

# 19 | 极致优化（下）：如何实现高性能的 C 程序？

2022-01-26 于航

《深入C语言和程序运行原理》

课程介绍 >



讲述：于航

时长 10:53 大小 9.98M



你好，我是于航。

在上一讲中，我介绍了几个用于编写高性能 C 代码的实用技巧。今天，我们继续聊这个话题，来讨论其他几种常见的 C 代码和程序优化技巧，它们分别是利用循环展开、使用条件传送指令、尾递归调用优化，以及为编译器指定更高的编译优化等级。

## 技巧五：循环展开（Loop Unrolling）

为了让你更好地理解“循环展开”这个优化技巧背后的原理，我们先从宏观角度看看 CPU 是如何运作的。

早期的 CPU 在执行指令时，是以串行的方式进行的，也就是说，一个指令的执行开始，需要等待前一个指令的执行完全结束。这种方式在实现上很简单，但存在的问题也十分明显：由于

领资料



指令的执行是一个涉及多个功能单元的复杂过程，而在某一时刻，CPU 也只能够对指令进行针对当前所在阶段的特定处理。

那么，将 CPU 处理指令的流程划分为不同阶段，并让它对多条指令同时进行多种不同处理，这样是否可以进一步提升 CPU 的吞吐量呢？事实正是如此。

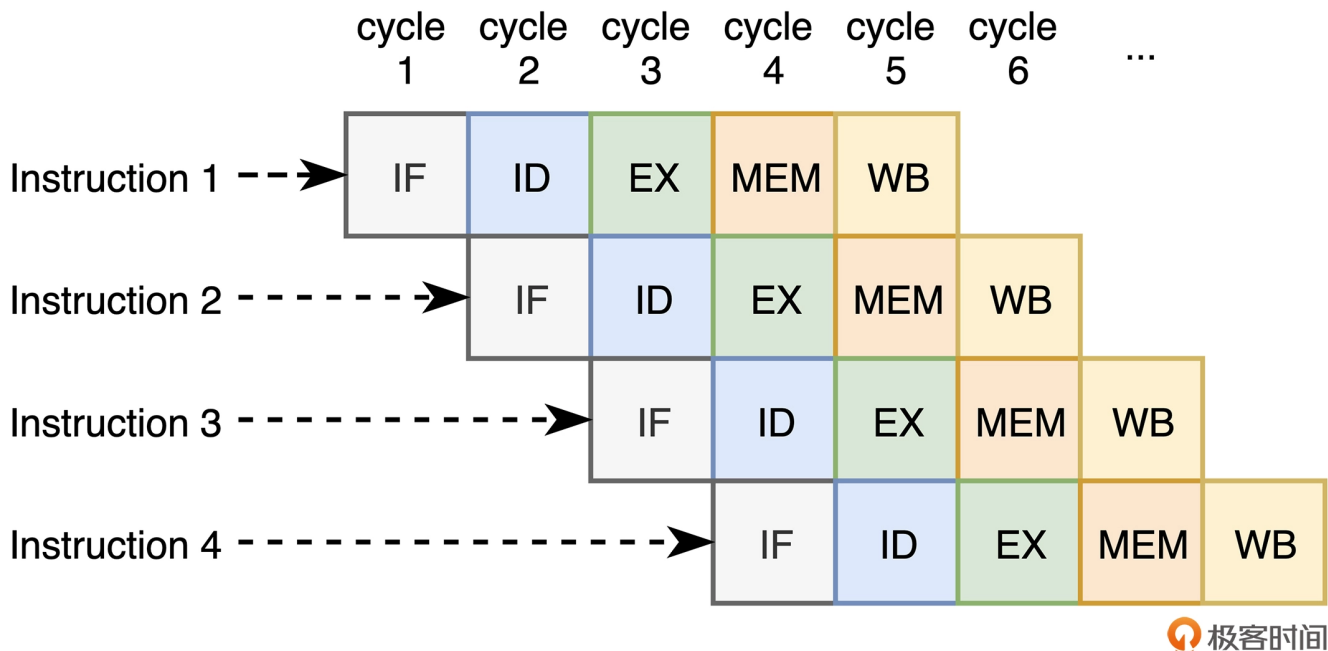
现代 CPU 为了进一步提升指令的执行效率，通常会将单一的机器指令再进行拆分，以达到指令级并行的目的。比如，对于一个基本的五级 RISC 流水线来说，CPU 会将指令的执行细分为指令提取（IF）、指令译码（ID）、指令执行（EX）、内存访问（MEM），以及寄存器写回（WB）共五个步骤。

