13 后端优化-ARM

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆ 掌握指令调度问题和方法
- ⋆ 掌握窥孔优化方法
- 了解并行优化

13.1 指令调度

13.1.1 指令调度问题

现代 CPU 普遍采用流水线结构执行指令。一条指令通常需要经历取指、译码、执行、访存、回写等多个阶段。当一条指令进入后续阶段后，前面的硬件资源即可用于处理新的指令，因此多条指令能够在不同阶段同时执行。这种现象称为指令级并行（Instruction-Level Parallelism，ILP）。然而，不同指令的执行开销存在较大差异。例如，整数加法通常只需 1 个周期即可完成，而访存、乘法和除法等操作往往具有更长的执行时延。同时，程序中的指令之间还存在数据依赖关系，部分指令必须等待其前驱指令完成后才能执行。这些因素都会限制指令级并行性的发挥。因此，编译器后端需要研究如下问题：在保证程序语义不变且不违反数据依赖关系的前提下，如何调整指令的执行顺序，从而减少流水线停顿，提高处理器资源利用率，并缩短程序执行时间。这个问题称为指令调度（Instruction Scheduling）。

以 ARM-A72 官方软件优化手册给出的性能数据为参照 [1]，我们得到常用指令的运行时延和吞吐大致如表 13.1 所示。

表 13.1: ARM-A72 指令开销

指令	时延	吞吐	执行单元
`ldr`	4	1	L
`str`	1	1	S
`add`	1	2	I0/I1
`sub`	1	2	I0/I1
`mul`	3	1	M
`madd`	3	1	M
`sdiv`	7	1/7	M
`mov`	1	2	I0/I1
`adr`	1	2	I0/I1
`b`	1	1	B
`bl`	1	1	B, I0/I1
`ret`	1	1	B

需要注意，时延（Latency）和吞吐（Throughput）是两个不同的概念。时延表示一条指令从开始执行到结果可被后续指令使用所需的周期数；吞吐表示处理器平均每个周期最多能够发射多少条该类指令。例

如，ARM-A72 中的乘法指令时延约为 3 个周期，但乘法流水线支持每周期启动一条新的乘法指令，因此其吞吐仍可达到 1。表中的加载时延是假设数据命中 L1 Cache 时的典型情况。若发生 Cache Miss，实际时延可能远高于表中数据。其中，数据加载和乘除运算的时延较高，加载和乘法的吞吐基本不受时延高的影响，但除法的吞吐非常低。加法和减法指令具有两个执行单元 I0 和 I1，因此吞吐可以翻倍；A72 中其它执行单元只有一个。

需要注意，不同 CPU 微结构的执行资源配置可能存在较大差异。例如 ARM Cortex-A77 相比 A72 增加了多个执行单元，从而能够支持更高的指令并行度。因此，实际编译器往往需要针对目标处理器选择不同的调度策略。

13.1.2 指令重排

本节主要讨论数据依赖对指令调度的影响。假设程序中存在两条指令 A 和 B，且 A 先于 B 执行，则两条指令之间可能存在如下依赖关系：

- 真依赖（Read After Write，RAW）：A 的运算结果是 B 的操作数，B 必须等待 A 执行完成后才能执行。
- 反依赖（Write After Read，WAR）：B 会改写 A 读取的操作数，因此 B 必须等待 A 完成读取后才能执行。
- 输出依赖（Write After Write，WAW）：A 和 B 都会写入同一个寄存器或存储位置，为保证最终结果正确，必须保持二者的写入顺序。

其中，真依赖反映了程序实际的数据流关系，无法通过寄存器重命名消除；反依赖和输出依赖属于名称依赖，通常可以通过寄存器重命名技术消除。

表 13.2: 一段汇编代码和开销分析示例

编号	指令	开始时间	结束时间
I1	`ldr x9, [sp, #-12]`	1	4
I2	`ldr x10, [sp, #-16]`	2	5
I3	`add x9, x9, x10`	6	6
I4	`ldr x10, [sp, #-20]`	7	10
I5	`ldr x11, [sp, #-24]`	8	11
I6	`sdiv x11, x10, x11`	12	18
I7	`str x11, [sp, #-24]`	19	19
I8	`ldr x10, [sp, #-28]`	20	23
I9	`mul x10, x9, x10`	24	26
I10	`str x10, [sp, #-28]`	27	27

