Blackfin在线培训课程

课程单元：面向Blackfin®处理器的C语言编程及优化
主讲人：Alan Anderson

第1章：概念和工具

第1a节：导言

大家好，我是模拟器件公司（ADI）的性能优化工程师，我的名字叫Alan Anderson。今天我们将讨论面向Blackfin处理器的C语言编程，以及如何实现程序优化。我们将介绍各种优化工具，并以客户应用基准测试中的各种场景为例，进行说明。提高应用程序运行速度的方法多种多样，包括自动优化、修改程序、以及重新编写部分代码等。

本课程单元将分为几个小节。首先，我将介绍一些基本概念和工具，以及关于Blackfin处理器的一般情况。然后，我们将讨论紧凑数字循环和信号处理内核向量化等概念。之后，我将解释什么是所谓的控制代码、决策代码、条件分支等等。从

第2小节到课程结束，我们讨论的内容都将涉及Blackfin处理器的存储性能，在当今现代化的处理器中，存储性能可谓至关重要。最后，我将以实际应用的基准测试为例，总结本课程单元的学习要点。

现在开始第1小节。我们将介绍一些基本概念，并讨论关于C语言编程的一般情况以及可用的工具。那么，第一个问题是，为什么要使用C语言？因为其他支持人工组合的编程语言的效率低得多，而使用C语言编程，程序员可以直接将编写好的C代码导入Blackfin处理器，并快速执行运行测试。此外，C语言程序的维护也更简便。

C语言的缺点是，ANSI C不是一种专门的信号处理语言，它更适用于系统设计，而不是数学运算。因此，在对DSP处理器进行C语言编程时，通常需要程序员在某些关键的代码区域进行人工组合。此外，信号处理应用自身也在不断演进，它们不断开拓新的领域，相比之下，诸如ANSI C等标准语言的发展步伐要缓慢得多。

在对C程序进行性能优化时，不仅可以利用编译器进行自动优化，程序员自己也大有可为，尤其是在编译器无能为力的算法优化部分。一般而言，算法优化总是比编译器优化更有效。因此，程序员应当首先进行算法优化，评估当前采用的算法是否适于在Blackfin处理器上运行。在评估C程序时，可以检查其一般运行情况以及所谓的“别名判别”特性，即，编译器是否能够始终理解应用正在调用的数据。如果应用采用了间接指针，那么编译器将不得不采取保守策略。

除了对C程序进行优化，程序员还必须考虑所用处理器的处理能力。在处理能力方面，不同的目标处理器各有所长，各自具有针对某种数学运算特性而专门编写的指令，如腓特比（Viteribi）运算、位多路复用、向量乘法和加法等。程序员必须考虑，采用自己选择的算法，是否能够实现这些运算。

最后，还要考虑不可移植的更改。我们前面讨论的优化都是维持一个可以在任何平台上运行的C程序。如果程序员要在程序中添加某些针对Blackfin处理器的代码，那么，更改后的程序将不可导入其他处理器。程序员可以重新编写部分C代码、添加编译指示甚或在代码行中添加汇编语句。我们始终建议程序员保留一个C模型备份，以便进行验证。此外，有一点非常重要，虽然有时可以通过优化程序代码来提高程序的运行速度，但在更多的时候，需要针对应用采用的特定硬件，对程序进行专门化来提高其运行速度。最终，程序员将构建一个速度更快，但也更大、更复杂，并且不适于导入其他处理器的专门程序。换句话说，应用优化需要付出代价。

C语言实现了所谓的“统一计算模型”，也就是说，C程序员可以假定其编写的程序能够在任何平台上运行。但在性能方面，程序员必须认识到，不同的平台将以不同的方式支持C计算模型。例如，如果某个处理器具备本机浮点指令，那么，C代码将假定该处理器可以支持64位浮点运算。

但是，Blackfin处理器不能本机支持浮点数据类型，需要借助软件仿真器，以相当慢的速度实现浮点运算。因此，程序员在选择算法和编码方法时必须考虑到这一点。C语言还可以实现一个很大的平面内存模型。实际上，对于当今现代化的处理器，内存性能有着举足轻重的影响。此外，在不同的处理器上，C语言编程可谓大相径庭，就连一些非常简单的方面也不尽相同，如整数的长度，是16位短整数还是32位整数？

C语言可能难以满足信号处理应用在累加、向量化、SIMD（单指令多数据）和小数运算等方面的要求。因此，令C程序员感到郁闷的是，目标处理器的特性将决定应用的性能。在设计应用时，必须考虑到这一点。虽然C程序可以轻松导入其他处理器，但是，如果想要获得高性能，就不能忽视所采用硬件的特性。

下面，我将从三个层面来讨论应用优化。首先是程序优化，也就是说，尽可能利用编译器的自动优化特性，实现代码优化。如果用户以前曾做过微控制器编程，那么，会很习惯通过编译器优化来加快程序的运行速度。对于信号处理循环，编译器优化相当有效，可以实现近20倍的速度提升。

但是，必须注意的是，编译器优化只对部分代码区域有效。编译器优化不能改变程序的算法、不会对程序进行全局更改、也不能大规模重新排列数据。对编译器而言，正确性始终比提高性能更重要。而程序员的任务就是为编译器提供一个安全的环境，让它放心大胆地生成运行速度更快的代码，否则，它将采取保守策略。

应用优化的第二步就是修改程序，我们只会在不得已的情况下，才对程序进行必要的修改。要将导入处理器的现成的C程序精心构建成所谓的“优化C程序”，程序员可以在其中添加一些编译指示，其他编译器会将这些编译指示当作代码注释，从而维持一定程度的可移植性。程序员还可以在一定范围内，对内置函数和存储限定符等代码进行同样的修改。

最后，程序员可以有针对性地重新编写部分代码，也就是说，针对Blackfin处理器的特性，修改程序。为了最大限度地缩短产品上市时间，建议程序员采用这个应用优化模型。首先，确定性能目标，然后，将C程序导入Blackfin处理器，再依次执行上述优化步骤，就能实现性能目标。

这张幻灯片旨在再次强调未经优化的代码与优化代码的差别。右上角显示的是一个很小的信号处理循环，如果启用了编译优化器，那么，将生成右下角的这个极其紧凑的代码。但是，如果没有启用优化器，那么，将生成左侧屏面中的代码。大家不必费力去弄懂这个代码，我只是想指出它有多么冗长。与微控制器不同的是，在Blackfin处理器上，优化代码比未经优化的代码的运行速度快得多。因此，程序员最好尽早进行代码优化。

第1b节：优化工具

在提高程序的运行速度时，最重要的是有的放矢。任何C程序员都能在任何存储页打开任何C程序，然后觉得，“这个代码看起来不怎么样，我要优化它，我可以让它运行得更快。”结果，尽管我们整天没完没了地优化代码，却成效甚微。关键在于，专注于程序中真正发挥最大作用的那部分代码。而且，当一个程序导入新的目标硬件后，其中最重要的代码段未必如程序员所料。

为此，我们提供了一个杰出的工具——统计监测器。统计监测器负责监视程序在硬件上的执行情况，并且完全不会干扰程序的运行。相比于其他陈旧的监测方法，这种方法非常先进。采用以往的监测方法，我们必须插入跟踪代码，这会干扰程序的运行，并改变定时特性。而统计监测器则只在程序运行时监视程序计数器，完全不会影响定时。这种方法非常准确，将显示出由于I/O存储延迟等各种原因对定时造成的影响。程序员还可以对正常的I/O输入和输出数据进行统计监测。

问题的关键是，程序员绝不能自以为知道应用中哪部分代码的功劳最大。必须进行测量，直觉一点都靠不住。这里存在着“2/8定律”，即，在应用程序中，通常有80％的代码都无关紧要，而真正需要程序员努力优化的代码只占20％。这张图片是正在运行的统计监测器的截屏。左侧的窗格中列出的是函数及其运行时间百分比。只要双击这些函数，右侧窗格中就会按代码行逐行显示类似的信息。这样，程序员就可以极为快速地详细查看应用中的“热点”代码段。

