编译器优化那些事儿（8）：指令调度概述

指令调度简介指令调度是指对程序块或过程中的操作进行排序以有效利用处理器资源的任务[1]。指令调度的目的就是通过重排指令，提高指令级并行性，使得程序在拥有指令流水线的CPU上更高效的运行。指令调度优化的一个必要前提就是CPU硬件支持指令并行，否则，指令调度是毫无意义的。根据指令调度发生的阶段，可以把其分为静态调度和动态调度[2]。(1)静态调度：发生在程序编译时期。静态调度由编译器完成，在生成可执行

鲲鹏小助手 · 2022-12-20 08:49:00 发布

指令调度简介

指令调度是指对程序块或过程中的操作进行排序以有效利用处理器资源的任务[1]。指令调度的目的就是通过重排指令，提高指令级并行性，使得程序在拥有指令流水线的CPU上更高效的运行。指令调度优化的一个必要前提就是CPU硬件支持指令并行，否则，指令调度是毫无意义的。

根据指令调度发生的阶段，可以把其分为静态调度和动态调度[2]。

(1)静态调度：发生在程序编译时期。静态调度由编译器完成，在生成可执行文件之前通过指令调度相关优化，完成指令重排。

(2)动态调度：发生在程序运行时期。需要提供相应的硬件支持，比如乱序执行(OoOE: out-of-order execution)，此时指令的发射顺序和执行顺序可能是不一致，但CPU会保证程序执行的正确性。

无论是静态调度还是动态调度，都是通过指令重排以提高指令流水，进而提高程序执行性能。静态调度和动态调度二者相辅相成，可以弥补对方的一些天然不足，协同完成指令流水优化，提高程序性能。本文主要介绍静态调度，如无特殊说明，后续指令调度均指静态指令调度。

