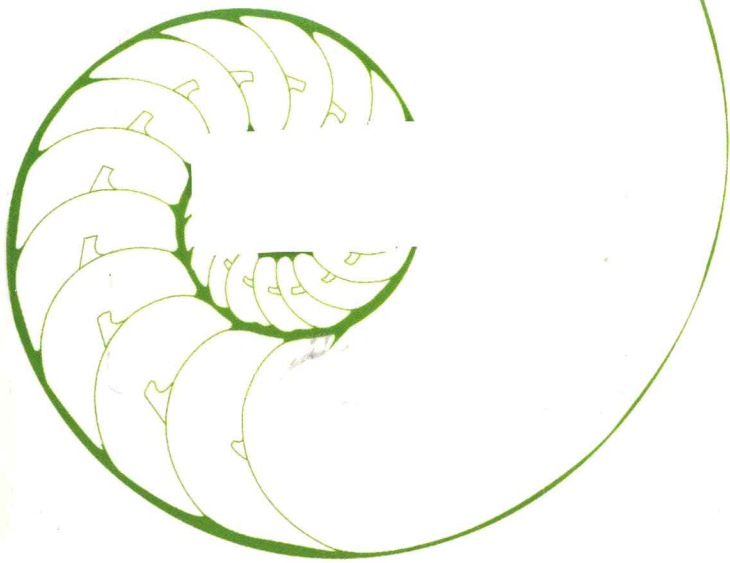
TURING
图灵程序设计丛书
usp lab.

让你的程序飞起来

C程序性能优化

20个实验与达人技巧

[日] 片山善夫◎著
何本华 居福国◎译

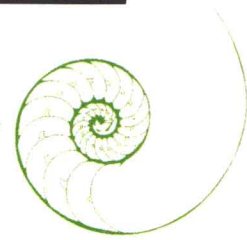


人民邮电出版社
POSTS & TELECOM PRESS

让你的程序飞起来

C程序性能优化

20个实验与达人技巧



本书作者精通C程序性能优化，具有近二十年的C语言编译器和解释器开发经验，还为实时图像处理专用芯片开发过C编译器。

作者从CPU与编译器的运行机制讲起，带领读者一步步了解程序的执行成本、编译器的优化选项等，总结出许多C程序性能优化的技巧，并将这些技巧通过实验的方式进行讲解，简明易懂，使人印象深刻。书中带有大量的代码实例，使读者不仅能够了解代码优化的原理，还能够轻松地在实践中加以应用。

图灵社区：www.ituring.com.cn
图灵微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010) 51095186 转 604

分类建议 计算机/编程语言/C语言

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-30000-3
定价：29.00元

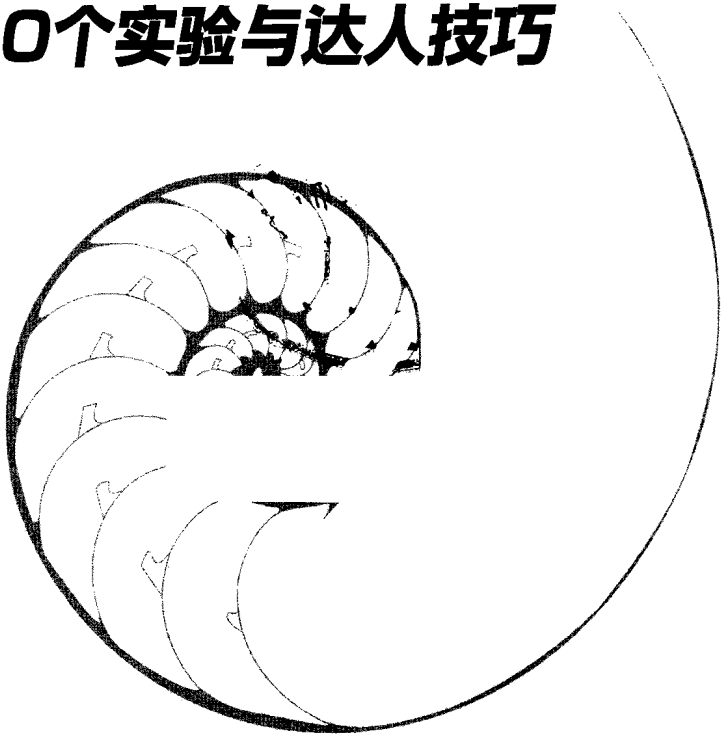
TURING
图灵程序设计丛书
usp lab.