如果程序员是在模拟器而不是硬件上进行开发，那么，也可以利用模拟器提供的这种特性。进一步讲，利用统计监测器，程序员还可以详尽查看每一行代码的运行情况。将鼠标指针移动至需要查看的代码行的监测信息上，然后单击鼠标右键，屏幕将出现一个“混合模式”对话框。利用混合模式，程序员可以在屏幕显示的C代码行中插入机器指令。

现在，我们可以清楚地查看程序中的每一条汇编指令占用的应用运行时间百分比。通过代码优化，这种排序可能略有变化。但是，这个排序的真正意义是，程序员可以轻松查明耗用时间明显较多的指令。例如，在本例中，每条指令的平均运行时间为总运行时间的0.04％，而其中有一条指令耗费的时间却是平均时间的4倍以上。那么，当程序员对应用进行最低层的优化时，就要自问——为什么？这是怎么回事？能不能重新编写代码，以缩短其运行时间？在这个例子里，我们发现了一个管线停顿，再下来，我们又发现了一个调用指令占用了大量时间，控制转移消耗了几个周期。

在对Blackfin处理器进行编程时，必须牢记，这个处理器是通过较长的管线来实现高速度的。也就是说，改变一条具备条件分支的管线的方向，将会产生一些停顿周期，从而导致时延。同样地，程序员还必须注意程序结构，如间接指针，管线必须等待加载地址，才能通过该地址解除参考，找到这个链式结构的下一部分。

这不仅降低了调用效率，而且也让编译器难以应付。当处理器最终运行完毕这个链时，编译器将无从知晓这个数据从哪儿来，并因而不能确知这个数据是否会与任何

其他数据或任何寄存器的内容相冲突。C编译器将竭尽所能，消除这些停顿。它将重新排列指令顺序，让有用的代码占用这些停顿周期。但是，C编译器的能力是有限的，取决于代码的长度以及C编译器是否真的理解这个代码，能不能积极地移动代码。我们提供的工程师注释EE197阐明了关于这些停顿状态的详细信息。

如果希望了解关于这些停顿的更多信息，全面了解到底是出了事，那么，程序员可以利用模拟器提供的管线浏览器（Pipeline Viewer）。这里显示的视图是我特意生成的，我在程序中找到感兴趣的循环，然后重复单步执行这个循环，就出现了这种情况。看来，产生的停顿太多了。一般而言，C程序员对管线的理解不需要细到这个程度，只要观察这些黄色显示的停顿标记就行了。随着重复单步执行这个循环，这些黄色的停顿标记将呈梯形排列——出现了停顿状况，只要了解这点就够了。如果将鼠标指针移动至这些黄色标记上，然后按下<Ctrl>键，屏幕将出现一个弹出窗框，表明这个停顿状态的类型。

这就是这种说明信息。

第2章：优化DSP内核

第2a节：优化技术

下面，我们将讨论一些运算密集型循环。在C语言中，控制代码和信号处理代码并没有形式上的差异，只是我们人为地将C代码划分为不同类型的代码。编译器只知道自己处理的是C程序，而不论这些代码究竟属于哪种类型。

在信号处理应用中，大多数程序在开始时都采用浮点数据，然后，会转换为固定点数据或小数实现。经过我们的不懈努力，Blackfin处理器能够尽快实现支持浮点数据。现在，程序员可以选择采用我们的“严格IEEE浮点数据”，其样本性能如右侧的绿色屏面所示。或者，如果程序员认为自己的程序无需执行严格IEEE浮点数据所要求的非数字或NAN检查，则可以采用我们提供了另一套仿真程序，获得更快的运行速度，速度提高如右侧比率所示。我们发现，其实很少有应用真的需要进行严格的NaN处理，因此，这种快捷仿真程序非常有用。

另外，我们发现，许多客户都希望在应用的某些部分一直保持浮点数据。Blackfin处理器不仅运行速度极快，而且允许程序员在应用中保留令人惊叹的大量浮点数据。

接下来，就要从浮点数据转换成小数运算。Blackfin处理器的指令集包含了大量的运算指令，可以快速执行固定点和小数运算，如乘加器和MACshift指令等。但是，编译器和库却不可用于这种代码，因为这种代码不属于可移植ANSI C代码。经过长期的实践，我们研究出了多种不同的方法，程序员可以根据自己的需要进行选择。

我们提供了小数内建指令，允许程序员通过类似于函数调用的方法，调用特定的小数算术运算符，并最终映射至一条机器指令。我们提供了多种小数类型，fract16和fract32，虽然这些不是C语言原生小数类型，但有助于程序员进行编程。我们发现ETSI运算符是一套非常有用的可移植小数运算符，稍后我们将详细讨论这一点。此外，还有C++小数类，通过运算符过载，可以实现堪称最为自然的小数表达式，看起来就像是一个C程序。

当然，小数运算的速度比我们在开始时介绍的浮点模拟运算快了上百倍。不过，实现小数运算的方法不只一种，这也引起了许多程序员的莫大兴趣，他们纷纷试着利用标准的C代码，编写出可以实现小数运算的程序，尽管小数运算从来不是可移植C语言的强项。在这个循环里，是一个执行小数乘加运算的可移植C代码——Blackfin处理器上的一条机器指令。

利用这种表达方式，程序员可以保持代码的可移植性。编译器也能够理解这个代码，并将其转换为相应的指令。C语言对溢出的表达有些含糊不清，因此，在发生溢出时，编译器将假定出现了饱和状态，以便进行信号处理。

应当注意的是，相比于带符号变量，无符号变量的条件要严格得多。因此，在使用带符号变量时，编译器更可能选择这种模式。由于没有40位数据类型，所以无法进行加法运算。如果编写了一个像这样要求进行换算的循环代码，那么，每一次循环都要进行换算，而不是只在最后进行换算。

当然，如果利用多个小数运算符，将这个代码编写成一个复杂的表达式，那么，编译器将无法理解这个代码的意图，因此，程序员应该考虑采用更加显而易见的表达方式。这里显示的就是一种更为简单明了的表达方式。这就是所谓的小数内建指令。Blackfin处理器附带的技术文档中列出了这种指令，这里仅举一条为例。这些指令看起来就像函数调用，但实际是映射至一条指令。不仅程序员可以轻松编写这些指令，编译器也能轻松读懂这些指令。编译器可以准确理解这些指令的含义及其作用，因而不会对与之相关的优化造成任何影响。

这里列出了ETSI内建指令，这是由欧洲电信标准协会定义的内建指令，主要用于GSM和AMR编解码器等程序。不过我发现，如今有相当多的编译器制造商也支持ETSI内建指令，因此，采用ETSI内建指令的精确的小数表达式可以进行移植。这些ETSI内建指令具备非常清晰的定义，程序员可以利用可移植C代码，实现这些内建指令。我们的许多客户都非常愿意采用符合标准的程序，通过编译这些代码，快速实现性能目标。强烈推荐程序员采用这些内建指令。

在编程时，经常遇到的另一个问题是，是否应该使用指针或阵列。对于当今功能强大的优化编译器而言，程序中有没有指针确实无所谓。人们通常认为指针更利于硬件使用，但其实，具备默认长度的阵列更有用。利用阵列，编译器可以更加准确地理解两个重叠的数据，而这往往是编译器进行积极优化时难以克服的一个关卡。如果必须选择一种，最好选择阵列，不过也许其实无所谓。

第2b节：紧凑循环

下面，介绍信号处理应用中采用的紧凑循环。在根据统计监测数据，对代码进行优化之前，编译器将粗略假定，程序大部分时间都在执行内部循环。因此，在程序中编写仅重复一两次的内部循环是很不明智的，将会适得其反。