下面我们以代码 13.2 为例分析其中的指令依赖关系和时延。由于指令 `I3` 的操作数 `x9` 和 `x10` 分别是指令 `I1` 和 `I2` 的执行结果，因此指令 `I3` 真依赖于 `I1` 和 `I2`。指令 `I4` 会更新 `I3` 读取的寄存器 `x10`，因此 `I4` 反依赖于 `I3`。基于该方法对所有指令间的依赖关系进行分析，我们可得到图 13.1 。

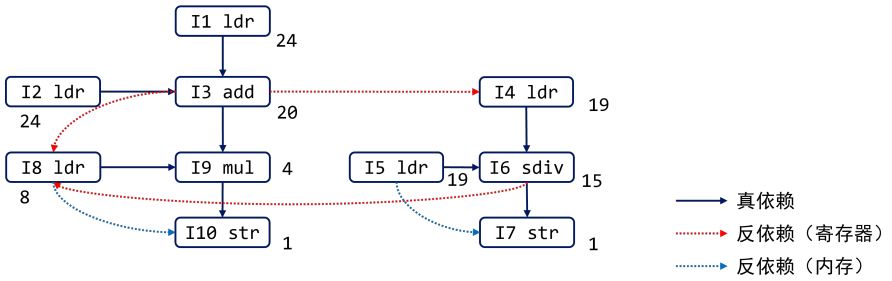

图 13.1: 代码 13.2 中指令间的依赖关系和时延

为了评估一条指令对程序整体运行时间的影响，我们定义程序时延（Critical Path Length，CPL）为：从当前指令开始执行到程序结束所需的最长时间。对于没有后继节点的指令，其程序时延等于自身时延；对于其它指令，其程序时延等于自身时延与所有后继节点程序时延最大值之和，即

$$CPL(i) = Latency(i) + \max_{j \in Succ(i)} CPL(j)$$

其中，(Succ(i)) 表示指令 (i) 的所有后继节点。

在图 13.1 中，指令 `I7` 和 `I10` 没有后继节点，因此其程序时延均为 1。指令 `I9` 的自身时延为 3，且依赖于 `I10`，因此其程序时延为 4；同理，指令 `I8` 的程序时延为 8。指令 `I6` 同时依赖于 `I7` 和 `I8`，因此其程序时延为 15，即自身时延 7 加上两条后继路径中较长的一条。按照上述方法自底向上计算所有节点的程序时延后，可以发现程序的执行瓶颈主要集中在程序时延较高的指令上。程序时延较高的指令位于关键路径附近，其延迟更容易影响整个程序的完成时间。因此在调度过程中，通常优先选择程序时延较高且已满足依赖关系的指令执行，以尽可能缩短关键路径长度。

根据程序时延计算结果，可得到各指令的优先级顺序：`I1=I2>I3>I4=I5>I6>I8>I9>I7=I10` 据此对代码进行重排，可得到表 13.3 中的结果。重排前程序的执行周期数为 27，重排后降低至 26，说明合理的指令调度能够在不改变程序语义的前提下提高处理器资源利用率并缩短程序运行时间。

表 13.3: 程序 13.2 指令重排后的结果

编号	指令	开始时间	结束时间
I1	`ldr x9, [sp, #-12]`	1	4
I2	`ldr x10, [sp, #-16]`	2	5
I3	`add x9, x9, x10`	6	6
I4	`ldr x10, [sp, #-20]`	7	10
I5	`ldr x11, [sp, #-24]`	8	11
I6	`sdiv x11, x10, x11`	12	18
I8	`ldr x10, [sp, #-28]`	19	22
I9	`mul x10, x9, x10`	23	25
I7	`str x11, [sp, #-24]`	24	24
I10	`str x10, [sp, #-28]`	26	26

13.1.3 消除反依赖

进一步对上述程序的性能瓶颈进行分析，可以发现 `I6` 与 `I8` 之间存在反依赖关系。由于 `I8` 会写入寄存器 `x10`，而 `I6` 需要读取 `x10` 的值，因此 `I8` 必须等待 `I6` 完成后才能执行。同理，`I4` 与 `I3` 之间也存在类似的反依赖关系。这类依赖并非真实的数据流依赖，而是由于多条指令复用了同一个寄存器名称所

