第10章　指令级并行性

每一个现代高性能处理器都能够在一个时钟周期内执行多条指令。在一个具有指令级并行机制的处理器上一个程序能够以多快的速度运行？这可是一个“价值十亿美元的问题”。对这个问题的回答要考虑下列因素：

1）该程序中潜在的并行性。

2）该处理器上可用的并行性。

3）从原来的顺序程序中抽取并行性的能力。

4）在给定的指令调度约束之下找到最好的并行调度方案的能力。

如果一个程序中的所有运算之间都是高度依赖的，那么再多的硬件或采用并行化技术都无法使这个程序快速并行执行。关于并行化的限制方面已经有了很多研究。典型的非数值应用有很多固有的依赖性。比如，这些程序具有很多依赖于数据的分支，使得哪怕预测一下下面将执行哪条指令都变得很困难，更不要说去决定哪些运算可以并行执行了。因此，这个领域中的研究工作集中在放松调度约束的技术，包括引入新的体系结构特性，而不是调度技术本身。

数值应用（比如科学计算和信号处理）往往具有更好的并行性。这些应用处理大型的聚合数据结构。在该结构的不同元素上的运算通常是相互独立的，可以并行地执行。在高性能通用机器和数字信号处理器中都提供了附加的硬件资源来利用这些并行性。这些程序通常具有简单的控制结构和规则的数据访问模式。已经有一些静态技术可以用来从这些程序中抽取出可用的并行性。这类应用的代码调度很有意思也很重要，因为它们允许大量的独立运算被映射到大量的资源上运行。

并行性抽取和并行执行的调度可以通过软件静态完成，也可以通过硬件动态进行。实际上，即使是具有硬件调度机制的机器也可以辅以软件调度。本章将首先解释使用指令级并行性的一些基本问题。不管是硬件管理的并行性还是软件管理的并行性，这些问题都是一样的。然后我们给出并行性抽取所需的基本数据依赖性分析。这些分析也可用于指令级并行性之外的其他优化。我们将在第11章中看到这些分析技术的其他应用。

最后，我们给出代码调度中的基本思想。我们将描述一个用于基本块调度的技术，并给出一个方法来处理通用程序中高度数据依赖的控制流，最后给出一个称为“软件流水线化”的技术。软件流水线化技术主要用于数值计算程序的调度。

10.1　处理器体系结构

当我们考虑指令级并行性的时候，通常设想的是一个在每个时钟周期内发出多条运算指令的处理器。实际上，如果使用流水线（pipelining）的概念，即使一个机器每个时钟周期[⊖]发送一条运算指令，我们仍然能够得到指令级并行性。下面，我们将首先解释流水线的概念，然后再讨论多指令发送。

⊖ 在含义明确的时候，我们将把时钟“嘀嗒”或者时钟周期简称为“时钟”。

10.1.1 指令流水线和分支延时

在实践中，不管是高性能超级计算机还是普通的机器，每个处理器都使用指令流水线(instruction pipeline)。使用指令流水线，每个时钟周期都可以取得一个新指令，而此时前面的指令还在流水线中执行。图 10-1 显示的是一个简单的 5 阶段指令流水线：它首先获取指令(IF)，对该指令解码(ID)，执行运算(EX)，访问内存(MEM)，然后回写结果(WB)。该图显示了指令 i、$i+1$、$i+2$、$i+3$ 和 $i+4$ 是如何在同一时刻并行运行的。图中的每一行对应于一个时钟周期，而每一列指明了各条指令在各时钟周期中所在的阶段。

	i	$i+1$	$i+2$	$i+3$	$i+4$
1.	IF				
2.	ID	IF			
3.	EX	ID	IF		
4.	MEM	EX	ID	IF	
5.	WB	MEM	EX	ID	IF
6.		WB	MEM	EX	ID
7.			WB	MEM	EX
8.				WB	MEM
9.					WB

图 10-1 在一个 5 阶段指令流水线中的五个连续指令

如果在后续指令需要某条指令的结果时此结果已经可用，那么处理器就可以在每个时钟周期内发出一条指令。分支指令特别容易出现问题，因为只有在它们被获取、解码并执行之后，处理器才能够知道下面该执行哪条指令。很多处理器假设分支不会跳转，投机性地选取下一条指令并解码。但是当这个分支真的需要跳转的时候，指令流水线被清空并获取分支跳转的目标指令。因此，分支跳转引入了为获取分支跳转目标而引起的延时，并使得指令流水线“打嗝”。先进的处理器使用硬件根据分支运行的历史来预测它们的结果，并从预测的目标位置预取指令。但是如果分支预测错误，依然会出现分支延时。

10.1.2 流水线执行

有些指令的执行需要几个时钟周期。一个常见的例子是内存加载运算。即使某次内存访问的目标数据已经在高速缓存中，高速缓存仍然需要多个时钟周期才会返回数据。如果一条指令的后继指令在不需要该指令的运算结果时可以立刻往下执行，我们就说该指令的执行被流水线化(pipelined)了。因此，即使一个处理器在每个时钟周期内只能发送一条指令，但仍然可能在同一时刻有多条指令在它们各自的阶段上执行。如果最深的执行流水线有 n 个阶段，那么在同一时刻最多可允许 n 条指令处于执行状态。请注意，不是所有的指令都是完全流水线化的。虽然浮点数加法和乘法通常都被完全地流水线化了，但更加复杂且很少执行的浮点数除法却没有做到这一点。

大多数通用处理器动态地检测连续指令之间的依赖关系，并在指令的运算分量尚不可用时自动阻塞这些指令的执行。有些处理器，特别是手持设备中的嵌入式芯片，则把依赖关系检查工作留给软件来做，以便简化硬件并降低能耗。在这种情况下，相应的编译器负责在必要时向代码中插入“no-op”指令(即不做任何处理的指令——译者注)，以保证需要某条指令的计算结果时该结果一定可用。

10.1.3 多指令发送

通过在每个时钟周期发送多条指令，处理器可以在同一时刻运行更多指令。可同时执行的指令数目是指令发送宽度和指令执行流水线中平均阶段数目的乘积。

和流水线处理类似，多发送机器的并行性既可以通过硬件管理，也可以通过软件管理。依靠软件管理其并发性的机器称为 VLIW(非常长指令字，Very-Long-Instruction-Word)机器，而那些使用硬件管理其并发性的机器称为超标量(superscalar)机器。顾名思义，VLIW 机器的指令字宽度比一般指令字更长。每条这样的指令字是要在同一时钟周期内发送的多条指令的编码。编译器决定哪些运

算将被并行地发送，并把这些信息在机器代码中明确地编码。另一方面，超标量机器有一个普通的指令集，并且具有普通的顺序执行语义。超标量机器自动检测指令之间的依赖关系，并在这些指令的运算分量变得可用时发送它们。有些处理器同时包含 VLIW 和超标量两种功能。

简单的硬件指令调度器按照指令获取的顺序执行指令。如果指令调度器碰到一个依赖前面指令的指令，那么该指令及其全部后继指令必须等待依赖关系的解除(即等待它所需的其他指令的计算结果变得可用)。如果有一个静态指令调度器能够把相互独立的运算按执行顺序放在一起，那么这样的机器显然能够从这个静态指令调度器获益。

更加复杂的指令调度器可以“颠三倒四”地执行指令。指令可以被单独地阻塞，直到被阻塞指令所需的所有值都已经生成后再继续执行。即使是这些指令调度器也可以从静态调度获益，因为硬件指令调度器只有有限的空间来缓冲那些必须被阻塞的指令。静态调度可以把相互独立的指令放得比较靠近，以便更好地利用硬件设施。更重要的是，不管动态指令调度器有多复杂，它都不能执行它还没有获取的指令。当处理器不得不执行一个未预见的分支时，它只能在新近获取的指令中寻找并行性。编译器可以设法保证这些新获取的指令可以并行执行，以此来增强动态指令调度器的性能。

10.2 代码调度约束

代码调度是程序优化的一种形式，它应用于由代码生成器生成的机器代码。代码调度要遵守下面三种约束：

1) *控制依赖约束*。所有在原程序中执行的运算都必须在优化后的程序中执行。

2) *数据依赖约束*。优化后的程序中的运算必须和原程序中的相应运算生成相同的结果。

3) *资源约束*。调度不能够超额使用机器上的资源。

这些调度约束保证了优化后的程序和原程序生成同样的结果。但是，因为代码调度改变了运算执行的顺序，所以优化后的程序执行时某一点上的内存状态可能和顺序执行时任一点上的内存状态都不匹配。如果一个程序的执行因异常或用户设定的断点而中断时，就会产生问题。因此经过优化的程序比较难以调试。请注意，这个问题不是代码调度专有的，所有的优化技术都会出现这个问题，包括部分冗余消除(见 9.5 节)和寄存器分配(见 8.8 节)。

10.2.1 数据依赖

显然，如果两个运算不接触同一个变量，那么改变这两个运算的执行顺序肯定不会影响它们的执行结果。实际上，即使这两个运算读取同一个变量的值，我们仍然可以交换它们的执行次序。只有当一个运算向一个变量写值，而另一个运算对这个变量执行读或写运算时，改变它们的执行次序才会改变它们的结果。这样的一对操作之间被认为存在*数据依赖*(data dependence)关系，并且它们的相对执行顺序必须保持不变。有三种类型的数据依赖关系：

1) *真依赖*：写之后再读。如果一个写运算后面跟随一个对同一个位置的读运算，那么这个读操作就依赖于被写入的值，这种依赖关系被认为是一个真依赖关系。

2) *反依赖*：读之后再写。如果一个读运算之后跟随一个对同一个位置的写运算，我们说存在一个从读运算到写运算的反依赖关系。写运算本质上不依赖于读运算，但是如果写运算在读运算之前发生，那么这个读运算将读取到错误的值。反依赖关系是强制式编程的一个副产品。这种语言中同一个内存位置可以在不同时刻存放不同的值。这不是一个“真”依赖关系，可以把值存放在不同的位置上以达到消除反依赖关系的目的。

3) *输出依赖*：写之后再写。对同一个位置的两个写运算之间有输出依赖关系。如果违反了这个关系，那么在这两个运算都完成之后，被写内存位置上存放的是错误的值。

反依赖关系和输出依赖关系被称为*存储相关的依赖*(storage-related dependence)。这些都不是"真"的依赖关系，可以通过使用不同的内存位置存放不同的值来消除这些依赖关系。请注意，数据依赖关系对于内存访问和寄存器访问同样有效。

10.2.2 寻找内存访问之间的依赖关系

要检查两个内存访问之间是否有数据依赖关系，我们只需要指出它们是否可能指向同一个内存位置，而不需要知道到底访问哪个位置。比如，虽然我们可能不知道指针 p 到底指向哪里，但仍然可以指出两个访问 *p 和 *(p + 4)(原文为(*p) + 4——译者注)不可能指向同一个位置。总的来说，数据依赖关系是不可能在编译时刻完全确定的。除非能够证明两个运算指向不同的位置，否则编译器必须假设它们可能会指向同一个位置。

例 10.1 给定下面的代码序列

```
1)  a   =  1;
2)  *p  =  2;
3)  x   =  a;
```

除非编译器知道 p 不可能指向 a，否则它必须决定这三个运算需要顺序执行。从语句(1)到语句(2)有一个输出依赖关系，从语句(1)、(2)到语句(3)有两个真依赖关系。 □

数据依赖分析对于程序所使用的程序设计语言是很敏感的。对于非类型安全的语言，比如 C 和 C++，一个指针可以被强制转换，指向任何类型的数据对象，因此要证明任意一对基于指针的内存访问之间的独立性需要复杂的分析过程。即使对于局部变量和全局标量变量，除非可以证明它们的地址没有被程序中的指令存放到任何地方，它们也有可能通过指针被间接访问。在像 Java 这样的类型安全的语言中，不同类型的对象一定是相互独立的。类似地，栈中的局部简单变量不可能通过其他的变量名字进行访问。

发现正确的数据依赖关系需要多种不同类型的分析。我们将关注那些主要的问题。如果编译器想要检测一个程序中的所有数据依赖关系，它必须首先解决这些问题，并说明如何使用这些信息进行代码调度。后面的章节将说明这些分析是如何完成的。

数组的数据依赖分析

数组的数据依赖分析问题主要是区分数组元素访问中的下标值。比如，下面的循环

```
for (i = 0; i < n; i++)
    A[2*i] = A[2*i+1];
```

把数组 A 的奇数号元素拷贝到紧靠在该元素之前的偶数号元素中去。因为这个循环中所有的读/写运算的位置互不相同，这些访问之间没有任何依赖关系，所以循环中所有的迭代都可以并行执行。数组的数据依赖分析，简称为*数据依赖分析*(data-dependence analysis)，对于数值应用的优化来说是非常重要的。这个主题将在 11.6 节中详细讨论。

指针别名分析

如果两个指针指向同一个对象，我们就说它们*互为别名*(aliased)。指针别名的分析很困难，原因是一个程序中具有很多可能互为别名的指针。随着时间的发展，它们中的每个都可能指向无限多个动态对象。为了得到精确的信息，进行指针别名分析时必须跨越程序中的各个函数。这个主题从 12.4 节开始讨论。

过程间分析

对于通过引用传递参数的语言，需要使用过程间分析来确定是否同一个变量被当作两个或多个不同的参数进行传递。这类别名看起来可能会在不同的参数之间建立依赖关系。类似地，全局变量可能被用作参数，由此会建立参数访问和全局变量访问之间的依赖。在确定这些别名

时，第 12 章中讨论的过程间分析技术是必需的。

10.2.3 寄存器使用和并行性之间的折衷

在这一章中，我们将假设源程序的机器无关中间表示形式使用了无限多个*伪寄存器*(pseudoregister)。这些伪寄存器代表了可以分配到寄存器的变量。这些变量包括源程序中不能通过任何其他名字访问的标量，也包括由编译器生成的用于存放表达式的部分结果的临时变量。和内存位置不同，寄存器的命名是唯一的。因此可以很容易地为寄存器访问生成精确的数据依赖约束。

在中间表示形式中使用的无限多个伪寄存器最终必须被映射到在目标机器上可用的少量物理寄存器。把几个伪寄存器映射为同一个物理寄存器有一个副作用。这种映射会生成人为的存储依赖，这限制了指令级的并行性。反过来，并行执行指令产生了更多的存储需求，以便存放同时计算出来的值。因此，尽量降低寄存器使用数量的目标和最大化指令级并行性的目标直接冲突。下面的例 10.2 和例 10.3 说明了存储和并行性之间的典型折衷处理。

硬件寄存器重命名

指令级并行性首先是作为一种加快普通的顺序机器代码执行速度的手段在计算机体系结构中使用的。当时的编译器还不知道机器上的指令级并行性，其目标是优化寄存器的使用。它们仔细地重新排列指令以使所用的寄存器数目最少，但同时也使可用的并行性的数量减到最少。例 10.3 说明的是在表达式树的计算过程中最小化寄存器使用的同时也限制了它的并行性。

在顺序代码中的并行性太少了，计算机体系结构设计师不得不发明了*硬件寄存器重命名*(hardware register renaming)的概念，试图通过寄存器重命名来撤销寄存器优化所带来的影响。硬件寄存器重命名在程序运行时动态地改变寄存器的指派。它对机器代码进行解释，把本来存放在同一个寄存器中的值存放在不同的内部寄存器中，并把对这些值的使用修正到相应的内部寄存器。

因为人为的寄存器依赖约束首先是由编译器引入的，如果使用了一个认识到指令级并行性的寄存器分配算法，这些约束就可以被消除。当一个机器的指令集只能引用少量寄存器时，硬件寄存器重命名机制仍然是有用的。这种能力使得我们可以给出这个指令集体系结构的更好的实现，把代码中的由指令集体系结构规定的少量寄存器动态地映射到多得多的内部寄存器上。