基本上，编译优化器的工作就是解开循环。优化器会将代码展开，然后进行重新编排，以避免发生停顿，或者实现向量化运算。最常用的两种基本优化技术是向量化和软件管线。在下面这个窗口中，显示了一个软件管线的简单例子。通过软件管线优化，可以将循环中各个独立的迭代，合并到一个机器周期中。

例如，在这个循环的第3个迭代，保存第1个迭代的运算结果；并执行第2个迭代中的MAC运算；并为第3个迭代加载输入数据；全都在一个周期内完成。采用这种模式可以显著提升程序性能。这里，我介绍这个模式是为了帮助大家分辨自己的程序是否实现了最优性能。

现在详细解释一下。首先，这是一个简单的编译代码。我们依次加载2个16位的值，然后进行乘加运算。这个小循环的迭代运算消耗了三个周期。通过向量化，可以将这个循环的两个迭代合并起来，于是，我们一次加载一个32位的值，即两个16位的数据操作数。然后，进行向量化的乘加运算，即，两个单独的MAC运算，从而将运算速度提高了一倍。

最后，我们来看看软件管线的优化效果。加载数据，然后开始执行这个单周期循环。每次执行两个乘加运算，同时另外加载两个数据集。最后，在最下面，执行最后一个运算，退出这个循环。这样，就通过一个指令，完成了这个循环的两个迭代运算。

基本上，我们将运算速度提高了6倍。但是，程序员必须谨慎检查程序是否真的实现了这些优化，如果没有，就得反问自己，为什么？编译器还需要哪些信息？为什么没能完成优化？

对此，我的建议是“保持简单”。切勿自己动手展开这些内部循环。有时候，程序员导入Blackfin处理器的代码原本是用于一个不善于进行这种优化的编译器，其中包含了一个简单的循环，如最上面所示，然而，程序员将其重新编写为这样一个复杂的循环。也就是说，程序员人为地进行了优化。但是，对于Blackfin的编译器而言，这只会让事情变得更糟。编译器更难以弄清楚这个代码的本质含义，编译器更愿意对原来的代码进行处理，自己完成优化。不过，有时候程序员手动展开外部循环会有所助益。

同样地，程序员最好不要自己动手对循环实现软件管线优化。这里显示的是一个非常简单的代码，编译器可以轻而易举地理解这段代码。而在右侧的窗口中，显示的

是一个程序员越俎代庖，替编译器完成了所谓“优化”的代码。对不起，我刚才引用了编译器工程师们的行话。

必须小心避免的几件事。首先，这里举例说明了定标器的相关性。这个等式的左侧设置了一个“X”值，右侧又表示要读取这个“X”值，也就是说，编译器无法合并其迭代。下面，这个包含阵列的代码也出现了同样的问题。在等式左侧，阵列A中设定了一个值；同时，这个等式表示要读取阵列A，并且所读取的阵列A中的索引值本身又是一个阵列表达式，因此，编译器搞不清究竟要读取阵列A的哪个部分。这样，就无法执行矢量化和软件管线优化。

下面列举的情况就是编译器能够处理的。这是一个加法运算。虽然等式左右两侧都有一个“X”值，但由于是加法运算，所以编译器可以对其进行优化。对于加法或乘法一类的结合运算，其运算顺序是可以根据需要重新调整的，对此，编译器当然愿意欣然效劳。

最后一个例子。这里产生了一个编译器工程师们所说的归纳变量，也就是说，在运算过程中，每一个迭代运算都会生成固定量的值。编译器可以相当容易地看清楚全局——在运算中要索引左侧和右侧的阵列A；这两个索引永远不会相互重叠，总是相隔一段固定的距离。编译器会根据这个信息，对程序进行矢量化。

如果编译器不能对程序进行优化，则可以对程序的循环结构进行测试，试试解开这些循环。有时候，可以将内部循环和外部循环合并为一个单循环，虽然这个循环可能会非常复杂，但是，优化器将仅专注于内部循环。因此，如果在外部循环中要执行许多操作，那么，将其整合到一个更大的内部循环中会显著提高其运行效率。此外，程序员还可以颠倒循环的嵌套顺序。如果程序在调用数据时需要跨越一段很长的距离，那么，编译器将无法对其进行优化。这时，就可以颠倒循环的嵌套顺序，实现连续的数据调用。

待会我们将讨论存储效应。存储性能对应用的影响可谓举足轻重。因此，如果程序中有多个循环需要调用外接存储器中保存的同一个数据，那么，最好将这些循环统一形成一个更大的循环，从而只需要对该数据执行一次读取。请记住，在进行向量化时，如果无法通过标准的C代码实现想要的运算，则可以利用这些明示的小数内建指令。这个向量化内建指令可以要求编译器同时执行两个运算。编译器不必弄清楚这个操作是否安全，你已经向它下达了明确的命令。

在对像这样的紧凑循环进行优化时，最好让编译器借助硬件循环。Blackfin处理器可以实现零开销硬件循环，其效率比通过条件分支返回循环起点的方式高得多。此外，这种硬件循环也很微妙，能够表示编译器完全理解了这个循环。如果编译器发送了LSETUP指令，那么，就表示编译器很愿意处理这个循环，即将对其进行向量化和软件管线优化。反之，如果编译器没有生成硬件循环，则说明编译器可能对这个循环不甚满意。

总的来讲，妨碍编译器生成硬件循环的原因有两个，因此，只需要检查两个嵌套最深的循环。第二，如果循环中存在任何令编译器迷惑不解的函数调用，那么，编译器将不会使用硬件循环。如果这是一个非常简单的局部函数，编译器可以充分理解该函数的意义，那么，编译器仍将生成硬件循环。此外，除了采用正规入口，循环中不得包含控制转移。不过，程序也可以跳出循环，编译器仍将生成硬件循环，虽然这会降低整个循环的效率。

好了，关于硬件循环就讨论到这里吧。现在，再回过头来看看向量化条件。成功实现向量化的条件屈指可数。第一，必须执行顺序存储器存取，因为我们需要同时读取两个操作数。第二，必须知道该数据的对齐方式，因为Blackfin处理器仅读取4字节数据，例如，4字节界限。

第三，编译器必须确知循环的迭代数量。如果循环运行并不频繁，编译器就不会对其进行向量化。单数次数的迭代也会令编译器发生错误，通常会生成序言进程（Prolog）和收尾进程（Epilog）。第四，也是最重要的一点是，数据的别名判别。也就是说，循环中索引的两个数据阵列或缓冲数据是否会重叠。如果编译器对此存有疑虑，就会驻足不前。

现在，我们详细解释数据的对齐方式。程序员可以假定，编译器会帮自己安排好数据，例如，所有最上层的阵列都将位于4字节界限上。不过，程序员也可以帮编译器传递地址——在传递一个数据阵列的地址时，应当传递该阵列的开始地址。例如，如果阵列中的零元素中有一个标记或计数，那么，传递阵列中的第一个、第二个或第三个元素的地址就没有用，反而会让事情变得混乱。应当始终坚持传递阵列的开始地址。

如果编译器无法确知数据的对齐方式，它将插入两种循环形式，以帮助程序员。它将插入一个决策，询问这个数据是否对齐？它将在运行时对此进行评估，然后迅速跳转至对齐的向量化循环或者未经对齐的非向量化循环，从而大幅提升程序的性能。但是，这会使这段代码的长度比需要的长一倍。因此，最好在开始时让编译器对数据的对齐方式一清二楚。

这里有两个例子，分别演示了有助于和阻碍编译器进行优化的两种数据格式。例如，如果程序要处理由两个16位数据（实数和虚数）构成的复数，那么，像这样采用两个单独的阵列来安排这个复数就会加剧编译器的任务，因为它必须从两个不同的地方进行加载，而且每次只能加载16位数据。但是，如果将这些交替读取的实数和虚数数据合并到一个阵列中，那么，只需要进行32位数据加载，就可以同时读取这个复数的两个部分。