导致的名称依赖。对于这类依赖，可以通过寄存器重命名（Register Renaming）技术消除，即为后续写入操作分配新的寄存器，从而避免不必要的顺序约束。

例如，将 `I4` 中的寄存器 `x10` 替换为 `x12`，并相应修改其后续使用位置；同时将 `I8` 中的寄存器 `x10` 替换为 `x13`。经过寄存器重命名后，原有的反依赖关系被消除，更多指令可以并行执行。

表 13.4: 程序 13.3 寄存器重命名后的结果

编号	指令	开始时间	结束时间
I1	`ldr x9, [sp, #-12]`	1	4
I2	`ldr x10, [sp, #-16]`	2	5
I3	`add x9, x9, x10`	6	6
I4	`ldr x12, [sp, #-20]`	7	10
I5	`ldr x11, [sp, #-24]`	8	11
I6	`sdiv x11, x12, x11`	12	18
I8	`ldr x13, [sp, #-28]`	13	16
I9	`mul x13, x9, x13`	17	19
I7	`str x11, [sp, #-24]`	20	20
I10	`str x13, [sp, #-28]`	21	21

寄存器重命名后，部分反依赖边被消除，程序中的关键路径也随之发生变化。我们对表 13.4 中的指令重新进行依赖关系和程序时延分析，得到图 13.2 所示的结果。此时各指令的程序时延优先级变为：`I4=I5>I1=I2>I6=I8>I3>I9>I7=I10`

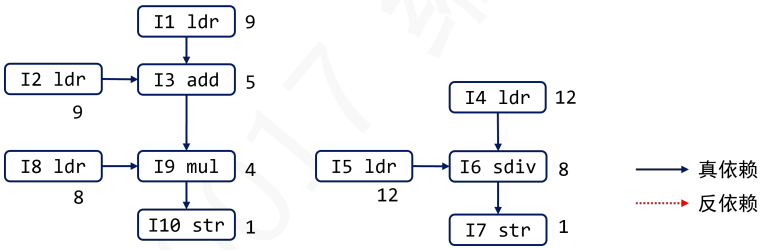

图 13.2: 代码 13.4 中指令间的依赖关系和时延

接下来根据新的程序时延优先级再次进行指令调度，得到表 13.5 中的结果。需要注意的是，当多条指令具有相同的程序时延时，仅依据程序时延无法唯一确定调度顺序，还需要结合后续资源竞争和关键路径情况进行选择。在本例中，若交换 `I6` 与 `I8` 的顺序，或交换 `I7` 与 `I10` 的顺序，都会导致最终调度结果变差，从而增加程序的总执行周期数。

表 13.5: 程序 13.4 指令重排后的结果

编号	指令	开始时间	结束时间
I4	`ldr x12, [sp, #-20]`	1	4
I5	`ldr x11, [sp, #-24]`	2	5
I1	`ldr x9, [sp, #-12]`	3	6
I2	`ldr x10, [sp, #-16]`	4	7
I8	`ldr x13, [sp, #-28]`	5	8
I6	`sdiv x11, x12, x11`	6	12
I3	`add x9, x9, x10`	8	8
I9	`mul x13, x9, x13`	9	11
I10	`str x13, [sp, #-28]`	12	12
I7	`str x11, [sp, #-24]`	13	13

与原始代码相比，经过寄存器重命名和指令重排后，程序的执行周期数由 27 进一步降低至 13。该例子说明，消除名称依赖能够显著增加可调度指令的数量，从而提高指令级并行性并改善程序性能。

13.1.4 应用考量

前面的分析仅局限于单个基本块内部的局部调度问题。实际编译器还会考虑跨基本块甚至跨循环的数据依赖关系，并结合控制流信息进行全局指令调度，以获得更好的优化效果。此外，在真实处理器中，指令执行时间不仅受到数据依赖的影响，还会受到缓存命中率、分支预测准确率以及执行单元竞争等因素的影响，因此实际调度策略往往比本章介绍的方法更加复杂。现代处理器通常还具备超标量发射、乱序执行和分支预测等动态优化机制，可以在运行时进一步挖掘指令级并行性。其中，以 Tomasulo 算法 [2] 为代表的乱序执行技术能够在硬件层面动态完成指令调度和寄存器重命名，从而隐藏部分数据相关带来的延迟。