例 10.2 下面的代码使用伪寄存器 t1 和 t2 把位于位置 a 和 c 上的变量的值分别复制到在位置 b 和 d 上的变量中。

```
LD t1, a    // t1 = a
ST b, t1    // b  = t1
LD t2, c    // t2 = c
ST d, t2    // d  = t2
```

如果已知所有被访问的内存位置都互不相同，那么上面的复制过程可以并行进行。但是，如果为了尽量降低所用寄存器的数量而把 t1 和 t2 赋给同一个寄存器，那么复制过程就只能顺序进行了。□

例 10.3 传统的寄存器分配技术的目标是尽可能减少一个计算过程所需要的寄存器数目。考虑表达式

```
(a + b) + c + (d + e)
```

图 10-2 显示了它的语法树。如图 10-3 的机器代码所示，可以使用 3 个寄存器来完成这个表达式的计算。

但是，对寄存器的复用使得计算串行化。唯一可以并行执行的运算是把位置 a 和 b 的值加载到寄存器，以及把位置 d 和 e 的值加载到寄存器。因此并行地完成这个计算共需要 7 步。

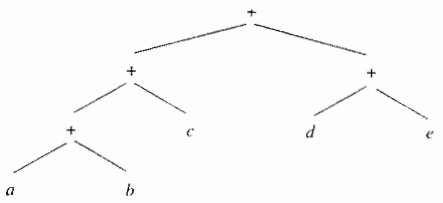

图 10-2 例 10.3 中的表达式树

```
LD r1, a        // r1 = a
LD r2, b        // r2 = b
ADD r1, r1, r2  // r1 = r1+r2
LD r2, c        // r2 = c
ADD r1, r1, r2  // r1 = r1+r2
LD r2, d        // r2 = d
LD r3, e        // r3 = e
ADD r2, r2, r3  // r2 = r2+r3
ADD r1, r1, r2  // r1 = r1+r2
```

图 10-3 图 10-2 中表达式的机器代码

假如我们使用不同的寄存器来存放各个部分和，这个表达式可以在 4 步内完成求值。这个步数正好是图 10-2 中的表达式树的高度。图 10-4 给出了这样的并行计算过程。 □

r1 = a r6 = r1+r2 r8 = r6+r3 r9 = r8+r7	r2 = b r7 = r4+r5	r3 = c	r4 = d	r5 = e

图 10-4 图 10-2 中表达式的并行求值过程

10.2.4 寄存器分配阶段和代码调度阶段之间的顺序

如果在代码调度之前进行寄存器分配，那么得到的代码往往会有很多存储依赖，而这会限制代码调度。另一方面，如果在寄存器分配之前先进行代码调度，那么得到的代码调度方案可能需要太多的寄存器，以至于寄存器溢出(spilling)会抵消指令级并行性所带来的好处。所谓寄存器溢出是指把一个寄存器中的内容保存到一个内存位置上，使得该寄存器可以用于其他目的。一个编译器应该首先分配寄存器然后再进行代码调度吗？还是应该按照相反顺序处理？或者我们同时解决这两个问题？

为了回答上面的问题，我们必须考虑被编译程序的特性。很多非数值应用没有那么多可用的并行性。把少量的寄存器专门用于保存表达式的临时结果就足够了。我们可以首先应用 8.8.4 节中所述的着色算法，为所有非临时变量分配寄存器，然后进行代码调度，最后为临时变量分配寄存器。

这个方法对于数值应用的效果就不太好(数值应用中有很多大型表达式)。我们可以使用层次化的方法来处理。代码优化首先从最内层循环开始，按照从内向外的顺序进行。首先进行指令调度，此时假设可以给每个伪寄存器分配一个独占的物理寄存器。然后进行寄存器分配，并在需要的地方加入处理寄存器溢出的代码，然后再次对代码进行调度。然后我们对较外层的循环重复这个过程。当把同一个外层循环中的多个内层循环一起考虑时，同一个变量可能在不同内层循环中被分配到不同的寄存器中。我们可以改变寄存器的分配方案，以避免把值从一个寄存器复制到另一个寄存器。在 10.5 节中，我们将在特定调度算法的上下文环境下进一步讨论寄存器分配和指令调度之间的关系。

10.2.5 控制依赖

对一个基本块内的运算进行调度是相对容易的。因为一旦控制流到达基本块的开头，所有

的指令都必然会执行。只要满足所有的数据依赖，一个基本块内的指令可以进行任意重新排序。遗憾的是，基本块通常都很小，对非数值程序而言尤其如此。一个基本块平均只有大约五条指令。另外，同一个基本块内的运算经常是紧密相关的，因此很少有并行性。可见，利用基本块之间的并发性是至关重要的。

一个优化后的程序必须执行原程序中的所有运算。它可以比原程序执行更多的指令，前提是额外增加的指令没有改变程序所做的计算。为什么执行额外的指令能够加快一个程序的执行速度？如果我们知道一条指令可能会执行，而且有空闲的资源来“免费”执行这个指令，我们就可以先投机性地(speculatively)执行这条指令。如果这个投机是正确的，那么程序的执行速度就会变快。

如果指令 i_2 的结果决定了指令 i_1 是否执行，那么就说指令 i_1 是控制依赖(control-dependent)于指令 i_2 的。控制依赖的概念和块结构程序中的嵌套层次相对应。明确地说，在 if-else 语句

```
if (c) s1; else s2;
```

中，s1 和 s2 是控制依赖于 c 的。类似地，在 while 语句

```
while (c) s;
```

中，循环体 s 控制依赖于 c。

例 10.4 在代码片段

```
if (a > t)
    b = a*a;
d = a+c;
```

中，语句 b = a * a 和 d = a + c 和此片断中的其他部分都没有数据依赖关系。语句 b = a * a 依赖于比较表达式 $a > t$。但是，语句 d = a + c 不依赖于这个比较表达式，它可以在任何时刻运行。假设乘法运算 $a * a$ 不会引起任何副作用，那么它就可以被投机地执行，前提是只有在发现 a 大于 t 之后才把结果写入 b 中。 □

10.2.6 对投机执行的支持

内存加载指令是能够从投机执行中获得很大好处的指令类型。当然，内存加载是很常见的。它们有比较长的执行延时，加载指令中使用的地址通常可以预先知道，且结果可以存放到一个新的临时变量中而不会破坏任何其他变量的值。遗憾的是，如果它们的地址是非法的，内存加载可能会引发异常，因此投机性地访问非法地址可能会使一个正确的程序意外地停止执行。另外，预测错误的内存加载可能引起额外的高速缓存脱靶和页面错误，这些问题的代价都非常大。

例 10.5 在代码片段

```
if (p != null)
    q = *p;
```

中，如果 p 的值是 null，投机性地对 p 解引用可能会使得正确的程序停止执行。 □

很多高性能处理器都提供了特殊的功能来支持投机性内存访问。下面我们给出其中一些最重要的特殊功能。

预取指令

人们发明了预取(prefetch)指令，以便在数据被使用之前将其从内存移动到高速缓存。一个预取指令向处理器表明该程序可能很快就要使用特定内存字。如果指定的内存位置不可用，或者访问该位置会引起页面错误，那么处理器可以直接忽略这个指令。否则，如果该数据不在高速缓存中，处理器将把该数据从内存移动到高速缓存。

毒药位

另一个体系结构特征被称为毒药位(poison bit)。人们发明毒药位以便投机性地把数据从内存加载到寄存器文件。该机器上的每个寄存器都增加了一个毒药位。如果访问了非法内存，或者被访问的页面不在内存中，处理器并不立刻引发异常，而是仅仅设置目标寄存器的毒药位。只有当其毒药位被置位的寄存器中的内容被使用时才会引发一个异常。

带断言的执行

因为分支运算的开销很大，而预测错误的分支的开销更大(见10.1节)，人们发明了带断言的指令(predicated instruction)以减少一个程序中的分支数量。一条带断言的指令和一条普通指令类似，但是它有一个额外的断言运算分量，作为它的执行条件。只有在该断言被满足时指令才会执行。

例如，一个带条件的移动指令 CMOVZ R2，R3，R1 的语义是只有当寄存器 R1 的值为零时寄存器 R3 的内容才会被移动到寄存器 R2。假设 a、b、c 和 d 分别分配到寄存器 R1、R2、R4、R5 中，下面的代码

```
if (a == 0)
    b = c+d;
```

可以使用如下两条机器指令实现：

```
ADD   R3, R4, R5
CMOVZ R2, R3, R1
```

这个转换把一系列具有控制依赖关系的指令替换为只有数据依赖关系的指令。替换后的这些指令可以和相邻的基本块合并，形成更大的基本块。更重要的是，使用这些代码，处理器就不会产生预测错误，因此保证了指令流水线的平滑运行。

带断言的执行也是有代价的。即使最后不需要执行带断言的指令，处理器也必须获取该指令并解码。静态调度器必须保留执行它们所需要的资源，并保证所有可能的数据依赖都得到满足。除非机器拥有的资源大大多于不使用带断言指令时所需要的资源，否则不应该过度使用带断言指令。

动态调度机器

使用静态调度的机器的指令集明确地定义了哪些指令可以并行执行。但是，回顾一下10.1.2节，有些机器的体系结构允许到运行时刻再确定哪些指令可以并行运行。使用动态调度，同样的机器代码可以在同一系列的不同机器上运行。这些机器实现了同样的指令集，但是拥有不同数量的并行执行支持设施。实际上，机器代码级的兼容是动态调度机器的一个主要优点。

用软件方式在编译器中实现的静态调度器可以帮助(用机器硬件实现的)动态调度器更好地利用机器资源。在为一个动态调度机器构造一个静态调度器时，我们几乎可以照搬为静态调度机器设计的调度算法，只是新算法不需要明确地生成原算法放置在调度方案中的 no-op 指令。这个问题将在10.4.7中进一步讨论。

10.2.7 一个基本的机器模型

很多机器可以使用下面的简单模型表示。一个机器 $M = <R, T>$ 由下列元素组成：

1) 一个运算类型的集合 T。这些运算类型包括加载、保存、算术运算等。

2) 一个代表硬件资源的向量 $R = [r_1, r_2, \cdots]$，其中 r_i 表示第 i 种资源的可用单元的数目。典型资源的例子包括：内存访问单元、算术逻辑单元(ALU)和浮点功能单元。

每条指令具有一组输入运算分量、一组输出运算分量和一个资源需求。每一个输入运算分量都有一个对应的输入延时。这个延时表示输入值(相对于运算开始时刻)必须在什么时候可用。典型的输入运算分量的延时是零，表明立刻就需要使用这些值。类似地，每个输出运算分量有一个对应的输出延时。这个延时表明了运算结果(相对于运算开始时刻)什么时候可用。

每个 t 类型的机器运算指令需要使用的资源可以建模为一个二维的资源预约表(resource-reservation table)，RT_t。该表的宽度是机器中资源的种类数量，它的长度是该运算使用资源的时间长度。表格中的条目 $RT_t[i,j]$ 表示在 t 类型的运算指令被发出 i 个时钟周期后该运算指令占用的第 j 种资源的数量。为了简化表示方法，如果 i 指向一个表格中不存在的条目(也就是 i 比执行该运算所需的时钟数大)，我们假定 $RT_t[i,j]=0$。当然，对于任何 t、i 和 j，$RT_t[i,j]$ 必须小于或等于 $R[j]$，也就是该机器拥有的第 j 种资源的总数。

典型的机器运算指令在其被发出时只占用一个单元的资源。有些运算可能使用多个功能单元。比如，一个相乘再相加的指令可能在第一个时钟周期使用一个乘法器，在第二个周期使用一个加法器。有些运算(比如除法运算)可能需要占用一个资源多个时钟周期。完全流水线化(fully pipelined)的运算是指那些在每个时钟周期都可以发出一条指令的运算，虽然这些运算的结果可能要等到几个时钟周期之后才可用。我们不需要明确地对一条流水线的各个阶段的资源建模，只需要用一个单元对第一个阶段建模就可以了。占用了某条流水线的第一阶段的运算一定能够在下一个时钟周期进入下一个阶段。

10.2.8 10.2 节的练习

练习 10.2.1：图 10-5 中的多个赋值语句具有某些依赖关系。对于下列的每个语句对，将它们之间的依赖关系按照下列四种情况进行分类。(1)真依赖，(2)反依赖，(3)输出依赖，(4)无依赖关系(即两条指令可以按照任何顺序出现)。

1) 语句(1)和(4)。
2) 语句(3)和(5)。
3) 语句(1)和(6)。
4) 语句(3)和(6)。
5) 语句(4)和(6)。

```
1)  a = b
2)  c = d
3)  b = c
4)  d = a
5)  c = d
6)  a = b
```

图 10-5 一组展示了数据依赖性的赋值语句序列

练习 10.2.2：严格按照括号顺序(即不使用交换律和结合律来改变加法的顺序)对表达式 $((u+v)+(w+x))+(y+z)$ 求值。给出寄存器层次的机器代码，要求此代码具有尽可能大的并行性。

练习 10.2.3：对下列表达式重复练习 10.2.2。

1) $(u+(v+(w+x)))+(y+z)$
2) $(u+(v+w))+(x+(y+z))$

如果我们不是要把并行性最大化，而是要最小化所用的寄存器数目，这个计算过程将执行多少步？通过将并行性最大化，我们省下了多少步骤？

练习 10.2.4：练习 10.2.2 中的表达式可以使用图 10-6 中的指令序列来执行。如果我们有足够多的并行机制，执行这些指令需要多少步？

```
1)  LD r1, u          // r1 = u
2)  LD r2, v          // r2 = v
3)  ADD r1, r1, r2    // r1 = r1 + r2
4)  LD r2, w          // r2 = w
5)  LD r3, x          // r3 = x
6)  ADD r2, r2, r3    // r2 = r2 + r3
7)  ADD r1, r1, r2    // r1 = r1 + r2
8)  LD r2, y          // r2 = y
9)  LD r3, z          // r3 = z
10) ADD r2, r2, r3    // r2 = r2 + r3
11) ADD r1, r1, r2    // r1 = r1 + r2
```

图 10-6 一个算术表达式的使用最少寄存器的实现

！练习 10.2.5：使用 10.2.6 节中的条件拷贝指令 CMOVZ 来翻译例 10.4 中的代码片断。机器代码中的数据依赖关系是什么？

10.3 基本块调度

我们现在可以开始讨论代码调度算法了。我们从最简单的问题开始：对一个由机器指令组成的基本块进行调度。给出这个问题的最优解的复杂度是 NP 完全的。但是在实践中，一个典型的基本块只有少量相互之间高度约束的运算，因此使用简单的调度算法就足够了。我们将介绍一个称为*列表调度*(list scheduling)的简单且非常高效的算法来解决这个问题。

10.3.1 数据依赖图

我们把每个由机器指令组成的基本块表示成为一个*数据依赖图*(data-dependence graph)，$G=(N,E)$，其中结点集合 N 表示基本块中机器指令的运算，而有向边集合 E 表示运算之间的数据依赖约束。G 的结点集合和边集按照如下方式构造：

1）在 N 中的每个运算 n 有一个资源预约表 RT_n，其值就是 n 的运算类型所对应的资源预约表。

2）E 中的每条边 e 有一个表示延时的标号 d_e。该标号表明目标结点必须在源结点发出后至少 d_e 个时钟周期之后发出。假设运算 n_1 之后跟有运算 n_2，并且两条指令访问同一个内存位置，访问的延时分别为 l_1 和 l_2。也就是说，该位置上的值在第一条指令开始之后的第 l_1 个时钟周期生成，且第二条指令在其开始后的第 l_2 个时钟周期需要这个值。请注意，在通常情况下 $l_1=1$ 而 $l_2=0$。那么，E 中有一个延时标号为 l_1-l_2 的边 $n_1\rightarrow n_2$。

例 10.6 考虑一个可以在每个时钟周期内执行两个运算的机器。其中第一个运算必须是分支运算或者以下形式的 ALU 运算：

```
OP dst, src1, src2
```