另外，程序中经常出现利用Malloc()函数灵活构建的二维或更高维度的阵列。其中包括一行指针，并且这行指针上面的每一行数据都是一个单独的Malloc()请求。从程序的角度而言，这个阵列非常精致、非常灵活，但实际上，它却妨碍了编译器进行优化，因为优化器不知道这些阵列行之间的关系。它不知道这些阵列行在存储空

间里是否相邻，而且也没有发现这些阵列行之间存在恒定的偏移。因此，大大地降低了程序的性能。最好是为这个阵列添加一个简单的二维说明。

下面讨论我在前面曾经提到过的在内存中相隔一定距离调用数据，这个例子来自一个客户应用的基准测试。在第一个例子中，我们根据内部循环控制变量“k”，选择数据元，并乘以一个常数。也就是说，将以该常数的长度为间隔距离，读取数据，而这会阻碍编译器对该程序进行向量化。但是，如果我们注意到这是一个执行加法和乘法的结合运算，那么，我们可以重新排列这些数据项，从而得到第二种形式的程序，按控制变量“k”连续读取数据。结果，这段程序的运行速度提高了6倍。

接下来，我们要来讨论别名判别特性。要实现编译器自动优化，就必须妥善处理这个最让人头痛的环节。引起编译器疑虑的指针主要是从外部进入的指针，可能是自变量或全局变量。此外，具备多个目的的指针，以及C++代码中的基准参数等，也会令编译器不知所云。从根本上讲，程序员应当仔细检查所有的指针，并不断地自问，编译器是不是始终知道我正在索引的是哪段数据？或者，这些指针会不会让编译器觉得含糊不清？

下面这个例子也许能够说明这一点。如果程序员要使用一个全局变量来控制一个FOR循环，那么，每一次程序员在这个循环中编写一个阵列，编译器都会担心这个阵列会不会与这个全局变量相重叠。与此相反，如果程序员在这里使用了一个局部变量，那么，对编译器来讲事情就清楚明了多了。

现在，我们可以归纳出一个利用编译器进行自动优化的模式。首先，最好不要对程序做出任何更改；其次，可以进行不同程度的干预。要对信号处理循环进行优化，程序员可以借助程序间分析。彻底检查整个应用，编译该应用的所有代码，从每一个程序块收集信息，如程序调用了哪些函数、使用了哪些自变量，以及关于全局变量的对齐方式和常数传播情况的信息等。然后，将信息传递回包含了我们试图对其进行向量化的循环的程序块。接着，利用该信息，进行第二轮编译。

当然，最好是程序员一开始就阐明一切情况。编译器将试图按程序员可能采取的方式，传递信息。必须牢记，这是一个与上下文无关的分析。编译器并不知道程序将要使用的确切路径，它将考虑所有的可能性，同时对所有路径进行分析。

第2c节：编译指示

对于这类问题，我推荐的首选解决方案是采用像这样一个局部的可移植干预。这里列出了两个编译指示，“noalias”和“vector_for”。将其插入循环前面，即可为编译器设定循环条件。

“noalias”编译指示的含义是，任何加载和保存操作都不会发生重叠。这样，编译器就不必再顾虑这一点。当然，如果这个断言为假，那么，程序就会产生错误的结

果。此外，也可以为这个说明添加一个限定词“restrict”，表示该指针不能创建别名。

下面有一个与之类似的“vector_for”编译指示，其含义是，告知编译器可以安全地同时执行这个循环中的两个迭代。需要强调指出的是，如果程序员将这段代码移植到其他程序中，那么，这些编译指示将被视作代码注释，不会影响程序运行。对于数据对齐方式，则可以向编译器声称，数据的对齐方式将会非常有用。不幸的是，这不是一个编译指示，而是一个可执行语句。因此，只能将这个语句插入部分位置。

不过，通常应该将这个语句插入函数的开头，根据从外部输入的参数，通知编译器数据元的对齐方式。可以将“all_aligned”编译指示置于循环前端，告知编译器已经完成了数据对齐。如果程序中出现了未完全对齐的数据，那么，程序员甚至可以在这个编译指示中添加一个可选参数，告知编译器该数据并未对齐。这样，编译器将为循环生成一个序言进程，并从第一个对齐的位置开始执行程序。

这里列出的最后一个编译指示是“different_ banks”，其含义是指示编译器在一个周期内同时执行两个加载或一个加载和一个保存操作。虽然这种操作方式会在内存中造成子块冲突，但这个编译指示将鼓励编译器，不用担心这一点。编译器对数据在内存中的布局所知甚少，因此，这个编译指示就是在告诉它，程序员已经妥善安排好了数据布局，完全不必担心会出现问题。

最理想的情况是，程序员利用常量或字来编写FOR循环。如果必须使用变量，则可以利用这个编译指示循环计数，告知编译器关于这个循环的运行状态的信息。第一个参数是一个最短行程计数，表示这个循环的迭代运算至少需要耗费这么长时间。中间的这个参数是最长行程计数，表示这个循环的迭代运算最多需要耗费的时间，从而帮助编译器判断，是不是值得费力对这个循环进行优化。此外，在进行软件管线和向量化时，编译器也可以根据这个行程信息，了解数据调用情况。

第2d节：易变性

还有一个非常有用的限定符——“Volatile（易变性）”。“Volatile”将告知编译器，数据可能因受到外部影响而改变。细想起来，其实数据总是有可能发生变化。在任何真实的应用中，都会有外设通过DMA传输向应用输入数据，或者有某种硬件设备向应用提供数据。编译器必须知道，它绝对不能对可能从底层发生改变的数据做出任何假定。实际上，缺少“Volatile”限定符是产生“Support（支持）”请求的最主要的原因，因为如今的编译优化器已经非常强大、积极，它会找出看起来无所事事的循环，并将之删除。因此，必须在程序中加入“Volatile”限定符，以告知编译器，存在外部影响因素。与之相反的另一个限定符“const（常量）”也非常有用，可以告知编译器该阵列绝对不会发生变化。

采用循环寻址也可以加速信号处理，即，以模数形式运行缓冲数据。如今，可移植C代码也能够实现循环寻址，因为编译器可以识别出寻址表达式中使用了模数运算符，例如，最上面列出的这个运算符。此外，如果编译器无法确知能不能安全地采用循环寻址，但程序员自己知道它确实可以，那么，程序员可以利用“force-circbuf”选项，告知编译器，可以信心十足地采用循环寻址进行优化。

最后，我们还提供了两个明示内建指令，以告知编译器，程序员已经完成了优化。

必须牢记，也有编译器鞭长莫及之处。例如，编译器不可能在可移植C代码中生成FFT运算中的自动位倒序寻址。面对一个真正复杂的指令，例如，Blackfin处理器提供的功能强大的“Viterbi”指令，或者“Bitmux”指令，编译器也只能束手无策。

几个星期以前，我们对一个客户的应用进行了基准测试，通过插入一条“bitMUX”指令，其关键循环的运行速度提高了25倍。这种优化方式堪称百试不爽。程序员必须了解所用处理器的符号集，熟悉可以调用这些功能强大的独立指令的内建指令，并且能够在程序中插入这些内建指令。这是关于这个符号集的另一个例子——“Count Ones”指令。这个例子来自一个真实的客户应用。这个可移植C代码的意思是，计数一个字中出现的位。如果利用适当的指令，那么，其效率将提高60倍。