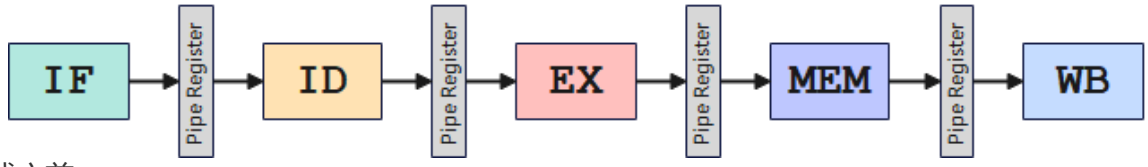
现代计算机的指令并行方案：

现代计算机的三种并行模式：流水线、超标量、多核。其中流水线和超标量与指令调度相关性更强，下面简单介绍一下这两种模式。

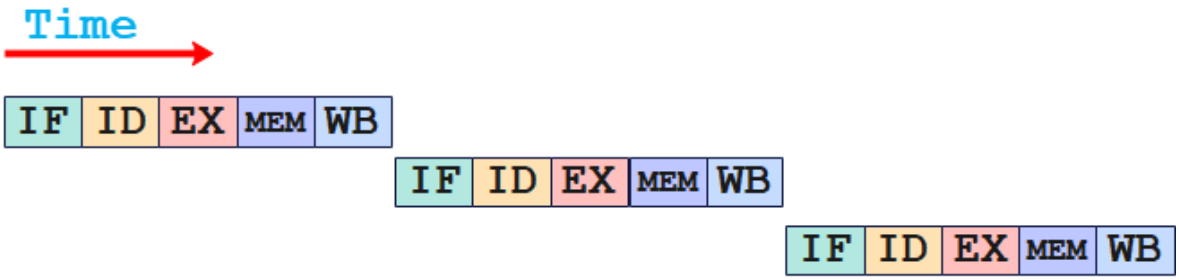
(1)流水线

将指令执行过程分成多个阶段，每个阶段使用不同的硬件资源，从而使得多条指令的执行时间可以重叠。

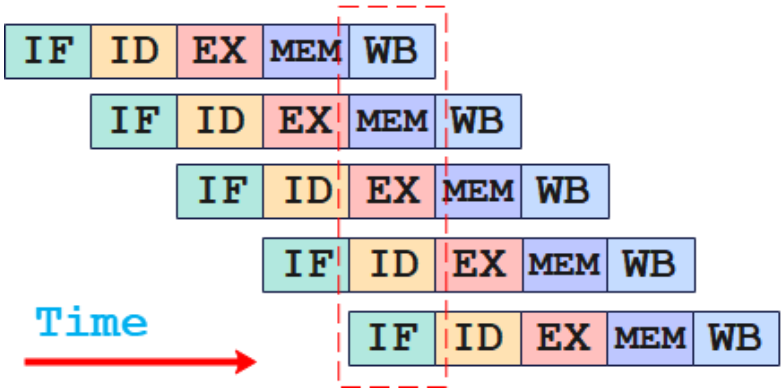
经典五段式流水线：IF(取指)、ID(译码)、EX(执行)、MEM(访存)、WB(回写)。在五段式流水线中将一条指令的执行过程分成了5个阶段。



a. 使能流水线之前



b. 使能流水线之后

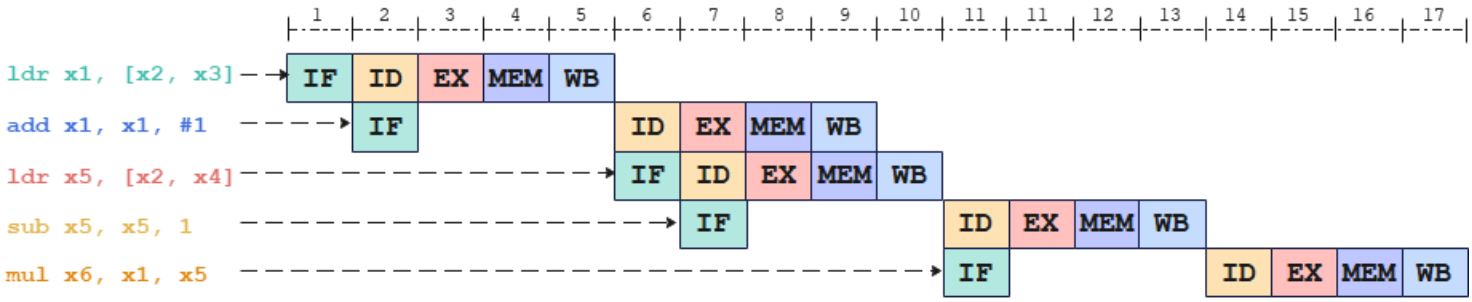


在最优情况下，一个cycle中，由于指令执行的每个阶段使用不同的硬件资源，不存在竞争关系，从而可以使每个指令执行在不同的阶段。而由于数据依赖等原因的存在，流水线的并行程度一般很难达到最优，具体的并行程度需要依赖于指令调度的效果。

对于如下原始指令序列

```
ldr    x1, [x2, x3]
add    x1, x1, #1
ldr    x5, [x2, x4]
sub    x5, x5, #1
mul    x6, x1, x5
```