让你的程序飞起来

C程序性能优化

20个实验与达人技巧

[日] 片山善夫◎著
何本华 居福国◎译



人民邮电出版社
北京

图书在版编目 (CIP) 数据

C 程序性能优化：20 个实验与达人技巧 /（日）片山善夫著；何本华，居福国译 . -- 北京：人民邮电出版社，2013.1

（图灵程序设计丛书）

ISBN 978-7-115-30000-3

Ⅰ. ①C… Ⅱ. ①片… ②何… ③居… Ⅲ. ①C 语言－程序设计 Ⅳ. ① TP312

中国版本图书馆 CIP 数据核字（2012）第 268131 号

内 容 提 要

本书从 CPU 与编译器的运行机制讲起，带领读者一步步了解程序的执行成本、编译器的优化选项等，总结出许多 C 程序性能优化的技巧，并以实验的方式进行了讲解，简明易懂，使人印象深刻。书中带有大量的代码实例，使读者不仅能够了解代码优化的原理，还能够轻松地在实践中应用。

本书适合有一定基础的 C 语言编程人员阅读。

图灵程序设计丛书

C 程序性能优化：20 个实验与达人技巧

原书名：C プログラム高速化研究班～コードを高速化する 20 の実験と達人の技～

原出版社：有限会社ユニバーサル・シェル・プログラミング研究所

◆ 著　[日] 片山善夫

译　何本华　居福国

责任编辑　傅志红

执行编辑　乐　馨

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号

邮编　100061　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

三河市海波印务有限公司印刷

◆ 开本：880×1230　1/32

印张：4.75

字数：181 千字　　2013 年 1 月第 1 版

印数：1-4 000 册　　2013 年 1 月河北第 1 次印刷

著作权合同登记号　图字：01-2012-5554 号

ISBN 978-7-115-30000-3

定价：29.00 元

读者服务热线：(010)51095186 转 604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

目录

第 1 章

CPU与编译器概论

1.1 高速路与人行道

近年来，台式机的CPU主频已达到2~3GHz，就连iPhone等智能手机和便携式终端的CPU主频率亦可达到0.5~1GHz。

这些CPU，有如在高速路上奔驰的跑车一样。试想一下，如果高速路上遍布着红绿灯和人行道，那跑车的性能就不能完全发挥出来。

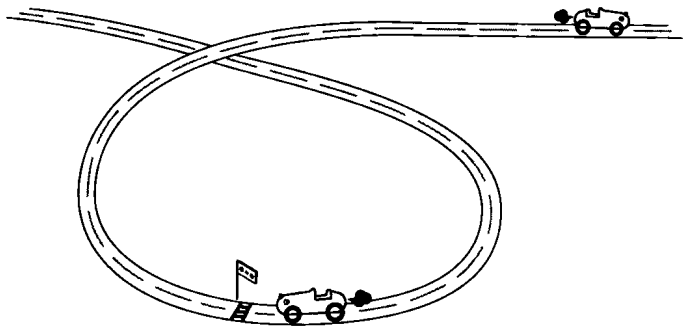


图1-1 高速路与人行道

无论在高速路上跑得多快，一旦遇上红绿灯和人行道那该怎么办？想必就算是高性能跑车，在这个时候减速也会影响它的性能发挥。即使等到了绿灯，再次加速行驶，但遇到下一个红绿灯时也不得不再次减速。实际生活中的高速路上是没有人行道的，但计算机中的程序却是在“有红绿灯的高速路”上工作着。