最后，我们要来讨论终极威慑力量。如果上述措施都不能使编译器执行优化，那么，程序员可以在C程序内部，利用这些内联asm()语句，以汇编语言编写程序。这些asm()语句是模仿[GNU] asm语句的，二者其实并不相同。过去，asm()语句曾是编译器在进行优化时几乎无法逾越的障碍，因为编译优化器在循环中发现汇编语句后，会对一切产生怀疑。但是现在，程序员可以通过一个“Clobbered”寄存器字段，确切地告知编译器，哪些寄存器受到了asm()语句的影响。这样，编译器在优化时就可以绕过这些寄存器，寄存器也会变得更加有用。

如果程序员利用asm()语句修改数据，也会引起编译器的迷惑，那么，程序员可以插入“clobber”字段，表明内存已修改。显然，这是一个很含混的声明：不精确。右侧的例子演示了如何使用asm()语句。在本例中，我们生成了一个在信号处理应用中非常有用的运算基元，目前处理器中还不能提供这个运算基元，也没有采用C语言编写的任何其他形式的运算基元。也就是说，如果处理器中不具备运算基元，那么，程序员可以通过这种方式构建自己的运算基元。

如果觉得有帮助，程序员也可以在C函数中添加这个“clobber”寄存器字符串。例如，如果要调用一个小小的汇编函数，那么，程序员就可以借助“clobber”字符串让编译器知道，将要修改哪个函数。

好了，关于紧凑式运算循环的介绍就到这里。

第3章：优化控制代码

第3a节：条件分支

现在，我们将介绍如何优化控制代码。控制代码就意味着更多标准的可移植C代码，程序中包含了更多分支，需要做出更多决策。和以前一样，如果程序员在配置时选中了“基于统计监测数据进行优化”选项，那么，编译器将自动试图解决大多数问题。

在这个优化过程中，首先要模拟用户的程序。通过模拟，从头到尾跟踪每一次控制转移。建议采用编译模拟器来进行模拟，因为其速度比周期精确模拟器快数百倍。这样，编译器就可以知道每个分支的执行路径，然后，相对轻松地亲自运行程序。根据跟踪获得的信息，编译器将重新编译程序，并尽可能优化所有的分支。如果分支选择两个执行路径的概率各占50％，那么，分支优化就没什么效果。但是，如果一个方向明显优于另一个方向，那么，分支优化将显著提升程序运行速度。一般而言，应用可以实现10％至15％的性能提升。

需要指出的是，这个性能提升幅度是采用任何数据运行程序所实现的一般结果，因此，程序员应当选择适当的样本数据。人们有时候更关注最坏情况而不是一般情况。编译器非常积极地进行分支优化有时也会给程序员增添烦恼。它的目的不是实现静态的预测跳转，而是直接转移。如果程序中包含一个条件表达式，那么，编译器将不会进行跳转，而是直接转至相继的下一条指令，因此不会产生任何停顿。这就是编译器的目标。但这也意味着，编译器可能必须大规模重新编排代码，因此，在完成分支优化后，程序中的汇编块会突然变化位置。

或者，如果程序员不喜欢使用手动选项或不能进行模拟，并且更愿意使用自动选项，则可以手动选择这个选项。程序员可以在C代码中插入“expected true”和“expected false”语句，告知编译器希望分支选择哪个路径。这个例子里，对一个错误调用使用了“expected false”语句。需要检查程序，以确定是否应当调用这个错误的函数，通常编译器将不会进入这个循环，从而避免调用错误的函数。“expected false”语句的含义是，告知编译器出现了极为异常的情况，然后，它将重新编排代码，通过分支结束调用错误函数，或者，更为常见的是，它会毫不迟疑地直接转至下一条指令。

这是一个可以替代条件表达式的简单函数；“max”运算符可以替代条件分支。Blackfin处理器提供了计算最小值、最大值和绝对值的单独指令，所以，我们可以用一个单周期机器指令替代原本可能需要耗用好几个周期的条件分支。对于32位运算和部分16位运算，编译器将为你代劳，完成这个优化步骤，不过，有时候，16位运算需要程序员的人工干预。需要注意的是，“min”和“max”运算符仅适用于带符号值。因此，程序员还必须考虑，是否需要使用无符号变量。

也可以通过其他方式消除这些条件表达式。如果你编写了一个包含“if”或“if then else”语句的循环，那么，这个循环必定包含一些引起长时间停顿的条件分支。不过，你可以将这个循环重新编写成这样，将其模式颠倒过来。将“if”语句置于循环

外部，并在两个程序块中编写两个相同的“FOR”循环，这样就能以同样的代码长度，实现更快的运行速度。

Blackfin处理器拥有一个非常快的条件指令，称为“预测指令”，可以在一个周期内，测试条件并完成条件寄存器赋值。只用一个周期。

编译器生成的代码应该可以实现这一点。如果不能，那么，你就该考虑能不能将代码弄得混乱一点，好让编译器生成这种形式的代码。你需要找出阻碍编译器生成这种代码的原因。一般而言，你要考虑的问题应该是我们所说的“推测性执行”。例如，如果最初的表达式为：如果a<A>，那么X为<表达式1>，否则为<表达式2>。如果我们将这个代码转换为下面这些形式之一，即，在任何情况下，都要无条件地执行<表达式1>和<表达式2>，然后再考虑条件，完成赋值。最后一个语句更加接近机器指令的模式，因此，更易于锁定（即，编译器更容易识别出这个指令模式）。不过，也许关键在于我们将这两个表达式移到了条件的前面，使之成为无条件执行的语句。

那么，如果其中一个表达式包含了可能导致地址错误的代码，会怎么样？这将导致编译器认为，不能移动这个表达式，这会产生程序错误。因此，这个问题得靠程序员来解决。最好分清编译器和程序员各自的职责。在对一个循环进行软件管线优化时，程序员总是想在阵列结束后额外加载一个数据元，这就会导致推测性执行。我们发现，程序员总想将其数据加载到虚拟内存的末端，从而导致非法存取。

因此，编译器会小心从事，不会默认执行软件管线优化。不过，Blackfin处理器提供了一个编译器选项——“-extra-loads”，用于告知编译器，可以安全地推测性加载阵列的末端。

这里显示的是消除条件分支的另一个例子。这个代码十分巧妙，我在这里举这个例子是为了激发大家的创造性思维。这个例子来自真实的客户应用的基准测试。代码开始运行，检查一个标记阵列，也许是个Boolean阵列，检查其条件是否为真。然后，取决于该标记，它会加上缓冲数据乘以64或者减去缓冲数据乘以64。

那么，如果在这个“关键”阵列中如果没有1或0，我们该加上，还是减去64？因此，你可以编写一个完全不包含任何条件的循环表达式。其速度将快得多。不过这个例子很好地演示了编译器也不是万能的，它不能更改这个数据阵列的内容，还得靠程序员自己动手。

第3b节：除法运算

和浮点数据一样，Blackfin处理器也不能通过单周期指令支持除法运算。有些指令可以执行一个位的除法，然而要执行32位除法，代价将非常高昂。因此，应当尽量避免在循环中进行除法运算。必须始终牢记，模数运算符也意味着除法运算。这里

有一个经典的窍门——2的乘方。如果是进行除以2的乘方的运算，则可以将这个表达式右移。

如果除数为无符号变量，那么，这个运算只需要一个周期。不过为了确保绝对安全，如果除数为带符号变量，那么，编译器实际上必须执行这个巧妙的表达式。通过这里显示的这些机器指令，对带符号变量除数进行安全的右移，需要耗费六个周期。因此，程序员应该考虑，在这种情况下，是否真的需要使用带符号或无符号变量。

还要记得考虑，编译器必须能够理解除数的性质。如果除数是一个常量或者字，那么，编译器可以判断出除数是不是2的乘方。但是，如果除数是一个变量或全局变量，那么，编译器能不能理解该变量的含义？该不该简化这个程序？