然而，动态调度并不能完全替代编译器优化。处理器只能依据运行时可见的信息进行决策，而编译器能够利用程序的全局信息提前消除冗余依赖、调整指令布局并降低运行时调度开销。因此，静态指令调度仍然是现代编译器后端的重要优化手段。编译器生成的代码质量越高，处理器就越容易发挥其硬件并行能力。有兴趣的同学可以进一步思考和比较静态调度与动态调度两种方法的优缺点。

13.2 窥孔优化

由于指令选择和寄存器分配的主要目标是实现与 IR 语义等价的翻译，而非直接生成最优代码，因此在代码生成过程中不可避免地会引入一些明显的冗余指令。窥孔优化（Peephole Optimization）是一种基于局部模式匹配的后端优化技术，其目标是在较小的指令窗口内识别这些冗余模式，并将其替换为功能等价但效率更高的指令序列。

窥孔优化通常不依赖复杂的数据流分析，实现简单、开销较低，因此被广泛应用于实际编译器中。具体的冗余模式与前端 IR 设计、指令选择策略以及寄存器分配算法密切相关。下面列举几种常见的冗余模式及其对应的窥孔优化方法。

13.2.1 冗余 mov

窥孔优化一般根据指令数窗口大小涉及不同的冗余模式。单条指令冗余模式主要是同一寄存器之间的 mov，可以直接删除。

```
mov x2, x2
```

对于窗口大小为 2 的情况，如果后一条指令重新定义了某个寄存器，且前一条指令产生的结果在此之前没有被使用，则前一条指令属于死定义，可以删除。

例如，对于连续的两条 mov 指令，第二条指令覆盖了第一条指令对寄存器 x10 的写入，因此第一条 mov 指令可以删除。

```
mov     x10, #0
mov     x10, x9
```

类似地，在 ldr-mov 模式中，若加载得到的值未被使用且随后被新的定义覆盖，则 ldr 指令可以删除。

```
ldr     x10, [sp]
mov     x10, x9
```

对于窗口大小为 3 的情况，还可能出现短距离的数据传递链。例如：

```
mov     x10, #0
mov     x11, x10
mov     x10, x9
```

仅观察任意两条相邻指令时，上述代码均不存在可直接删除的冗余指令。然而，若同时观察三条指令，则可以发现寄存器 x10 中的常量值仅用于向 x11 传递一次，随后立即被重新定义。因此可以将常量直接传播到 x11，从而消除中间的数据传递过程。类似的模式还包括 ldr-mov-mov、ldr-mov-ldr 等情况。

13.2.2 冗余 store-load

对于窗口大小为 2 的情况，如果连续出现 str-ldr 指令，且两条指令访问同一个内存位置，则后续的 ldr 指令可以替换为寄存器之间的 mov 操作。

```
str     x9, [sp]
ldr     x10, [sp]
```

由于 ldr 读取的值必然来自前一条 str 写入的数据，因此无需再次访问内存，只需将寄存器 x9 中的值传递给 x10 即可。若加载目标寄存器与存储源寄存器相同，则 ldr 指令可以进一步删除。

13.3 利用 CPU 特性优化

除了通过指令调度和窥孔优化改进已有代码外，编译器还可以利用目标处理器提供的特殊硬件特性进一步提升程序性能。这类优化通常与具体处理器架构密切相关，因此具有较强的平台相关性。

本节介绍两类典型的 CPU 特性：数据预取（Prefetch）和 SIMD。

13.3.1 数据预取

处理器访问内存的开销通常远高于寄存器运算。当程序访问的数据不在缓存中时，需要从更低层次的存储系统中加载数据，从而产生较大的访存延迟。数据预取的基本思想是在数据真正被使用之前提前发起加载请求，使数据尽可能提前进入缓存，从而减少后续访存等待时间。