第二个运算必须是如下形式的加载运算或者保存运算：

```
LD dst, addr
ST addr, src
```

其中的加载运算(LD)是完全流水线化的并占用两个时钟周期。但是，一个加载运算后面可以立刻跟一个向被读内存地址进行写运算的保存运算 ST。所有其他的运算都在一个时钟周期内完成。

资源预约表的图示方法

把一个运算的资源预约表用实心和空心方块组成的网格可视化地表示出来是非常有用的。在网格中，每一列对应于目标机器上的一种资源，而每一行表示该运算执行中的一个时钟周期。假设对于每种类型的资源，这个运算最多只需要一个单元，我们就可以使用实心方块表示 1，用空心方块表示 0。另外，如果该指令是完全流水线化的，那么只需要指明在第一行中使用的资源，相应的资源预约表变成了单独的一行。

例如，这个表示方式在例 10.6 中使用。在图 10-7 中，我们可以看到各个资源预约表都是单行的。其中的两个加法运算需要“alu”资源，而加载和保存运算需要“mem”资源。

图 10-7 中显示的是一个基本块例子的依赖图和它的资源需求。我们可以想像 R1 是一个栈指针，用来通过诸如 0 或者 12 这样的偏移量访问栈中的数据。第一条指令向寄存器 R2 中加载数

据，直到两个时钟周期之后这个数据才变得在 R2 中可用。这就是从第一条指令到第二及第五条指令的边的标号为 2 的原因，这两条指令都需要 R2 中的值。类似地，从第三条指令到第四条指令的边也有标号表明延时为 2；第四条指令需要被加载到 R3 中的值，而这个值要在第三条指令开始之后两个时钟周期才变得可用。

因为我们不知道 R1 和 R7 的值之间有什么样的关系，所以不得不考虑地址 8(R1)和地址 0(R1)相同的可能性。也就是说，最后一条指令可能正在把值保存到第三条指令读取数据的位置。我们正使用的机器模型允许我们在从某个位置开始读取数据的一个时钟周期之后把数据存放到这个位置上，即使被读出的数据需要再等一个时钟周期才出现在寄存器中。这就是从第三条指令到最后一条指令的边的标号为 1 的原因。这也同样是从第一条指令到最后一条指令有一条标号为 1 的边的原因。其他标号为 1 的边产生的原因是指令间的数据依赖关系，或者当 R7 取某些值时可能产生的依赖关系。

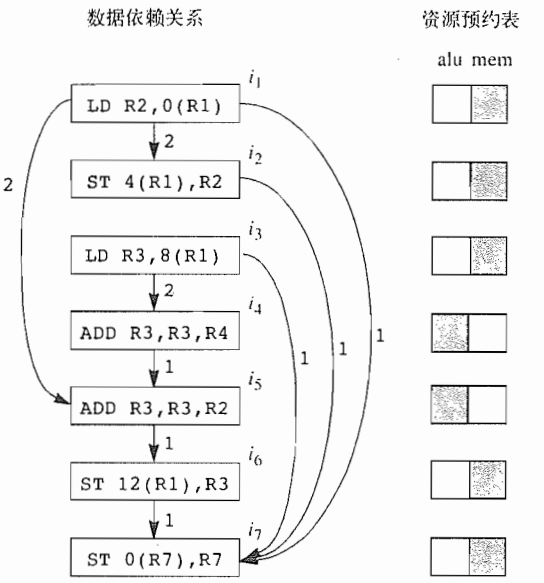

图 10-7 例 10.6 的数据依赖图

□

10.3.2 基本块的列表调度方法

基本块调度的最简单的方法是以“带有优先级的拓扑排序”访问数据依赖图的各个结点。因为一个数据依赖图中不可能有环，因此总是至少存在一个各个结点之间的拓扑顺序。但是，在所有可能的拓扑排序中，有些排序可能比其他排序更好。我们将在 10.3.3 节中讨论选择拓扑排序的一些策略。但是现在我们仅仅假设存在某种算法来选择一个较好的拓扑排序。

下面我们将讨论的列表调度算法按照被选中的带优先级的拓扑排序访问各个结点。最后，这些结点并不一定按照它们被访问的顺序进行调度。但是指令被尽可能早地放置在调度方案中，因此指令被调度的顺序往往和它们被访问的顺序差不多。

更详细地讲，算法根据每个结点和之前已调度的结点之间的数据依赖约束，计算出能够执行该结点的最早时间位置。然后，算法根据一个资源预约表来检验该结点所需要的资源是否得到满足。这个资源预约表收集了至今已经分配出去的资源的信息。该结点被安排在最早的能够获得足够资源的时间位置上。

算法 10.7 对一个基本块进行列表调度

输入： 一个机器－资源向量 $R=[r_1,r_2,\cdots]$，其中 r_i 是第 i 种资源的可用单元的数目；一个数据依赖图 $G=(N,E)$。N 中的每个运算 n 的标号是它的资源预约表 RT_n；E 中的每个边 $e=n_1\rightarrow n_2$ 都有标号 d_e，表明了 n_2 不能在 n_1 执行之后的 d_e 个时钟周期之内执行。

输出： 一个调度方案 S。它把 N 中的每个运算映射到时间位置中。各个运算在方案所确定的时间位置开始执行，就可以保证所有的数据依赖关系和资源约束都得到满足。

方法： 执行图 10-8 中的程序。关于什么是“带优先级的拓扑排序”的讨论将在 10.3.3 节中给出。

```
RT = 一个空的资源预约表;
for (按照带优先级的拓扑排序访问 N 中的每个结点 n) {
    s = max_{e=p→n in E}(S(p) + d_e);
        /* 根据一个指令的各个前驱在何时开始,
           计算这个指令最早可以在何时开始 */
    while (存在 i 使得 RT[s + i] + RT_n[i] > R)
        s = s + 1;
            /* 进一步把这个指令后延, 直到所需资源
               都变得可用为止 */
    S(n) = s;
    for (所有 i)
        RT[s + i] = RT[s + i] + RT_n[i]
}
```

图 10-8 一个列表指令调度算法

10.3.3 带优先级的拓扑排序

列表调度算法不会回溯，它对每个结点进行一次且只进行一次指令调度。它使用一个启发式的优先级函数来从已经就绪的结点中选择下一个调度的结点。下面是一些关于结点的所有可能的带优先级的拓扑排序的性质：

- 如果不考虑资源约束，最短的调度方案可以根据关键路径(critical path)给出。所谓关键路径就是数据依赖图中的最长路径。一个可以被用作优先级函数的度量是结点的高度(height)，就是从这个结点开始的最长路径的长度。
- 从另一方面考虑，如果所有的运算都是独立的，那么调度方案的长度受到可用资源的约束。关键资源就是具有最大的资源使用/可用数量比值的资源。所谓资源使用/可用数量比值是指对资源的使用和可用资源的单元数目的比值。使用较多关键资源的运算具有较高的优先级。
- 最后，我们可以使用源代码中的顺序来解决运算之间难分先后的问题，在源程序中先出现的运算应该首先被安排。

例 10.8 对于图 10-7 中的数据依赖关系，它的关键路径的(包含了执行最后一条指令的时间)长度是6 个时钟周期。也就是说，关键路径是最后的五个结点，从 R3 的加载运算开始到对 R7 的保存运算结束。这条路径中的所有边上的总延时是 5，此外我们还要再加上执行最后一条指令所需的 1 个时钟周期。

使用结点高度作为优先级函数，算法 10.7 找到了一个如图 10-9 所示的优化的调度方案。请注意，因为 R3 的加载具有最大的高度，因此安排这条指令首先执行。R3 和 R4 的加法在第二个时钟周期就有足够的资源，但是一条加载指令有 2 个时钟周期的延时，因此我们把这个加法安排到第三个时钟周期。也就是说，在第 3 个时钟周期开始之前我们不能保证 R3 中已经保存了需要的值。□

调度方案		资源预约表 alu	mem
	LD R3,8(R1)		
	LD R2,0(R1)		
ADD R3,R3,R4			
ADD R3,R3,R2	ST 4(R1),R2		
	ST 12(R1),R3		
	ST 0(R7),R7		

图 10-9 将列表调度算法应用到图 10-7 后得到的结果

10.3.4 10.3 节的练习

练习 10.3.1：对于图 10-10 中的每个代码片段，画出数据依赖图。

```
1)  LD R1, a
2)  LD R2, b
3)  SUB R3, R1, R2
4)  ADD R2, R1, R2
5)  ST a, R3
6)  ST b, R2
```
a)

```
LD R1, a
LD R2, b
SUB R1, R1, R2
ADD R2, R1, R2
ST a, R1
ST b, R2
```
b)

```
LD R1, a
LD R2, b
SUB R3, R1, R2
ADD R4, R1, R2
ST a, R3
ST b, R4
```
c)

图 10-10 练习 10.3.1 的机器代码

练习 10.3.2：假设一个机器具有一个 ALU 资源(用于 ADD 和 SUB 运算)和一个 MEM 资源(用于 LD 和 ST 运算)。假设除了 LD 运算需要两个时钟周期之外，其余所有运算都只需要一个时钟周期。但是，如例 10.6 中所说，在一个对某内存位置的 LD 运算执行一个时钟周期之后就可以执行对同一个位置的 ST 运算。为图 10-10 中的每个代码片断寻找一个最短调度方案。

练习 10.3.3：在如下假设下重复练习 10.3.2。

1) 该机器具有一个 ALU 资源和两个 MEM 资源。

2) 该机器具有两个 ALU 资源和一个 MEM 资源。

3) 该机器具有两个 ALU 资源和两个 MEM 资源。

练习 10.3.4：使用例 10.6 中的机器模型(和练习 10.3.2 一样)：

1) 为图 10-11 中的代码画出数据依赖图。

2) 对于问题(1)得到的数据依赖图，全部关键路径包括哪些？

3) 假设有无限多个 MEM 资源，对于这七条指令的所有可能的调度方案是什么？

```
1)  LD R1, a
2)  ST b, R1
3)  LD R2, c
4)  ST c, R1
5)  LD R1, d
6)  ST d, R2
7)  ST a, R1
```
图 10-11 练习 10.3.4 的机器代码

10.4 全局代码调度

对于一个具有中等数量的指令并行机制的机器，通过压缩各个基本块而得到的调度方案往往会留下很多空闲的资源。为了更好地利用机器资源，有必要考虑把一些指令从一个基本块移动到另一个基本块的代码生成策略。同时考虑多个基本块的策略称为全局调度(global scheduling)算法。为了正确地进行全局调度，我们需要考虑的问题不仅包括数据依赖关系，还包括控制依赖。我们必须保证

1) 所有在原程序中执行的指令都会在优化后的程序中运行，并且

2) 虽然优化后的程序可以投机性地执行一些额外指令，但这些指令不能产生任何有害的副作用。

10.4.1 基本的代码移动

让我们首先通过一个简单的例子来研究一下指令移动可能涉及的问题。

例 10.9 假设我们有一个可以在单个时钟周期内同时执行任意两条指令的机器。除了加载运算有两个时钟周期的延时外，其余每个运算的执行延时为一个时钟周期。为简单起见，我们假设例子中所有的内存访问都是正确的，且访问的数据都在高速缓存中。图 10-12a 显示了一个包括三个基本块的简单流图。其中的代码被扩展为图 10-12b 所示的机器指令。因为数据依赖关系，每个基本块中的所有指令必须顺序执行。实际上，每个基本块中都必须插入一个 no-op 指令。

假设变量 a、b、c、d 和 e 的地址互不相同，并且这些地址被分别存放在寄存器 R1 ~ R5 中。因此不同基本块中的计算之间没有数据依赖关系。我们发现，不管是否选择图中的分支跳转，基本块 B_3 中的所有运算都会被执行，因此它可以和基本块 B_1 中的运算并行执行。我们不能把 B_1 中

的运算移动到 B_3，因为需要它们来决定分支跳转的出口。

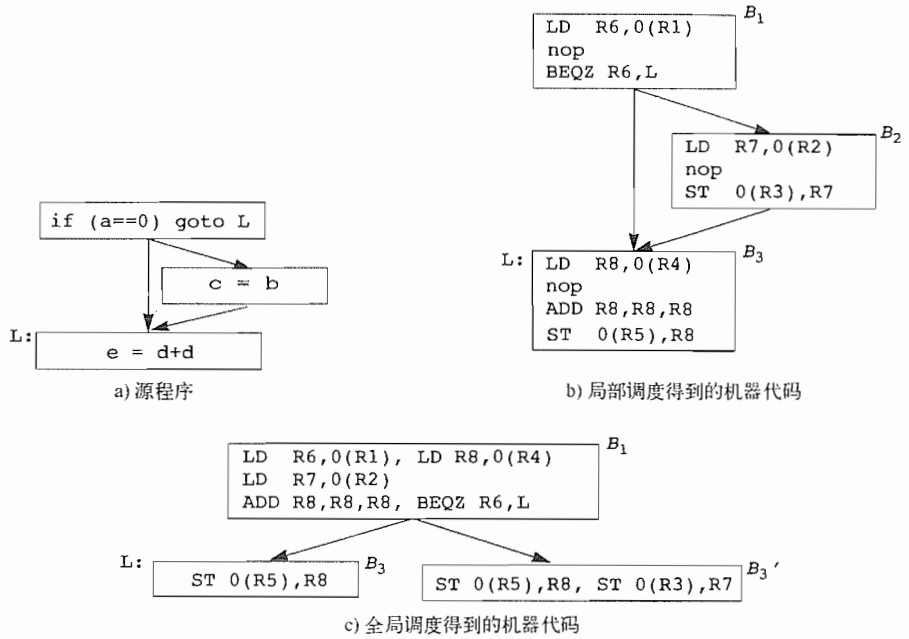

图 10-12　例 10.9 中全局调度之前和之后的流图

B_2 中的运算和基本块 B_1 中的测试指令之间具有控制依赖关系。我们可以在基本块 B_1 中投机性执行 B_2 中的加载运算而不会产生任何附加开销，并且只要该分支执行，就可以节约两个时钟周期。

保存运算不应该投机性地执行，因为它们覆写了某个内存位置上的原值。但是可以延迟执行一个保存运算。我们不能直接把 B_2 中的保存运算放到基本块 B_3 中，因为只有当控制流经过基本块 B_2 时才能执行这个保存运算。但是，我们可以把保存运算放在 B_3 的一个拷贝中。图 10-12c 中显示了经过这样优化后的调度方案。优化后的代码在 4 个时钟周期内执行完毕，这和单独执行基本块 B_3 所需的时间一样。 □

例 10.9 表明我们可以沿着一个执行路径上下移动指令。在这个例子中，每一对基本块都有一个不同的“支配关系”，因此关于何时以及如何在每一对基本块之间移动指令的考虑是不同的。如 9.6.1 节中所讨论的，如果每一个从控制流图入口处到达基本块 B' 的路径都经过一个基本块 B，那么就认为 B 支配 B'。类似地，如果从 B' 到达流图出口处的路径都经过 B，我们说 B 反向支配(postdominate) B'。当 B 支配 B' 并且 B' 反向支配 B 的时候，我们就说 B 和 B' 是控制等价的(control equivalent)，其含义是一个基本块会被执行当且仅当另一个基本块也会被执行。对于图 10-12 中的例子，假设 B_1 是流图入口，且 B_3 是出口，则

1）B_1 和 B_3 是控制等价的：B_1 支配 B_3 而 B_3 反向支配 B_1。

2）B_1 支配 B_2，但是 B_2 不反向支配 B_1。

3）B_2 不支配 B_3 但是 B_3 反向支配 B_2。

在一条路径上的一对基本块之间也可能既不具有支配关系，也不具有反向支配关系。

10.4.2　向上的代码移动

我们现在仔细考查把一个运算沿着一条路径向上移动意味着什么。假设我们希望把一个运

算从基本块 *src* 沿着一条控制流路径向上移动到基本块 *dst*。同时假设这样的移动没有违反任何数据依赖关系，并且使得从 *dst* 到 *src* 的路径运行得更快。如果 *dst* 支配 *src* 并且 *src* 反向支配 *dst*，那么被移动的运算会在它应该运行的时候被恰好运行一次。

