

复图 5-14 右边的模板 n 次, 就能得到这张图。我们可以看到, 程序有两条数据相关链, 分别对应于操作 `mul` 和 `add` 对程序值 `acc` 和 `data+i` 的修改。假设浮点乘法延迟为 5 个周期, 而整数加法延迟为 1 个周期, 可以看到左边的链会成为关键路径, 需要 $5n$ 个周期执行。右边的链只需要 n 个周期执行, 因此, 它不会制约程序的性能。

图 5-15 说明在执行单精度浮点乘法时, 对于 `combine4`, 为什么我们获得了等于 5 个周期延迟界限的 CPE。当执行这个函数时, 浮点乘法器成为了制约资源。循环中需要的其他操作——控制和测试指针值 `data+i`, 以及从内存中读数据——与乘法器并行地进行。每次后继的 `acc` 的值被计算出来, 它就反馈回来计算下一个值, 不过只有等到 5 个周期后才能完成。


其他数据类型和运算组合的数据流与图 5-15 所示的内容一样, 只是在左边的形成数据相关链的数据操作不同。对于所有情况, 如果运算的延迟, L 大于 1, 那么可以看到测量出来的 CPE 就是 L , 表明这个链是制约性能的关键路径。

2. 其他性能因素

另一方面, 对于整数加法的情况, 我们对 `combine4` 的测试表明 CPE 为 1.27, 而根据沿着图 5-15 中左边和右边形成的相关链预测的 CPE 为 1.00, 测试值比预测值要慢。这说明了一个原则, 那就是数据流表示中的关键路径提供的只是程序需要周期数的下界。还有其他一些因素会限制性能, 包括可用的功能单元的数量和任何一步中功能单元之间能够传递数据值的数量。对于合并运算为整数加法的情况, 数据操作足够快, 使得其他操作供应数据的速度不够快。要准确地确定为什么程序中每个元素需要 1.27 个周期, 需要比公开可以获得的更详细的硬件设计知识。

总结一下 `combine4` 的性能分析: 我们对程序操作的抽象数据流表示说明, `combine4` 的关键路径长 $L \cdot n$ 是由对程序值 `acc` 的连续更新造成的, 这条路径将 CPE 限制为最多 L 。除了整数加法之外, 对于所有的其他情况, 测量出的 CPE 确实等于 L , 对于整数加法, 测量出的 CPE 为 1.27 而不是根据关键路径的长度所期望的 1.00。

看上去, 延迟界限是基本的限制, 决定了我们的合并运算能执行多快。接下来的任务是重新调整操作的结构, 增强指令级并行性。我们想对程序做变换, 使得唯一的限制变成吞吐量界限, 得到接近于 1.00 的 CPE。

 **练习题 5.5** 假设写一个对多项式求值的函数, 这里, 多项式的次数为 n , 系数为 a_0, a_1, \dots, a_n 。对于值 x , 我们对多项式求值, 计算

$$a_0 + a_1x + a_2x^2 + \dots + a_nx^n \quad (5.2)$$

这个求值可以用下面的函数来实现, 参数包括一个系数数组 `a`、值 `x` 和多项式的次数 `degree` (等式 (5.2) 中的值 n)。在这个函数的一个循环中, 我们计算连续的等式的项, 以及连续的 x 的幂:

```
1  double poly(double a[], double x, long degree)
2  {
3      long i;
4      double result = a[0];
5      double xpwr = x; /* Equals x^i at start of loop */
6      for (i = 1; i <= degree; i++) {
7          result += a[i] * xpwr;
8          xpwr = x * xpwr;
9      }
10     return result;
11 }
```