可以看到测量值有些不太精确。对于整数求和的 CPE 数更像是 23.00，而不是 22.68；对于整数乘积的 CPE 数则是 20.0 而非 20.02。我们不会“捏造”数据让它们看起来好看一点儿，只是给出了实际获得的测量值。有很多因素会使得可靠地测量某段代码序列需要的精确周期数这个任务变得复杂。检查这些数字时，在头脑里把结果向上或者向下取整几百分之一个时钟周期会很有帮助。

未经优化的代码是从 C 语言代码到机器代码的直接翻译，通常效率明显较低。简单地使用命令行选项“-O1”，就会进行一些基本的优化。正如可以看到的，程序员不需要做什么，就会显著地提高程序性能——超过两个数量级。通常，养成至少使用这个级别优化的习惯是很好的。(使用-Og 优化级别能得到相似的性能结果。)在剩下的测试中，我们使用 -O1 和-O2 级别的优化来生成和测量程序。

5.4　消除循环的低效率

可以观察到，过程 combine1 调用函数 vec_length 作为 for 循环的测试条件，如图 5-5 所示。回想关于如何将含有循环的代码翻译成机器级程序的讨论(见 3.6.7 节)，每次循环迭代时都必须对测试条件求值。另一方面，向量的长度并不会随着循环的进行而改变。因此，只需计算一次向量的长度，然后在我们的测试条件中都使用这个值。

图 5-6 是一个修改了的版本，称为 combine2，它在开始时调用 vec_length，并将结果赋值给局部变量 length。对于某些数据类型和操作，这个变换明显地影响了某些数据类型和操作的整体性能，对于其他的则只有很小甚至没有影响。无论是哪种情况，都需要这种变换来消除这个低效率，这有可能成为尝试进一步优化时的瓶颈。

```
1   /* Move call to vec_length out of loop */
2   void combine2(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6
7       *dest = IDENT;
8       for (i = 0; i < length; i++) {
9           data_t val;
10          get_vec_element(v, i, &val);
11          *dest = *dest OP val;
12      }
13  }
```

图 5-6　改进循环测试的效率。通过把对 vec_length 的调用移出循环测试，我们不再需要每次迭代时都执行这个函数

函数	方法	整数		浮点数	
		+	*	+	*
combine1	抽象的-O1	10.12	10.12	10.17	11.14
combine2	移动 vec_length	7.02	9.03	9.02	11.03

这个优化是一类常见的优化的一个例子，称为*代码移动*(code motion)。这类优化包括识别要执行多次(例如在循环里)但是计算结果不会改变的计算。因而可以将计算移动到代码前面不会被多次求值的部分。在本例中，我们将对 vec_length 的调用从循环内部移动