4, 5, 6
  - (3+1+1+1)/2 = 3
- MEM: used by 1, 3
  - (2+1)/1 = 3
- BR: used by 7
  - 1 / 1 = 1

ResMII = MAX(3,3,1) = 3

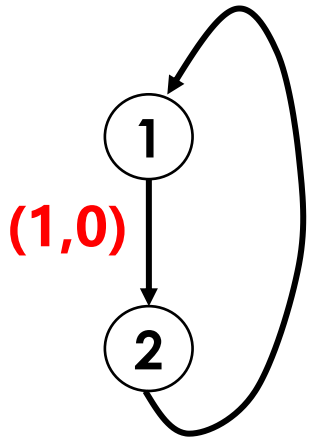
12.8.3 模调度方法

计算RecMII

- 由于存在循环携带的依赖关系，数据依赖可以形成环

$$RecMII = \max_{C \text{ a cycle in G}} \left\{ \frac{\sum_{e \in C} d_e}{\sum_{e \in C} \delta_e} \right.$$

$$\delta_e = \begin{cases} 0 & \text{loop - independent} \\ > 0 & \text{loop - carried dependent} \end{cases}$$



$\Sigma d_e = 1 + 3 = 4$

$\Sigma \delta_e = 0 + 1 = 1$

RecMII = 4/1 = 4

(delay, dependence distance)

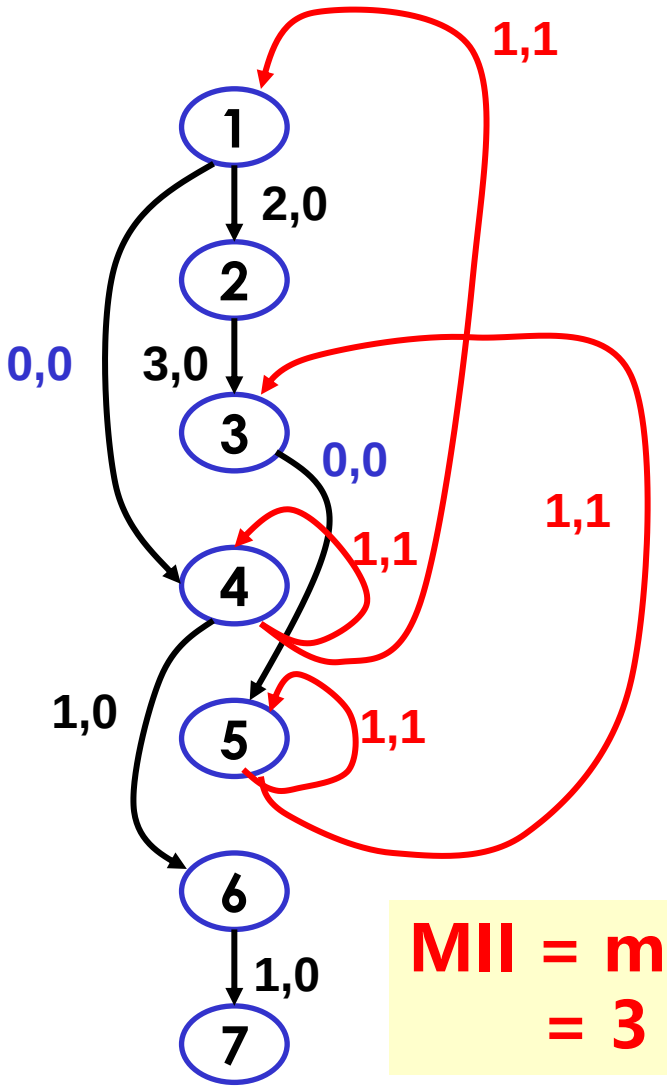
12.8.3 模调度方法

计算RecMII示例

```
1:  r3 = load(r1)
2:  r4 = r3 * 26
3:  store (r2), r4
4:  r1 = r1 + 4
5:  r2 = r2 + 4
6:  p1 = cmpp (r1 < r9)
7:  brct p1 Loop
```

- (4) : 1/1 = 1
- (5) : 1/1 = 1
- (1,4): (1+0)/(1+0) = 1
- (3,5): (1+0)/(1+0) = 1