取决于应用要求的精确度，Blackfin处理器可以支持两种类型的除法运算。如果计算结果和除数都是16位数据，那么，这个运算最少需要约40个周期或者更快。但是，如果执行完全32位除法运算，那么，最多可能需要400个周期。这个差距相当可观，非常值得进行优化。在开始进行除法运算之前，最好考虑要不要进行换算，将32位数据转换成16位数据。

出人意料的是，如果编译器确信有一个内部循环会完成大量任务，并且想借助硬件循环，那么，它可能会生成一个除法。生成硬件循环需要输入这个循环包含的迭代数量。如果编译器处理的对象是变量，那么，它可能会认为冒险采用除法，计算出FOR循环变量包含的迭代数量是值得一试的。因此，最好是使用常量或字。

这是另外一个有趣的小窍门，演示了程序员/编译器执行的另一种除法。在对一个真实的客户应用进行的基准测试中，我们对比率进行了测试。如果“X/Y > A/B”是源代码，那么，只要利用高中代数知识，将这个表达式转换成“X * B > A * Y”，即，将除法转换成简单的乘法，就可以将其运算速度提高数百倍。但问题是，只有在不会发生溢出的情况下，才能进行这种转换。例如，如果x和b是12位值，那不会有问题，因为乘法的结果为24位值，没有超出32位值。但如果这两个值是17位值呢？那么，就有可能发生溢出。编译器不会冒险使数据发生溢出，必须由程序员来考虑8位、16位、32位和64位数据之间的精确度。因此，程序员必须首先做出决策，编译器不能自动进行优化。

第3c节：高级优化

Blackfin处理器将成组的寄存器与函数单元相关联，从而加快了运行速度、降低了功耗。例如，Blackfin处理器既有运算寄存器或d-reg寄存器，又有地址寄存器或p-reg寄存器。但是，在这两类寄存器之间传输数据会产生停顿。这种现象虽然不是经常发生，但最好能够避免出现停顿。

例如，在进行复杂的地址运算时，就会因在运算寄存器和地址寄存器之间传递数据而产生停顿。如果是顺序或者以2或4为间隔，读取一个阵列，那么，不会有什么问题。但如果是一个包含奇数数量元素的结构阵列，那么，当以9或11或类似奇数为间距，读取这个阵列时，Blackfin处理器可能需要在d-reg寄存器中进行任意乘法运算，然后，再将计算结果返回。这样的话，来回两次传输就会产生两次停顿。因此，程序员应当考虑简化地址运算，并注意地址运算指令的执行。

Blackfin处理器身兼两职，既可执行DSP，又可充当微控制器。这两种功能面临的一个共同问题是，整数的长度。在进行DSP运算时，通常采用16位数据。而在用作微处理器，运行决策代码时，就需要处理32位数据。关键是，对于不同长度的数据而言，Blackfin处理器的指令系统架构并不是对称的。换句话说，Blackfin处理器不能对任何长度的数据支持所有的运算。

执行条件分支其实就是进行32位数据比较运算。因此，如果你在程序中将某个“短数据”设定为局部变量，并对这个局部变量进行比较，那么，在每一次比较时，编译器都必须将这些短数据转换成32位数据。因此，最好一开始就将控制代码变量设置为32位数据，而只在数据阵列中采用16位数据。对于循环控制变量尤为如此。对于16位数据，编译器总是担心会发生溢出。

与整数长度有关的另一个问题是乘法运算的代价。一个32位乘法运算需要消耗三个周期，而一个16位乘法运算只需要一个周期。程序员甚至还可以通过向量化，在一个周期中完成两个16位乘法运算。也就是说，16位乘法运算的速度更快，所以，在编程时要注意，尽可能避免32位乘法运算，而将其转换为16位乘法运算。此外，在我们先前讨论过的地址运算中也可以利用这个特点。如果要乘以一个特殊的地址增量，那么，在标准C程序中，这将是一个32位乘法运算。

普通的控制代码通常包含大量非常小的函数。这不利于进行优化。如果编译器无法理解这些函数最终将产生什么结果，可能会造成什么样的改变，它就不会对其进行优化。因此，可以在设置编译器时，选中自动优化选项，并要求进行函数内联。通过对函数进行内联，也就是说，将该函数的代码主体插入调用点，就不用执行需要耗用数个周期的调用指令；也不用花费几个周期来返回调用程序，而且也不必生成参数和返回数值，从而大大提高了程序的运行效率。

此外，同样重要的是，函数内联后，优化器可以更加清楚明白地了解这些函数的作用，可以明确地知道其运行结果。不过，函数内联会增加两个控制转移之间的代码的长度。调用函数就是执行一个控制转移，如果消除了这个控制转移，就需要处理一段更长的代码。对优化器而言，一段比较长的代码为它提供了更加广阔的施展空间，它可以重新安置指令，有效利用停顿周期，大幅提高代码的运行效率。

当然，如果程序员不留心的话，函数内联会显著增加代码的长度。因此，一方面，让编译器自动执行优化，另一方面，程序员也要利用内联限定符来人为地控制被内联的函数。

硬件循环是控制代码编程中的另一个小问题。需要注意的是，硬件循环并不完全符合C语言对FOR循环的定义。

C程序中的FOR循环可以完全没有迭代，而硬件循环则假定至少会执行一次迭代。因此，必须在循环之前插入编译器提供的保护码，以检查循环到底会不会执行迭代。虽然这会让人觉得烦闷，不过，通过限定常量范围或者利用我之前介绍过的“pragma loop_count”语句，就可以轻松避免这种情况。

第4章：存储性能

第4a节：存储代价

好了，现在，我们要来讨论存储性能。这个问题似乎与C代码优化没什么关系。然而严酷的事实是，当处理器的运行速度突飞猛进时，存储器却滞足不前。如今，外接存储器和片上L1内存之间已经存在着巨大的性能差异。我们不得不承认这一点，要加快C程序的运行速度，就必须解决这个问题。

在设置Blackfin处理器时，程序员一般都会选择需要的功率，然后选择尽可能最快的运行速率。但如果受应用存储性能的影响，执行存储器存取操作需要消耗大量周期，那么，建议你不妨考虑使用一个像这样的表格。这里列出了所有可用的总线速率。只需要将总线速率稍微提高一点，就能显著提升整个应用的性能；不过，可能达不到原来的内核速率。当存储总线是影响应用性能的最重要的因素时，就可以借助这个表格，选择适当的运行速率。

这个表格列出了所有理论上可用的性能值，其中这些黄色的横条代表最实用的性能组合。出现这样的间隔是因为我们使用的EZkit评估板采用了特定的时钟频率。如果你的应用中包含一个可变信号和时钟信号发生器，那么，你可以随需任意调节这些数值。利用高速缓存，就能轻松解决存储性能问题。只要启用高速缓存，Blackfin处理器就会自动将正在调用的数据加载到L1内存中，并且只要这些数据被反复调用，就会一直保存在L1内存中。

只要在程序中编写一个连接器选项和一个控制变量，就能启用高速缓存。其他的课程单元将详细介绍这些关于单独控制一条指令以及数据高速缓存的内容。这里还涉及一个有趣的特性——“回写”模式，其作用是告知处理器，不需要随时将高速缓存中发生变化的数据拷贝回外接存储器。可以通过更改CPLB配置表，启用“回写”模式，稍后我将演示。

除了利用高速缓存，还可以直接将至关重要的代码和数据保存到L1数据内存中，从而提高应用的运行速度。通过在C程序中编写一个像这样的语句，就可以实现这一点。这其实是在程序中加入了一个“内存段”指示。在数据布局文件，即LDF文件中，你可以找到“L1data”这个代码段名称。影响应用存储性能的原因有多种，包括

存取外接存储器，以及在外接存储器中转换存储行。在一个典型的Blackfin处理器设置中，每个存储行最多可以保存4 Kb数据。打开一个存储行，然后再转移至另一个存储行需要消耗存储周期。通过将数据分散保存到不同的存储体中，就可以避免转换存储行，下一张幻灯片将列表演示这种方法的效果。