在这种情况下，当第一条机器指令经过了指令提取阶段的处理后，即使该条指令还没有被完全执行完毕，CPU 也可以立即开始处理下一条机器指令。因此，**从宏观上来看，机器指令的执行由串行变为了并行，程序的执行效率得到了提升。**

其中，指令提取是指从内存中读取出机器指令字节的过程。CPU 根据得到的指令字节，在译码阶段，从相应的寄存器中获得指令执行所需要的参数。而在执行阶段，ALU 可以选择执行指令明确的操作，或者是计算相关内存引用的有效地址等操作。随后，在访存阶段，根据指令要求，CPU 可以将数据写回内存，或从内存中读出所需数据。类似地，在写回阶段，CPU 可以将指令执行得到的结果存入寄存器。

而当五个阶段全部执行完毕后，CPU 会更新指令指针（PC），将其指向下一个需要执行的指令。你可以通过下图来直观地理解这个过程：





那么，如何将 CPU 的吞吐量最大化呢？相信你心中已经有了答案。我们需要做的，就是让 CPU 在执行程序指令时，能够以满流水线的方式进行。

但现实情况并非总是这样理想。这里，我要介绍的代码优化技巧“循环展开”便与此有关。让我们先来看一段代码：

```

1 #define LEN 4096
2 int data[LEN] = { ... };
3 int foo(void) {
4     int acc = 1;
5     for (int i = 0; i < LEN; ++i) {
6         acc = acc * data[i];
7     }
8     return acc;
9 }

```

复制代码

在这段代码中，我们定义了一个名为 `data` 的全局整型数组，并在其中存放了若干个值。而函数 `foo` 则主要用来计算该数组中所有数字的乘积之和。

领资料

此时，如果我们在 `main` 函数中调用函数 `foo`，CPU 在实际执行它的代码时，`for` 循环的每一轮都会产生两个数据相关：循环控制变量 `i` 的下一个值依赖于本次循环变量 `i` 在经过自增运算后得到的结果值。同样地，计数变量 `acc` 的下一个值也依赖于该变量在当前循环中经过乘积计算后的结果值。

而这两个数据相关会导致 CPU 无法提前计算下一轮循环中各个参与变量的值。而只有在寄存器写回，或内存访问阶段执行完毕，也就是变量 `acc` 和 `i` 的值被最终更新后，CPU 才会继续执行下一轮循环。

那么，应该如何优化这个过程呢？我们直接来看优化后的代码：

 复制代码

```
1 #define LEN 4096
2 int data[LEN] = { ... };
3 int foo(void) {
4     int limit = LEN - 1;
5     int i;
6     int acc0 = 1;
7     int acc1 = 1;
8     for (i = 0; i < limit; i += 2) { // 2x2 loop unrolling.
9         acc0 = acc0 * data[i];
10        acc1 = acc1 * data[i + 1];
11    }
12    for (; i < LEN; ++i) { // Finish any remaining elements.
13        acc0 = acc0 * data[i];
14    }
15    return acc0 * acc1;
16 }
```

可以直观地看到，参与到程序执行的局部变量变多了。而这里的主要改动是，我们为函数 `foo` 中的循环结构应用了 **2x2 循环展开**。

循环展开这种代码优化技术，主要通过增加循环结构每次迭代时计算的元素个数，来减少循环次数，同时优化 CPU 的指令集并行与流水线调度。而所谓的 **2x2**，是指在优化后的代码中，循环的步长变为了 2，且循环累积值被分别存放在两个独立变量 `acc0` 与 `acc1` 中。

循环展开带来的最显著优化，就是减少了循环的迭代次数。使用多个独立变量存储累积值，各个累积值之间就不会存在数据相关，而这就增大了 CPU 多个执行单元可以并行执行这些指令的机会，从而在一定程度上提升了程序的执行效率。

 领资料

需要注意的是，循环展开一方面可以带来性能上的提升，另一方面它也会导致程序代码量的增加，以及代码可读性的降低。并且，编译器在高优化等级下，通常也会对代码采用隐式的循环展开优化。因此，在大多数情况下，我们并不需要手动地改变代码形式来为它应用循环展开，

