

6.035

2005年秋

讲座14&15

指令调度时序

简单机器模式

- 指令以一定顺序执行
 - 指令获得、解码、执行，存储结果
 - 一个时间只能执行一条指令
- 对于分支指令，必要的话重新获得一个执行开始位置
 - 检查分支情况
 - 分支指令可能定义了下一条指令的位置

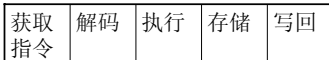
Saman Amarasinghe

6.035 ©MIT Fall 1998

2

简单执行模式

- 5步管道



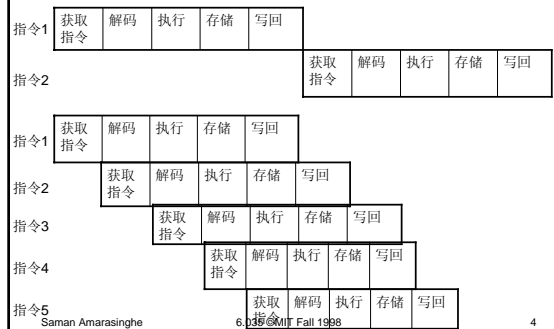
- 获取指令：获取下一条需要执行的指令
- 解码：判断指令是什么
- 执行：执行ALU操作
- 存储：写入存储器mem.op
- 写回：将结果写回

Saman Amarasinghe

6.035 ©MIT Fall 1998

3

简单执行模式



Saman Amarasinghe

6.035 ©MIT Fall 1998

4

从简单机器模式到实际机器模式

- 更多管道级
 - 奔腾 5
 - 奔腾 pro 10
 - 奔四 (130um) 20
 - 奔四 (90um) 31
- 不同的指令执行需要不同的时间
- 如果一条指令需要的结果还没有得出的话，硬件则保存管道。.

Saman Amarasinghe

6.035 ©MIT Fall 1998

5

实际机器模式内容

- 大多数现代处理器都有多个执行单元（超标量体系结构）
 - 如果指令序列是正确的，多个指令则在同一周期内执行
 - 重要的是指令序列一定要是正确的

Saman Amarasinghe

6.035 ©MIT Fall 1998

6

调配限制

- 数据相关
- 控制相关
- 资源限制

指令间的数据相关

- 如果两条指令获取同一个变量，则他们数据相关
- 相关的种类
 - 真相关：写→读
 - 反相关：读→写
 - 输出相关：写→写
- 如果两条指令相关，则
 - 指令执行顺序不可颠倒
 - 降低调度的可能性

计算相关

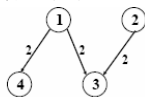
- 对于基本块，忽视具体指令，计算相关性
- 确定寄存器相关是很简单的
 - 是同一个寄存器么？
- 对于存储器访问
 - 简单：基址+偏移量1? =基址+偏移量2
 - 数据相关分析：a[2i]?=a[2i+1]
 - 程序间分析：全局? =参数
 - 指针别名分析：p1? =p

相关性描述

- 对每一个基本块，都使用一个独立DAG图（有向图）
- 网点为指令，弧代表相关性

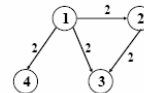
相关性描述

- 对每一个基本块，都使用一个独立DAG图（有向图）
- 网点为指令，弧代表相关性
 - 1: $r2 = *(r1 + 4)$
 - 2: $r3 = *(r1 + 8)$
 - 3: $r4 = r2 + r3$
 - 4: $r5 = r2 - 1$
- 弧上标注有延迟：
 - $v(i \rightarrow j) = i$ 和 j 初始化需要的时间差减去 i 的执行时间



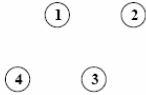
示例

```
1: r2 = *(r1 + 4)
2: r3 = *(r1 + 8)
3: r4 = r2 + r3
4: r5 = r2 - 1
```



另一个示例

```
1: r2 = *(r1 + 4)
2: *(r1 + 4) = r3
3: r3 = r2 + r3
4: r5 = r2 - 1
```



