代码里充斥着 if-else 分支有什么不好吗？除了可维护性，对程序运行效率有什么影响吗？

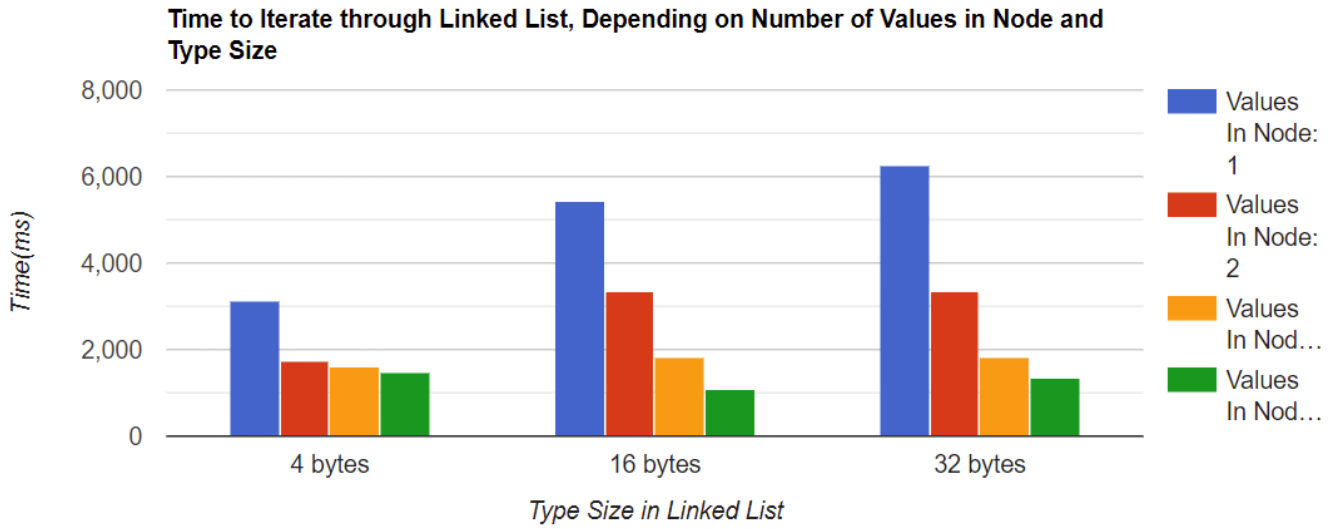
代码
代码质量
编程技巧

代码里充斥着 if-else 分支有什么不好吗？除了可维护性，对程序运行效率有什么影响吗？

代码里充斥着 if-else 分支有什么不好吗？除了可维护性，对程序运行效率有什么影响吗？
被浏览
743,285

> *有篇国外的文章，详细的分析了一些避免分支结构的方法，同时做了基准测试，结果显示目前来说if else性能不算差。*
> *译者注：原文<How branches influence the performance of your code and what can you do about it?>*

这是关于底层优化的第三篇文章，前面两篇为：



函数调用的代价与优化16 赞同 · 0 评论文章

我们已经涵盖了与数据缓存和函数调用优化有关的前两个主题，接下来将讨论有关于分支相关的内容。所以分支有什么特别的嘛？

分支（亦或跳转）是最常用的指令类型之一。在统计学上，每 5 条指令就会存在一个分支相关的指令。 分支可以有条件地或无条件地改变程序的执行流程。对于 CPU来说， 高效的分支实现对于良好的性能至关重要。

在我们解释分支如何影响 CPU 性能之前，先简单介绍一下 CPU 的内部组织形式。

CPU内部的组织形式

今天的许多现代处理器（但不是全部，特别是用于嵌入式系统的一些处理器）具有以下一些或全部特征：

- *指令流水(Pipline)*:管道允许CPU同时执行一条以上的指令。能实现这一效果的原因是因为 CPU 将每条指令的执行分成几个阶段，每条指令处于不同的执行阶段。汽车工厂也采用同样的原则：在任何时候，工厂都有50辆汽车在同时生产，例如，一辆车正在喷漆，发动机正在安装在另一辆车上，车灯正在安装在第三辆车上，等等。指令流水可以很短，只有几个阶段（如三个阶段），也可以很长，有很多阶段（如二十个阶段）。（我在我的一篇博客中有用到指令流水来优化程序，不太明确的可以看下）。
- *失序执行(Out of order execution)* :在程序员的视角下，程序指令是一个接一个执行的。但是在 CPU 视角下情况是完全不同的：CPU 不需要按照指令在内存中出现的顺序执行指令。在执行期间，CPU的一些指令将被阻塞，等待来自内存的数据或等待其他指令的数据。CPU 跑去执行后面那些没被阻塞的指令。当阻塞的指令被激活后，那些没被阻塞的指令已经执行完毕。这样可以节省CPU周期。
- *推测性执行(Speculative execution)* :CPU 可以预先执行一些指令，即使它不是 100% 确定这些指令需要被执行。例如，它将猜测一个条件性分支指令的结果，然后在完全确定将进行分支之前开始执行分支目的地的指令。如果后来CPU发现猜测（推测）是错误的，它将取消预先执行指令的结果，一切都将以没有推测的方式出现。
- *分支机构预测(Branch prediction)* :现代的 CPU 有特殊的电路，对于每个分支指令都会记住其先前的结果：已跳转的分支或未跳转的分支。当下一次执行相同的分支指令时，CPU 将利用这些信息来猜测分支的目的地，然后在分支目的地开始预测性地执行指令。在分支预测器是正确的情况下，这会提高程序性能。

所有现代处理器都有指令流水系统，以便更好地利用 CPU 的资源。而且大多数处理器都有分支预测和推测执行。就失序执行而言，大多数低端的低功耗处理器都没有这个功能，因为它消耗了大量功耗，而且速度的提高并不巨大。 但不要太看重这些，因为这些信息可能在几年后就会过时。

你可以阅读 Jason Robert Carey Patterson: *Modern Microprocessors - a 90 minutes guide*来了解更多现代处理器的特性。

现在让我们来谈谈这些 CPU 的特性是如何影响分支的。

CPU如何处理分支的?