如果 src 不反向支配 dst

这种情况下，存在一条经过 *dst* 但是没有到达 *src* 的路径。此时会执行一个多余的运算。除非被移动的运算没有任何有害的副作用，否则这个代码移动就是非法的。如果被移动的运算是“免费”执行的(即它只使用那些本来会被闲置的资源)，那么这次代码移动没有产生开销。只有当控制流到达 *src* 的时候这次代码移动才是有益的。

如果 dst 不支配 src

这种情况下存在一条没有首先经过 *dst* 就到达 *src* 的路径。我们需要在这样的路径中插入被移动运算的拷贝。根据 9.5 节中对部分冗余消除的讨论我们可以知道如何准确做到这一点。我们把这个运算的拷贝放置在一组基本块中，这组基本块形成了一个将入口基本块和 *src* 分割开的割集。在每个插入这个拷贝的地方，下列约束必须满足：

1) 该运算的运算分量必须和原运算的运算分量具有相同的值。

2) 运算的结果没有覆盖掉可能在后面使用的值。

3) 此运算本身的结果没有在到达 *src* 之前被覆盖掉。

这些拷贝使得 *src* 中的原指令完全冗余，因此可以被消除。

我们把这个运算指令的额外拷贝称为补偿代码(compen sation code)。9.5 节讨论过，可以在关键边上插入基本块来放置这些拷贝。补偿代码可能使得某些路径的执行变慢。因此，只有当被优化路径的执行频率高于其他未被优化的路径时，这个代码移动才会提高程序执行的性能。

10.4.3 向下的代码移动

假设我们感兴趣的是把一个运算从基本块 *src* 沿着一条控制流路径向下移动到基本块 *dst*。我们可以像上面介绍的那样考虑这样的代码移动。

如果 src 不支配 dst

在这种情况下，存在一条没有先访问 *src* 就到达 *dst* 的路径。同样，在这种情况下会执行一个额外的运算。遗憾的是，向下代码移动经常用于写运算。这种运算具有副作用，会覆盖原来的值。我们可以设法绕过这个问题，方法是复制从 *src* 到 *dst* 的路径上的基本块，并且只在 *dst* 的新拷贝中放置这个运算。另一个方法是，如果可以在使用带断言的指令时使用这种指令。我们用基本块 src 的卫式断言作为被移动运算的卫式断言。请注意，这些带断言的指令只能被安排在由计算该断言的基本块所支配的基本块中，否则该断言的值会不可用。

如果 dst 不反向支配 src

和上面的讨论一样，我们必须插入补偿代码以使得被移动的运算在所有没有到达 *dst* 的路径上都被执行了。这个转换仍然和部分冗余消除类似，不同之处在于运算的拷贝被放置在基本块 *src* 之后、把 *src* 和流图出口处分开的割集中。

关于向上和向下代码移动的总结

从上面的讨论中可知，我们看到存在一组可能的全局代码移动的方法。这些方法的收益、代价以及实现复杂度各不相同。图 10-13 中给出了这些代码移动方法的总结。图中的各行对应于下面四种情况：

1) 在控制等价的基本块之间移动指令最简单且性价比最高。不需要执行额外的运算，也不需要补偿代码。

2) 在向上(向下)代码移动中，如果源基本块不反向支配(支配)目标基本块，那么就可能需

要执行额外的运算。当该额外运算能够免费执行并且通过源基本块的路径被执行时，这个代码移动就是有益的。

3）在向上(向下)代码移动中，如果目标基本块不支配(反向支配)源基本块，就需要补偿代码。带有补偿代码的路径的运行可能会变慢，因此保证被优化的路径具有较高的执行频率是很重要的。

4）最后一种情况把第二和第三种情况的不利之处合并了起来：可能既需要执行额外运算，又需要补偿代码。

	向上：*src*反向支配 *dst* 向下：*src* 支配 *dst*	*dst* 支配 *src* *dst* 反向支配 *src*	投机 代码复制	补偿代码
1	是	是	否	否
2	否	是	是	否
3	是	否	否	是
4	否	否	是	是

图 10-13 代码移动的总结

10.4.4 更新数据依赖关系

如下面的例 10.10 所示，代码移动可能会改变运算之间的数据依赖关系。因此在每次代码移动之后都必须更新数据依赖关系。

例 10.10 对于图 10-14 中显示的流图，对 x 的两个赋值之一可以被向上移动到顶部的基本块，因为这样的转换保持了原程序中的所有依赖关系。但是，一但我们把其中一个赋值语句上移，就不能再移动另一个。更明确地说，我们看到在代码移动之前顶部的基本块的出口处 x 是不活跃的，但是在移动之后就变得活跃了。如果一个变量在一个程序点上活跃，那么我们不能把对该变量的投机性定值移动到该程序点的前面。 □

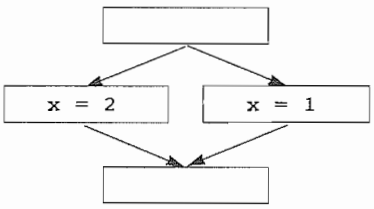

图 10-14 说明因为代码移动而改变数据依赖关系的例子

10.4.5 全局调度算法

在上一节中，我们看到代码移动对某些路径有益，但是会损害另外一些路径的性能。好消息是指令并不是生而平等的。实际上，我们知道，一个程序的 90% 以上的执行时间被花在不到 10% 的代码上。因此，我们可以把目标确定为使得频繁执行的路径更快运行，虽然有可能降低不频繁路径的运行速度。

编译器有多种技术来估算执行频率。我们有理由假设在最内层的循环中的指令比外层循环中的指令执行得更频繁，也有理由假设选择向回跳转分支的使用频率高过不选择这个分支的使用频率。另外，转向程序出口或者异常处理例程的分支不大可能被选择执行。但是，最好的频率估算来自于动态获取的程序运行剖面。在这个技术中，程序经过插装以记录程序运行时刻各个条件分支的出口选择情况。然后，程序就在有代表性的输入上运行，确定程序总体的运行行为。人们发现应用这个技术得到的结果相当精确。这样的信息可以反馈给编译器，由编译器在其优化过程中使用。

基于区域的调度

现在我们描述一个简单的全局调度器，它支持两种最容易的代码移动：

1）把运算向上移动到控制等价的基本块。

2）把运算向上移动一个分支，移动到一个支配前驱中。

9.7.1 节介绍过，一个区域是一个控制流图的子集，它只能通过一个入口基本块到达。我们可以把任何过程表示为一个区域的层次结构。顶层区域包括整个过程，而嵌套在其中的子区域代表该过程中的自然循环。我们假设控制流图是可归约的。

算法 10.11 基于区域的调度。

输入：一个控制流图和一个机器 - 资源描述。

输出：一个调度方案 S。它把每条指令映射到一个基本块和一个时间位置。

方法：执行图 10-15 中的程序。其中的一些术语缩写的含义是很明显的：*ControlEquiv*(B)表示和基本块 B 控制等价的基本块的集合，而作用于一个基本块集合的 *DominatedSucc* 表示下面的基本块集合：它们是该基本块集合中一个或多个基本块的后继，且被该集合中的所有基本块支配。

```
for（按照拓扑排序访问各个区域R，使得内层区域先于
        外层区域被访问）
    {计算数据依赖关系；
     for（按照带优先级的拓扑排序访问R中的每个基本块B）{
          CandBlocks = ControlEquiv(B) ∪
               DominatedSucc(ControlEquiv(B));
          CandInsts = CandBlocks中已可以被调度的指令;
          for（t = 0, 1, ... 直到B中的所有指令都已经调度完毕）{
               for（按照优先顺序访问CandInsts中的每个指令 n）
                    if（n在时刻t上没有资源冲突）{
                         S(n) = ⟨B, t⟩;
                         更新已分配资源的信息;
                         更新数据依赖关系；
                    }
               更新CandInsts;
          }
     }
}
```

图 10-15 一个基于区域的全局调度算法

算法 10.11 中的代码调度从最内层的区域开始逐渐扩展到最外层。在对一个区域进行调度时，每一个内嵌的子区域都被当作一个黑盒子，指令不能移入或移出某个子区域。但是，只要指令的数据依赖关系和控制依赖关系都得到满足，我们可以绕着子区域移动这些指令。

算法忽略了所有流回区域头基本块的控制和依赖边，因此最后得到的控制流和数据依赖图都是无环的。对每个区域中的各基本块的访问遵守拓扑排序。这个顺序保证了只有在该基本块所依赖的所有指令都被调度好之后才对这个基本块进行调度。将被安排在一个基本块 B 中的指令来自于所有和基本块 B 控制等价的基本块（包含 B），以及这些等价基本块的、被 B 支配的直接后继。

为各个基本块创建调度方案时使用的是列表调度算法。该算法有一个候选指令列表 *CandInsts*，这个列表中包含了候选基本块中的所有其前驱已调度好的指令。该算法逐个时钟周期地构造调度方案。对于每个时钟周期，它按照优先级顺序检查 *CandInsts* 中的各条指令，在资源允许的情况下把指令安排在该时钟周期上。然后，算法 10.11 更新 *CandInsts* 并重复这个过程，直到 B 中所有指令都被安排完毕。

CandInsts 中的指令的优先级顺序所使用的优先级函数和 10.3 节中讨论过的函数类似。然而，我们作了一个重要的修改。我们把较高的优先级赋予那些来自和基本块 B 控制等价的基本块的指令，而对来自于后继基本块的指令赋予较低的优先级。这么做的原因是后一种类型的指令只是在基本块 B 中被投机性执行。 □

循环展开

在基于区域的调度中，一个循环中迭代之间的界限是代码移动的障碍。来自一个迭代的运算不能和来自其他迭代的运算重叠。可缓解这一问题的简单且高效的技术是在代码调度之前少量地展开该循环。如下的 for 循环

```
for (i = 0; i < N; i++) {
    S(i);
}
```

可以被写成图 10-16a 所示的代码。类似地，如下的 repeat 循环

```
repeat
    S;
until C;
```

可以被写成图 10-16b 所示的代码。循环展开在循环体中产生了更多的指令，使全局调度算法找到更多的并行性。

```
for (i = 0; i+4 < N; i+=4) {
    S(i);
    S(i+1);
    S(i+2);
    S(i+3);
}
for ( ; i < N; i++) {
    S(i);
}
```

a) 展开一个for循环

```
repeat {
    S;
    if (C) break;
    S;
    if (C) break;
    S;
    if (C) break;
    S;
} until C;
```

b) 展开一个repeat循环

图 10-16 循环的展开

相邻压缩

算法 10.11 只支持 10.4.1 节中描述的前两种形式的代码移动。需要引入补偿代码的代码移动有时也是有用的。支持这种代码移动的方法之一是在基于区域的调度之后再跟一个简单的处理过程。在这个过程中，我们可以检查各对连续执行的基本块，检查是否有运算可以在它们之间上移或下移，以改进这些基本块的执行时间。如果找到了一对这样的基本块，我们检查是否需要在别的路径中复制将被移动的指令。如果移动之后的预期收益是正的，就可以进行这样的代码移动。

这个简单的扩展能够有效提高循环的性能。比如，它可以把一个迭代开始处的运算移动到上一个迭代的结尾，同样也可以把运算从第一个迭代移动到循环之外。对于较紧密的循环，即每个迭代过程只执行少量指令的循环，这种优化特别具有吸引力。但是，由于每个代码移动的决定是局部地独立做出的，因此这个技术的效果受到一定的限制。

10.4.6 高级代码移动技术

如果我们的目标机器是静态调度的，并且具有丰富的指令级并行机制，我们可能需要更加积极的调度算法。下面是有关进一步扩展代码移动技术的一些高级描述：

1）为了便于进行下面的扩展，我们可以在那些从具有多个前驱的基本块出发的控制流边上增加新的基本块。在代码调度结束后将删除这些基本块中的空基本块。一个有用的启发式规则是试图把指令移出几乎为空的基本块，使得这个基本块可以被完全删除。

2）在算法 10.11 中，各基本块内执行的代码在此基本块被访问时一次性调度完毕。这个简单的方法已经足够了，因为这个算法只能够把运算上移到前面的支配基本块。为了进行需要额外补偿代码的代码移动，我们选用了一个略微不同的方法。当我们访问基本块 B 的时候，只对 B 以及和 B 控制等价的基本块中的指令进行调度。我们首先试图把这些指令放置到之前已经被访问过的前驱基本块中。这些基本块已经有了一个部分调度方案。我们试图找到一个目标基本块，使得移动后可以改进一个频繁执行的路径，然后在其他路径上放置这条指令的拷贝来保证移动的正确性。如果指令不能向上移动，那么它们和以前一样在当前的基本块中进行调度。

3）在以拓扑顺序访问基本块的算法中，实现向下的代码移动要更加困难一些，原因是目标基本块还在等待调度。虽然机会较少，但还是存在一些机会进行这样的代码移动。我们会移动所有同时满足下列条件的运算：

① 它们可以被移动。

② 在原来的基本块中它们不能免费执行。

如果目标机器有很多没有使用到的硬件资源，这个简单的策略会很有效。

10.4.7 和动态调度器的交互

一个动态调度器的优势是它可以根据运行时刻的情况产生新的调度方案，而不必在运行之前对所有可能的调度进行编码。如果目标机器有一个动态调度器，那么静态调度器的主要功能是保证尽早获得高延时指令，使得动态调度器可以尽早发出这些指令。

高速缓存脱靶是一类不可预测的事件，它们可能使得程序的性能有很大的不同。如果可以使用数据预取指令，那么静态调度器可以较早地放置这些预取指令以使得在需要相应数据时，数据已经在高速缓存之中。静态调度器通过这种方式有效地帮助动态调度器。如果没有预取指令可用，编译器可以估算一下哪些运算可能会发生高速缓存脱靶，并试图早一点发出这些运算指令。

如果在目标机器上没有动态调度机制，静态调度器必须保守地处理调度问题，把每对具有数据依赖关系的运算分开，使它们之间的间隔不小于最小延时。但是，如果有动态调度器可用，编译器只需要把具有数据依赖关系的运算指令按照正确的顺序排列，以保证程序的正确性。为了得到最佳性能，编译器应该给较有可能发生的依赖赋予较长的延时，给不大可能发生的依赖赋予较短的延时。

分支的错误预测是性能下降的重要原因之一。因为错误预测会带来较长的时间损失，较少执行路径上的指令仍然会对整个执行时间产生较大的影响。所以，应该给这类指令赋予较高的优先级，以便降低错误预测的代价。

10.4.8 10.4 节的练习

练习 10.4.1：指出如何展开一般的 while 循环

```
while (C)
    S;
```

! 练习 10.4.2：考虑代码片断：

```
if (x == 0) a = b;
else a = c;
d = a;
```

假设有一个机器使用了例 10.6 中的延时模型（即加载运算需要两个时钟周期，其他指令需要一个时钟周期）。同时假设该机器可以一次执行任意两条指令。为这个片断找出一个最短的执行方法。不要忘记考虑在各个复制步骤中使用哪个寄存器最好。同时，记住利用 8.6 节中描述的寄存器描述符所提供的信息，以避免不必要的加载和保存运算。

10.5 软件流水线化

正如在本章的引言部分所讨论的，数值应用往往具有很大的并行性。特别地，它们经常含有各次迭代之间相互完全独立的循环。从并行化的角度看，这些被称为 do-all 循环的循环很有吸引力，原因是它们的迭代可以并行执行，其加速比和循环的迭代数目呈线性关系。具有很多迭代的 do-all 循环具有的并行性足以充满一个处理器的所有资源。能否充分利用循环中的可用并行性完全依赖于调度器的能力。本节描述一个被称为*软件流水线化*（software pipelining）的算法。它可以同时对整个循环进行调度，充分利用各个迭代之间的并行性。