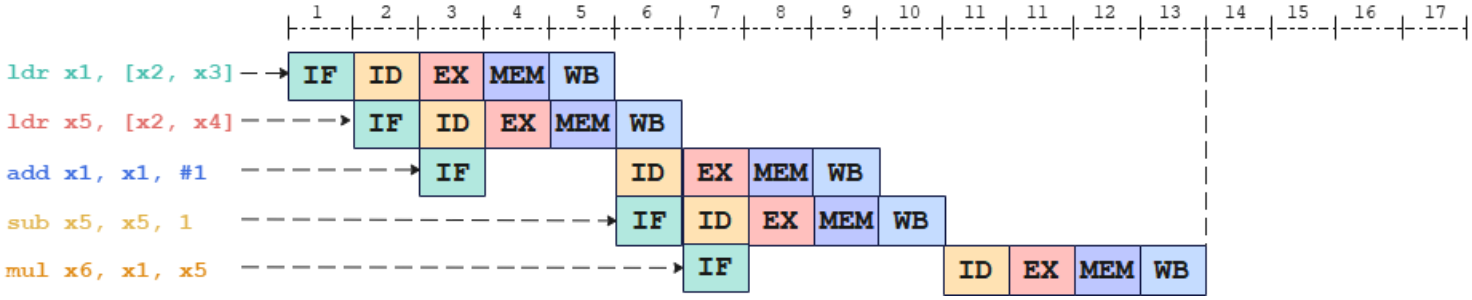
在指令调度之前，耗时17个cycle：



在指令调度之后，耗时13个cycle：

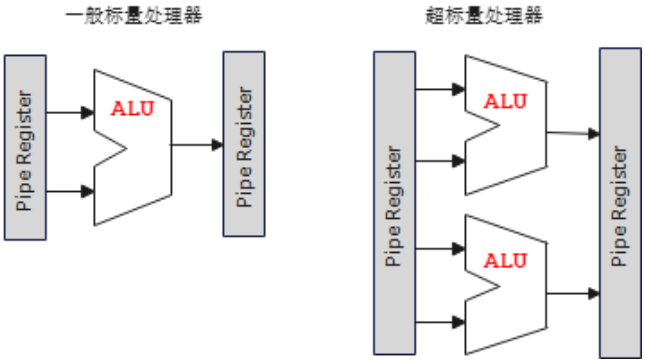
```
ldr    x1, [x2, x3]
ldr    x5, [x2, x4]
```

```
add     x1, x1, #1
sub     x5, x5, #1
mul     x6, x1, x5
```

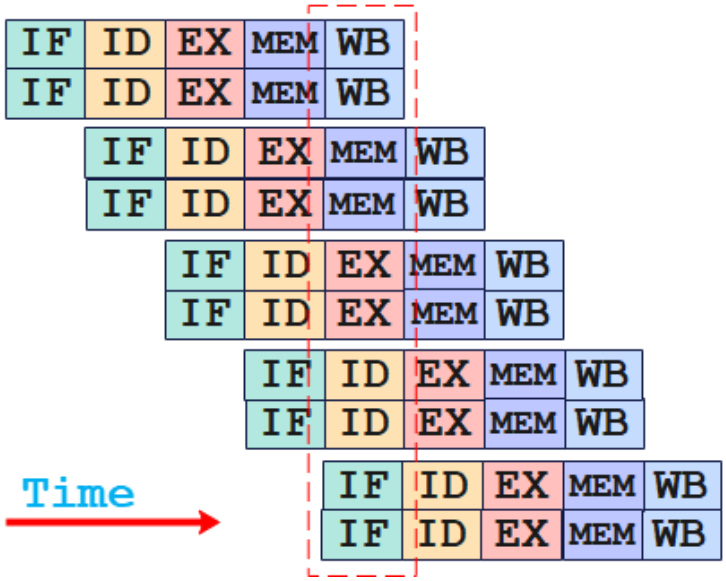


(2)超标量

具备超标量结构的CPU在一个内核上集成了多个译码器、ALU等单元。相比于具备普通流水线技术的CPU，具备超标量技术的CPU可以在同一个阶段执行多条处在相同阶段的指令。



超标量流水线：



指令调度与寄存器分配的关系：

讲到指令调度，不可避免的会想到寄存器分配，而指令调度和寄存器分配之间可以说具有相互约束、相互作用的关系。

指令调度通过重排指令顺序，降低指令间依赖，提高程序的并行度，相应的，改变指令的执行时机也会改变指令所使用的寄存器的生命周期；而寄存器分配又是挖掘程序的局部性，尽量缩短寄存器的生命周期，以能够让更多的数据直接存储在寄存器中。

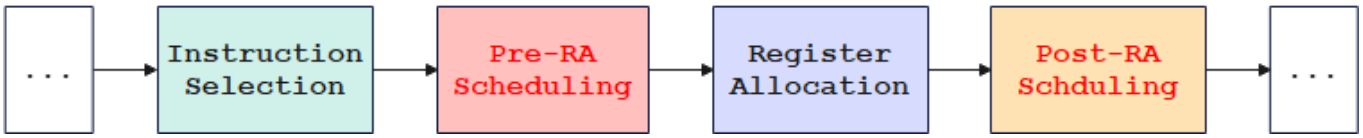
寄存器分配同样也会影响指令调度，例如当对寄存器的需求超过寄存器数量时，会选择增加一些访存指令，这些指令也需要纳入到指令调度的考虑范畴之内。