下面介绍一个极为简单易行的解决方案：在标准LDF文件中设置“PARTITION_EZKIT_SDRAM”宏命令，处理器就会将代码保存到一个存储体中、将堆阵保存到下一个存储体中、再将数据保存倒再下一个存储体中，依此类推。只要在“EEMBC Mpeg4”编码器这个典型的大程序中添加这段代码，就能将应用性能提高6.5%，而无需深刻理解其背后的存储原理。

如果想弄得更明白一些，可以看看这张有趣的表格。虽然这个课程的主题是C代码优化，不过，这些直观的数据会加深你的理解。如果读取的是L1内存，那么，耗用的周期数量将如最上面这一行所示，即，每次调用只需要一个周期，这很好。但是，如果你的程序很大，溢出至外接存储器，那么，数据调用就需要消耗第三行（L3）中列出的周期数量。连续读取16位数据，将消耗40个周期。如果要跨不同的存储行读取数据，也就是说，要打开和关闭不同的存储行，则需要141个周期，非常惊人。

当然，只要程序完成了适当的优化，一般不会出现这种现象。例如，如果启用了高速缓存，那么，读取数据将消耗第二行所列周期数量。你可以看出，连续读取耗用的周期数量少多了，即使跨行读取，也还可以。如果将代码分散保存到各个存储体中，则可以完全避免因跨行读取而耗用的周期。

不过这仍然需要7.7个周期，因为我们是执行连续读取，就像信号处理应用接收到的数据流。使用高速缓存的真正优势是，可以最大限度地提高重复调用数据的效率。因此，在编程时需要考虑调用数据的模式。将需要重复调用的数据保存到高速缓存中，然后反复调用，而不是一次又一次地从外接存储器中读取这个数据。

DMA调用也许是解决之道，其他课程将讨论这个问题。写操作的情况也和读操作类似。对L1内存进行写操作只需要一个周期，如果是对高速缓存进行写操作则需要更多周期。右侧列出的是进行跨行写操作所需耗用的周期数量，在这种情况下，应该考虑采用回写模式。这里我要强调指出的是，上述现象不是Blackfin处理器本身的缺陷，而完全是由于其核心频率很高，即使速度最快的存储器的性能也与之相差甚远而造成的。

第4b节：代码运行速率与占用空间之比较

关于存储器，C程序员还要考虑另一个问题——缩小代码占用的空间，以便将这个代码尽可能保存到高速缓存甚或L1内存中。代码越小，程序员就可以将更多的代码保存到这些高速存储器中。此外，如果代码非常大，就会被保存到外接存储器中，占用最终部署中的存储资源。因此，编译器工具提供了不同的程序优化选项。

在默认状态下，编译器将全力以赴实现最快运行速度，不过，程序员也可以要求编译器最大限度地缩短代码长度。也就是说，要求编译器避免进行任何可能增加代码长度的优化。

在DSP内核中，这种情况尤为极端，因为编译器总想尽可能展开循环。因此，程序员可以向编译器发出非常具体的指示。不仅可以要求编译器对某个文件进行编译、尽量缩小其尺寸，而对另一个文件则要求尽量提高其运行速度；还可以直接在函数中添加下面列出的这些“pragma”标记，精确控制程序优化。

通过对运行速度优化和代码长度优化进行分析，我们可以将引起变化的原因分为几个类别。这些优化措施将代码缩短了50％。其中，通过展开循环和优化代码，实现了将近50％的优化。而另一半功劳则应该归函数内联。这也意味着，程序员必须严格控制需要对哪些函数进行内联。此外，还有其他一些次要的影响因素。

值得注意的是这个“-Ofp”选项，它可以平衡堆栈和参数，最大限度地利用16位偏移，从而缩短代码的长度。每一个小小的举措都能优化代码。此外，我要强调的是，代码长度优化不会对代码的功能造成丝毫影响。凡是经运行速度优化的代码能做到的，它都能做到，包括中断等等。

当然，我们也提供了一个自动优化选项，帮助程序员轻松实现优化。通过这个“-Ov”滑动块，程序员可以选择从1到100的值，向编译器表明，希望对应用执行某种程度的运行速度和代码长度折衷优化。要使用这个选项，先要选中“基于统计监测数据进行优化”选项。通过在模拟器上运行这个程序，详尽了解其中各段代码的运行情况。然后，编译器就能对每个代码块对应用性能的重要性了然于胸。

对于很少执行的代码块，将进行代码长度优化，而不是运行速度优化。基于这个原则，编译器将根据每次运行速度优化将使代码长度增加多少，针对用户的应用，提供一个度身定制的灵活的优化方案。对用户而言，这个优化选项既简单，又有效。编译器包揽了所有复杂的任务。

将优化结果绘制成图表，就是这个样子，不论是最大限度缩短代码长度，还是最大限度提高运行速度，实际的优化效果都不太好。而在接近坐标原点的位置，这几个折衷优化点就能实现很好的优化效果。对于Ov滑动块，这些优化点相当于30至70之间的折衷值。这个范围内的任意折衷值都能实现运行速度和代码长度的合理平衡。

如果用户对编译器自动优化并不完全满意，或者无法使用模拟器，则可以动手进行优化。这张表列出的是依次单独对这个应用中的每一个文件进行运行速度优化和代码长度优化，然后，再测量其优化效果。结果很有意思。这一栏是进行运行速度优化的结果，只有黄色亮显的文件才实现了显著的性能提升。也就是说，应用中的大部分代码都只需要尽可能缩短其代码长度，而对运行速度几乎没什么影响。

因此，可以不必理会这28个文件中的20个文件，而只对余下的几个文件进行运行速度优化。右侧的几栏数据显示了进行代码长度优化的效果。蓝色亮显的文件的长度明显缩短。有趣的是，这些文件与黄色亮显的文件并不一定重合，这就意味着，可以对这些文件进行代码长度优化而不会影响其性能。

从这些表格可以看出，程序员可以列出一个运行速度和代码长度的折衷优先列表，根据自己应用的情况，制定一些决策。除了完全依靠编译器自动优化，程序员也可以通过这种方法，分析问题。你会惊奇地发现，原来不是所有文件都需要进行优化。

第5章：举例

第5a节：数据结构

好了，在最后这个小节，我们将以几个真实的客户应用的基准测试为例，回顾我们先前讨论的内容，演示在实践中如何应用这些优化技术。

第一个例子很简单，取自一个复杂的数据结构。显而易见，这个程序是在执行两个MAC运算，并未进行向量化，每行代码执行一个MAC运算。

对程序员而言，这段代码可能已经很好了。它可能已经实现了性能目标，你不想再进一步，不想再继续对着这堆代码绞尽脑汁。很好。不过，如果你需要缩短这个代码的运行周期，那么，就得考虑，为什么不对其进行向量化呢？因为，这个程序执行了三个加载，但不是在一个周期内完成这三个加载。回想一下我们先前讨论过的复数的情况，一个复数包含两个相邻的16位虚数和16位实数，那么，为什么不如法炮制，也将这两个16位加载合并成一个32位的加载？

原因是它们属于同一个数据结构。这里还有另一个16位元素。因此，这个结构阵列总共包含6个字节。也就是说，在这个结构阵列中，有一个元素是对齐地址，而另一个则是非对齐地址，所以，我们不能进行32位加载。编译器无力独自解决这个问题，但程序员可以考虑改变这个数据结构。要做到这一点其实很容易，只要将这个数据拷贝到一个局部阵列中，并保持实数和虚数相邻，然后运行这个拷贝就可以了。

这样，我们并未对程序员的数据结构进行太大改动，而是做了一个相关数据的局部拷贝，但收获颇丰，我们获得了想要的代码结构。优化的向量化MAC运算。这里，需要注意的一个小问题是，有多种可能的表达式，但是，使用这个[K*2]能够向编译器暗示，这个小阵列实现了4字节对齐。

