发生在它们第一次变成错误时。另一方面，值 src1[i1]和 src2[i2]之间的比较(第6行)，对于通常的数据来说，都是非常难以预测的。这个比较控制一个条件分支，运行在随机数据上时，得到的 CPE 大约为 15.0(这里元素的数量为 $2n$)。

重写这段代码，使得可以用一个条件传送语句来实现第一个循环中条件语句(第6～9行)的功能。

5.12　理解内存性能

到目前为止我们写的所有代码，以及运行的所有测试，只访问相对比较少量的内存。例如，我们都是在长度小于 1000 个元素的向量上测试这些合并函数，数据量不会超过 8000 个字节。所有的现代处理器都包含一个或多个高速缓存(cache)存储器，以对这样少量的存储器提供快速的访问。本节会进一步研究涉及加载(从内存读到寄存器)和存储(从寄存器写到内存)操作的程序的性能，只考虑所有的数据都存放在高速缓存中的情况。在第 6 章，我们会更详细地探究高速缓存是如何工作的，它们的性能特性，以及如何编写充分利用高速缓存的代码。

如图 5-11 所示，现代处理器有专门的功能单元来执行加载和存储操作，这些单元有内部的缓冲区来保存未完成的内存操作请求集合。例如，我们的参考机有两个加载单元，每一个可以保存多达 72 个未完成的读请求。它还有一个存储单元，其存储缓冲区能保存最多 42 个写请求。每个这样的单元通常可以每个时钟周期开始一个操作。

5.12.1　加载的性能

一个包含加载操作的程序的性能既依赖于流水线的能力，也依赖于加载单元的延迟。在参考机上运行合并操作的实验中，我们看到除了使用 SIMD 操作时以外，对任何数据类型组合和合并操作来说，CPE 从没有到过 0.50 以下。一个制约示例的 CPE 的因素是，对于每个被计算的元素，所有的示例都需要从内存读一个值。对两个加载单元而言，其每个时钟周期只能启动一条加载操作，所以 CPE 不可能小于 0.50。对于每个被计算的元素必须加载 k 个值的应用，我们不可能获得低于 $k/2$ 的 CPE(例如参见家庭作业 5.15)。

到目前为止，我们在示例中还没有看到加载操作的延迟产生的影响。加载操作的地址只依赖于循环索引 i，所以加载操作不会成为限制性能的关键路径的一部分。

要确定一台机器上加载操作的延迟，我们可以建立由一系列加载操作组成的一个计算，一条加载操作的结果决定下一条操作的地址。作为一个例子，考虑函数图 5-31 中的函数 list_len，它计算一个链表的长度。在这个函数的循环中，变量 ls 的每个后续值依赖于指针引用 ls->next 读出的值。测试表明函数 list_len 的 CPE 为 4.00，我们认为这直接表明了加载操作的延迟。要弄懂这一点，考虑循环的汇编代码：

```
    Inner loop of list_len
    ls in %rdi, len in %rax
1   .L3:                        loop:
2     addq    $1, %rax            Increment len
3     movq    (%rdi), %rdi        ls = ls->next
```

```
1   typedef struct ELE {
2       struct ELE *next;
3       long data;
4   } list_ele, *list_ptr;
5
6   long list_len(list_ptr ls) {
7       long len = 0;
8       while (ls) {
9         len++;
10        ls = ls->next;
11      }
12      return len;
13  }
```

图 5-31　链表函数。其性能受限于加载操作的延迟