从 CPU 的角度看，分支的代价是高昂的。

当一条分支指令进入处理器流水时，在解码和计算其目的地之前，并不知道分支目的地。分支指令后面的指令可以是：1）直接跟在分支后面的指令；2）分支目的地的指令。

对于有指令流水的处理器来说，这就是一个问题。为了保持指令流水饱和并避免减速，处理器需要在处理器解码分支指令之前就知道分支目的地。取决于处理器的设计方式，它可以：

1. 暂停指令流水（专业技术名称*stall the pipeline*），停止解码指令，直到解码完分支指令并知道分支目的地。然后继续执行指令流水。
2. 紧随分支之后的加载指令。如果后来发现这是一个错误的选择，处理器将需要刷新流水线并开始从分支目的地加载正确的指令。
3. 分支预测器是否应该加载紧随分支之后的指令或分支目的地的指令。分支预测器还需要告诉指令流水哪里是分支的目的地（否则，需要等到流水解决了分支的目的地之后，才新指令加载到流水中）。

现在，除了一些非常低端的嵌入式处理器之外，很少有人会采用这种 1）这种方法。只是让处理器什么都不做是对其资源的浪费，所以大多数处理器会做2）。具有2）处理方式的处理器在低端嵌入式系统和面向低端市场处理器中很常见。常见的台式机和笔记本电脑CPU都采取 3）处理方式。

带有分支预测器的CPU上的分支

如果处理器有一个分支预测器和推测执行，如果分支预测器是正确的，那么分支预测有较小的代价。万一不是的话，分支预测具有较高的代价。这对于具有较长的指令流水的 CPU 来说尤其如此，在这种情况下，CPU 需要在预测错误的情况下刷新许多指令。错误预测的准确代价是不一样的，但是一般的规则是：CPU 越贵，分支预测错误的代价越高。

有一些分支很容易预测，当然也有一些分支则很难预测。为了说明这一点，想象一下一个算法，该算法在一个数组中循环并找到最大的元素。条件 `if (a[i] < max) max = a[i]` 对于一个有随机元素的数组来说，大多数时候条件都为假。 现在想象一下第二个算法，计算小于数组平均值的元素数量。 `if (a[i] < mean) cnt++` 分支预测器在随机数组中很难预测的。

关于推测性执行的一个简短说明。推测性执行是一个更广泛的术语，但在分支的背景下，它意味着对分支的条件进行推测（猜测）。现在经常出现的情况是，分支条件不能被推测，因为 CPU 正在等待数据或者正在等待其他指令的完成。推测性执行将允许 CPU 至少执行几条在分支主体内的指令。当分支条件最终被评估时，这项工作可能会变成有用的，从而使 CPU 节省了一些周期，或者是没用的，CPU 会把预测的相关内容清空。

了解分支汇编

C 和 C++ 中的分支由一个需要判断的条件和一系列需要在条件满足的情况下执行的命令组成。在汇编层面，条件判断和分支通常是两条指令。请看下面这个 C 语言的小例子：

```
if (p =/= nullptr) {
    transform(p);
} else {
    invalid++;
}
```

汇编程序只有两类指令：比较指令和使用比较结果的跳转指令。所以上面的C++例子大致对应于下面的伪汇编程序。

```
p_not_null = (p =/= nullptr)
    if_not (p_not_null) goto ELSE;
    transform(p);
    goto ENDIF;
ELSE:
    invalid++;
ENDIF:
```

判断原有的C条件 `(p =/= nullptr)` ，如果它是假的，则执行对应于else分支的指令。否则，执行与if分支的主体相对应的指令。

同样的行为可以用稍微不同的方式实现。将原本跳转到 ELSE 的部分和 if 部分进行调换。像下面这样：

```
p_not_null = (p =/= nullptr)
    if (p_not_null) goto IF:
    invalid++;
    goto ENDIF;
IF:
    transform(p);
ENDIF:
```

大多数时候，编译器将为原始的 C++ 代码生成如同第一个代码的汇编，但开发者可以使用 GCC 内置程序来影响这一点。我们将在后面讨论如何告诉编译器要生成什么样的代码。

你也许会有疑问，为什么需要做上述的操作？在一些 CPU 上跳转的代价比不跳转的代价昂贵。在这些场景下，告诉编译器如何构建代码可以带来更好的性能。

分支和向量化

分支影响你的代码性能的方式比你能想到的要多。我们先来谈谈矢量化的问题（你可以在这里找到更多关于矢量化和分支的信息）。大多数现代 CPU 都有特殊的向量指令，可以处理同一类型的多个数据（AVX）。例如，有一条

指令可以从内存中加载 4 个整数，另一条指令可以做4个加法，还有一条指令可以将 4 个结果存回内存。

矢量代码可以比其标量代码快几倍。编译器知道这一点，通常可以在一个称为自动矢量化的过程中自动生成矢量指令。但自动矢量化有一个限制，这个限制是由于分支结构的存在。考虑一下下面的代码：

```
for (int i = 0; i < n; i++) {
    if (a[i] > 0) {
        a[i]++;
    } else {
        a[i]--;
    }
}
```

这个循环对于编译器来说很难矢量化，因为处理的类型取决于变量：如果值 a[i] 是正的，我们做加法；否则，我们做减法。不存在指令对正数做加法，对负数做减法。处理的类型根据数据值的不同而不同，这种代码很难被矢量化。

一句话：若在编译器支持向量化的情况下，循环内部的分支使编译器的自动矢量化难以实现或完全无法实现。而在循环内的不进行分支可以带来很大的速度改进。

一些优化程序的技巧

在谈论技术之前，让我们先定义两件事。当我们说*条件概率*时，实际上我们的意思是条件为真的几率是多少。 有的条件大部分是真的，有的条件大部分是假的。还有一些条件，其真或假的机会是相等的。

具有分支预测功能的 CPU 很快就能弄清哪些条件大多是真的，哪些是假的，你不应该指望在这方面有任何性能退步。然而，当涉及到难以预测的条件时，分支预测器预测正确的概率为50%。这部分是隐藏的潜在的优化空间。

另外一件事，我们将使用一个术语*计算密集、高代价的条件*。这个术语实际上可以意味着两件事：1）需要大量的指令来计算它，或者2）计算所需的数据不在缓存中，因此一条指令需要很多时间才能完成。第一个对计数指令(PC)可见，第二个则不可见，但也非常重要。如果我们以随机的方式访问内存，数据很可能不在缓存中，这将导致指令流水停滞和性能降低。