第5b节：整个应用

下面介绍最后一个例子。这是我们对一个客户应用从头到尾执行的基准测试，旨在评估该应用在Blackfin处理器上运行时是否实现了客户的性能目标。这是一个快速评估。一般而言，第一步是插入定时点和检查代码，以便了解该应用的优化效果。在初始状态下，这个应用运行需要5,000万个周期。现在，启用优化选项，运行周期减少至950万，这再一次说明了为什么确实需要进行优化。

然后，启用高速缓存，消除了许多讨厌的存储效应，周期数量进一步减至140万。刚开始，就使运行速度大幅提高。接下来，我们检查了全局优化情况：IPA选项、对齐函数等等，检查这些优化的效果；然后，我们继续执行更加有趣的优化。我们启用了统计监测器，得到一个需要依次对其进行优化的“热点”代码段优先顺序列表；这个列表也能帮助我们进一步理解这个应用的运行情况。最后，我们设置预期的总线速率（单位为MHz），然后执行存储优化。

下面，介绍如何对这些热点代码段进行优化。

第一个热点代码段可真让我们汗颜。Blackfin处理器提供了一个很棒的ETSI小数运算实现，客户的程序默认采用了这个代码段。然而，这个代码的运行速度似乎很慢，我仔细检查这段代码时，才发现，原来调用对象是ETSI函数，而不是一个机器指令。因为客户并不知道，必须设置我们提供的“#define ETSI_SOURCE”标志，才能实现ETSI模式。

因此，我们改进了介绍ETSI模式的技术文档。只要完成了这个设置，就能显著提升应用性能，在这个代码段，周期数量从187,000减至141,000。

第二个热点是一个关键循环，其中包含三个条件跳转。这个代码采用16位数据，这上面有一个“if”语句，中间一个“if”语句，下面还有一个“if”语句。因此，这个代码中的大部分周期都是这些条件分支产生的停顿周期。初始运行周期数量为67,000。

我们的任务是尽量消除这些停顿周期。第一个条件跳转的目的是使用标准C库中的一个“max”指令，因此，只要像这样直接表达出来，或者使用内建指令就可以了。这样，运行周期数量从67,000减至40,000。编译器未能执行自动优化的原因是，这个代码采用16位数据。

下面，我们来看看第二个条件跳转。这就是这个条件跳转。如果满足这个条件，就把它保存到内存中。这里，我又使用了“MAX”指令。下面这个表达式中，我们无条件地保存到内存中。编译器未能执行自动优化的原因是，这改变了存储存取。现在，每一次都要执行写操作。这个表达式不只是增加了存储总线的负载，而且可能产生地址错误。因此，必须由程序员来做出这个决策。这可以减少6,500个运行周期。

第三个，也就是最后一个条件跳转看起来像是一个预测指令，一个单周期条件分支。如果“CC”，那么，就给寄存器赋值。我重新编排了这个代码，形成了这种模

式。现在的问题是，我们无条件执行的许多操作原本都是有条件执行的。编译器不会冒这个险，必须由程序员迫使它这么做。

现在，每一次都要执行保存操作。因此，又减少了6,500个运行周期。

在刚才的条件跳转优化中，我们将变量“maxval”的定义从2字节改为4字节，从而避免了强制数据类型转换，并且减少了一个代码运行周期，这在整个应用中相当于2,000个周期。

最后得到的代码就是这个样子。不再是一个条件跳转。而是一段很短的代码。事实上，如果实现了软件管线优化，这个代码的运行时间还可以再减少一个周期。我们发现，可以利用“pragma no_alias”实现这一点。

现在，这个循环的运行速度比原来提高了3.5倍，完全没有任何跳转停顿。这里，我要强调的是，程序员不一定非要进行这种优化。只是如果非常需要减少程序的运行周期，就可以采用这些技巧。除非想要节省运行时间，否则，没必要把自己的程序弄得乱糟糟的。

统计监测器查出的下一个焦点代码段是“memcpy()”。客户应用在外接存储器的缓冲区之间传输数据。我们很难干预这个过程，要知道，这是一个客户应用，因此，我们并不理解这个程序的所有操作。因此，我们只是试着启用了“回写”高速缓存模式。结果，显著提高了运行效率。

只要在表文件“CPLBtab.H”中修改一个条目，就能启用“回写”高速缓存模式。这里演示了如何进行修改。

关于“memcpy()”函数，你们应当知道，这些函数将产生4字节数据对齐。这是Blackfin处理器提供的memcpy()函数的一个特性。它将动态检查接收到的数据是否对齐，如果是，则转换至字级循环。否则，将执行传统的逐字节拷贝可移植C代码。问题是，逐字节拷贝的速度非常慢。实际上，执行32位字循环的速度比逐字节拷贝快8倍。因此，很有必要考虑，输入memcpy()函数的数据是否正确对齐。此外，输入请求，每次向memcpy()函数输入一两个字节，也会降低其运行效率。

因此，我们与客户讨论了如何重新对齐这个应用的数据缓冲区，以便充分利用这个特性。

统计监测器发现的另一个问题代码段是这个包含四个MAC运算的循环，原来这段代码仅利用了Blackfin处理器提供的两个累加器之一。接着，我们又对这个代码段进行了测试以查出症结所在。也许，问题就出在这个循环的复杂性本身。因此，我们将这个循环分割成两个单独的循环，就像现在这个样子。于是，每个循环包含两个MAC运算，各自使用一个累加器。这个办法很管用。我们得到了这个业经优化的向量化双MAC运算代码，运行周期减少了20,000个。

现在，评估进入尾声。对应用进行评估的最后一个步骤是在目标平台上运行应用，这个应用的目标平台是BF532处理器。在前面的讨论中，我们一直是在BF533开发平台上进行各种优化。应用的运行时间已经从150万个周期减至73万个周期。换句话说，通过对统计监测器查出的这些热点代码段进行优化，我们已经将应用性能提升了一倍。

通过设置总线，还可以进一步提高应用性能。将总线速率提高至这个值。这个应用的存储性能有限，因此，我们最大限度地提高了总线速率，将其设置为130.5 MHz。

最后，我们要考虑的是，应该如何优化利用L1内存。在默认状态下，优化工具会按先来后到的顺序，将最早接收到的任何数据保存到高速L1内存中。因此，只要有选择性地将最经常调用的数据保存到L1内存中，就能加快应用运行速度。由于应用不是我们编写的，所以，这个优化环节的关键在于，确定需要将哪些数据保存到L1内存中。

我们借助了开始时的暗示信息。通过研究统计监测器提供的指令综合视图，我们发现一条保存指令的运行时间比与之相邻的其他指令长了12倍，显然这个指令在存储方面存在问题。于是，我们往回追溯出这条指令的用户数据缓冲区，并将该缓冲区映射到L1内存中。然后，我们又检查了高速缓存的运行情况。优化工具提供了一个高速缓存浏览器，借助这个浏览器，用户可以查看有没有任何数据段导致高速缓存中发生大量存储冲突，如果有，就将之转移到L1 data内存段。最终，应用的运行周期缩短为747,000。评估圆满完成，客户非常满意。

最后，我要指出的是，所有这些优化都是可选的。如果用户将应用导入Blackfin处理器后，直接就能实现目标性能，那很好。如果希望进一步提高应用的运行速度，则可以采取上述优化技术。程序员可以登录我们的Blackfin网站，查阅参考资料。包括我我们先前讨论过的关于时延的工程师注释，以及Blackfin编译器手册《让C代码实现最优性能》，其中有一章非常有用，更加详细地介绍了许多优化技术。

如有任何其他疑问，请与我们联系。我们的Blackfin技术支持团队将随时帮助程序员解决应用性能问题。希望今天的课程能对大家有所帮助，谢谢观看！