内循环，用执行 get_vec_element 代码的内联函数结果替换对数据元素的访问。我们称这个新版本为 combine4b。这段代码执行了边界检查，还通过向量数据结构来引用向量元素。

```
1  /* Include bounds check in loop */
2  void combine4b(vec_ptr v, data_t *dest)
3  {
4      long i;
5      long length = vec_length(v);
6      data_t acc = IDENT;
7
8      for (i = 0; i < length; i++) {
9          if (i >= 0 && i < v->len) {
10             acc = acc OP v->data[i];
11         }
12     }
13     *dest = acc;
14 }
```

然后，我们直接比较使用和不使用边界检查的函数的 CPE：

函数	方法	整数		浮点数	
		+	*	+	*
combine4	无边界检查	1.27	3.01	3.01	5.01
combine4b	有边界检查	2.02	3.01	3.01	5.01

对整数加法来说，带边界检测的版本会慢一点，但对其他三种情况来说，性能是一样的。这些情况受限于它们各自的合并操作的延迟。执行边界检测所需的额外计算可以与合并操作并行执行。处理器能够预测这些分支的结果，所以这些求值都不会对形成程序执行中关键路径的指令的取指和处理产生太大的影响。

2. 书写适合用条件传送实现的代码

分支预测只对有规律的模式可行。程序中的许多测试是完全不可预测的，依赖于数据的任意特性，例如一个数是负数还是正数。对于这些测试，分支预测逻辑会处理得很糟糕。对于本质上无法预测的情况，如果编译器能够产生使用条件数据传送而不是使用条件控制转移的代码，可以极大地提高程序的性能。这不是 C 语言程序员可以直接控制的，但是有些表达条件行为的方法能够更直接地被翻译成条件传送，而不是其他操作。

我们发现 GCC 能够为以一种更“功能性的”风格书写的代码产生条件传送，在这种风格的代码中，我们用条件操作来计算值，然后用这些值来更新程序状态，这种风格对立于一种更“命令式的”风格，这种风格中，我们用条件语句来有选择地更新程序状态。

这两种风格也没有严格的规则，我们用一个例子来说明。假设给定两个整数数组 a 和 b，对于每个位置 i，我们想将 a[i]设置为 a[i]和 b[i]中较小的那一个，而将 b[i]设置为两者中较大的那一个。

用命令式的风格实现这个函数是检查每个位置 i，如果它们的顺序与我们想要的不同，就交换两个元素：

```
1  /* Rearrange two vectors so that for each i, b[i] >= a[i] */
2  void minmax1(long a[], long b[], long n) {
3      long i;
```