现在转到编程技巧。这里有几个技巧，通过重写程序的关键部分，使你的程序运行得更快。 但是请注意，这些技巧也可能使你的程序运行速度变慢，这将取决于：1）你的CPU是否支持分支预测。2）你的CPU是否必须等待来自内存的数据。因此，请做基准测试！

加入条件--高代价和低代价的条件

加入条件是 (cond1 && cond2) 或 (cond1 || cond2) 类型的条件。根据 C 和 C++ 标准，在 (cond1 && cond2) 的情况下，如果 cond1 是假的，cond2 将不会被评估。同样地，在 (cond1 || cond2) 的情况下，如果 cond1 为真，cond2 将不会被评估（短路断路）。

因此，如果你有两个条件，其中一个条件比较简单，另一个条件比较复杂，那么就把简单的条件放在前面，复杂的条件放在后面。这将确保复杂的条件不会被无谓地评估。

优化 if/else 指令链

如果你在代码的关键部分有一连串的 if/else 命令，你将需要查看条件概率和条件计算强度，以便优化该链。比如说：

```
if (a > 0) {
    do_something();
} else if (a == 0) {
   do_something_else();
} else {
    do_something_yet_else();
}
```

现在想象一下，（a < 0 ）的概率为 70%，（a > 0）为 20%，（a == 0）为 10%。在这种情况下，最合理的做法是将上述代码重新编排成这样。

```
if (a < 0) {
    do_something_yet_else();
} else if (a > 0) {
   do_something();
} else {
    do_something_else();
}
```

使用查询表来代替switch

当涉及到删除分支时，查询表（LUT）会很方便。不幸的是，在 switch 语句中，大多数时候分支是很容易预测的，所以这种优化可能会变成没有任何效果。尽管如此，这里还是需要提一下：

```
switch(day) {
    case MONDAY: return "Monday";
    case TUESDAY: return "Tuesday";
   ...
    case SUNDAY: return "Sunday";
    default: return "";
};
```

上述语句可以用LUT来实现：

```
if (day < MONDAY || day > SUNDAY) return "";
char* days_to_string = { "Monday", "Tuesday", ... , "Sunday" };
return days_to_string[day - MONDAY];
```

通常情况下，编译器可以为你做这项工作，通过用查找表代替 switch。然而，不能保证这种情况会发生，你需要看一下编译器的向量化报告。

还有一个叫做计算标签的 GNU 语言扩展，允许你使用存储在数组中的标签来实现查找表。它对实现解析器非常有用。例如下面代码所示：

```
static const void* days[] = { &&monday, &&tuesday, ..., &&sunday };
goto days[day];
monday:
    return "Monday";
tuesday:
    return "Tuesday";
...
sunday:
    return "Sunday";
```

将最常见的情况从 switch 中移出

如果你正在使用 switch 命令，而且有一种情况似乎是最常见的，你可以把它从 switch 中移出来，给它一个特殊的处理。继续上一节的例子：

```
day get_first_workday() {
     std::chrono::weekday first_workday = read_first_workday();
    if (first_workday == Monday) { return day::Monday; }
    switch(first_workday) {
        case Tuesday: return day::Tueasday;
        ....
    };
}
```

重写加入条件

如前所述，在连接条件的情况下，如果第一个条件有一个特定的值，第二个条件根本不需要被评估。编译器是如何做到这一点的呢？以下面这个函数为例：

```
if (a[i] > x && a[i] < y) {
    do_something();
}
```

现在假设 a[i]>x 和 a[i]<y ，判断起来很简单（所有数据都在寄存器或缓存中），但很难预测。这个代码将转化为以下伪汇编程序。

```
if_not (a[i] > x) goto ENDIF;
if_not (a[i] < y) goto ENDIF;
do_something;
ENDIF
```

你在这里得到的是两个难以预测的分支。如果我们用 & 而不是 && 连接两个条件，我们会：

1. 强制一次评估两个条件：&操作符是算术和操作，它必须评估两边。
2. 让条件更容易预测，从而降低分支错误预测率：两个完全独立的条件，概率为50%，若是一个联合条件，其真实概率为25%。
3. 两个分支合二为一变成一个分支。

操作符 & 评估两个条件，在生成的汇编中，只有一个分支，而不是两个。同样的情况也适用于运算符 || 和其孪生运算符 | 。

请注意：根据C++标准，bool类型的值为0表示假，任何其他值表示真。C++ 标准保证逻辑运算和算术比较的结果永远是 0 或 1，但不能保证所有的 bool 类型的变量都只有这两个值。 你可以通过对其应用!!操作符来规范化 bool 变量。

告诉编译器哪个分支有更高的概率

GCC和CLANG提供了一些关键字，程序员可以用这些关键字来告诉他们哪些分支的概率更高。例如：

```
#define likely(x)      __builtin_expect(!!(x), 1)
#define unlikely(x)    __builtin_expect(!!(x), 0)
if (likely(ptr)) {
    ptr→do_something();
}
```

通常我们通过宏 likely 和 unlikely 来使用 __builtin_expect，因为它们的语法很麻烦，不方便使用。当这样注释时，编译器将重新安排 if 和 else 分支中的指令，以便最优化地使用底层硬件。请确保条件概率是正确的，否则就会出现性能下降。

使用无分支算法

一些算法可以通过一些技巧转化为无分支的算法。例如，下面的一个函数 abs 使用一个技巧来计算一个数字的绝对值。你能猜到是什么技巧嘛?

```
int abs(int a) {
  int const mask =
        a >> sizeof(int) * CHAR_BIT - 1;
    return  = (a + mask) ^ mask;
}
```

有一大堆无分支的算法，这个列表在网站 Bit Twiddling Hacks。

使用条件加载（conditional loads）而不是分支

许多 CPU 都支持有条件移动指令（conditional move），可以用来删除分支。下面是一个例子：

```
if (x > y) {
    x++;
}
```

可以改写为

```
int new_x = x + 1;
x = (x > y) ? new_x : x; // the compiler should recognize this and emit a conditional branch
```