读者在用C或C++编写程序时，常常会在程序里设置很多“红绿灯和人行道”。虽然需要减速的原因有很多，但只要去掉其中几个主要的障碍，程序的运行速度就会提高几十倍。

那么，哪些程序在扮演着“高速路上的人行道”呢？我们该如何规避它？要想解答这些问题，我们就必须了解CPU的构造和工作原理，以及编译器运行的相关知识。

在这一章中，我们会在探讨性能优化的具体方法之前，先对CPU的构造与编译器的运行原理进行简单的讲解。

1.2 编译器是如何运作的

大多数程序员在日常编程中很少会直接用到CPU中的指令（即机器语言）。这主要是因为直接使用机器语言比较繁琐，所以我们选择人类更容易理解的语言来编程，然后再通过编译器将其翻译成机器语言。但是，编译器能否准确地将人类的逻辑思维转换为相应的机器语言呢？在这里，我们先来研究一下编译器到底是如何运作的。

比如，使用GCC按以下步骤将程序编译为目标代码(即汇编语言程序)。

1. 读取源程序并进行解析。将字符分离出来整理成较容易统计的形式，收集参数与函数名等标识符。
2. 对收集到的参数与标识符进行内存地址分配（即后文将提到的寄存器），将内存地址与参数或函数对应起来。
3. 根据逻辑程序生成汇编语言程序。

接下来，汇编编译器将已生成的汇编语言转换成机器语言的目标程序，链接器将目标程序和外部模块连接起来（图1-2）。

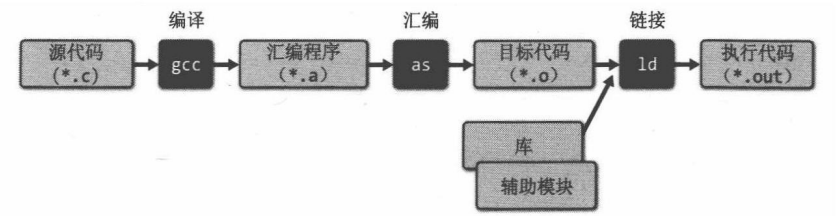


图1-2 从源程序到执行代码的实现过程

近代的编译器实现了在编译过程中，让所生成的程序在更短的时间内得到相同的结果，达到更高的效率。其实现的方法多种多样，比如说，编译器扫描程序后，将多余的操作忽略，修改指令的运行顺序以使CPU处理得更快等。

优化与程序的调优有着密不可分的关联，在后面的章节中将会提到。

编译后的汇编语言程序

我们来看看由 GCC 生成的汇编语言程序。程序 1-1 是为检验而编写的小程序。

程序 1-1　10 次加 1 运算的程序

```
#include <stdio.h>
int a, b;
main()
{
    a = 0;
    do {
        b += a + 1;
        a++;
    } while (a < 11);
}
```

如果在编译此程序时加上 -S 选项，如“gcc -S test.c”，就会出现如程序 1-2 这样的汇编语言程序（该程序在 X86 系列 64 位环境下进行编译）。

程序 1-2　编译后的汇编语言程序（部分）

```
        .text
.globl main
        .type   main, @function
main:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        movl    $0, a(%rip)       ……变量 a 赋值为 0
.L2:
        movl    a(%rip), %eax     ……变量 a 的值放到寄存器 %eax 中
        leal    1(%rax), %edx     ……（变量 a）+1 的值放入 %edx(%rdx 的低 32 位 ) 中
        movl    b(%rip), %eax     ……将变量 b 的值放入 %eax（%rax 的低 32 位）中
        leal    (%rdx,%rax), %eax  ……%rdx 和 %rax 的和放入 %eax 中
        movl    %eax, b(%rip)     ……%eax 的结果赋值给变量 b
```

```
        movl    a(%rip), %eax     ……变量 a 的值放入 %eax 中
        addl    $1, %eax          ……对 %eax 进行加 1 运算
        movl    %eax, a(%rip)     ……%eax 的结果赋值给变量 a
        movl    a(%rip), %eax     ……变量 a 的值放入 %eax 中
        cmpl    $10, %eax         ……将 %eax 的值与 10 进行比较
        jle     .L2               ……小于等于 10 的话跳到 L2
        leave                     ……释放栈中的变量
        ret                       ……程序跳出
.LFE2:
        .size   main, .-main
        .comm   a,4,4             ……分配给 a 以 4 为边界基准的 4 字节内存
        .comm   b,4,4             ……分配给 b 以 4 为边界基准的 4 字节内存
        …
```