10.5.1 引言

在本节中，我们将使用例 10.12 中的 do-all 循环来解释软件流水线化。我们首先说明跨越迭代的调度极其重要，原因是在单一迭代过程中的运算之间的并行性相对较小。然后，我们说明循环展开技术通过让被展开迭代相互重叠执行来提高程序的性能。但是，被展开循环之间的界限仍然为代码移动设置了障碍，还有很多可改进性能机会没有被循环展开技术充分利用。另一方面，软件流水线化技术将多个连续的迭代持续地交叠执行，直到所有迭代执行完毕为止。这个技术使得软件流水线化技术可以生成高效紧凑的代码。

例 10.12 下面是一个典型的 do-all 循环：

```
for (i = 0; i < n; i++)
    D[i] = A[i]*B[i] + c;
```

上面循环中各个迭代对不同的内存位置执行写运算，而这些被写的位置不同于任何被读的位置。因此各个迭代之间没有内存依赖关系，所有的迭代都可以并行地进行。

在本节中，我们选择下面的模型作为目标机器。在这个模型中，

- 机器可以在同一个时钟周期内发出：一个加载运算、一个保存运算、一个算术运算和一个分支运算。
- 机器有一个如下形式的循环回归运算指令

  `BL R, L`

  这个运算把寄存器 R 的值减一，并且在结果不为 0 的情况下跳转到位置 L。
- 内存运算有一个自动加一的寻址模式，通过寄存器之后的 ++ 符号表示。在每次访问之后，寄存器自动地加一，指向接下来的一个地址。
- 算术运算是完全流水线化的。每个时钟周期可以启动一个算术运算，但是结果要到 2 个时钟周期后才可用。所有其他指令的执行延时为一个时钟周期。

如果每次只对一个迭代进行调度，那么在这个机器模型上得到的最好调度方案如图 10-17 所示。该图中也指明了一些有关数据布局的假设：寄存器 R1、R2 和 R3 存放数组 A、B 和 D 的开始地址，寄存器 R4 存放常量 c，而寄存器 R10 存放值 $n-1$，这个值在循环之外计算。这个计算过程大部分是串行的，共需要 7 个时钟周期。只有循环回归运算指令的执行和迭代的最后一个运算的执行重叠。 □

```
      //   R1, R2, R3 = &A, &B, &D
      //   R4         = c
      //   R10        = n-1

L:    LD  R5, 0(R1++)
      LD  R6, 0(R2++)
      MUL R7, R5, R6
      nop
      ADD R8, R7, R4
      nop
      ST  0(R3++), R8        BL R10, L
```

图 10-17 例 10.12 的局部调度代码

一般来说，我们可以展开一个循环的多个迭代来获得较好的硬件利用率。但是这么做会增加代码的大小，会对程序的整体性能产生负面影响。因此，我们必须选择一个折衷方案，选择适当的迭代展开次数以选择最大的性能提升，但是又不能过度扩展代码。下面的例子解释了这种折衷方案。

例 10.13 虽然在例 10.12 中循环的各个迭代内部几乎找不到并行性，但各个迭代之间依然具有很多并行性。循环展开技术把该循环的多个迭代放到一个大基本块中，然后使用一个简单的列表调度算法来对这些运算进行调度，使之并行运行。如果把我们的例子中的循环展开四次并把算法 10.7 应用于展开得到的代码，那么就可以得到图 10-18 显示的调度方案（为简单起见，我们忽略寄存器分配的细节）。这个循环执行了 13 个时钟周期，或者说每个迭代执行 3.25 个周期。

```
L:  LD
    LD
        LD
    MUL LD
        MUL  LD
    ADD      LD
        ADD       LD
    ST       MUL  LD
        ST        MUL
             ADD
                  ADD
             ST
                  ST   BL (L)
```

图 10-18 例 10.12 的未展开的代码

一个被 k 次展开的循环至少需要 $2k+5$ 个时钟周期，得到的吞吐量是每 $2+5/k$ 个时钟周期一个迭代。因此，我们展开的迭代越多，循环就运行得越快。当 $k \to \infty$ 时，一个完全展开的循环可以平均每两个时钟周期执行一次迭代。但是，我们展开的迭代越多，得到的代码也越大。我们当然承担不起把一个循环的全部迭代都展开的代价。把这个循环展开 4 次生成了有 13 条指令的代码，执行

时间是最优情况的 163%；把这个循环展开 8 次生成了带有 21 条指令的代码，执行时间是最优情况的 131%。反过来，如果我们希望执行时间只是最优情况的 110%，我们需要把这个循环展开 25 次，这将产生带有 55 条指令的代码。 □

10.5.2 循环的软件流水线化

软件流水线化提供了一个方便的优化方法，能够在优化资源使用的同时保持代码的简洁。让我们用一个连续的例子来说明这个想法。

例 10.14 图 10-19 中显示的是把例 10.12 展开 5 次之后得到的代码(我们再次忽略了对寄存器使用方面的考虑)。第 i 行中显示的是在第 i 个时钟周期发出的所有运算指令；第 j 列中显示的是第 j 次迭代的全部运算。请注意，相对于各个迭代的开始时间，每个迭代都有同样的调度方案，同时要注意每个迭代都在前一个迭代开始两个时钟周期之后开始。可见，这个调度方案满足所有的资源和数据依赖约束。

时钟	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
1	LD				
2	LD				
3	MUL	LD			
4		LD			
5		MUL	LD		
6	ADD		LD		
7			MUL	LD	
8	ST	ADD		LD	
9				MUL	LD
10		ST	ADD		LD
11					MUL
12			ST	ADD	
13					
14				ST	ADD
15					
16					ST

图 10-19 例 10.12 经过 5 次迭代展开后得到的代码

我们看到，在第 7 和第 8 个时钟周期运行的运算和在第 9 和第 10 个周期运行的运算是一样的。第 7 和第 8 个时钟周期执行的运算来自原程序中的前四个迭代。第 9 和第 10 个时钟周期执行的运算也是来自四个迭代，不过这次是来自第 2 到第 5 个迭代。实际上，我们可以不停地执行同样的多运算指令对，不断有一个最老的迭代退出，又有一个新的迭代加入，直到运行完所有的迭代。

如果假设这个循环至少有 4 个迭代，那么这样的动态行为可以用图 10-20 中显示的代码简洁地编码。图中的每一行对应于一条机器指令。第 7 行和第 8 行形成了一个两个时钟周期的循环。这个循环将执行 $n-3$ 次，其中 n 是原循环中的迭代次数。 □

```
1)        LD
2)        LD
3)        MUL   LD
4)              LD
5)              MUL   LD
6)        ADD         LD
7)  L:                MUL   LD
8)        ST    ADD           LD    BL (L)
9)                            MUL
10)             ST    ADD
11)
12)                   ST    ADD
13)
14)                         ST
```

图 10-20 例 10.12 的经软件流水线化的代码

上面描述的技术被称为*软件流水线化*技术，因为这是原本用于硬件流水线调度的技术在软件中的对应。我们可以把这个例子中各个迭代执行的调度方案当作一个 8 阶段的流水线。每两个时钟周期就可以在这条流水线上启动一个新迭代。在开始的时候，这条流水线中只有一个迭代在运行。在第一个迭代进行到第三阶段的时候，第二个迭代开始进入它的第一个执行阶段。

到第 7 个时钟周期时，流水线被前面四个迭代充满。在稳定状态下有四个连续的迭代同时运行。每当流水线中最老的迭代退出时就有一个新的迭代被启动。当我们运行完所有的迭代时，流水线开始排空，其中的所有迭代运行结束。用来填充流水线的指令序列(在这个例子中是第 1 到第 6 行)被称为*序言*(prolog)；第 7 和第 8 行被称为*稳定状态*(steady state)；用来排空流水线的指令序列(即第 9 行到第 14 行)被称为*尾声*(epilog)。

对于这个例子，我们知道这个循环不可能运行得比每两个时钟周期一个迭代更快。原因是目标机器每个时钟周期只能发出一个读指令，而每个迭代有两个读指令。上面的经软件流水线化的循环在 $2n+6$ 个时钟周期内执行完毕，其中 n 是原循环的迭代次数。当 $n \to \infty$ 时，这个循环的通量接近每两个时钟周期一次迭代。因此，和循环展开技术不同，软件调度可以用一个非常简洁的代码序列给出最优调度方案的编码。

请注意，对于单个迭代而言，这个调度方案的运行时间并不是最短的。和图 10-17 中显示的局部优化的调度方案相比，这个方案在 ADD 运算之前引入了一个延时。引入这个延时是调度策略之一，其目的是使这个调度方案可以在保证没有资源冲突的情况下每两个时钟周期启动一个迭代。如果我们坚持使用局部紧凑的调度方案，为了避免资源冲突，各次启动之间的间隔不得不延长到 4 个时钟周期，而吞吐率将被减半。这个例子说明了流水线调度的一个重要原则：必须小心选择调度方案以便优化吞吐量。虽然一个局部紧凑的调度方案可以使完成一个迭代的时间降到最低，但是在流水线化之后得到的吞吐量却可能是次优的。

10.5.3 寄存器分配和代码生成

我们首先讨论例 10.14 中经过软件流水线化的循环的寄存器分配。

例 10.15 在例 10.14 中，第一个迭代中的乘法运算结果在第 3 个时钟周期生成，在第 6 个时钟周期使用。在这两个时钟周期之间，第二次迭代中的这个乘法运算又在第 5 个时钟周期生成一个新的结果，这个值在第 8 个时钟周期使用。这两次迭代的结果必须保存到不同的寄存器中，以防止它们之间互相干扰。因为干扰只会在两个相邻的迭代之间发生，使用两个寄存器就可以避免这种干扰：一个寄存器用于奇数次迭代，另一个寄存器用于偶数次迭代。因为奇数次迭代的代码和偶数次迭代的代码不同，稳定状态循环的代码大小是原来的两倍。这个代码可以用于执行任何具有大于等于 5 的奇数次迭代的循环。

```
if (N >= 5)
    N2 = 3 + 2 * floor((N-3)/2);
else
    N2 = 0;
for (i = 0; i < N2; i++)
    D[i] = A[i]* B[i] + c;
for (i = N2; i < N; i++)
    D[i] = A[i]* B[i] + c;
```

图 10-21 例 10.12 中循环在源语言层次上的展开

为了处理迭代次数小于 5 的循环和具有偶数次迭代的循环，我们生成的代码在源语言层次上和图 10-21 中的代码等价。第一个循环被流水线化了，它的机器语言层次的等价表示见图 10-22。图 10-21 的第二个循环不需要优化，因为它最多迭代 4 次。 □

```
1.       LD R5,0(R1++)
2.       LD R6,0(R2++)
3.       LD R5,0(R1++)   MUL R7,R5,R6
4.       LD R6,0(R2++)
5.       LD R5,0(R1++)   MUL R9,R5,R6
6.       LD R6,0(R2++)   ADD R8,R7,R4
7.  L:   LD R5,0(R1++)   MUL R7,R5,R6
8.       LD R6,0(R2++)   ADD R8,R9,R4   ST 0(R3++),R8
9.       LD R5,0(R1++)   MUL R9,R5,R6
10.      LD R6,0(R2++)   ADD R8,R7,R4   ST 0(R3++),R8   BL R10,L
11.                      MUL R7,R5,R6
12.                      ADD R8,R9,R4   ST 0(R3++),R8
13.
14.                      ADD R8,R7,R4   ST 0(R3++),R8
15.
16.                                     ST 0(R3++),R8
```

图 10-22 在例 10.15 中经过软件流水线化和寄存器分配之后得到的代码

10.5.4 Do-Across 循环

软件流水线化技术也可以用于各个迭代之间存在数据依赖关系的循环。这样的循环被称为 *do-across* 循环。

例 10.16 代码

```
for (i = 0; i < n; i++) {
    sum = sum + A[i];
    B[i] = A[i] * b;
}
```

的两个连续迭代之间具有数据依赖关系，因为前一次的 sum 值和 $A[i]$ 相加得到新的 sum 值。如果目标机器可以提供足够的并行性，这个求和运算可以在 $O(\log n)$ 时间内完成。但是为了本次讨论，我们假设必须遵守所有的顺序依赖关系，上面的加法必须以原来的顺序完成。因为我们假设的机器模型需要两个时钟周期才能完成一个 ADD 运算，所以循环的运行不可能快过每两个时钟周期一个迭代。给该机器增加更多的加法器和乘法器都不会使循环运行得更快。像这样的 do-across 循环的吞吐量受迭代之间的依赖链的限制。

图 10-23a 显示了每个迭代的最好的局部紧凑的调度方案，经过软件流水线化处理的代码在图 10-23b 中显示。这个软件流水线化的循环每两个时钟启动一次迭代，因此运行的速度是最优的。 □

```
    // R1 = &A; R2 = &B
    // R3 = sum
    // R4 = b
    // R10 = n-1

L:  LD R5, 0(R1++)
    MUL R6, R5, R4
    ADD R3, R3, R4
    ST R6, 0(R2++)          BL R10, L
```

a) 最好的局部紧凑的调度方案

```
    // R1 = &A; R2 = &B
    // R3 = sum
    // R4 = b
    // R10 = n-2

    LD R5, 0(R1++)
    MUL R6, R5, R4
L:  ADD R3, R3, R4          LD R5, 0(R1++)
    ST R6, 0(R2++)          MUL R6, R5, R4   BL R10, L
                            ADD R3, R3, R4
                            ST R6, 0(R2++)
```

b) 调度方案的软件流水线化版本

图 10-23 一个 do-across 循环的软件流水线化

10.5.5 软件流水线化的目标和约束

软件流水线化的主要目标是使一个长时间运行的循环的吞吐量最大，次要目标之一是使生成代码保持合理的大小。换句话说，经过软件流水线化的循环应该有一个较小的流水线稳定状态。我们可以要求每个迭代的相对调度方案相同，并要求各个迭代启动的时间间隔相同，从而得到一个较小的稳定状态。因为循环的吞吐量是启动间隔的倒数，所以软件流水线化的目标是使这个间隔最小化。

一个数据依赖图 $G=(N,E)$ 的软件流水线调度方案可以描述为

1）一个启动间隔 T。

2）一个相对调度方案 S。对每个运算，它给定了该运算相对于它所处迭代的开始时刻的执行时间。

因此，如果从 0 开始计算时钟周期的话，第 i 个迭代的一个运算 n 将会在第 $i \times T + S(n)$ 个时钟周期上运行。和所有其他的调度问题一样，软件流水线化有两种约束：资源和数据依赖关系。

下面我们详细讨论每一种约束。

模数资源预约

令一个机器的资源表示为 $R=[r_1,r_2,\cdots]$，其中 r_i 表示第 i 种资源的可用数目。如果一个循环的单次迭代需要 n_i 个单元的第 i 种资源，那么一条流水线化的循环的平均启动间隔至少是 $\max_i(n_i/r_i)$ 个时钟周期。软件流水线化要求在任何一对相邻迭代之间的启动间隔是一个常量值。因此，它的启动间隔至少是 $\max_i\lceil n_i/r_i\rceil$ 个时钟周期。如果 $\max_i(n_i/r_i)$ 小于 1，那么把源代码少量展开几次就有助于提高代码效率。

例 10.17 让我们回到图 10-20 所示的经过软件流水线化处理的循环。回顾一下，目标机器可以在每个时钟周期内发出一个加载指令、一个算术运算指令、一个保存指令和一个循环回归分支指令。因为这个循环有两个加载运算、两个算术运算和一个保存运算，所以根据资源约束，这个循环的最小启动间隔是 2 个时钟周期。