编译器会将第 2 行的命令，写成对变量 x 的条件性加载，并发出条件性移动指令。不幸的是，编译器对何时发出条件分支有自己的内部逻辑，而这并不总是如开发者所期望的。你可以通过内联汇编的方式来强制条件加载（后面会有介绍）。

但是要注意，无分支版本做了更多的操作。无论 x 是否大于 y ，x 都会执行加一操作。 加法是一个代价很低的操作，但对于其他代价高的操作（如除法），这种优化可能造成性能下降。

用算术运算来实现无分支

有一种方法可以通过巧妙地使用算术运算来实现无分支。例子：

```
// 使用分支
if (a > b) {
    x += y;
}
// 不使用分支
x += -(a > b) & y;
```

在上面的例子中，表达式-(a > b) 将创建一个掩码，若条件不成立的时候，掩码为 0，当条件成立的时候掩码为 1。

条件性赋值的一个例子：

```
// 使用分支
x = (a > b) ? val_a : val_b;
// 不使用分支
x = val_a;
x += -(a > b) & (val_b - val_a);
```

上述所有的例子都使用算术运算来避免分支。当然，根据你的 CPU 的分支预测错误惩罚和数据缓存命中率，这也可能不会带来性能提升。

一个在循环队列中移动索引的例子：

```
// 带分支
int get_next_element(int current, int buffer_len) {
    int next = current + 1;
    if (next == buffer_len) {
        return 0;
    }
    return next;
}
// 不带分支
int get_next_element_branchless(int current, int buffer_len) {
    int next = current + 1;
    return (next < buffer_len) * next;
}
```

重新组织你的代码，以避免分支的出现

如果你正在编写需要高性能的软件，你肯定应该看一下 面向数据的设计原则。下面将简单叙述一个技巧。

假设你有一个叫做animation的类，它可以是可见的或隐藏的。处理一个可见的animation与处理一个隐藏的animation是完全不同的。有一个包含animation的列表叫animation_list，你的处理方式看起来像这样：

```
for (const animation& a: animation_list) {
   a.step_a();
   if (a.is_visible()) {
      a.step_av();
   }
```

```
   a.step_b();
   if (a.is_visible) {
       a.step_bv();
}
```

分支预测器真的很难处理上述代码，除非animation是按照可见度排序的。有两种方法来解决这个问题。一个是根据is_visible()对animation_list中的动画进行排序。第二种方法是创建两个列表，animation_list_visible和animation_list_hidden，然后重写代码如下：

```
for (const animation& a: animation_list_visible) {
   a.step_a();
   a.step_av();
   a.step_b();
   a.step_bv();
}
for (const animation& a: animation_list_hidden) {
   a.step_a();
   a.step_b();
}
```

所有的条件分支都消失了。

使用模板来删除分支

如果一个布尔值被传递给函数，并且在函数内部作为参数使用，你可以通过把它作为模板参数传递来删除这个布尔值。例如：

```
int average(int* array, int len, bool include_negatives) {
    int average = 0;
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (include_negatives) {
            average += array[i];
        } else {
            if (array[i] > 0) {
                average += array[i];
                count++;
            }
        }
    }
    if (include_negatives) {
         return average / len;
    } else {
        return average / count;
    }
}
```

在这个函数中， include_negatives 的这个条件被多次判断。要删除判断，可以将参数作为模板参数而不是函数参数传递。

```
template <bool include_negatives>
int average(int* array, int len) {
    int average = 0;
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (include_negatives) {
            average += array[i];
        } else {
            if (array[i] > 0) {
                average += array[i];
                count++;
            }
        }
    }
    if (include_negatives) {
         return average / len;
```

```
    } else {
        return average / count;
    }
}
```

通过这种实现方式，编译器将生成两个版本的函数，一个包含 include_negatives，另一个不包含（以防对该参数有不同值的函数的调用）。分支完全消失了，而未使用的分支中的代码也不见了。

但是你调用函数的方法有一些区别，如下所示：

```
int avg;
bool should_include_negatives = get_should_include_negatives();
if (should_include_negatives) {
    avg = average<true>(array, len);
} else {
    avg = average<false>(array, len);
}
```

这实际上是一种叫做分支优化的编译器优化。 如果在编译时知道 include_negatives 的值，并且编译器决定内联函数，它将删除分支和未使用的代码。我们用模板的版本保证了这一点，而未使用模板的原始版本则不一定能做到这点。

编译器通常可以为你做这种优化。如果编译器能够保证 include_negatives 这个值在循环执行过程中不会改变它的值，它可以创建两个版本的循环：一个是它的值为真的循环，另一个是它的值为假的循环。这种优化被称为 *loop invariant code motion* ，你可以在我们关于 循环优化 的帖子中了解更多信息。使用模板可以保证这种优化发生。

其他一些避免分支的技巧

如果你在代码中多次检查一个条件，你可以通过检查一次该条件，然后多做一些代码复制来达到更好的性能。例如：

```
if (is_visible) {
    hide();
}
process();
if (is_active) {
    display();
}
```

可替换为：

```
if (is_visible) {
    hide();
    process();
    display();
} else {
    process();
}
```

我们也可以引入一个两个元素数组，一个用来保存条件为真时的结果，另一个用来保存条件为假时的结果。例如：

```
int larger = 0;
for (int i = 0; i < n; i++) {
    if (a[i] > m) {
        larger++;
    }
}
return larger;
```

可以被替换为：

```
int result[] = { 0, 0 };
for (int i = 0; i < n; i++) {
    result[a>i]++;
```

```
}
return result[1];
```

实验

现在让我们来看看最有趣的部分：实验。我们决定做两个实验，一个是与遍历一个数组并计算具有某些属性的元素有关。这是一种缓存友好的算法，因为硬件预取器可以很好的预取数据。

第二种算法是我们在关于[[缓存友好程序设计指南 id=22260e6c-fd75-4db0-9f05-98dc201b30fb]]文章中介绍的经典的二分查找算法。由于二分查找的性质，这种算法对缓冲区完全不友好，大部分的速度上的瓶颈来自于对数据的等待。

