code/opt/vec.c

```
1  data_t *get_vec_start(vec_ptr v)
2  {
3      return v->data;
4  }
```

code/opt/vec.c

```
1  /* Direct access to vector data */
2  void combine3(vec_ptr v, data_t *dest)
3  {
4      long i;
5      long length = vec_length(v);
6      data_t *data = get_vec_start(v);
7
8      *dest = IDENT;
9      for (i = 0; i < length; i++) {
10         *dest = *dest OP data[i];
11     }
12 }
```

图 5-9　消除循环中的函数调用。结果代码没有显示性能提升，但是它有其他的优化

令人吃惊的是，性能没有明显的提升。事实上，整数求和的性能还略有下降。显然，内循环中的其他操作形成了瓶颈，限制性能超过调用 get_vec_element。我们还会再回到这个函数(见 5.11.2 节)，看看为什么 combine2 中反复的边界检查不会让性能更差。而现在，我们可以将这个转换视为一系列步骤中的一步，这些步骤将最终产生显著的性能提升。

5.6　消除不必要的内存引用

combine3 的代码将合并运算计算的值累积在指针 dest 指定的位置。通过检查编译出来的为内循环产生的汇编代码，可以看出这个属性。在此我们给出数据类型为 double，合并运算为乘法的 x86-64 代码：

```
    Inner loop of combine3.  data_t = double, OP = *
    dest in %rbx, data+i in %rdx, data+length in %rax
1   .L17:                                   loop:
2     vmovsd  (%rbx), %xmm0                   Read product from dest
3     vmulsd  (%rdx), %xmm0, %xmm0            Multiply product by data[i]
4     vmovsd  %xmm0, (%rbx)                   Store product at dest
5     addq    $8, %rdx                        Increment data+i
6     cmpq    %rax, %rdx                      Compare to data+length
7     jne     .L17                            If !=, goto loop
```

在这段循环代码中，我们看到，指针 dest 的地址存放在寄存器%rbx 中，它还改变了代码，将第 i 个数据元素的指针保存在寄存器%rdx 中，注释中显示为 data+i。每次迭代，这个指针都加 8。循环终止操作通过比较这个指针与保存在寄存器%rax 中的数值来判断。我们可以看到每次迭代时，累积变量的数值都要从内存读出再写入到内存。这样的读写很浪费，因为每次迭代开始时从 dest 读出的值就是上次迭代最后写入的值。

我们能够消除这种不必要的内存读写，按照图 5-10 中 combine4 所示的方式重写代码。引入一个临时变量 acc，它在循环中用来累积计算出来的值。只有在循环完成之后结果才存放在 dest 中。正如下面的汇编代码所示，编译器现在可以用寄存器%xmm0 来保存