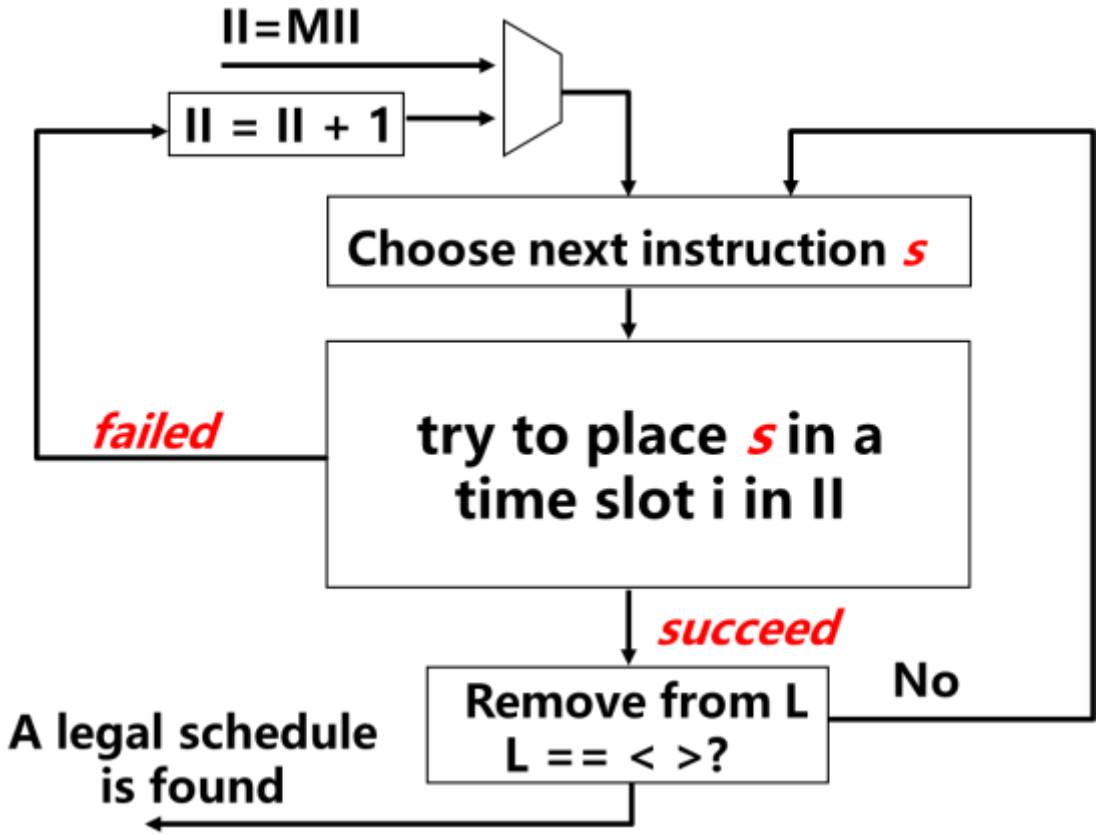
RecMII = max(1,1,1,1) = 1



MII = max(RecMII, ResMII) = 3

12.8.3 模调度方法

- 通过迭代回溯寻找服从资源约束和数据依赖约束的调度
  - 首先确定一个MII，开始尝试调度
  - 如果MII不能形成一个调度，则增加MII，重新开始调度



12.8.3 模调度方法

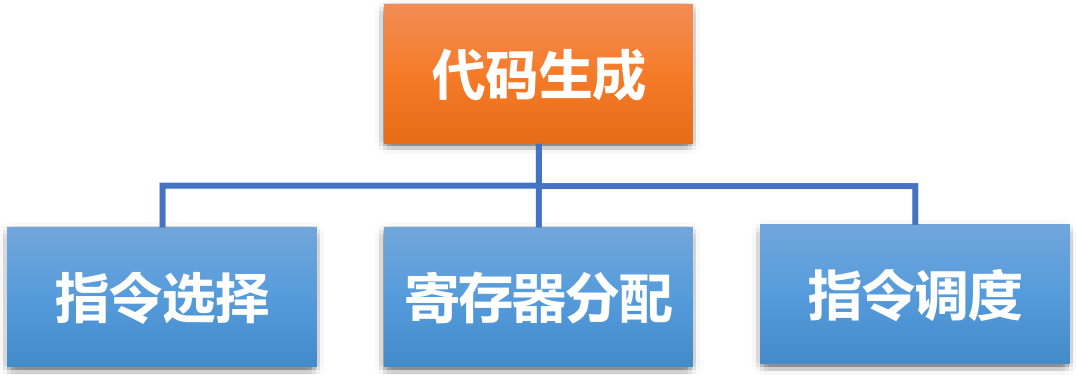
- MII不一定是一个现实可行的最小启动间隔
  - 资源的使用模式非常复杂
  - 并发指令可能因资源冲突而导致延迟
- 即使一个调度存在MII，模调度方法不一定能够找到该MII对应的调度
  - 允许有限的撤回调度和重调度
  - 对每条指令可被尝试调度的次数进行了限制