为了测试，我们使用了三种不同架构的芯片：

- **AMD A8-4500M quad-core x86-64** 处理器，每个单独的内核有16 kB的L1 数据缓存，一对内核共享 2M 的 L2 缓存。这是一个现代指令流水处理器，具有分支预测、推测执行和失序执行功能。根据技术规格，该 CPU 的错误预测惩罚（misprediction penalty）约为 20 个周期。
- **Allwinner sun7i A20 dual-core ARMv7** 处理器，每个核心有 32kB 的 L1 数据缓存和 256kB 的 L2 共享缓存。 这是一个廉价的处理器，旨在为嵌入式设备提供分支预测和推测执行，但没有失序执行。
- **Ingenic JZ4780 dual-core MIPS32r2** 处理器，每个内核有 32kB 的 L1 数据缓存和 512kB 的 L2 共享数据缓存。这是一个用于嵌入式设备带指令流水的处理器，有一个简单的分支预测器。根据技术规范，分支预测错误的惩罚约为3个周期。

计算实例

为了证明代码中分支的影响，我们写了一个非常小的算法，计算一个数组中大于给定给定元素的数量。代码可在我们的Github仓库中找到，只需在 2020-07-branches 目录中输入 make counting 。

这里是最重要的函数：

```
int count_bigger_than_limit_regular(int* array, int n, int limit) {
    int limit_cnt = 0;
    for (int i = 0; i < n; i++) {
        if (array[i] > limit) {
            limit_cnt++;
        }
    }
    return limit_cnt;
}
```

如果让你来写这个算法，你可能会想出上面的办法。

为了能够进行恰当的测试，我们用优化级别 -O0 编译了所有的函数。在所有其他的优化级别中，编译器会用算术来代替分支，并做一些繁重的循环处理，并掩盖了我们想要看到的东西。

分支错误预测的代价

让我们首先测试一下分支错误预测给我们带来了多少损失。 我们刚才提到的算法是计算数组中所有大于 limit 的元素。因此，根据数组的值和 limit 的值，我们可以在 if (array[i] > limit) { limit_cnt++ }中调整 (array[i] > limit) 为真的概率。

我们生成的输入数组的元素在 0 和数组的长度（arr_len）之间均匀分布。然后，为了测试错误预测的代价，我们将 limit 的值设置为 0（条件永远为真），arr_len / 2（条件在50%的时间内为真，难以预测）和 arr_len（条件永远为假）。下面是测量结果：

条件永远为true	条件随机	条件永远为false	
Runtime (ms)	5533	14176	5478
Instructions	14G	13.5G	13G
Instructions per cycle	1.36	0.50	1.27
Branch misspredictions (%)	0%	32.96%	0%

上表的数据为：数组长度=1M，在AMD A8-4500M上查找1000个。

在 x86-64 上，不可预测条件的代码版本的速度要慢三倍。发生这种情况是因为每次分支被错误预测时，指令流水都要被刷新。

下面是ARM和MIPS芯片的运行时间：

条件永远为true	条件随机	条件永远为false	
ARM	30.59s	32.23s	25.89s
MIPS	37.35s	35.59s	31.55s

上表为在 MIPS 和 ARM 芯片上的运行时间，数组长度为 1M，查找量为 1000。

根据我们的测量，MIPS 芯片没有错误预测的惩罚（和规格上的描写不同）。在 ARM 芯片上有一个小的惩罚，但肯定不会像 x86-64 芯片那样急剧。

我们能解决这个问题吗？向下阅读。

使用无分支方法

现在让我们根据我们之前给你的建议重写条件。下面是三个重写了条件的实现：

```
int count_bigger_than_limit_branchless(int* array, int n, int limit) {
    int limit_cnt[] = { 0, 0 };
    for (int i = 0; i < n; i++) {
        limit_cnt[array[i] > limit]++;
    }
    return limit_cnt[1];
}
int count_bigger_than_limit_arithmetic(int* array, int n, int limit) {
    int limit_cnt = 0;
    for (int i = 0; i < n; i++) {
        limit_cnt += (array[i] > limit);
    }
    return limit_cnt;
}
int count_bigger_than_limit_cmove(int* array, int n, int limit) {
    int limit_cnt = 0;
    int new_limit_cnt;
    for (int i = 0; i < n; i++) {
        new_limit_cnt = limit_cnt + 1;
        // The following line is pseudo C++, originally it is written in inline assembly
        limit_cnt = conditional_load_if(array[i] > limit, new_limit_cnt);
    }
    return limit_cnt;
}
```

我们的代码有三个版本：

- count_bigger_than_limit_branchless在(如上文*其他一些避免分支的技巧*) 内部使用一个小的两元素数组来计算当数组中的元素大于和小于 limit 时的情况。
- count_bigger_than_limit_arithmetic利用表达式 (array[i] > limit) 只能有 0 或 1 的值这一事实，用表达式的值来增加计数器。
- count_bigger_than_limit_cmove计算新值，然后使用*条件移动*来加载它，如果条件为真。我们使用内联汇编来确保编译器会发出 cmov 指令。

请注意所有版本的一个共同点。 在分支内部，有一项我们必须做的工作。当我们删除分支时，我们仍然在做这项工作，但这次即使我们不需要这项工作的情况下仍然去做这项工作。这使得我们的CPU执行更多的指令，但我们希望通过减少分支错误预测和提高每周期的指令比率来获得性能提升。

在x86-64架构上测试无分支代码

我们的三种不同的策略是如何在性能上显示出避免分支的？以下是可预测条件下的结果。

Regular	Branchless	Arithmetic	Conditional Move	
Runtime (ms)	5502	7492	6100	9845
Instructions executed	14G	19G	15G	19G
Instructions per cycle	1.37	1.37	1.33	1.04

上表是数组长度=1M，在AMD A8-4500M上查找1000个可预测的分支的结果。

正如你所看到的，当分支是可预测的，Regular 的实现是最好的。这种实现方式还具有最小的执行指令数量和最佳的周期指令比。

始终错误条件的运行时间与始终正确条件的运行时间差别不大，这适用于所有四个实现。除常规实现外，所有使用其他方法实现的性能都是一样的。在Regular 实现中，每周期指令数降低，但执行的指令数也降低，没有观察到速度上的差异。

当分支无法预测时，会发生什么？性能看起来会完全不同。

Regular	Branchless	Arithmetic	Conditional Move	
Runtime (ms)	14225	7427	6084	9836
Instructions executed	13.5G	19G	15G	19G
Instructions per cycle	0.5	1.38	1.32	1.04