在 AArch64 中，数据预取指令为 `PRFM`。预取的基本单位通常为一个 Cache Line。通过不同的参数可以指定预取到 L1 或 L2 Cache，以及读预取或写预取等不同策略。

```
; PRFM <type>, [<base>, <offset>]

PRFM PLDL1KEEP, [x0, #256]    ; 提前预取未来数据

... ; 若干指令

ldr  w1, [x0, #256]           ; 实际访问该数据

; PLDL2KEEP: 读操作，预取到L2 Cache
; PSTL1KEEP: 写操作，预取到L1 Cache
; PSTL2KEEP: 写操作，预取到L2 Cache
```

上述代码展示了软件预取的基本思想：在数据真正被访问之前，通过 `PRFM` 提前向缓存系统发出加载请求。在预取指令与实际访存指令之间插入足够多的其它计算或访存操作，使得数据有时间从下层存储系统加载到缓存中。当后续执行 `ldr` 时，数据有较大概率已经位于 Cache 中，从而降低 Cache Miss 带来的访存延迟。

数据预取需要准确预测未来的访存行为。如果预取距离过近，则数据尚未完成加载；如果预取距离过远，则数据可能在真正使用前被缓存淘汰。因此，预取距离通常需要根据程序的访存模式和处理器特性进行调整。由于自动分析访存行为较为困难，实际编译器对此类优化通常较为保守，在性能关键代码中往往需要程序员根据具体算法特征进行辅助优化。

13.3.2 SIMD 指令

SIMD（Single Instruction Multiple Data）是一种数据级并行技术。其基本思想是利用一条指令同时完成多组数据的相同运算，从而提高处理器吞吐率。

AArch64 提供的 SIMD 扩展称为 NEON [3]。NEON 包含 32 个 128 位向量寄存器 `v0-v31`。例如，一个 128 位寄存器可以同时存储 4 个 32 位整数，因此一次向量加法指令即可完成 4 组整数加法运算。

下面的代码实现了两个长度为 4 的 32 位整数向量加法：

```
ldr q0, [x8, _x@PAGEOFF]
ldr q1, [x8, _y@PAGEOFF]
add.4s v0, v1, v0
```

与标量代码相比，SIMD 能够显著提高循环、矩阵运算、图像处理以及机器学习等场景中的计算性能。现代编译器能够在部分情况下自动完成向量化，将普通循环转换为 SIMD 指令。但自动向量化受到数据

依赖关系和程序结构的限制，因此在性能要求较高的场景下，程序员仍常通过 C 语言提供的库手动编写 SIMD 代码。

参考文献

[1] Arm Cortex-A72 Software Optimization Guide, `https://developer.arm.com/documentation/uan0016/latest/`.

[2] Robert M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units." IBM Journal of Research and Development, 1967.

[3] Neon Programmer Guide for Armv8-A Coding for Neon 4.0, `https://developer.arm.com/documentation/102159/0400`, 2023

练习

1) 下列 ARM 代码是编译时未进行任何优化得到的，分析其中的冗余和产生原因，手动对这段代码进行优化。

```
LBB0_0:
  str x0, [sp, #24]
  str x1, [sp, #16]
  str 0, [sp, #8]
  b LBB0_1
LBB0_1:                                 ;
  ldr x8, [sp, #8]
  ldr x9, [sp, #16]
  cmp x8, x9
  b.ge  LBB0_3
  b LBB0_2
LBB0_2:
  ldr x11, [sp, #8]
  ldr x9, [sp, #24]
  ldr x10, [sp, #8]
  add x8, x10, #1
  str x8, [sp, #8]
  ldr x8, [x9, x10, lsl #2]
  add x8, x8, x11
  str w8, [x9, x10, lsl #2]
  ldr x11, [sp, #8]
  ldr x9, [sp, #24]
  ldr x10, [sp, #8]
  add x8, x10, #1
  str x8, [sp, #8]
  ldr x8, [x9, x10, lsl #2]
  add x8, x8, x11
  str w8, [x9, x10, lsl #2]
  b LBB0_1
LBB0_3:
  add sp, sp, #32
  ret
```

2) 设计程序样例，使得指令调度算法可以在支持乱序执行的 CPU 上可以产生明显优化效果，并通过实验验证。

3) 设计程序样例，使得数据预取可以产生明显优化效果，并通过实验证明。