图 10-24 显示了四个连续迭代在不同时刻的资源需求。随着被启动迭代数量的增加，所需的资源也越来越多，最终达到稳定状态下的最大资源需求。令 RT 为表示单个迭代所需资源的资源预约表，并令 RT_S 表示循环的稳定状态所需要的资源。RT_S 把每隔 T 个时钟周期启动的四个相邻迭代所需的资源组合起来。RT_S 表的第 0 行所需的资源是 $RT[0]$、$RT[2]$、$RT[4]$ 和 $RT[6]$ 所需资源的总和。类似地，表中第 1 行所需资源对应于 $RT[1]$、$RT[3]$、$RT[5]$ 和 $RT[7]$ 所需资源的总和。也就是说，稳定状态下第 i 行所需资源可以由下面的公式给出：

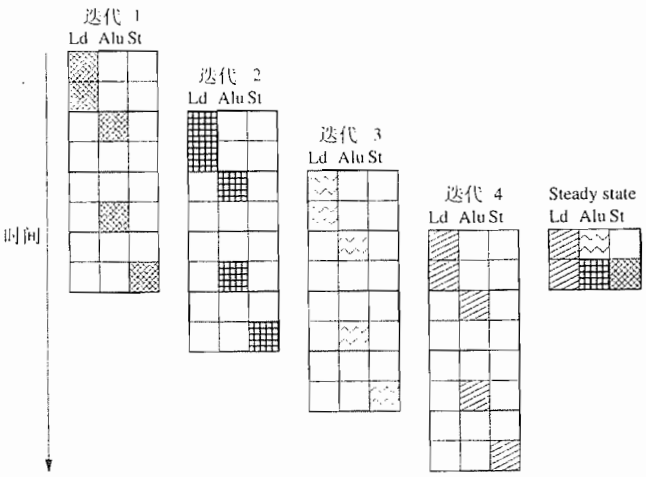

图 10-24 例 10.13 中的代码的四个连续迭代的资源需求

$$RT_S[i]=\sum_{t\mid(t \bmod 2)=i} RT[t]$$

我们把表示稳定状态的资源预约表称为这条流水线化的循环的*模数资源预约表*(modular resource-reservation)。

为了检查软件流水线调度方案是否存在冲突，我们只需要检查模数资源预约表中指出的资源需求。如果在稳定状态下的资源需求可以被满足，那么在序言和尾声中，即在稳定状态循环之前和之后的代码部分中，资源需求也一定可以被满足。□

总的来说，给定一个启动间隔 T 和单个迭代的一个资源预约表 RT，流水线化的调度方案在一个资源向量为 R 的机器上没有资源冲突，当且仅当 $RT_S[i]\leqslant R$ 对 $i=0,1,\cdots,T-1$ 都成立。

数据依赖约束

在软件流水线化中的数据依赖关系和我们至今为止遇到的依赖关系不同，因为它们可能会形成环。一个运算可能会依赖于前一个迭代中同一个运算的结果。现在仅仅在依赖边上加上一个表示延时的标号已经不够了，我们还需要区分同一个运算在不同迭代中的实例。如果在第 i 次迭代中的运算 n_2 必须在第 $i-\delta$ 次迭代中的运算 n_1 执行至少 d 个时钟之后才可以执行，那么我们给依赖边 $n_1\rightarrow n_2$ 加上标号 $<\delta,d>$。令 S 是软件流水线化的调度方案，它是一个从数据依赖图中

的结点到整数的函数，并令 T 为启动时间间隔的目标，那么

$$(\delta \times T) + S(n_2) - S(n_1) \geqslant d$$

其中的迭代距离 δ 必须是非负的。而且，给定一个由数据依赖图的边组成的环，至少一个边具有正的迭代距离。

例 10.18 考虑下面的循环，并假设我们不知道 p 和 q 的值：

```
for (i = 0; i < n; i++)
    *(p++) = *(q++) + c;
```

我们必须假设任何一对 *(p++) 和 *(q++) 都可能访问同一个内存位置。因此，所有的读和写运算都必须按照原来的串行顺序进行。假设本例中目标机器和例 10.12 中描述的机器有同样的特性，这段代码的数据依赖边如图 10-25 所示。但是，请注意我们忽略了循环控制指令。本来应该有这条指令的，它要么计算并测试 i 的值，要么根据 R1 或 R2 的值来完成这个测试。 □

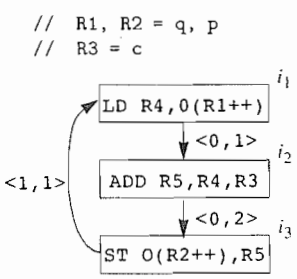

图 10-25 例 10.18 的数据依赖图

如下例所示，两个相关运算之间的迭代距离可以大于 1：

```
for (i = 2; i < n; i++)
    A[i] = B[i] + A[i-2];
```

在第 i 个迭代中写入的值在两个迭代之后才会被用到。因此在保存 $A[i]$ 的运算和加载 $A[i-2]$ 的依赖边之间的迭代距离是 2。

一个循环中出现的数据依赖环还对循环的执行吞吐量增加了另一个限制。比如，图 10-25 中的数据依赖环限定了两个连续迭代中的加载运算之间必须有至少 4 个时钟周期的延时。也就是说，循环的执行不可能快过每 4 个时钟周期一次迭代。

一个被流水线化的循环的启动间隔不小于

$$\max_{c\text{是一个}G\text{中的圈}} \left\lceil \frac{\sum_{e\text{在}c\text{中}} d_e}{\sum_{e\text{在}c\text{中}} \delta_e} \right\rceil$$

个时钟周期。

总结一下，每个被软件流水线化的循环的启动间隔受到每个迭代的资源使用情况限制。也就是说，对于每一类资源，启动间隔必须不小于一次迭代所需该类资源的数目除以机器上该类资源的可用数量所得的商。另外，如果循环中存在数据依赖环，那么它的启动间隔还必须不小于这个环中的延时总数除以环中迭代距离之和得到的商。这些量的最大值定义了启动间隔的下界。

10.5.6 一个软件流水线化算法

软件流水线化的目标是找到一个具有最小启动间隔的调度方案。这个问题是 NP 完全的，并且可以被写成一个整数线性规划问题。我们已经说明，如果知道最小的启动间隔，那么调度算法可以在放置各个运算时使用模数资源预约表来避免资源冲突。但是只有当我们找到一个调度方案之后才能知道最小启动间隔是什么。我们怎样才能解开这样的循环套？

我们可以按照上面讨论的方法根据循环的资源需求和依赖环计算得到启动间隔的下界。我们已知启动间隔必须大于这个下界。如果我们可以找到一个调度方案使得启动间隔就是这个下界，那么就找到了最优的调度方案。如果我们找不到这样的调度方案，可以再使用大一点的启动间隔进行尝试，直到找到一个符合要求的调度方案为止。请注意，如果使用启发式搜索而不是穷尽搜索，那么这个过程找到的可能不是最优的调度方案。

我们能否找到接近下界的调度方案依赖于相应数据依赖图的性质以及目标机的体系结构。如果依赖图是无环的，并且每条机器指令只需要一个单元的某种资源，那么我们可以很容易地找到最优的调度方案。如果可用的硬件资源超过了有环的依赖图所需要的资源，那么也很容易找到启动间隔接近于下界的调度方案。对于这些情况，建议一开始就把下界作为初始的启动间隔目标，然后逐渐把目标增加一个时钟周期，并且每增加一次进行一次调度尝试。另一种可能性是使用二分搜索法来寻找启动间隔。我们可以把列表调度法为单次迭代生成的调度方案的长度作为启动间隔的上界。

10.5.7 对无环数据依赖图进行调度

为简单起见，我们现在假设即将进行软件流水线化处理的循环只包含一个基本块。在10.5.11节中将放宽这个假设条件。

算法 10.19 对一个无环依赖图进行软件流水线化处理。

输入： 一个机器资源向量 $R=[r_1,r_2,\cdots]$，其中 r_i 表示第 i 种资源的可用单元数量；一个数据依赖图 $G=(N,E)$。N 中的每个运算 n 用它的资源预约表 RT_n 作为标号；E 中的每条边 $e=n_1\rightarrow n_2$ 上有标号 $<\delta_e,d_e>$。这个标号表示 n_2 只能在往前第 δ_e 个迭代中的结点 n_1 执行 d_e 个时钟周期之后才可以执行。

输出： 一个经过软件流水线化的调度方案 S 和一个启动间隔 T。

方法： 执行图10-26中的程序。 □

算法10.19将无环的数据依赖图进行软件流水线化处理。这个算法首先基于图中运算的资源需求找到启动间隔的界限 T_0。然后它尝试以 T_0 为启动间隔的目标，寻找一个软件流水线化的调度方案。如果算法不能为当前目标找到一个调度方案，它就不断增加启动间隔并重复尝试。

```
main() {
    T0 = max_j ⌈ (Σ_{n,i} RTn(i,j)) / rj ⌉;
    for (T = T0, T0 + 1, ..., 直到 N 中的所有结点都已经被调度完毕 ) {
        RT = 一个具有 T 行的空的资源预约表 ;
        for (按照带优先级的拓扑顺序访问 N 中的每个结点 n ) {
            s0 = max_{E中的边 e=p→n} (S(p) + de);
            for (s = s0, s0 + 1, ..., s0 + T - 1)
                if (NodeScheduled(RT, T, n, s)) break;
            if ( n无法在RT中调度 ) break;
        }
    }
}

NodeScheduled(RT, T, n, s) {
    RT' = RT;
    for (在 RTn 中的每一行 i )
        RT'[(s + i) mod T] = RT'[(s + i) mod T] + RTn[i];
    if (对于所有i, RT'(i) ≤ R) {
        RT = RT';
        S(n) = s;
        return true;
    }
    else return false;
}
```

图 10-26 无环依赖图的软件流水线化算法

这个算法在每次尝试中使用了一个列表调度方法。它使用一个模数资源预约表 RT 来跟踪流水线的稳定状态所要求的资源。运算按照拓扑顺序进行调度，以便总是能够通过推迟运算来满足数据依赖关系。为了调度一个运算，它首先根据数据依赖约束找到一个下界 s_0。然后，它调用 *NodeScheduled* 来检测在稳定状态上可能发生的资源冲突。如果发现了资源冲突，该算法试图把这个运算安排在下一个时钟周期。因为资源冲突检测的取模特性，如果发现该运算在连续 T 个时钟周期上都有冲突，那么继续尝试也不会有用。此时，这个算法认为对当前启动间隔目标的尝试已经失败，继续尝试另一个启动间隔。

把各个运算尽早安排的启发式规则往往会使单个迭代的调度方案的长度最小化。但是，尽早安排一条指令可能会加长某些变量的生命期。比如，加载数据的运算往往会被较早安排，有时候会在数据被使用前很早就执行。处理这个问题的一个简单的启发规则是逆向地调度一个依赖图，理由是加载运算通常要多于保存运算。

10.5.8 对有环数据依赖图进行调度

依赖环明显地增加了软件流水线化的复杂性。当按照拓扑顺序对一个无环图中的运算进行调度时，被调度的运算之间的数据依赖关系只能给出每个运算位置的下界。结果，算法总是能够通过推迟运算来满足数据依赖关系。有环的图没有“拓扑排序”的概念。实际上，给定一个环中的一对运算，放置一个运算会限定第二个运算的位置的下界和上界。

令 n_1 和 n_2 是一个依赖环中的两个运算，S 是一个软件流水线调度方案，而 T 是这个调度方案的启动间隔。一个带有标号 $<\delta_1, d_1>$ 的依赖边 $n_1 \rightarrow n_2$ 对 $S(n_1)$ 和 $S(n_2)$ 加上了如下约束：

$$(\delta_1 \times T) + S(n_2) - S(n_1) \geqslant d_1$$

类似地，一个带有标号 $<\delta_2, d_2>$ 的依赖边 $n_2 \rightarrow n_1$ 增加了如下约束：

$$(\delta_2 \times T) + S(n_1) - S(n_2) \geqslant d_2$$

因此

$$S(n_1) + d_1 - (\delta_1 \times T) \leqslant S(n_2) \leqslant S(n_1) - d_2 + (\delta_2 \times T)$$

一个图的强连通分量(Strongly Connected Component，SCC)是满足如下条件的一个结点集合，其中的每个结点都可以从集合中的所有其他结点到达。对 SCC 中的一个结点进行调度将会从上下两个方向限制其他各个结点的可行时间。如果存在一个从 n_1 到 n_2 的路径 p，那么有

$$S(n_2) - S(n_1) \geqslant \sum_{e \text{在} p \text{中}} (d_e - (\delta_e \times T)) \tag{10.1}$$

请注意下面的情况：

1）沿着任何一个环，各个边上的 δ 值的总和必须为正。如果和是 0 或者负数，就表明环中的一个运算要么必须在它自己之前执行，要么所有迭代中的该运算都在同一时钟周期执行。

2）一个迭代中的各运算的调度方案和所有迭代中的调度方案相同，这个要求实质上就是“软件流水线”的含义。结果，一个环上的延时(即数据依赖图中边的标号的第二个元素)的总和除以环上的迭代距离的总和所得的商就是启动间隔 T 的一个下界。

当我们把这两点联系起来，就可以看到，如果 p 是一个环，那么对于任何可行的启动间隔 T，式(10.1)的右边部分的值必然是负数或零。由此可见，对于结点位置的最强约束来自于简单路径——那些不包含环的路径。

因此，对于每个可行的启动间隔 T，计算每对结点之间的数据依赖关系的传递效果就等同于寻找从第一个结点到达第二个结点的最长的简单路径。不仅如此，因为环不会增加一条路径的长度，所以可以用一个简单的动态规划算法在没有“简单路径”需求的情况下来寻找最长路径。这样得到的长度也一定是最长简单路径的长度(见练习 10.5.7)。

例 10.20 图 10-27 显示了有四个结点 a、b、c、d 的数据依赖图。每个结点上附加了该结点的资源预约表，每条边上附加了它的迭代距离和延时。假设这个例子中的目标机器的每一种资源都有一个单元。因为对第一种资源有三处使用，而第二种资源有两处使用，所以启动间隔必须不小于 3 个时钟。在这个图中有两个连通分量：第一个是只包含了

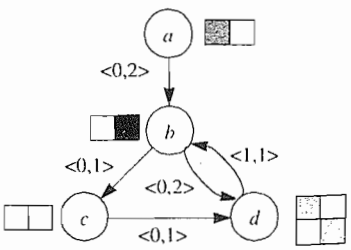

图 10-27 例 10.20 中的依赖图和资源需求

结点 a 的分量，第二个包含了结点 b、c 和 d。最长的环 b、c、d、b 的总延时是 3 个时钟周期。这个环把相隔一个迭代的结点连接起来。因此，根据数据依赖环约束得到的启动间隔的下界也是 3 个时钟周期。

对 b、c 或 d 中的任何一个进行调度都会对分量中的其他结点产生约束。令 T 为启动间隔。图 10-28 显示了传递依赖关系。图 10-28a 显示了每条边的延时和迭代距离 δ。其中的延时是直接表示的，而 δ 则是通过在延时上“加”上 $-\delta T$ 来表示的。

如果两个结点之间存在简单路径，那么图 10-28b 中就显示了这两个结点之间的最长简单路径的长度。表中的项是图 10-28a 中给出的路径上各条边的表达式的和。然后，在图 10-28c 和图 10-28d 中，我们看到的表达式是将图 10-28b 的表达式中的 T 替换为两个相关值(即 3 和 4)之后得到的表达式。根据不同的 T 值，两个结点 n_1 和 n_2 的调度时间位置之差 $S(n_2)-S(n_1)$ 必须不小于在图10-28c 或图 10-28d 中的项 (n_1,n_2) 的值。