上表是数组长度=1M，在AMD A8-4500M上查找 1000 个不可预测的分支的结果。

Regular 实现的性能最差。每周期的指令数要差很多，因为由于分支预测错误，指令流水必须被重新刷新。对于其他方法实现的代码，性能和上表几乎没有任何变化。

有一件值得注意的事情。 如果我们用 -O3 编译选项编译这个程序，编译器不会按照 Regular 实现去实现。因为分支错误预测率很低，运行时间和 Arithmetic 实现的运行时间非常接近。

在ARMv7上测试

在 ARM 芯片的情况下，性能看起来又有所不同。由于作者不熟悉 ARM 的汇编程序，所以我们没有显示条件移动 (Conditional Move) 实现的结果。

Condition predictability	Regular	Arithmetic	Branchless
总是true	3.059s	3.385s	4.359s
无法预测	3.223s	3.371s	4.360s
总是false	2.589s	3.370s	4.360s

这里，Regular 版本是最快的。Arithmetic 版和 Branchless 版并没有带来任何速度上的提高，它们实际上更慢。

请注意，具有不可预测条件的版本的性能最差的。这表明该芯片有某种分支预测功能。然而，错误预测的代价很低，否则我们会看到在这种情况下，其他的实现方式会更快。

在MIPS32r2上测试

下面是MIPS的结果：

Condition predictability	Regular	Arithmetic	Branchless	Cmov
总是true	37.352s	37.333s	41.987s	39.686s
无法预测	35.590s	37.353s	42.033s	39.731s
总是false	31.551s	37.396s	42.055s	39.763s

从这些数字来看，MIPS 芯片似乎没有任何分支错误预测，因为运行时间完全取决于常规执行的指令数量（与技术规范相反）。对于 Regular 执行来说，条件为真的次数越少，程序就越快。

另外，分支代价似乎是相对较低的，因为在条件总是真的情况下，Arithmetic 实现和普通实现有相同的性能。其他的实现方式会慢一些，但不会太多。

用 likely 的和 unlikely 的来注释分支

我们想测试的下一件事是，用 "likely"和 "unlikely"注释分支是否对分支的性能有任何影响。我们使用了与之前相同的函数，但我们对临界条件做了这样的注释，`if (likely(a[i] > limit) limit_cnt++`。我们使用优化级别 O3 来编译这些函数，因为在非生产优化级别上测试注释的行为没有意义。

使用 GCC 7.5 的 AMD A8-4500M 给出了一些意外的结果。下面是这些结果。

条件预测	Likely	Unlikely	不声明
总是true	904ms	1045ms	902ms
总是false	906ms	1050ms	903ms

在条件被标记为可能的情况下，总是比条件被标记为不可能的情况快。仔细想想这并不完全出乎意料，因为这个 CPU 有一个好的分支预测器。Unlikely的版本只是引入了额外的指令，没有必要。

在我们使用 GCC 6.3 的 ARMv7 芯片上，如果我们使用 likely 或 Unlikely 的分支注解，则完全没有性能差异。编译器确实为两种实现方式生成了不同的代码，但两种方式的周期数和指令数大致相同。我们的猜测是，如果不采取分支，这个CPU不会性能提升，这就是为什么我们看到性能既没有增加也没有减少的原因。

在我们的 MIPS 芯片和 GCC 4.9 上也没有性能差异。GCC 为 likely 和 Unlikely 的函数版本生成了相同的汇编。

结论：就 likely 和 Unlikely 的宏而言，我们的调查表明，它们在有分支预测器的处理器上没有任何帮助。不幸的是，我们没有一个没有分支预测器的处理器来测试那里的行为。

联合条件

为了测试 if 子句中的联合条件，我们这样修改我们的代码。

```
int count_bigger_than_limit_joint_simple(int* array, int n, int limit) {
    int limit_cnt = 0;
    for (int i = 0; i < n/2; i+=2) {
        // The two conditions in this if can be joined with & or &&
        if (array[i] > limit && array[i + 1] > limit) {
            limit_cnt++;
        }
    }
    return limit_cnt;
}
```

基本上，这是一个非常简单的修改，两个条件都很难预测。唯一不同的就是第四行代码`if (array[i] > limit && array[i + 1] > limit)` 。 我们想测试一下，使用操作符 && 和操作符 & 来连接条件是否有区别。我们称第一个版本为 *simple*，第二个版本为 *arithmetic*。

我们用 -O0 编译上述函数，因为当我们用 -O3 编译它们时，算术版本在 x86-64 上非常快，而且没有分支错误预测。这表明编译器已经完全优化掉了这个分支。

下面是所有三种架构的结果，以防这两种条件都难以用来优化预测：

Joint simple	Joint arithmetic	
x86-64	5.18s	3.37s
ARM	12.756s	15.317s
MIPS	13.221s	15.337s

上述结果表明，对于具有分支预测器和高错误预测惩罚的 CPU 来说，使用 & 要快得多。但是，对于错误预测惩罚较低的 CPU 来说，使用 && 的速度更快，仅仅是因为它执行的指令更少。

二分查找

为了进一步测试分支的行为，我们采用了我们在关于缓存友好程序设计指南文章中用来测试缓冲区预取的二进制查找算法。源代码在github 仓库里， 只要在 2020-07-branches 目录下输入 make binary_search 即可运行。

这里是实现二分查找的核心代码：

```
int binary_search(int* array, int number_of_elements, int key) {
    int low = 0, high = number_of_elements-1, mid;
    while(low ≤ high) {
        mid = (low + high)/2;
        if (array[mid] == key) {
            return mid;
        }
        if(array[mid] < key) {
            low = mid + 1;
        } else {
            high = mid-1;
        }
    }
```

```
    return -1;
}
```