控制相关和资源限制

- 目前，我们可以只考虑基本块
- 目前，我们可以看看简单管道

示例

结果

| | |
|-----------------------|----------|
| 1: lea var a, %rax | 1 cycle |
| 2: add \$4, %rax | 1 cycle |
| 3: inc %E11 | 1 cycle |
| 4: mov 4(%rsp), %r10 | 3 cycles |
| 5: add %r10, 8(%rsp) | 4 cycles |
| 6: and 16(%rsp), %rbx | 4 cycles |
| 7: imul %rax, %rbx | 3 cycles |

| | | | | | | | | | | | |
|---|---|---|---|----|----|---|---|----|----|----|---|
| 1 | 2 | 3 | 4 | 指令 | 指令 | 5 | 6 | 指令 | 指令 | 指令 | 7 |
|---|---|---|---|----|----|---|---|----|----|----|---|

指令调度算法

- 思路
 - 做出相关DAG图的拓扑结构
 - 思考在不造成停顿的前提下，如何对指令进行调度
 - 如果不造成停顿则对指令进行调度，同时其前面的指令都已经进行了调度
- 理想执行序列为NP完全环
 - 必要时使用启发算法

指令调度算法

- 为每一个基本块创制一份相关DAG图
- 拓扑类型
 - READY = 网点没有前面的指令
 - 在READY为时空循环
 - 在无停顿的前提下对READY的网点进行调度
 - 更新READY

指令调度算法

- 从READY清单中挑选的启发式算法
 - 挑选相关图中拥有最长路径的叶节点
 - 挑选有最立即后续网点的网点
 - 挑选进入最不堵塞的管道（在超标量体系结构中）

指令调度算法

- 挑选相关图中拥有最长路径的叶节点
- 算法（对于网点x）
 - 如果没有前面的网点 $dx=0$
 - 对于x的后继网点, $dx=MAX(dy+Cxy)$
 - 翻转宽度优先访问顺序

指令调度算法

- 挑选有最立即后续网点的网点
- 算法（对于网点x）：
 - $Fx=x$ 的后续网点个数

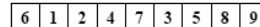
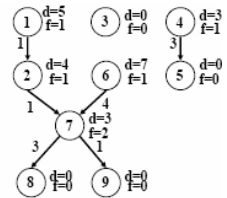
示例

结果

| | | |
|----|--------------------|----------|
| 1: | lea var a, %rax | 1 cycle |
| 2: | add \$4, %rax | 1 cycle |
| 3: | inc %r11 | 1 cycle |
| 4: | mov 4(%rsp), %r10 | 3 cycles |
| 5: | add %r10, 8(%rsp) | |
| 6: | and 16(%rsp), %rbx | 4 cycles |
| 7: | imul %rax, %rbx | 3 cycles |
| 8: | mov %rbx, 16(%rsp) | |
| 9: | lea var b, %rax | |

示例

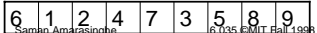
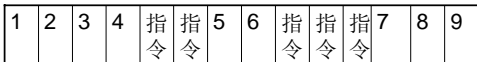
READY = { }



示例

结果

| | | |
|----|--------------------|----------|
| 1: | lea var a, %rax | 1 cycle |
| 2: | add \$4, %rax | 1 cycle |
| 3: | inc %r11 | 1 cycle |
| 4: | mov 4(%rsp), %r10 | 3 cycles |
| 5: | add %r10, 8(%rsp) | |
| 6: | and 16(%rsp), %rbx | 4 cycles |
| 7: | imul %rax, %rbx | 3 cycles |
| 8: | mov %rbx, 16(%rsp) | |
| 9: | lea var b, %rax | |



14个周期VS9个周期

资源限制

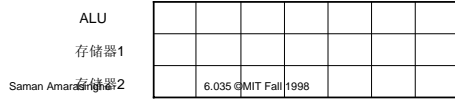
- 现代机器都会有很多的资源限制
- 超标量结构：
 - 可以运行少量的并行操作
 - 但有限制