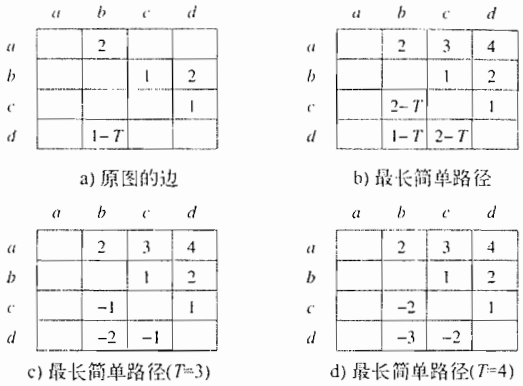

图 10-28 例 10.20 中的传递约束

比如，考虑图 10-28 中给出的表示从 c 到 b 的最长(简单)路径的项 $2-T$。从 c 到 b 的最长简单路径是 $c\rightarrow d\rightarrow b$。这条路径上的总延时是2，而 δ 的和是 1，它表明迭代编号必须加 1。因为 T 表示每个迭代和前一个迭代的时间差异，为 b 安排的时钟周期必须至少是安排给 c 的时钟周期之后的第 $2-T$ 个时钟周期。因为 T 至少是 3，我们实际上是说 b 必须被安排在 c 之前的 $T-2$ 个时钟周期或再晚一些，但是不能更早了。

请注意，考虑从 c 到 b 的非简单路径并不会产生更强的约束。我们可以在路径 $c\rightarrow d\rightarrow b$ 上加上由 d 和 b 组成的环的任意多次迭代。如果我们加上 k 个这样的环，因为路径的总延时为 3，而环上的 δ 的总和是 1，我们得到的路径长度为 $2-T+k(3-T)$。因为 $T\geqslant 3$，所以这个长度绝不会超过 $2-T$，即 b 的时钟和 c 的时钟的差的下界是 $2-T$，也就是我们考虑最长简单路径时得到的界限。

比如，从项 (b,c) 和 (c,d) 我们可以知道

$$S(c)-S(b)\geqslant 1$$
$$S(b)-S(c)\geqslant 2-T。$$

也就是说，

$$S(b)+1\leqslant S(c)\leqslant S(b)-2+T$$

如果 $T=3$，则

$$S(b)+1\leqslant S(c)\leqslant S(b)+1$$

等价地说，c 必须被安排在 b 后一个时钟周期上。但是，如果 $T=4$，则

$$S(b)+1\leqslant S(c)\leqslant S(b)+2$$

也就是说，c 可以被安排在 b 后的一个或两个时钟周期上。

给定所有点对之间的最长路径的信息，我们可以很容易地计算出由于数据依赖的原因，一个结点可以放置在什么位置。我们看到，当 $T=3$ 时放置结点 b 的位置是没有松弛度的，而当 T 增加的时候这个松弛度会增加。 □

算法 10.21 软件流水线化。

输入： 一个机器资源向量 $R=[r_1,r_2,\cdots]$，其中 r_i 表示第 i 种资源的可用单元的数量；一个数据依赖图 $G=(N,E)$。N 中的每个运算 n 的标号为它的资源预约表 RT_n；E 中的每条边 $e=n_1\rightarrow n_2$ 上有标号 $<\delta_e,d_e>$，这个标号表示 n_2 的执行时刻不能早于向前第 δ_e 个迭代中的结点 n_1 之后的 d_e 个时钟周期。

输出： 一个软件流水线化的调度方案 S 和一个启动间隔 T。

方法： 执行图 10-29 中的程序。 □

```
main() {
    E' = {e | e in E, δ_e = 0};
    T_0 = max( max_j ⌈ Σ_{n,i} RT_n(i,j) / r_j ⌉ , max_{c a cycle in G} ⌈ Σ_{e in c} d_e / Σ_{e in c} δ_e ⌉ );
    for (T = T_0, T_0 + 1, ... 或者直到 G 中的所有 SCC 都已经被调度完毕){
        RT = 一个 T 行的空资源预约表;
        E* = AllPairsLongestPath(G, T);
        for ( 以带优先级的拓扑顺序遍历 G 中的每个 SCC C ) {
            for ( 对 C 中的各个 n )
                s_0(n) = max_{e=p→n in E*, p scheduled} (S(p) + d_e);
            first = 某个使得 s_0(n) 取最小值的 n ;
            s_0 = s_0(first);
            for (s = s_0; s < s_0 + T; s = s + 1)
                if (SccScheduled (RT, T, C, first, s)) break;
            if ( C 不能在 RT 中调度 ) break;
        }
    }
}

SccScheduled(RT, T, c, first, s) {
    RT' = RT;
    if (not NodeScheduled (RT', T, first, s)) return false;
    for (按照 E' 中各条边的带优先级的拓扑排序
                访问 c 中余下的每个 n ) {
        s_l = max_{e=n'→n in E*, n' in c, n' scheduled} (S(n') + d_e - (δ_e × T));
        s_u = min_{e=n→n' in E*, n' in c, n' scheduled} (S(n') - d_e + (δ_e × T));
        for (s = s_l; s ≤ min(s_u, s_l + T - 1); s = s + 1)
            if (NodeScheduled(RT', T, n, s)) break;
        if ( n 不能够在 RT' 中调度 ) return false;
    }
    RT = RT';
    return true;
}
```

图 10-29 一个针对有环依赖图的软件流水线化算法

算法 10.21 在高层结构上和只能处理无环图的算法 10.19 类似。在本算法处理的情况中，最小的启动间隔不仅受到资源需求的限制，也受到图中数据依赖环的限制。整个图是按照每次处理一个强连通分量的方式进行调度的。通过把每个强连通分量当作一个单元，在强连通分量之间的边必然形成一个无环图。算法 10.19 的顶层循环按照拓扑顺序来调度图中的结点，而算法 10.21 的顶层循环按照拓扑顺序调度各个强连通分量。和前面一样，如果算法不能调度所有的分量，那么它就会尝试较大的启动间隔。请注意，如果给定一个无环的数据依赖图，算法 10.21 和算法 10.19 的做法是完全一样的。

算法 10.21 要计算得到额外两个边集：E'是所有的迭代距离为 0 的边，而 E^* 是所有点对之

间的最长路径边集。也就是说，对每个结点对(p,n)，只要有一条从p到n的路径，在E^*中就有一条边e，该边所关联的长度d_e是从p到n的最长简单路径的长度。对于启动间隔目标T的每一个取值都需要计算相应的E^*。也可以像我们在练习10.20中所做的那样，先使用T的符号化值一次性完成这个计算过程，然后在每一次迭代的时候把T替换为实际的启动间隔的值。

算法10.21使用了回溯。如果它不能完成一个SCC的调度，它就会延后一个时钟周期再次对整个SCC进行调度。这些调度尝试会持续T个时钟周期。回溯是很重要的，因为如例10.20所示，对于一个SCC中的第一个结点的调度安排可能会完全地决定所有其他结点的调度安排。如果这个调度方案不能和至今已经产生的调度方案配合，那么这次尝试就失败了。

在对一个SCC进行调度时，对该分量中的每个结点，此算法确定了满足E^*中的传递数据依赖关系的最早可调度的时间。然后，算法选择具有最早开始时间的结点作为第一个被调度的结点。然后，此算法调用*SccScheduled*，试图根据这个最早开始时间实际调度这个分量。如果尝试失败，此算法将逐次增大开始时间，不断尝试。该算法最多做T次尝试。如果T次尝试失败了，该算法就会尝试另一个启动间隔。

算法*SccScheduled*和算法10.19类似，但是有三大不同之处：

1）*SccScheduled*的目标是对输入的强连通分量在给定时间位置s上进行调度。如果该强连通分量的第一个结点不能被安排在s上，*SccScheduled*就返回*false*。在需要时，主函数*main*可以使用一个较晚的时间位置再次调用*SccScheduled*。

2）在强连通分量中的结点按照E'中的边集所确定的拓扑顺序进行调度。因为E'中的所有边的迭代距离都是0，这些边不会穿越任何迭代边界，也就不会形成环（穿越迭代边界的边被称为穿越循环的）。只有穿越循环的依赖会设置指令可调度位置的上界。因此，这个调度顺序以及尽早调度安排各条指令的策略把后继结点的可调度范围最大化了。

3）对于强连通分量，依赖关系既给出了一个结点的可调度范围的下界，又给出了其上界。*SccScheduled*计算了这些范围，并使用它们进一步限制调度尝试。

例10.22 让我们把算法10.21应用到例10.20中的有环的数据依赖图上。算法首先计算出这个例子的启动间隔的下界是3个时钟周期。注意，这个下界不可能达到。当启动间隔T是3时，图10-28中的传递依赖关系决定了$S(d)-S(b)=2$。把结点b和d安排在间隔两个时钟的位置会在长度为3的模数资源预约表中产生一个冲突。

图10-30说明了算法10.21是如何处理这个例子的。它首先试图找到一个启动间隔为3个时钟

尝试	启动间隔	结点	区间	调度安排	模数资源预约表
1	$T=3$	a b c	$(0,\infty)$ $(2,\infty)$ $(3,3)$	0 2 --	
2	$T=3$	a b c d	$(0,\infty)$ $(2,\infty)$ $(4,4)$ $(5,5)$	0 3 4 --	
3	$T=3$	a b c d	$(0,\infty)$ $(2,\infty)$ $(5,5)$ $(6,6)$	0 4 5 --	
4	$T=4$	a b c d	$(0,\infty)$ $(2,\infty)$ $(3,4)$ $(4,5)$	0 2 3 --	
5	$T=4$	a b c d	$(0,\infty)$ $(2,\infty)$ $(4,5)$ $(5,5)$	0 3 5 --	
6	$T=4$	a b c d	$(0,\infty)$ $(2,\infty)$ $(5,6)$ $(6,7)$	0 4 5 6	

图10-30 算法10.21在处理例10.20时的行为

周期的调度方案。这次尝试开始时，算法尽可能早地调度结点 a 和 b。但是，一旦结点 b 被安排在第二个时钟周期，结点 c 就只能安排在第 3 个时钟周期。这和结点 a 的资源使用相冲突。也就是说，a 和 c 在能够被 3 整除的时钟周期上都需要第一种资源。

这个算法执行回溯，试图延后一个时钟周期再对强连通分量 $\{b,c,d\}$ 进行调度。这一次结点 b 被安排在第三个时钟周期上，而结点 c 可以被成功地安排在第 4 个时钟周期上。但是，结点 d 不能被安排在第 5 个时钟周期上。也就是说，在能够被 3 整除的时钟周期上，b 和 d 都需要第二种资源。请注意，虽然至今为止找到的两个冲突都发生在除以 3 的余数都是 0 的时钟位置上，但是这只是一个巧合；在其他的例子中，冲突可能在余数为 1 或 2 的时钟周期上发生。

算法再次延后一个时钟周期尝试对强连通分量 $\{b,c,d\}$ 进行调度。但是，前面讨论过，当启动间隔是 3 个时钟周期时，这个强连通分量实际上永远不可能被成功地调度，因此这次尝试一定会失败。此时，这个算法放弃尝试，并试图找到一个启动间隔为 4 个时钟的调度方案。这个算法最终在第 6 次尝试时找到了最优调度方案。 □

10.5.9 对流水线化算法的改进

算法 10.21 是一个相当简单的算法，尽管人们发现它能够在实际的目标机器上很好地完成任务。这个算法中的要素包括：

1）使用一个模数资源预约表来检查稳定状态下的资源冲突。

2）需要计算传递依赖关系，以便在出现依赖环的时候找到各个结点可以被调度的合法范围。

3）回溯是有用的，而关键环（即给出了启动间隔 T 的最高下界的环）上的结点都必须一起重新调度，因为它们之间的时间间隔是没有松弛度的。

有很多种方法可以改进算法 10.21。比如，这个算法花了一段时间才发现对于简单的例子 10.22 来说，采用 3 个时钟的启动间隔是不可行的。我们可以首先对各个强连通分量进行独立调度，确定当前的启动间隔对于各个分量是否可行。

我们也可以改变结点被调度的顺序。算法 10.21 中使用的顺序有一些不利之处。第一，因为非平凡的 SCC 难以调度，所以首先对它们进行调度是较好的选择。第二，有些寄存器的生命期可能不需要那么长。因此期望能够使定值位置靠近使用位置。可行方法之一是首先调度带有关键环的强连通分量，然后向两端扩展调度方案。

10.5.10 模数变量扩展

如果一个标量变量的活跃范围处于循环的一个迭代之内，那么该标量变量被称为可私有化的（privatizable）。换句话说，一个可私有化变量不能在任何迭代的入口或者出口处活跃。这些变量会这样命名的原因是执行一个循环中的不同迭代的各个处理器可以拥有这些变量的私有拷贝，使得它们不会互相干扰。

变量扩展（variable expansion）指的是这样一种变换技术：它把一个可私有化的标量变量转换成为一个数组，并让循环的第 i 个迭代读写第 i 个元素。这个转换消除了一个迭代中的读运算和后一个迭代中的写运算之间的反依赖关系，以及不同迭代的写运算之间的输出依赖关系。如果所有的穿越循环的依赖关系都可以被消除，那么循环的各个迭代就可以并行执行。

消除穿越循环的依赖关系也就消除了数据依赖图中的环，这样可以大大提高软件流水线化的效率。如例 10.15 所示，我们不需要根据循环的迭代次数来完全扩展可私有化变量。同一时间内只能执行少量的迭代，而在同一时刻私有变量在其中活跃的迭代数量更少。因此，同一个内存位置可用于存放其生命周期不交叠的多个变量的值。更明确地讲，如果一个寄存器的生命周期是 l 个时钟，且启动间隔是 T，那么在一个时间点上只有 $q = \left\lceil \frac{l}{T} \right\rceil$ 个值是活跃的。我们可以为该

变量分配 q 个寄存器，而第 i 个迭代中的变量使用第（$i \bmod q$）个寄存器。我们把这种转换称为模数变量扩展（modular variable expansion）。

存在不同于启发式的方法吗？

我们可以把同时寻找最优软件流水线调度方案和寄存器分配方案的问题写成一个整数线性规划问题。虽然很多整数线性规划问题可以很快地得出解，但有些问题需要特别长的时间。在编译器中使用一个求解整数线性规划问题的程序时，我们必须能够在它无法在某个预设时间内完成解答时退出求解过程。

这个方法曾经在一个目标机器上（SGI R8000）实验性地尝试过，结果发现规划求解器可以在一个合理的时间内为大部分试验程序找到最优解决方案。我们发现，用启发式方法得到的调度方案和最优解相当接近。这个结果说明，至少对于那个目标机器，使用整数线性规划方法是没有什么意义的。从一个软件工程师的角度来看尤其如此。因为整数线性规划求解程序可能不会按时结束，在编译器中实现某种启发式调度程序仍然是必要的。一旦有了一个这样的启发式调度器，也就不需要再去实现一个基于整数规划技术的调度器了。

算法 10.23 使用模数变量扩展技术的软件流水线化。

输入： 一个数据依赖图和一个机器资源描述。

输出： 两个循环，一个经过软件流水线化处理，另一个没有。

方法：

1）从输入的数据依赖图中删除和可私有化变量相关的穿越循环的反依赖关系和输出依赖关系。

2）使用算法 10.21 对第一步得到的数据依赖图进行软件流水线化。令 T 是已经找到相应调度方案的启动间隔，L 是一个迭代的调度方案的长度。

3）对于每个可私有化变量 v，依据得到的调度方案计算 q_v，即 v 所需要的最小寄存器数目。令 $Q = \max_v q_v$。

4）生成两个循环：一个经过软件流水线化的循环和一个没有被流水线化的循环。被软件流水线化的循环有

$$\left\lceil \frac{L}{T} \right\rceil + Q - 1$$

个迭代的拷贝，各个拷贝之间相距 T 个时钟。它有一个带有

$$\left(\left\lceil \frac{L}{T} \right\rceil - 1 \right) T$$

条指令的序言部分，一个带有 QT 条指令的稳定状态和一个具有 L-T 条指令的尾声部分。插入一个从稳定状态的尾部到稳定状态顶端的循环回归指令。

分配给可私有化变量 v 的寄存器数目是

$$q'_v = \begin{cases} q_v & \text{如果 } Q \bmod q_v = 0 \\ Q & \text{否则} \end{cases}$$