除非是在那些你确定编译器没有进行优化，并且手动循环展开可以带来显著性能提升的情况下。

## 技巧六：优先使用条件传送指令

通常来说，CPU 指令集中存在着一类指令，它们可以根据 CPU 标志位的不同状态，有条件地传送数据到某个特定位置，这类指令被称为“**条件传送指令**”。举个例子，指令 `cmove` 接收两个参数 `S` 和 `R`，当 CPU 标志寄存器中的 `ZF` 置位时，该指令会将 `S` 中的源值复制到 `R` 中。

与条件传送指令类似的还有另外一类指令，它们被称为“**条件分支指令**”。顾名思义，这类指令在执行时，会根据 CPU 标志位的不同状态，选择执行程序不同部分的代码。比如指令 `jz`，该指令接收一个参数 `L`，当 CPU 标志寄存器中的 `ZF` 置位时，该指令会将下一条待执行指令修改为 `L` 所在内存位置上的指令。

对于 C 代码中的某些逻辑，使用条件传送指令与条件分支指令都能够正确完成任务。但在程序的执行效率上，这两种方式却可能带来极大的差别。而这主要是由于条件分支指令可能会受到 CPU 分支预测错误带来的惩罚。

现代 CPU 一般都会采用投机执行，其中的一个场景是：处理器会从它预测的，分支可能会发生跳转的地方取出指令，并提前对这些指令进行译码等操作。处理器甚至会在还未确认预测是否正确之前，就提前执行这些指令。之后，如果 CPU 发现自己预测的跳转位置发生错误，就会将状态重置为发生跳转前分支所处的状态，并取出正确方向上的指令，开始重新处理。

由此，上述两种指令在 CPU 的内部执行上便产生了不同。由于条件分支指令会导致 CPU 在指令实际执行前作出选择，而当 CPU 预测错误时，状态的重置及新分支的重新处理过程会浪费较多的 CPU 周期，进而使程序的运行效率下降。相对地，条件传送指令不会修改处理器的 PC 寄存器，因此它不会导致 CPU 需要进行分支预测，也就不会产生这部分损失。

至于 CPU 是如何进行分支预测的，相关内容超出了这门课的范畴，这里我就不详细介绍了。但你需要知道的是，在发生类似问题时，我们可以进一步观察程序，并尝试使用条件传送指令优化这些逻辑。为了方便你理解，我们来看个例子。你可以看看下面这段代码中函数 `foo` 的实现细节：



复制代码

```
1 #define LEN 1024
2 void foo(int* x, int* y) {
```



```

3  int i;
4  for (i = 0; i < LEN; i++) {
5      if (x[i] > y[i]) {
6          int t = x[i];
7          x[i] = y[i];
8          y[i] = t;
9      }
10 }
11 }

```

函数 `foo` 接收两个整型数组 `x` 与 `y`，并依次比较这两个数组中位于相同索引位置上的元素，最后将较大者存放到数组 `y` 的对应位置上。我们可以看到，在遍历数组的过程中，我们在循环结构内使用了 `if` 语句，来判断数组 `x` 中的元素值是否大于数组 `y` 对应位置上的元素。而在代码实际编译时，`if` 语句通常会由对应的条件分支指令来实现。因此，在循环结构的“加持”下，由 CPU 分支预测错误引发的惩罚，在经过每一轮迭代的累积后，都可能会变得更加可观、更加明显。

下面，我们就来使用条件传送指令优化这段代码。条件传送指令一般会用于实现 C 语法中的三元运算符 `?:`，因此对上述代码的优化过程也就显而易见：

 复制代码

```

1  #include <stdio.h>
2  #define LEN 16
3  void foo(int* x, int* y) {
4      int i;
5      for (i = 0; i < LEN; i++) {
6          int min = x[i] < y[i] ? x[i] : y[i];
7          int max = x[i] < y[i] ? y[i] : x[i];
8          x[i] = min;
9          y[i] = max;
10     }
11 }

```

可以看到，这里我们没有使用 `if` 语句来判断，是否应该调整两个数组对应位置上的数字值，而是直接使用三元运算符，来将每一次迭代时的最大值与最小值结果计算出来，并拷贝到数组中的相应位置上。

通过这种方式，我们虽然解决了 CPU 分支预测失败带来的惩罚，但与此同时，每一次循环中也会多了几次比较与赋值操作。你可能想问：这样的一来一回真的可以提升性能吗？我的回答