上述算法是一种经典的二进制查找算法。我们将其称为 *regular* 实现。注意： 在第8-12行有一个重要的if/else条件，决定了查找的流程。由于二进制查找算法的性质，Array[mid]< key的条件很难预测。另外，对数组 [mid] 访问的代价很高，因为这些数据通常不在数据缓存中。我们用两种方法消除了这个分支，使用条件移动和使用算术运算。下面是这两个版本。

```
// Conditional move implementation
int new_low = low + 1;
int new_high = high - 1;
bool condition = array[mid] > key;
// The bellow two lines are pseudo C++, the actual code is written in assembler
low = conditional_move_if(new_low, condition);
high = conditional_move_if_not(new_high, condition);
// Arithmetic implementation
int new_low = mid + 1;
int new_high = mid - 1;
int condition = array[mid] < key;
int condition_true_mask = -condition;
int condition_false_mask = -(1 - condition);
low += condition_true_mask & (new_low - low);
high += condition_false_mask & (new_high - high);
```

条件移动的实现使用 CPU 提供的指令来有条件地加载准备好的值。

算术实现使用巧妙的条件操作来生成 condition_true_mask 和 condition_false_mask 。根据这些掩码的值，它将向变量 low 和 high 加载适当的值。

x86-64 上的二进制查找算法

下面是x86-64 CPU的性能比较，在工作集很大，不适合缓存的情况下。我们测试了使用 __builtin_prefetch 的显式数据预取和不使用的算法版本。

Regular	Arithmetic		
无数据预取	2.919s	3.623s	3.243s
有数据预取	2.667s	2.748s	2.609s

上面的表格显示了一些非常有趣的东西。我们的二进制查找中的分支不能被很好地预测，当没有数据预取时，我们的常规算法表现得最好。为什么？ 因为分支预测、推测性执行和失序执行使 CPU 在等待数据从内存到达时有事情可做。为了不占用这里的文字，我们将在以后再谈。

下面是工作集完全适合 L1 缓存时同一算法的结果：

Regular	Arithmetic	Conditional move	
无数据预取	0.744s	0.681s	0.553s
有数据预取	0.825s	0.704s	0.618s

与之前的实验相比，这些数字是不同的。当工作集完全适合L1数据缓存时， Conditional move 版本的算法是最快的，差距很大，其次是 Arithmetic 版本的算法。由于许多分支预测错误，regular 版本的表现很差。在工作集较小的情况下，预取并没有帮助。所有的数据都已经在缓存中，预取指令只是执行更多的指令，没有任何额外的好处。

ARM和MIPS上的二进制查找算法

对于 ARM 和 MIPS 芯片，预取算法比非预取算法要慢，所以我们不考虑预取。

下面是 ARM 和 MIPS 芯片在 4M 元素的数组上的二分查找运行时间。

Regular	Arithmetic	Conditional Move	
ARM	10.85s	11.06s	—
MIPS	11.79s	11.80s	11.87s

在 MIPS 上，所有三种类型的数字都大致相同。在 ARM 上，Regular 版本比Arithmetic 版本稍快一些。

下面是 ARM 和 MIPS 芯片在 10k 元素的阵列上的运行时间：

Regular	Arithmetic	Conditional Move	
ARM	1.71s	1.79s	—
MIPS	1.42s	1.48s	1.51s

工作集的大小并不改变性能之间的相对比例。在这些芯片上，与分支有关的优化并不产生速度的提高。

为什么在 x86-64 的大工作集上，带分支的二进制搜索最快?

现在让我们回到这个问题。在 x86-64 芯片中，我们看到，如果工作集很大，regular 版本是最快的。在工作集很小的情况下，Conditional Move 版本是最快的。当我们引入软件预取以提高缓存命中率时，我们看到 regular 版本的优势正在消失。为什么?

失序执行的局限性

为了解释这一点，请记住，我们正在谈论的 CPU 是高端 CPU，具有分支预测、推测执行和失序执行功能。所有这些都意味着，CPU 可以并行地执行几条指令，但它一次可以执行的指令数量是有限的。这一限制是由两个因素造成的：

- 处理器中的资源数量是有限的。例如，一个典型的高端处理器可能同时处理四条简单的算术指令、两条加载指令、两条存储指令或一条复杂的算术指令。当指令执行完毕后（技术术语是指令*retire*），资源变得可用，因此处理器可以处理新指令。
- 指令之间存在着数据依赖性。如果当前指令的输入参数依赖于前一条指令的结果，那么在前一条指令完成之前，当前指令不能被处理。它被卡在处理器中占用资源，阻止其他指令进入。

所有的代码都有数据依赖性，有数据依赖性的代码不一定是坏的。但是，数据依赖性降低了处理器每个周期所能执行的指令数量。

在顺序执行的 CPU 中，如果当前指令依赖于前一条指令，并且前一条指令还没有完成，那么流水线就会被停滞。如果是失序执行的 CPU，处理器将尝试加载被阻止的指令之后的其他指令。 如果这些指令不依赖于前面的指令，它们可以安全地执行。这是让CPU利用闲置资源的方法。

解释带分支的二分查找的性能

那么，这与我们的二进制搜索的性能有什么关系呢？下面将 regular 的核心部分改写为伪汇编程序：

```
element = load(base_address = array, index = mid)
    if_not (element < key) goto ELSE
    low = mid + 1
    goto ENDIF
```

```
ELSE:
    high = mid - 1
ENDIF:
    // These are the instructions at the beginning of the next loop
    mid = low + high
    mid = mid / 2
```

让我们做一些假设：操作 `element = load(base_address = array, index = mid)` 如果 `array[mid]` 不在数据缓存中，需要 300 个周期完成，若在缓存中只需要 3 个周期。分支条件`element < key`将在 50% 的时间会被正确预测（最坏情况下的分支预测）。分支错误预测的代价是 15 个周期。

让我们分析一下我们的代码是如何被执行的。处理器需要等待 300 个周期来执行第1行的 `load` 。由于它有 OOE（乱序执行），它开始在执行第 2 行的分支。第 2 行的分支依赖于第一行的数据，所以 CPU 不能执行它。然而，CPU 进行猜测并开始运行第 3、4、9 和 10 行的指令。若猜测正确，那么整个程序运行只花费 300 个机械周期。若猜测错误，需要额外的加上处理指令 6、9和10的时间，又多增加了15个周期。

解释带有条件移动(conditional move)的二分查找的性能

条件性移动的实现情况如何？下面是伪汇编：

```
element = load(base_address = array, index = mid)
    load_low =  element < key
    new_low = mid + 1
    new_high = mid - 1
    low = move_if(condition = load_low, value = new_low)
    high = move_if_not(condition = load_low, value = new_high)
    // These are the instructions at the beginning of the next loop
    mid = low + high
    mid = mid / 2
```