本讲小结

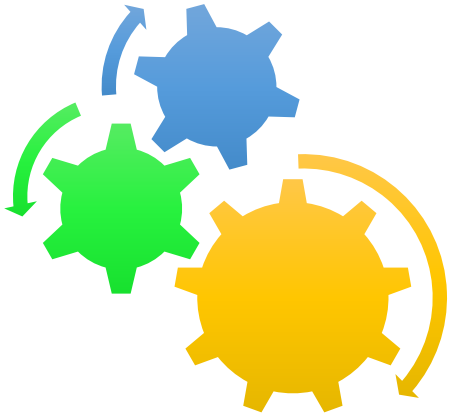
- 控制依赖与投机执行
- 跨基本块的代码移动方法
  - 向上代码移动和向下代码移动
  - src和dst之间的必经关系与后必经关系
- 轨迹调度
  - 轨迹构建，将轨迹看作是基本块调度，处理轨迹入口和出口副作用
- 软件流水
  - 针对循环的优化，将循环体的多个连续迭代中的指令组合起来

总结

- 编译后端：生成高质量目标代码 (与目标机相关的优化)



- 三个部分会相互影响
  - 指令选择影响寄存器分配和指令调度
  - 寄存器分配要重用寄存器，减少访存
  - 指令调度提高指令级并行，会增大寄存器分配压力
  - 需在降低访存延迟和提高指令级并行性之间折中
  - 指令选择→前指令调度→寄存器分配→后指令调度



先后问题?

总结

COMP CCRG Open MP

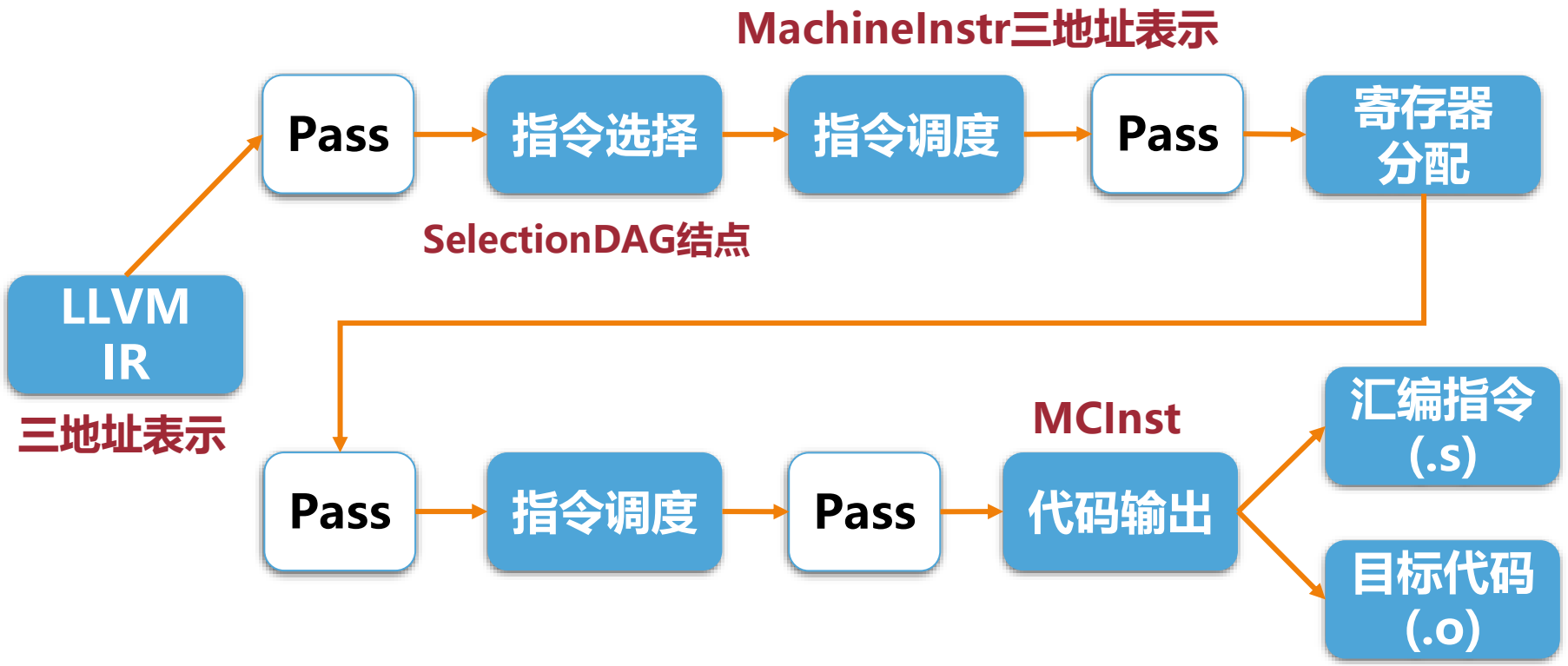
- 为什么不只在寄存器分配后才做指令调度?

