令，以及各个操作的时序特性。例如，编译器必须知道时序信息，才能够确定是用一条乘法指令，还是用移位和加法的某种组合。现代计算机用复杂的技术来处理机器级程序，并行地执行许多指令，执行顺序还可能不同于它们在程序中出现的顺序。程序员必须理解这些处理器是如何工作的，从而调整他们的程序以获得最大的速度。基于 Intel 和 AMD 处理器最近的设计，我们提出了这种机器的一个高级模型。我们还设计了一种图形数据流(data-flow)表示法，可以使处理器对指令的执行形象化，我们还可以利用它预测程序的性能。

了解了处理器的运作，我们就可以进行程序优化的第二步，利用处理器提供的指令级并行(instruction-level parallelism)能力，同时执行多条指令。我们会讲述几个对程序的变化，降低一个计算的不同部分之间的数据相关，增加并行度，这样就可以同时执行这些部分了。

我们以对优化大型程序的问题的讨论来结束这一章。我们描述了代码剖析程序(profiler)的使用，代码剖析程序是测量程序各个部分性能的工具。这种分析能够帮助找到代码中低效率的地方，并且确定程序中我们应该着重优化的部分。

在本章的描述中，我们使代码优化看起来像按照某种特殊顺序，对代码进行一系列转换的简单线性过程。实际上，这项工作远非这么简单。需要相当多的试错法试验。当我们进行到后面的优化阶段时，尤其是这样，到那时，看上去很小的变化会导致性能上很大的变化。相反，一些看上去很有希望的技术被证明是无效的。正如后面的例子中会看到的那样，要确切解释为什么某段代码序列具有特定的执行时间，是很困难的。性能可能依赖于处理器设计的许多细节特性，而对此我们所知甚少。这也是为什么要尝试各种技术的变形和组合的另一个原因。

研究程序的汇编代码表示是理解编译器以及产生的代码会如何运行的最有效手段之一。仔细研究内循环的代码是一个很好的开端，识别出降低性能的属性，例如过多的内存引用和对寄存器使用不当。从汇编代码开始，我们还可以预测什么操作会并行执行，以及它们会如何使用处理器资源。正如我们会看到的，常常通过确认关键路径(critical path)来决定执行一个循环所需要的时间(或者说，至少是一个时间下界)。所谓关键路径是在循环的反复执行过程中形成的数据相关链。然后，我们会回过头来修改源代码，试着控制编译器使之产生更有效率的实现。

大多数编译器，包括 GCC，一直都在更新和改进，特别是在优化能力方面。一个很有用的策略是只重写程序到编译器由此就能产生有效代码所需要的程度就好了。这样，能尽量避免损害代码的可读性、模块性和可移植性，就好像我们使用的是具有最低能力的编译器。同样，通过测量值和检查生成的汇编代码，反复修改源代码和分析它的性能是很有帮助的。

对于新手程序员来说，不断修改源代码，试图欺骗编译器产生有效的代码，看起来很奇怪，但这确实是编写很多高性能程序的方式。比较于另一种方法——用汇编语言写代码，这种间接的方法具有的优点是：虽然性能不一定是最好的，但得到的代码仍然能够在其他机器上运行。

5.1　优化编译器的能力和局限性

现代编译器运用复杂精细的算法来确定一个程序中计算的是什么值，以及它们是被如何使用的。然后会利用一些机会来简化表达式，在几个不同的地方使用同一个计算，以及降低一个给定的计算必须被执行的次数。大多数编译器，包括 GCC，向用户提供了一些对它们所使用的优化的控制。就像在第 3 章中讨论过的，最简单的控制就是指定优化级