一个超标量处理器的资源限制

- 示例
 - 一个全管道寄存器对寄存器单元
 - 所有的整数操作消耗一个周期
 - 一个全管道写存储器/读寄存器单元
 - 数据载入消耗两个周期
 - 数据存储消耗一个周期

资源限制条件下的指令调度算法

- 将超标量体系结构用多管道形式表现
 - 每个管道表示某种资源
- 示例
 - 一个单个寄存器到寄存器ALU单元
 - 一个两个周期管道写存储器/读寄存器单元



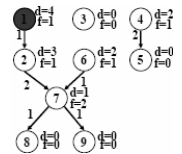
资源限制条件下的指令调度算法

- 为每一个基本块创制一份相关DAG图
- 拓扑类型
 - READY= 网点没有前面的指令
 - 在READY为空时循环
 - 假设 $n \in \text{READY}$ 标明该网点具有最高优先级
 - 将 n 调度到最早的时隙
 - 这样满足优先+资源限制
 - 更新READY

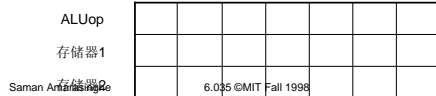
示例

```

1: lsw var_a, %eax
2: add 4(%eax), %eax
3: and %r15
4: mov 4(%eax), %e15
5: mov %e15, 8(%eax)
6: add 2(%eax), %eax
7: jmp %eax, %eax
8: lsw var_b, %eax
9: mov %eax, 16(%eax)
    
```



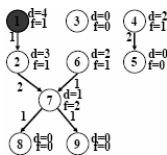
READY={1,6,4,3}



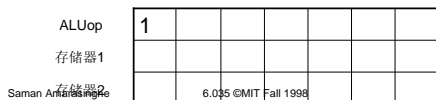
示例

```

1: lsw var_a, %eax
2: add 4(%eax), %eax
3: and %r15
4: mov 4(%eax), %e15
5: mov %e15, 8(%eax)
6: add 2(%eax), %eax
7: jmp %eax, %eax
8: lsw var_b, %eax
9: mov %eax, 16(%eax)
    
```



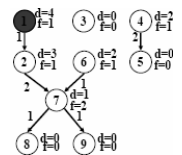
READY={1,6,4,3}



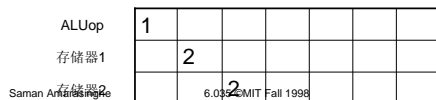
示例

```

1: lsw var_a, %eax
2: add 4(%eax), %eax
3: and %r15
4: mov 4(%eax), %e15
5: mov %e15, 8(%eax)
6: add 2(%eax), %eax
7: jmp %eax, %eax
8: lsw var_b, %eax
9: mov %eax, 16(%eax)
    
```



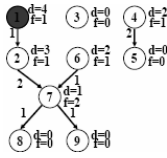
READY={1,6,4,3}



示例

```

1:  load word_0, %eax
2:  addl 4(%eax), %eax
3:  load word_1, %eax
4:  movl 4(%eax), %eax
5:  movl %eax, 0(%eax)
6:  addl 20(%eax), %eax
7:  andl %eax, %eax
8:  load word_2, %eax
9:  movl %eax, 16(%eax)
    
```



READY={1,6,4,3}

| | | | | | | | |
|-------|---|---|---|----------|---|--|--|
| ALUop | 1 | 6 | 3 | 7 | 8 | | |
| 存储器1 | 4 | 2 | 5 | | 9 | | |
| 存储器2 | 4 | 6 | 2 | MIT 1998 | | | |

Saman Amarasinghe

37

越过基本块调度

- 在一个基本块那的指令数量很小
 - 在一个基本块内调度，不能保证长管道的多个单元不闲置
- 需要处理控制相关
 - 越过基本块进行限制调度
 - 调度原则

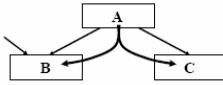
Saman Amarasinghe

6.035 ©MIT Fall 1998

38

越过基本块移动

- 向下移动来使基本块靠近



- 从A到B的路径并不执行A?