virtual registers	physical registers
r1 = load(r10)	R1 = load(R1)
r2 = load(r11)	R2 = load(R2)
r3 = r1 + 4	R5 = R1 + 4
r4 = r1 – r12	R1 = R1 – R3
r5 = r2 + r4	R2 = R2 + R1
r6 = r5 + r3	R2 = R2 + R5
r7 = load(r13)	R5 = load(R4)
r8 = r7 * 23	R5 = R5 * 23
store (r8, r6)	store (R5, R2)

Too many artificial ordering constraints!!!

总结

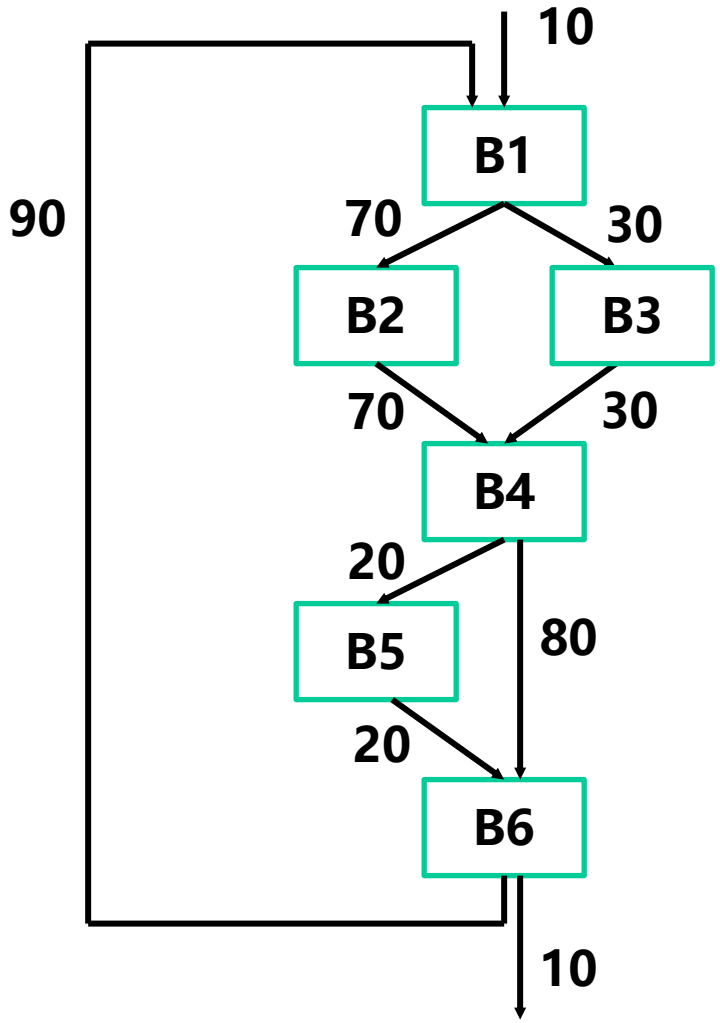
LLVM后端的各个阶段



用到了四种不同层次的指令表示: LLVM IR，SelectionDAG结点，MachineInstr，和MCInst

作业

- 对如下加权控制流图，进行轨迹调度。假设阈值为75，按照优先调度执行最频繁的轨迹的顺序，给出该控制流图的所有轨迹



参考资料

- 《高级编译器设计与实现》(鲸书) 第17章
- 《编译原理》(龙书) 第10章
- 《现代编译原理C语言描述》(虎书) 第20章
- 《编译器设计》第12章