所以说两者相互约束。可以知道，将指令调度问题和寄存器分配问题作为两个约束条件进行联合求解得到的解决方案是相对更优的，但由于无论是指令调度还是寄存器分配，都是很复杂的NP完全问题，综合考虑下，编译器一般会分别处理二者[1]。

在LLVM编译器的设计中，寄存器分配之前和寄存器分配之后都会执行指令调度。

(1)寄存器分配之前执行指令调度：当前LLVM IR中分配的寄存器为虚拟寄存器，寄存器数量不受限制，此时指令调度受到的约束最小，可以更大程度上提高指令并行度。但是在寄存器分配阶段，使用物理寄存器替换虚拟寄存器，由于物理寄存器数量有限，寄存器压力增大，可能产生寄存器spill场景影响程序性能。

(2)寄存器分配之后执行指令调度：寄存器分配阶段由于寄存器复用等情况会增加指令间依赖，破坏在寄存器分配之前做好的指令调度优化，所以在寄存器分配之后还要再次执行指令调度。



指令调度的问题与约束

指令调度受到多方面的约束，如数据依赖约束、功能部件约束、寄存器约束等[3]，在这些约束下，寻找到最优解，降低指令流水间的stall，就是指令调度的终极目标。

指令流水间的stall主要由数据型冒险、结构性冒险、控制型冒险导致。

(1)数据型冒险：当前指令的执行依赖与上一条指令执行的结果。数据型冒险共有三种：写后读(RAW)、读后写(WAR)、写后写(WAW)。数据冒险可能产生数据流依赖。

(2)结构型冒险：多条指令同时访问一个硬件单元的时候，由于缺少相应的资源，导致结构型冒险。

(3)控制型冒险：存在分支跳转，无法预测下一条要执行的指令，导致其产生的控制型冒险。

编译器解决上述冒险的方法就是通过插入 NOP 指令，增加流水间的stall来化解冒险。

下面简单介绍一下三种数据型冒险(即数据依赖)：

(1)写后读(RAW)：一条指令读取前一条指令的写入结果。写后读是最常见的一种数据依赖类型，这种依赖被称为真数据依赖(true dependence)。

```
x = 1;
y = x;
```

(2)读后写(WAR)：一条指令写入数据到前一条指令的操作数。这种依赖被称为反依赖或反相关(anti dependence)。

```
y = x;
x = 1;
```

(3)写后写(WAW)：两条指令写入同一个目标。这种依赖被称为输出依赖(output dependence)。

```
x = 1;
x = 2;
```

指令调度算法之表调度(List Scheduling)

表调度是一种贪心+启发式方法，用以调度基本块中的各个指令操作，是基本块中指令调度的最常见方法。基于基本块的指令调度不需要考虑程序的控制流，主要考虑数据依赖、硬件资源等信息。

表调度的基本思想：维护一个用来存储已经准备执行的指令的ready列表和一个正在执行指令的active列表，ready列表的构建主要基于数据依赖约束和硬件资源信息；根据调度算法以周期为单位来执行具体的指令调度，包括从列表中选择及调度指令，更新列表信息。

基本的表调度算法大致分为以下三步：

(1)根据指令间依赖，建立依赖关系图。

(2)根据当前指令节点到根节点的长度以及指令的latency，计算每个指令的优先级。

(3)不断选择一个指令，并调度它，

a. 使用两个队列维护ready的指令和正在执行的active的指令；

b. 在每个周期：选择一个满足条件的ready的指令并调度它，更新ready队列；检查active的指令是否执行完毕，更新active列表。

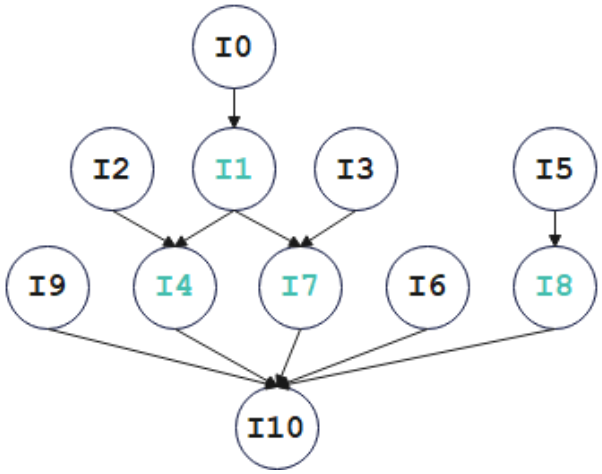
指令调度案例[4]

