
 **练习题 5.10** 作为另一个具有潜在的加载-存储相互影响的代码，考虑下面的函数，它将一个数组的内容复制到另一个数组：

```
1 void copy_array(long *src, long *dest, long n)
2 {
3     long i;
4     for (i = 0; i < n; i++)
5         dest[i] = src[i];
6 }
```

假设 *a* 是一个长度为 1000 的数组，被初始化为每个元素 *a[i]* 等于 *i*。

- 调用 `copy_array(a+1,a,999)` 的效果是什么？
- 调用 `copy_array(a,a+1,999)` 的效果是什么？
- 我们的性能测试表明问题 A 调用的 CPE 为 1.2(循环展开因子为 4 时，该值下降到 1.0)，而问题 B 调用的 CPE 为 5.0。你认为是什么因素造成了这样的性能差异？
- 你预计调用 `copy_array(a,a,999)` 的性能会是怎样的？

 **练习题 5.11** 我们测量出前置和函数 `psum1`(图 5-1) 的 CPE 为 9.00，在测试机器上，要执行的基本操作——浮点加法的延迟只是 3 个时钟周期。试着理解为什么我们的函数执行效果这么差。


下面是这个函数内循环的汇编代码：

Inner loop of psum1

a in %rdi, i in %rax, cnt in %rdx

| | |
|--------------------------------------|------------------|
| 1 .L5: | loop: |
| 2 vmovss -4(%rsi,%rax,4), %xmm0 | Get p[i-1] |
| 3 vaddss (%rdi,%rax,4), %xmm0, %xmm0 | Add a[i] |
| 4 vmovss %xmm0, (%rsi,%rax,4) | Store at p[i] |
| 5 addq \$1, %rax | Increment i |
| 6 cmpq %rdx, %rax | Compare i:cnt |
| 7 jne .L5 | If !=, goto loop |

参考对 `combine3`(图 5-14)和 `write_read`(图 5-36)的分析，画出这个循环生成的数据相关图，再画出计算进行时由此形成的关键路径。解释为什么 CPE 如此之高。

 **练习题 5.12** 重写 `psum1`(图 5-1)的代码，使之不需要反复地从内存中读取 *p[i]* 的值。不需要使用循环展开。得到的代码测试出的 CPE 等于 3.00，受浮点加法延迟的限制。

5.13 应用：性能提高技术

虽然只考虑了有限的一组应用程序，但我们能得出关于如何编写高效代码的很重要的经验教训。我们已经描述了许多优化程序性能的基本策略：

1) 高级设计。为遇到的问题选择适当的算法和数据结构。要特别警觉，避免使用那些会渐进地产生糟糕性能的算法或编码技术。

2) 基本编码原则。避免限制优化的因素，这样编译器就能产生高效的代码。

- 消除连续的函数调用。在可能时，将计算移到循环外。考虑有选择地妥协程序的模块性以获得更大的效率。

- 消除不必要的内存引用。引入临时变量来保存中间结果。只有在最后的值计算出来时，才将结果存放到数组或全局变量中。