 领资料

是：不要小看 CPU 指令并行处理能力的高效性，但也不要小看 CPU 分支预测带来的性能损耗。

## 技巧七：使用更高的编译优化等级

除了可以通过调整代码写法来优化程序运行外，我们还可以为编译器指定更高优化等级的选项，来让编译器自动为我们进行更多程序执行细节上的优化。

以 GCC 为例，它为我们提供了 -O0、-O1、-O2、-O3、-Os、-Ofast 等优化选项。我把它们各自的大致优化内容整理成了一张表格，你可以参考：

优化标记	优化内容简介
-O0	编译器采用的默认优化选项，编译耗时最短
-O1	编译器会尝试减小代码体积，并优化程序运行效率，但不会进行需要大量时间的编译优化。相较于 -O0，该选项将耗费更多的编译时间和内存
-O2	编译器会在 -O1 的基础上做进一步的代码优化，GCC 将应用几乎所有支持的、非空间换时间类的编译优化。相较于 -O1，该选项将耗费更多的编译时间和内存
-O3	编译器会在 -O2 的基础上做进一步的代码优化
-Os	编译器将在 -O2 的基础上，更侧重于优化生成二进制文件的体积
-Ofast	编译器会在 -O3 的基础上，再应用可能会违反 C 标准的更多优化策略
-Og	编译器将侧重优化程序的调试体验，该选项可以在保持快速编译和良好调试体验的同时，为代码应用合理的优化级别。对于需要支持调试的代码来说，它是比 -O0 更好的选择



## 技巧八：尾递归调用优化（Tail-Call Optimization）

尾递归调用优化也是一个重要的代码优化技巧。关于它的原理和代码编写方式，我已经在 [06 讲](#) 中为你介绍过，如果你觉得记忆有些模糊了，可以返回那一讲回顾下相关知识。

总的来看，尾递归调用优化通过将函数的递归调用过程优化为循环结构，减少了程序执行时对 call 指令的调用次数，进而减少了栈帧的创建与销毁过程，提升了程序的执行性能。并且你



需要注意，尾递归调用优化的效果在那些函数体本身较小，且递归调用次数较多的函数上体现得会更加明显。

## 总结

讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

今天我主要介绍了四种可用于实现高性能 C 程序的技巧：

- 循环展开让我们可以进一步利用 CPU 的指令级并行能力，让循环体执行得更快；
- 优先使用条件传送指令，让我们可以在一些特定的场景中，防止使用条件分支指令带来的 CPU 周期浪费；
- 使用更高的编译优化等级，让我们可以借编译器之手，利用更多“黑科技”进一步优化我们的代码；
- 尾递归调用优化让我们可以用循环代替递归，减少函数调用时的栈帧创建与销毁过程，让递归进行得更快。


## 思考题

最后，给你留个思考题：“达夫设备（Duff's Device）”有什么作用？它的实现原理是怎样的呢？欢迎在评论区告诉我你的发现。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

领资料

 赞 2  提建议



上一篇 18 | 极致优化（上）：如何实现高性能的 C 程序？

下一篇 20 | 生产加速：C 项目需要考虑的编码规范有哪些？

## 更多课程推荐

# 操作系统实战 45 讲

## 从 0 到 1, 实现自己的操作系统

彭东

网名 LMOS

Intel 傲腾项目关键开发者



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

### 精选留言 (4)

 写留言



术子米德

2022-01-26

优化，仅知道方法，非常容易出现伪优化

优化，确定度量方法，才能控制住优化真正效果

度量一段实现代码执行所需的耗时，即总指令数，以及每个时钟周期执行的指令数，即 $IPC=Instructions-Per-Cycle$ ，这两个指标抓住，大部分情况下打开编译器优化，就达到技巧所谓的优化效果

如果要有更多的优化，都是要选择新的算法或者结合业务和运行环境的各种适配性调优，语言层面的技巧开不出更多的花

共 3 条评论 >

 4

 领资料



八怪

2022-03-22

老师 `__builtin_expect` 能有效减少分支预测带来的性能损失吗？

作者回复: 如果合理使用的话（场景合适），理论上是可以的，Linux 内核里也有在用这些扩展函数。但实际使用时还是建议配合 `profiler` 检验一下优化效果。



**fee1in**

2022-01-28

而当五个阶段全部执行完毕后，CPU 会更新指令指针（PC），将其指向下一个需要执行的指令

应该是在IF结束后，更新PC把 不然跳转指令就会出问题

作者回复: 这里针对五级 RISC 流水线来说，实际上 PC 的值一般是在 IF 阶段就可以计算（预测）好的，然后在 WB 之后才会实际更新到 PC 寄存器中。



**liu\_liu**

2022-01-26

看了达夫设备的代码，原来 `switch case` 语句还可以这样用，涨见识了。



领资料