这里没有分支，因此没有分支错误预测的惩罚。让我们与 `regular` 实现有相同的假设。（操作`element = load(base_address = array, index = mid)`如果`array[mid]`不在数据缓存中，需要300个周期来完成，否则需要3个周期）。

这段代码的执行情况如下：处理器需要等待300个周期来执行第1行的加载。因为 CPU 具有 OOE(乱序执行)，它将会尝试执行第二行，很快 CPU 发现第二行依赖第一行的数据。因此，CPU 会继续向下探索执行第 3 和 4 行。CPU 无法执行第 5 和 6 行因为它们都依赖于第 2 行的数据。第 9 行的指令也无法执行，因为它依赖于第 5 行和第 6 行。第 10 行指令依赖于第 9 行，所以也无法执行。由于这里没有涉及到推测，到达指令 10 需要 300 个周期外加上执行指令2、5、6 和 9 的一些时间。

分支与条件移动的性能比较

现在让我们做一些简单的数学计算。在带有分支预测的二分查找的情况下，运行时间为：

```
MISSPREDICTION_PENALTY = 15 cycles
INSTRUCTIONS_NEEDED_TO_EXECUTE_DUE_MISSPREDICTION = 50 cycles

RUNTIME = (RUNTIME_PREDICTION_CORRECT + RUNTIME_PREDICTION_NOTCORRECT) / 2
RUNTIME_PREDICTION_CORRECT = 300 cycles
RUNTIME_PREDICTION_NOTCORRECT = 300 cycles + MISSPREDICTION_PENALTY + INSTRUCTIONS_NEEDED_TO_EXECUTE_DUE

RUNTIME = 332.5 cycles
```

如果是条件移动的版本，运行时间是：

```
INSTRUCTIONS_BLOCKED_WAITING_FOR_DATA = 50 cycles

RUNTIME = 300 cycles + INSTRUCTIONS_BLOCKED_WAITING_FOR_DATA = 350 cycles
```

正如你所看到的，若从内存中加载数据需要等待 300 个周期的情况下，分支预测（`regular`）版本平均快 17.5 个周期。

最后说一下

目前的处理器不对条件性移动（conditional moves）进行推测，只对分支进行推测。分支猜测使其能够掩盖缓慢的内存访问所带来的一些惩罚。条件性移动（conditional moves）（和其他去除分支的技术）消除了分支错误预测的惩罚，但引入了数据依赖性惩罚。处理器将更经常地被阻塞，并且可以推测地执行更少的指令。在高速缓存失误率较低的情况下，数据依赖性的惩罚比分支错误预测的惩罚要昂贵得多。因此，结论是：分支预测机制打破了一些数据的依赖性，有效地掩盖了 CPU 需要从内存中等待数据的时间。如果分支预测器的猜测是正确的，那么当数据从存储器到达时，很多工作已经完成。对于无分支的代码来说，情况并非如此。

总结

当我第一次开始写这篇文章时，我以为是一篇简单明了的文章，结论很短。孩子，我错了 让我们从感恩开始。

首先为编译器开发者喝彩。这个经验告诉我，编译器是使分支快速化的大师。他们知道每条指令的时间，他们可以使一般的分支具有良好性能。

第二个赞誉要归功于现代处理器的硬件设计师。在分支预测正确的情况下，硬件设计使分支成为代价相当低的指令之一。大多数时候，分支预测工作良好，这使得我们的程序运行顺畅。程序员可以专注于更重要的事情。

而第三个赞美之词又是给现代处理器的硬件设计师的。为什么？因为失序执行（OOE）。我们在二分查找例子中的实验表明，即使在分支错误预测率很高的情况下，等待数据然后执行分支比预测性地执行分支然后在错误预测的情况下刷新指令流水更昂贵。

关于分支优化的一般说明

我们在这里提出了一些建议，这些建议有些是通用的，每次都能在每个硬件上发挥作用，比如*优化if/else命令链*，或者*重新组织你的代码，以避免分支*。然而，这里介绍的其他技术比较有限，只能在某些条件下推荐使用。

要优化你的分支，你首先需要了解的是，编译器在优化它们方面做得很好。因此，我的建议是，这些优化在大多数时候是不值得的。让你的代码简单易懂，编译器会尽最大努力生成最好的代码，无论是现在还是将来。

第二件事也很重要：在优化分支之前，你需要确保你的程序以最佳方式使用数据缓存。在许多高速缓存的情况下，分支实际上是CPU性能的捍卫者。移除它们，你会得到不好的结果。首先改善数据缓存的使用，然后再处理分支。

你唯一需要关注的是你代码中的某一个或者两个关键代码，这一两个方法将会运行在特定的计算机上。我们的经验显示，在有些地方，从分支代码切换到无分支代码会带来更多的性能，但具体数字取决于你的CPU，数据缓存利用率，以及可能还有其他因素。 因此需要进行仔细的测试。

我还建议你使用内联汇编代码的形式编写分支的关键部分，因为这将保证你编写的代码不会被编译器的优化所破坏。当然，关键是你要测试你的代码的性能回归，因为这些似乎都是脆弱的优化。

分支的未来

我测试了两个便宜的处理器，它们都有分支预测器。如今，很难找到一款不带分支预测器的处理器。在未来，我们应该期待更复杂的处理器设计，即使在低端 CPU 中也是如此。随着越来越多的CPU采用失序执行，分支错误预测的惩罚将变得越来越高。 对于一个有性能意识的开发者来说，关注好分支将变得越来越重要。

扩展阅读

Agner’s Optimizing Software in C++: chapter 7.5 Booleans, chapter 7.12 Branches and switch statements

CPW: Avoiding Branches

Power and Performance: Software Analysis and Optimization by Jim Kukunas, Chapter 13: Branching

一些测试和感悟

什么时候采用分支优化，文章中并没有过多的笔墨。经过我的一些测试，发现了两个适用条件：

- 当前分支的branch-misses > 5%，这个指标是可以通过 perf 工具给出，当 branch-misses 较大时，采用文章中的方法，有可能会带来比较好的效果。
- 当前条件十分复杂，看看有没有优化的空间。