大致上，左边是标签，中间是 CPU 指令，右边是操作目标。以“.”开始的字符串表示指定汇编程序集的函数名。以“:”结束的字符行是标识符的定义。函数名和标识符以外的部分是实际被执行的指令集，与 CPU 的机器语言相对应。以“%”开头的是寄存器名，以“$”开头的是常量。

“a(%rip)”表示外部变量 a，“b(%rip)”表示外部变量 b，外部变量引用以 %rip 标识的寄存器（程序计数器标识程序接下来该执行的指令）所指的地址中相对应的位置。

从标识符 .L2 到 jle 指令是程序的循环部分，相当于 C 语言源程序的“do~while(a<11)”。即使没有使用过汇编语言，一看到程序当中的备注也能大致理解其运行原理。

X86 系列 CPU 的寄存器

CPU 内部有若干个高速存储单元，也就是常说的寄存器，它可以用来储存数据，还可以作为指示其储存地址的指针来使用。在下面的表格中，我们列举了 X86 系列 CPU 中常用的寄存器。

一直以来，32 位的 CPU 中所使用的寄存器为 32 位寄存器，64 位 CPU 中则使用扩展到 64 位的寄存器，更甚者追加到寄存器 r8~r15。扩展后的 64 位寄存器可以在 64 位模式下使用。

在64位环境中，通过使用增加的寄存器，可完成程序中函数参数的传递。rax被用于存放返回值，rdi、rsi、rdx、rcx、r8、r9分别用于存放第1~6个整数型参数[①]。

在32位环境中，将函数写入栈中，这样可以缩短执行时间。

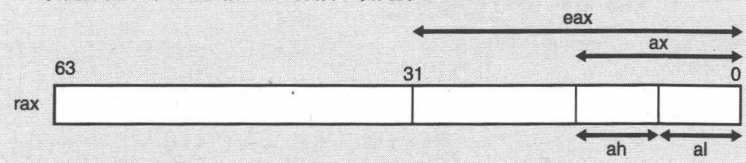
另外，对部分寄存器来说，寄存器的一部分也可进行数据存储和计算操作。比如，寄存器rax的低32位是eax，eax的低16位是ax；ax的高8位是ah，ax的低8位是al等。

x86/x64系列CPU的主要寄存器

寄存器名（64位环境）	用　途	寄存器名（32位环境）	用　途
rax	第一函数值，操作用	eax	函数值，操作用
rbx	寄存器变量	ebx	寄存器变量
rcx	操作用，第四函数变量	ecx	操作用
rdx	操作用，第三函数变量，第二函数值	edx	操作用
rsi	操作用，第二函数变量	esi	寄存器变量
rdi	操作用，第一函数变量	edi	寄存器变量
r8	操作用，第五函数变量	—	
r9	操作用，第六函数变量	—	
r10~r11	操作用	—	
r12~r15	寄存器变量	—	
rbp	基指针，寄存器变量	ebp	基指针，寄存器变量
rip	程序计数器（提示下一条指令）	eip	程序计数器（提示下一条指令）

*1 函数变量仅在64位模式下使用。

*2 寄存器变量在某些时候被分配为操作用寄存器。



部分寄存器

① 参考：System V Application Binary Interface（http://www.x86-64.org/documentation/abi.pdf）

添加优化选项后的结果

想必即使是不熟悉汇编语言的读者也能看出，前面所生成的代码太过冗余。虽然在计算时将内存上变量 a 和 b 的内容复制到了寄存器 %eax 上，但是如果能将变量 a 和 b 分别放在不同的寄存器上，然后仅对寄存器进行操作，这样岂不是更合理？