本案例选自卡内基梅隆大学(Carnegie Mellon University)的Compiler Design课程。

假设当前CPU有两个计算单元(即每个周期可以执行两条指令)；加法指令的latency为 2 cycles，其他指令为 1 cycle。

(1)根据数据依赖关系构建出依赖关系图。

```
I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1
```

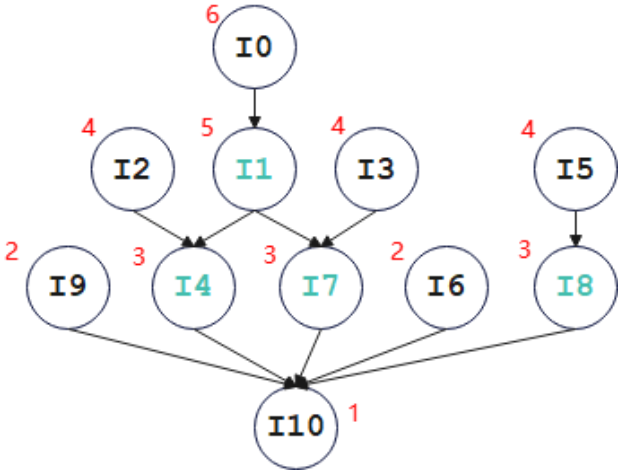


(2)计算指令节点优先级

优先级计算公式如下：

$$priority(x) = \begin{cases} latency(x) & \text{if x is a leaf} \\ max(latency(x) + max_{(x,y)\in E}(priority(y)), max_{(x,y)\in E'}(priority(y))) & \text{otherwise} \end{cases}$$

其中，x表示当前指令节点，y表示x的子节点，E表示 "true dependency"，E'表示 "anti-dependency"。



其中 I10 为叶节点，优先级为其latency，故结果为1；I4 为非叶节点，优先级为当前节点latency(I4 为加法指令，latency为2)+ 子节点的优先级，故结果为3。本例中无反依赖(anti-dependency)情形。

(3)执行调度

		Cycle		
I0	I2	0	I0	I2
I1	I5	1	I1	I3
I3	I8	2	I5	I6
I4	I7	3	I4	I7
I6	I9	4	I8	I9
I10	---	5	---	---
		6	I10	---

在实际执行调度时，对于同等优先级的指令，由于具体调度方案的不同，会出现不同的情况，例如本例中出现的场景，可以通过添加其他度量标准进一步优化优先级计算方案。尽管表调度方法不能保证得到最优调度结果，但它是接近最优解的。

本文只是简单介绍了最基本的表调度方法，在实际应用中，存在各种基于该方法的改进方案。关于LLVM编译器中的表调度算法，可以先自行阅读其源码，更多相关介绍，敬请期待。

结语

本文简单介绍了指令调度的基本概念，指令调度的原因与影响以及基本的指令调度算法。

指令调度作为NP完全问题目前依旧尚未有一个完美的解决方案，对指令调度算法的探索与优化尚有很大的发展空间。

LLVM之父Chris Lattner认为“编译器的黄金时代”已经降临[5]。随着计算机架构的复兴，未来的N年里将是每一位编译器工程师大显身手的时代。

参考

1. Keith D. Cooper, Linda Torczon. Engineering a Compiler (Second Edition).

2. https://zhuanlan.zhihu.com/p/360364235

3. Andrew W.Apple, Maia Ginsburg. Modern Compiler Implementation in C.

4. https://www.cs.cmu.edu/afs/cs/academic/class/15745-s19/www/lectures/L18-Instruction-Scheduling-pre-class.pdf

5. https://zhuanlan.zhihu.com/p/502730940