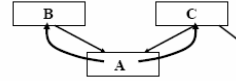
Saman Amarasinghe

6.035 ©MIT Fall 1998

39

越过基本块移动

- 向上移动来使基本块靠近



- 从A到C的路径并不执行A?

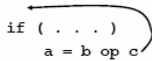
Saman Amarasinghe

6.035 ©MIT Fall 1998

40

控制相关

- 在越过基本块移动指令时的限制



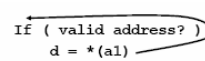
Saman Amarasinghe

6.035 ©MIT Fall 1998

41

控制相关

- 在越过基本块移动指令时的限制



Saman Amarasinghe

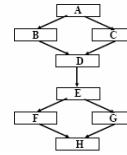
6.035 ©MIT Fall 1998

42

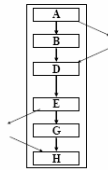
路径调度

- 找到基本块的最通用路径
- 把在路径上的基本块合并，并且将其视为一个块进行调度
- 如果执行偏离了路径，需要编制清除代码

路径调度

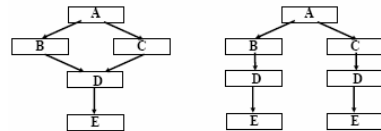


路径调度

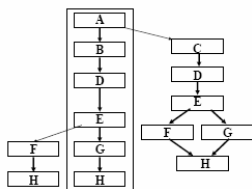


通过代码复制产生的大基本块

- 通过复制产生扩展后的大基本块
- 对大基本块进行调度



路径调度



循环调度

- 循环体通常较小
- 但是，由于大量反复的执行，在循环体上消耗了大量的时间
- 需要更好的方法来进行循环调度

循环示例

loop:

```

mov  (%rdi,%rax), %r10
imul %r11, %r10
mov  %r10, (%rdi,%rax)
sub  $4, %rax
mov  (%rdi,%rax), %rcx
imul %r11, %rcx
mov  %rcx, (%rdi,%rax)
sub  $4, %rax
bge  loop
    
```



循环解开

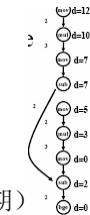
- 寄存器重命名
 - 在不同的迭代过程中使用不同的寄存器
- 清除无用的相关
 - 再次使用更多的寄存器清除真相关、反相关和输出相关
 - 有可能的话，清除相关链计算

循环示例

loop:

```

mov  (%rdi,%rax), %r10
imul %r11, %r10
mov  %r10, (%rdi,%rax)
sub  $8, %rax
mov  (%rdi,%rbx), %rcx
imul %r11, %rcx
mov  %rcx, (%rdi,%rbx)
sub  $8, %rbx
bge  loop
    
```



- 调度（每次叠代为4.5个周期）

| | | | | | | | | | |
|-----|-----|------|------|------|------|-----|-----|--|--|
| | | mov | | mov | mov | mov | | | |
| | mov | | mov | | mov | | mov | | |
| | | imul | | imul | | | bge | | |
| | | | imul | imul | | | bge | | |
| | | | | imul | imul | | | | |
| sub | | | | | sub | | | | |
| | sub | | | | | sub | | | |

软件流水操作

- 让循环间有一定重叠，这样空隙就可以被填满
- 找到稳定状态的视窗以：
 - 循环体的所有指令都被执行
 - 但是在不同的迭代过程执行

循环示例

- 汇编代码

```

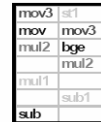
loop:
mov  (%rdi,%rax), %r10
imul %r11, %r10
mov  %r10, (%rdi,%rax)
sub  $4, %rax
bge  loop
    
```