接下来，我们将添加优化选项（-O），执行“gcc -S -O3 example.c”指令后再来编译看看。其结果如程序 1-3 所示，可以看得出所生成的代码更为简练。

程序 1-3　优化选项的效果

```
        .text
        .p2align 4,,15
.globl main
        .type   main, @function
main:
.LFB13:
        movl    b(%rip), %edx       ……变量 b 的值放到寄存器 %edx 中
        movl    $0, a(%rip)         ……变量 a 赋值为 0
        xorl    %eax, %eax          ……将寄存器 %eax 清零
        .p2align 4,,7
.L2:
        addl    $1, %eax            ……对寄存器 %edx 做加 1 运算
        addl    %eax, %edx          ……寄存器 %edx 与 %eax 相加
        cmpl    $10, %eax           ……将寄存器 %eax 的值与 10 相比
        jle     .L2                 ……小于等于 10 则跳到 L2
        movl    %edx, b(%rip)       ……将加法运算的结果赋值给变量 b
        movl    $11, a(%rip)        ……变量 a 赋值为 11
        ret
.LFE13:
        .size   main, .-main
        .comm   a,4,4
        .comm   b,4,4
        …
```

前面冗余的代码在循环内对外部变量逐一计算并赋值，但是在优化

后的代码中，我们用寄存器代替了外部变量，在最后灵活地将变量 a 赋值为 11，并且在循环代码前面使用了能缩短执行时间的逻辑操作指令 xorl（XOR），这也是为了提高效率。

这只是编译器进行优化的一个小小例子，但我们可以从中看出，直接生成的代码和优化后的代码之间存在哪些区别。

1.3 CPU 是如何运作的

上一节中我们介绍了程序经过编译器和汇编程序的转换后，最终得到机器语言的过程，也探讨了由 C 语言程序生成汇编语言的过程。下面我们再进一步探究 CPU 是如何执行机器语言程序的。

指令集架构与微架构

CPU 能够执行什么样的指令，或者说 CPU 所具备的指令集，称为 CPU 的指令集架构。

指令集架构是规定程序设计如何使用指令的规范，它包括寻址模式和寄存器构成、中断、异常处理等。所以，只要指令集架构是相同的，即使使用不同品牌的 CPU，所得出的结果也是一样的。比如，在上一节所展示的汇编语言程序中，只要是 x86_64 架构的 CPU，不管其厂家是 AMD 还是英特尔，结果都是一样的。

相反，如果指令集架构不同，那么 CPU 的指令也会相应发生变化。但是，不管使用的方法是否一样，几乎所有的 CPU 中都具备了以下基本指令。

- 算术运算
- 逻辑运算（AND，OR，XOR）
- 移位指令
- 条件比较指令
- 寄存器与内存之间的传送
- 跳转指令

另一方面，CPU 执行指令集架构的方法叫做微架构。各品牌为了使 CPU 更高效化，在设计上花费了不少功夫，所以即使是拥有同一指令集架构的 CPU，其性能和负荷方面的特点也是有所不同的。

高效编程的一个重要手段就是掌握 CPU 高效执行指令的方法。下面我们了解一下 CPU 的基本构造，以及它是如何执行指令的。

如何执行指令

所谓计算机，就是指按顺序执行分配到内存上的程序（指令的排列），并通过这一操作完成数据处理的机器。CPU 大致按以下顺序执行指令（图 1-3）。

1. 从内存读取指令（读取）
2. 解析所读取的指令（解码）
3. 提取操作目标的数据
4. 执行计算和条件对比等指令
5. 输出加工后的数据

图 1-3　指令的执行

第三步是从寄存器或者内存中读取出第四步执行指令时所需要的数据。

CPU 中有分管上述各功能的管理单元，解码负责支配执行指令时各单元的运行。第四步中的指令执行装置包含执行加减运算的算术逻辑单元（ALU）、逻辑运算器、乘法器、除法器、装载 / 记忆装置等单元，与所执行的指令结合使用。如下例，图 1-4 是在执行寄存器间的计算指令时各单元的运作情况，图 1-5 是在执行将内存数据装载到寄存器时各单元的运作情况。