在第 i 个迭代中的变量 v 使用的是被分配给 v 的第（$i \bmod q'_i$）个寄存器。

令 n 为源代码循环中表示迭代数目的变量。这个软件流水线化的循环被执行的前提是

$$n \geqslant \left\lceil \frac{L}{T} \right\rceil + Q - 1$$

循环回归分支的执行次数是

$$n_1 = \left\lfloor \frac{n - \left\lceil \frac{L}{T} \right\rceil + 1}{Q} \right\rfloor$$

因此，软件流水线化的循环所执行的源代码中的迭代的次数是

$$n_2 = \begin{cases} \left\lceil \frac{L}{T} \right\rceil - 1 + Qn_1 & \text{如果 } n \geqslant \left\lceil \frac{L}{T} \right\rceil + Q - 1 \\ 0 & \text{否则} \end{cases}$$

未被流水线化的循环执行的迭代数目是 $n_3 = n - n_2$。

例 10.24 在图 10-22 中经过软件流水线化的循环中，$L=8$，$T=2$ 且 $Q=2$。这个软件流水线化的循环有 7 个迭代的拷贝，其中的序言、稳定状态和尾声部分分别有 6、4、6 条指令。令 n 为源代码循环中的迭代次数。这个软件流水线化的循环在 $n \geqslant 5$ 的时候被执行，在这种情况下循环回归分支被执行

$$\left\lfloor \frac{n-3}{2} \right\rfloor$$

次，且软件流水线化的循环负责执行

$$3 + 2 \times \left\lfloor \frac{n-3}{2} \right\rfloor$$

个源代码循环中的迭代。 □

模数扩展会把稳定阶段代码的大小增加到 Q 倍。虽然如此，由算法 10.23 生成的代码仍然是相当精简的。在最坏情况下，经过软件流水线化的循环的指令数目是单个迭代的调度方案中指令数目的三倍。粗略地讲，把用来处理零星迭代的额外循环加在一起，整个代码的大小大约是原代码大小的四倍。这个技术通常应用于紧凑的内层循环，因此这样的代码增加量是可接受的。

算法 10.23 可以使用更多的寄存器来使代码的扩展量降到最低。我们可以通过生成更多的代码来降低对寄存器的使用。如果我们使用一个具有

$$T \times LCM_v q_v$$

条指令的稳定状态，我们最少可以为每个变量 v 使用 q_v 个寄存器。这里，LCM_v 是求解所有 q_v 的最小公倍数（即能够被所有 q_v 整除的最小整数）的函数，v 的取值范围是所有的可私有化变量。遗憾的是，即使对少量很小的 q_v 值，最小公倍数也可能变得相当大。

10.5.11 条件语句

如果可以使用带断言的指令，我们可以把控制依赖的指令转换成为带断言的指令。带断言的指令可以和其他指令一样进行软件流水线化处理。但是，如果在循环体内有很多依赖于数据的控制流，那么就更加适合使用 10.4 节中的算法进行调度。

如果一个机器没有带断言的指令，那么可以使用下面描述的*层次结构归约*（hierarchical reduction）技术来处理少量的依赖于数据的控制流。和算法 10.11 类似，在层次结构归约中，对一个循环控制结构的调度是从嵌套在最内层的结构开始，以从内到外的顺序进行调度的。当每个结构被调度时，整个结构被归约为一个结点。这个结点代表了它的所有组成部分和程序的其他部分之间的调度约束。然后，这个结点可以当作它外围的控制结构中的单个结点进行调度。当整个程序被归约为单个结点的时候，调度过程就结束了。

当处理一个带有“then”分支和“else”分支的条件语句时，我们首先独立地对各个分支进行调度。然后：

1）整个条件语句的约束被保守地设定为来自两个分支的约束的并集。

2）它的资源使用情况是各个分支所用资源的最大值。

3）它的先后次序约束是各个分支中此类约束的并集。通过假设两个分支都被执行就可以求得这个约束集合。

然后，这个结点就可以和其他结点一样进行调度。需要生成分别对应于两个分支的两组代码。任何被安排与这个条件语句并行执行的代码都需要在这两个分支中分别进行复制。如果多个条件语句相互交叠，那么对并行执行的每个分支组合都要生成单独的代码。

10.5.12 软件流水线化的硬件支持

人们提出了特殊的硬件支持机制来使软件流水线代码的大小降到最低。在 Itanium 体系结构中的*轮转寄存器文件*(rotating register file)就是这样的一个例子。轮转寄存器文件有一个*基寄存器*(base register)，可以把基寄存器中的内容加到代码中给定的寄存器编号来得到实际被访问的寄存器。我们只需要在每个迭代的边界上改变基寄存器中的内容，就可以让一个循环中的不同迭代使用不同的寄存器。Itanium 体系结构也支持广泛的带断言指令。断言不仅可以把控制依赖转换成数据依赖，它也可以用来避免生成序言代码和尾声代码。一个软件流水线化的循环体中包含了所有在序言和尾声中的指令。我们只需要为稳定状态生成代码，并适当地使用断言来抑制多余的运算，使得代码的运行效果就像是存在一个序言和一个尾声。

虽然 Itanium 的硬件支持机制提高了经软件流水线化的代码的密度，我们必须知道这种支持机制可不便宜。因为软件流水线化技术主要用于最内层循环，被流水线化处理的循环往往很小。原则上，对于那些预期会用于执行很多软件流水线化的循环且尽可能降低代码大小又很重要的机器，为软件流水线化提供专门的支持机制是合理的。

10.5.13 10.5 节的练习

练习 10.5.1：在例 10.20 中，我们说明了如何求出 b 和 c 之间的相对时钟距离的上下界。分别①为一般化的 T，②为 $T=3$，③为 $T=4$，计算另外五对结点的上下界。

练习 10.5.2：图 10-31 显示的是一个循环的循环体。a(R9)这样的地址是内存位置，其中 a 是一个常数，而 R9 是对该循环的迭代进行计数的寄存器。因为对于不同的迭代有不同的 R9 的值，所以可以假设该循环的每个迭代访问不同的位置。使用例 10.12 中的机器模型，按照下面的方法对图 10-31 中的循环进行调度。

```
1)  L:  LD  R1, a(R9)
2)      ST  b(R9), R1
3)      LD  R2, c(R9)
4)      ADD R3, R1, R2
5)      ST  c(R9), R3
6)      SUB R4, R1, R2
7)      ST  b(R9), R4
8)      BL R9, L
```

图 10-31 练习 10.5.2 的机器代码

1）尽量保持各个迭代紧致(即在每个算术运算之后只引入一个 nop 运算)，把该循环展开两次。该机器在任意时钟周期上只能做一次加载运算、一个保存运算、一个算术运算以及一个分支运算。在不破坏上面约束的情况下，调度第二次迭代使之在尽可能早的时刻开始。

2）重复(1)部分，但是把这个循环展开三次。同样，在遵守机器资源约束的情况下让各个迭代尽可能早地启动。

！3）在遵守机器约束的情况下构造完全流水线化的代码。在这一部分，可以在必要时引入 nop 运算，但是你必须每两个时钟周期启动一个新迭代。

练习 10.5.3：某一个循环需要 5 个加载运算、7 个保存运算和 8 个算术运算。假设有这样一

台机器，它的每个运算都能够在一个时钟周期内完成，并且有足够的资源在一个时钟周期内执行：

1）3 个加载运算，4 个保存运算和 5 个算术运算。

2）3 个加载运算，3 个保存运算和 3 个算术运算。

请问对于上面的两种情况，这个循环经软件流水线化后的启动间隔最小是多少？

! **练习 10.5.4**：使用例 10.12 中的机器模型，为下列循环

```
for (i = 1; i < n; i++) {
    A[i] = B[i-1] + 1;
    B[i] = A[i-1] + 2;
}
```

寻找最小的启动间隔以及对此循环的各个迭代的统一调度方案。请记住，对迭代的计数是通过寄存器的自动增一运算实现的，不需要专门的对 for 循环计数的运算指令。

! **练习 10.5.5**：请证明，如果每个运算都只需要一个单元的某种资源，算法 10.19 总能够找到一个使用启动间隔下界的软件流水线调度方案。

! **练习 10.5.6**：假设有一个结点集合为 a、b、c、d 的有环的数据依赖图。从 a 到 b 以及从 c 到 d 都有标号为(0，1)的边；从 b 到 c 及从 d 到 a 都有标号为(1，1) 的边。此外，再没有其他的边。

1）画出这个有环的依赖图。

2）计算记录了结点之间的最长简单路径的表。

3）如果启动间隔 T 的值为 2，指出最长简单路径的长度。

4）设 $T=3$，重复(3)。

5）对于 $T=3$ 的情况，在调度 a、b、c、d 所表示的各条指令时，它们之间的相对时间的约束是什么？

! **练习 10.5.7**：假设在一个有 n 个结点的图中没有长度为正的环，给出一个 $O(n^3)$ 的寻找该图中最长简单路径长度的算法。提示：修正 Floyd 的最短路径算法（见 A. V. Aho 和 J. D. Ullman，Foundations of Computer Science，Computer Science Press，New York，1992）。

!! **练习 10.5.8**：假设我们有一个带有三种指令类型的机器，我们把这三种指令称作 A、B 和 C。所有的指令都需要一个时钟周期，并且该机器可以在每个时钟周期执行每个类型的各一条指令。假设一个循环由六条指令组成，每种两个，那么一个软件流水线能够以 2 作为启动间隔执行这个循环式。但是，这六条指令的某些序列要求插入一个延时，而另外一些序列需要插入两个延时。在 90 种可能的由两个 A 型指令、两个 B 型指令和两个 C 型指令组成的序列中，多少个序列不需要延时？多少个序列需要一个延时？提示：在这三类指令中存在对称性，因此如果两个序列能够通过交换 A、B 和 C 的名字相互转换，那么它们就需要同样多的延时。比如，*ABBCAC* 一定和 *BCCABA* 一样。

10.6 第 10 章总结

- *体系结构问题*：被优化的代码调度利用了现代计算机体系结构的一些特性。这样的机器常常允许以流水线方式执行代码，也就是多条指令在同一个时刻处于不同的执行阶段。有些机器还允许多条指令在同一个时刻开始执行。
- *数据依赖*：在调度运算指令时，我们必须知道这些指令对于每个内存位置和寄存器的影响。如果一条指令必须在另一指令对某个内存位置写入之后才读取该位置的值，那么这两条指令之间具有真依赖关系。如果有一个对同一位置的读指令之后的写指令，那么两条指令之间就出现反依赖关系；当有两个对同一位置的写指令时就会出现输出依赖。

- 消除依赖关系：通过使用附加的位置存放数据，可以消除反依赖和输出依赖。只有真依赖不能被消除，并且在调度代码时必须保证遵守这类依赖关系。
- 基本块的数据依赖图：这些图表示了一个基本块中的语句之间的时间安排约束。图的结点对应于这些语句。从 n 到 m 的标号为 d 的边表明指令 m 的开始时刻必须比 n 的开始时刻晚至少 d 个时钟周期。
- 带优先级的拓扑排序：一个基本块的数据依赖图总是无环的，通常有很多个与这个依赖图一致的拓扑排序。为一个给定依赖图选择较好的拓扑排序的启发式规则之一是首先选择具有最长关键路径的结点。
- 列表调度：给定一个数据依赖图的带优先级的拓扑排序，我们可以按照这个顺序考虑对结点的调度。在对每个结点进行调度时，把每个结点安排在最早的满足下列条件的时钟周期上：满足图的边所蕴涵的时间安排约束，并且和所有之前已经调度好的结点的调度方案一致，同时满足该机器的资源约束。
- 基本块之间的代码移动：在某些情况下，可以把一些语句从它所在的基本块移动到该基本块的前驱或后继。进行这种移动的好处在于有机会在新的位置上并行执行新指令，而在原位置上可能没有这个机会。如果原基本块和新位置之间没有支配关系，那么有必要在某些路径上插入补偿代码，以保证不管控制流如何运行，被执行的总是相同的代码序列。
- do-all 循环：一个 do-all 循环的迭代之间不存在依赖关系，因此各个迭代都可以并行运行。
- do-all 循环的软件流水线化：软件流水线化技术充分利用了目标机器能够同时执行多条指令的能力。通过调度使得循环的各个迭代的开始时刻只相隔很短的时间。在此过程中可能需要在迭代中插入 no-op 指令以避免迭代之间产生机器资源冲突。结果，循环可以很快地执行，其中包括序言、尾声和(通常)较小的内部循环。
- do-across 循环：很多循环具有从每个迭代到后续迭代的依赖关系。这些循环称为 do-across 循环。
- do-across 循环的数据依赖图：为了表示一个 do-across 循环的指令之间的依赖关系，依赖图中的边的标号由两个值组成：必须的延时(和表示基本块的依赖图中的延时含义相同)以及在具有依赖关系的两条指令之间相隔的迭代数量。
- 循环的列表调度算法：为了调度一个循环，我们必须为所有的迭代选择同一个调度方案，并选择启动间隔，即连续迭代的启动时刻的间隔。这个算法还需要获取针对循环中不同指令的相对调度方案的约束。它通过计算两个结点之间的最长无环路径的长度来获得这种约束。算法求得的这些长度把启动间隔作为参数，因此给启动间隔设定了一个下界。

10.7 第 10 章参考文献

如果希望对处理器体系结构和设计进行更深入的研究，我们推荐 Hennessy 和 Patterson[5]。

数据依赖的概念首先出现在 Kuch、Muraoka 和 Chen[6]以及 Lamport[8]中，在多处理器和向量机编译代码的上下文中讨论。

指令调度首先在水平微代码调度中使用([2, 3, 11 和 12])。Fisher 在微代码压缩上的研究成果使他提出了 VLIW 机器的概念。在这种机器上，编译器可以直接控制运算的并行执行[3]。Gross 和 Hennessy[4]在第一个 MIPS RISC 指令集中使用指令调度方法来处理被延时的分支。本章的算法是基于 Bernstein 和 Rodeh[1]的研究成果的。他们的工作对具有指令级并行机制的机器的运算调度作出了更一般化的处理。

软件流水线化的基本思想首先由 Patel 和 Davidson[9]为硬件流水线调度而提出。软件流水线化技术首先由 Rau 和 Glaseser[10]用于为一个具有支持软件流水线化的特殊硬件机制的机器编译代码。这里描述的算法基于 Lam[7]，该文中假设没有特殊硬件的支持。

1. Bernstein, D. and M. Rodeh, "Global instruction scheduling for superscalar machines," *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 241–255.

2. Dasgupta, S., "The organization of microprogram stores," *Computing Surveys* **11**:1 (1979), pp. 39–65.

3. Fisher, J. A., "Trace scheduling: a technique for global microcode compaction," *IEEE Trans. on Computers* **C-30**:7 (1981), pp. 478–490.

4. Gross, T. R. and Hennessy, J. L., "Optimizing delayed branches," *Proc. 15th Annual Workshop on Microprogramming* (1982), pp. 114–120.

5. Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufman, San Francisco, 2003.

6. Kuck, D., Y. Muraoka, and S. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup," *IEEE Transactions on Computers* **C-21**:12 (1972), pp. 1293–1310.

7. Lam, M. S., "Software pipelining: an effective scheduling technique for VLIW machines," *Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328.

8. Lamport, L., "The parallel execution of DO loops," *Comm. ACM* **17**:2 (1974), pp. 83–93.

9. Patel, J. H. and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays," *Proc. Third Annual Symposium on Computer Architecture* (1976), pp. 159–164.

10. Rau, B. R. and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," *Proc. 14th Annual Workshop on Microprogramming* (1981), pp. 183–198.

11. Tokoro, M., E. Tamura, and T. Takizuka, "Optimization of microprograms," *IEEE Trans. on Computers* **C-30**:7 (1981), pp. 491–504.

12. Wood, G., "Global optimization of microprograms through modular control constructs," *Proc. 12th Annual Workshop in Microprogramming* (1979), pp. 1–6.