- 调度

| | | | | | | | | | | | | | | |
|-----|-----|------|------|------|------|------|------|------|------|------|------|------|--|--|
| mov | | mov1 | | mov2 | st | mov3 | st1 | mov4 | st2 | mov5 | st3 | | | |
| | mov | | mov1 | | mov2 | mov3 | mov3 | mov1 | mov4 | mov2 | st3 | mov3 | | |
| | | mul | | mul1 | | mul2 | bge | mul3 | bge1 | mul4 | bge2 | mul3 | | |
| | | | mul | | mul1 | | mul2 | bge | mul3 | bge1 | mul4 | bge2 | | |
| | | | | mul | | mul1 | | mul2 | | mul3 | | mul4 | | |
| | | | | | sub | | sub1 | | sub2 | | sub3 | | | |
| | | | | | | sub | | sub1 | | sub2 | | sub3 | | |

循环示例

- 4次迭代都有重叠
 - %r11的值不改变
 - 对 (%rdi,%rax) 有四个寄存器进行存储
 - 每个地址以4*4增加
 - 用四个寄存器保存%r10



```

loop:
mov  (%rdi,%rax), %r10
imul %r11, %r10
mov  %r10, (%rdi,%rax)
sub  $4, %rax
bge  loop
    
```

- 在这些块中的四块产生了四块的代码后，同一个寄存器就可以重新使用了，否则就需要将寄存器中的值存储在其他位置。

软件流水操作

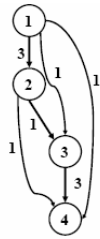
- 理想的资源使用
- 需要很多的寄存器
 - 在多次迭代中的值需要保留
- 相关性
 - 在分支指令需要在上次迭代中执行之前，需要在此次迭代中运行存储的指令
 - 数据的载入和存储是无序的（再次之前需要解决相关性）
- 代码生成
 - 生成前同步和后同步代码
 - 由于有多个块，所以无需寄存器

寄存器分配和指令调度

- 如果寄存器分配在指令调度之前的话
 - 限制调度的选择

示例

```
1: mov 4(%rbp), %rax
2: add %rax, %rbx
3: mov 8(%rbp), %rax
4: add %rax, %rcx
```



ALU操作数

| | | | | |
|------|---|---|---|---|
| | | 2 | | 4 |
| 存储器1 | 1 | | 3 | |
| 存储器2 | | 1 | | 3 |

示例

```
1: mov 4(%rbp), %rax
2: add %rax, %rbx
3: mov 8(%rbp), %r10
4: add %r10, %rcx
```



ALU操作数

| | | | |
|------|---|---|---|
| | | 2 | 4 |
| 存储器1 | 1 | 3 | |
| 存储器2 | | 1 | 3 |

寄存器分配和指令调度

- 如果寄存器分配在指令调度之前的话
 - 限制调度的选择

寄存器分配和指令调度

- 如果寄存器分配在指令调度之前的话
 - 限制调度的选择
- 如果指令调度在寄存器分配之前的话
 - 寄存器分配可能导致寄存器溢出
 - 会改编精心设计的调度方法!!!

超标量：这些晶体管都到哪去了？

- 在指令执行顺序外
 - 如果一条指令思索了，就跳过这条指令，开始执行非依赖相关的指令
 - 赞成观点：
 - 硬件调度
 - 容忍不可预测的延迟
 - 反对观点：
 - 指令窗口很小

超标量：这些晶体管都到哪去了？

- 寄存器重命名
 - 如果一个死锁管道寄存器有反相关和输出相关，则使用另外一个寄存器
 - 赞成观点：
 - 避免了反相关和输出相关
 - 反对观点：
 - 不能做更复杂的转换来清除相关

硬件vs.编译器

- 在超标量结构中，硬件和编译器调度可以很好的协调
- 在编译器不能预测的时候，硬件可以降低系统负担
- 编译器仍然可以很大的提升性能
 - 对调度的大视窗
 - 很多的程序改变增加了并行执行的可能性
- 编译器在没有硬件支持的前提下更加的关键
 - VLIW机器 (Itanium, DSPs)