第五章 优化程序性能

主要探讨内容：如何使用几种不同类型的程序优化技术，使程序运行的更快

编写高效程序的几种方式：

1. 选择一种适当的算法和数据结构；
2. 编写出编译器能够有效优化以转换成高效可执行代码的源代码；
   ① 不依赖机型底层的优化 （5.1-5.6）
   ② 依赖指定机型处理器微体系结构的优化 （5.7-5.9）
3. 针对处理量大的计算，采用并行计算（12章）或者异构并行计算。

本章内容：如何编写出**编译器**能**够有效优化**以**转换成高效可执行代码**的源代码。

5.1、编译优化器的能力和局限性

理想：编译器能够接受我们编写的任何代码，并产生尽可能高效的、具有指定行为的机器级程序。

现实：现代编译器采用了复杂的分析和优化形式，而且变得越来越好。然而，即使最好的编译器也受到**妨碍优化的因素**（optimization blocker）的阻碍。

编写程序方式中看上去只是一点小小的变动，都会引起编译器优化方式很大的变化，所以一定要先理解编译优化器的能力和局限性有哪些。

5.1.1 编译优化器的能力

对于C/C++程序，大多数的编译器会指定优化级别，以GCC为例子：gcc -o指令就可以设置优化级别：

-o0:关闭所有优化

-o1:最基本的优化级别，编译器试图以较少的时间生成更快以及体积更小的代码。

（后续代码示例使用-o1编译，目的是为了展示 C语言函数的不同方法如何影响编译器产生代码的效率，使得让-o1写出的C代码能够比更高级别优化选项编译的性能还要好）

-o2:推荐的优化级别，o1的进阶。 （目前被普遍接受）

-o3:较危险的优化等级，这个等级会延长编译时间，编译后会产生更大的二进制文件，会带来一些无法预知的问题。

-os:优化代码体积，通常适用于磁盘空间紧张或者CPU缓存较小的机器。

5.1.2 编译优化器的局限性

编译器对程序只使用***安全的优化***：对于程序可能遇到的所有可能情况，优化后的程序和未优化的版本有一样的行为。

在***安全的优化***中，主要有两个因素妨碍了优化

1. ***内存别名使用***
2. ***函数调用***

5.1.2.1内存别名使用

举个“栗子”：

哪个函数的效率更高？为什么？

将存储在由指针 yp指示的位置处的值两次加到指针xp指示的位置处的值

```
void twiddle1(long *xp, long *yp)
{
    *xp += *yp;
    *xp += *yp;
}

void twiddle2(long *xp, long *yp)
{
    *xp += 2 * *yp;
}
```

函数twiddle2效率更高一些。它只要求3次内存引用，而twiddle1需要6次(2次读*xp，2次读*yp，2次写*xp)。因此，如果要编译器编译过程twiddle1，我们会认为基于twiddle2执行的计算能产生更有效的代码。

不过，考虑xp等于yp的情况。此时，函数twiddle1会执行如下操作

```
*xp += *xp;
*xp += *xp;
```

结果就是*xp的值变为原来的4倍

而twiddle2会执行如下操作

```
*xp += 2* *xp;
```

结果却是*xp的值变为原来的3倍

所以编译器不会产生 Twiddle2 的代码作为 Twiddle1 的优化代码.

两个指针可能存在指向同一个内存位置的情况称为***内存别名使用***，在执行安全的优化时，编译器会假设不同的指针指向内存的同一个位置。

5.1.2.2函数调用

举个“栗子”：

func1和func2的行为是否相同？哪个效率更高？为什么？

```
long f();

long func1(){
    return f() + f() +f() +f();
}

long func2(){
    return 4 * f();
}
```

可以看出f1调用了f()四次，而f2()只调用了一次，函数的调用涉及到栈帧的操作这需要消耗一些系统资源，因此按理来说f2()的性能优于f1()，但是编译器针对这种情况同样不会进行优化，考虑到以下代码

```
long counter = 0 ;

long f(){
    return counter++;
}
```

f()函数有一个***副作用***——它修改了全局程序状态的一部分，改变调用它的次数会改变程序的行为

大多数的编译器不会去判断一个函数是否没有副作用，编译器会假设最糟糕的情况，并保持所有的函数调用不变。

优化方式：内联函数替换

对于 Func1() 我们可以用内联函数进行优化即将函数调用替换为函数体:

```
//优化后的版本
int Funct1Inline()
{
    int t = counter++;    // +0
    t += counter++;        // +1
    t += counter++;        // +2
    t += counter++;        // +3
    return t;
}
```

这样的转换不仅可以减少了函数调用的开销, 也允许编译器对代码进行进一步优化:

```
//编译器进一步优化
int Func1Opt()
{
    int t = 4 * counter + 6;
    counter += 4;
    return t;
}
```

5.1.3 GCC

GCC的优化能力并不突出，它能完成基本你的优化，但是它不会对程序进行更加“有进取心的”编译器所做的那种激进变化。因此，使用GCC的程序员必须花费更多的精力，以一种简化编译器生产高效代码的任务来编写程序。

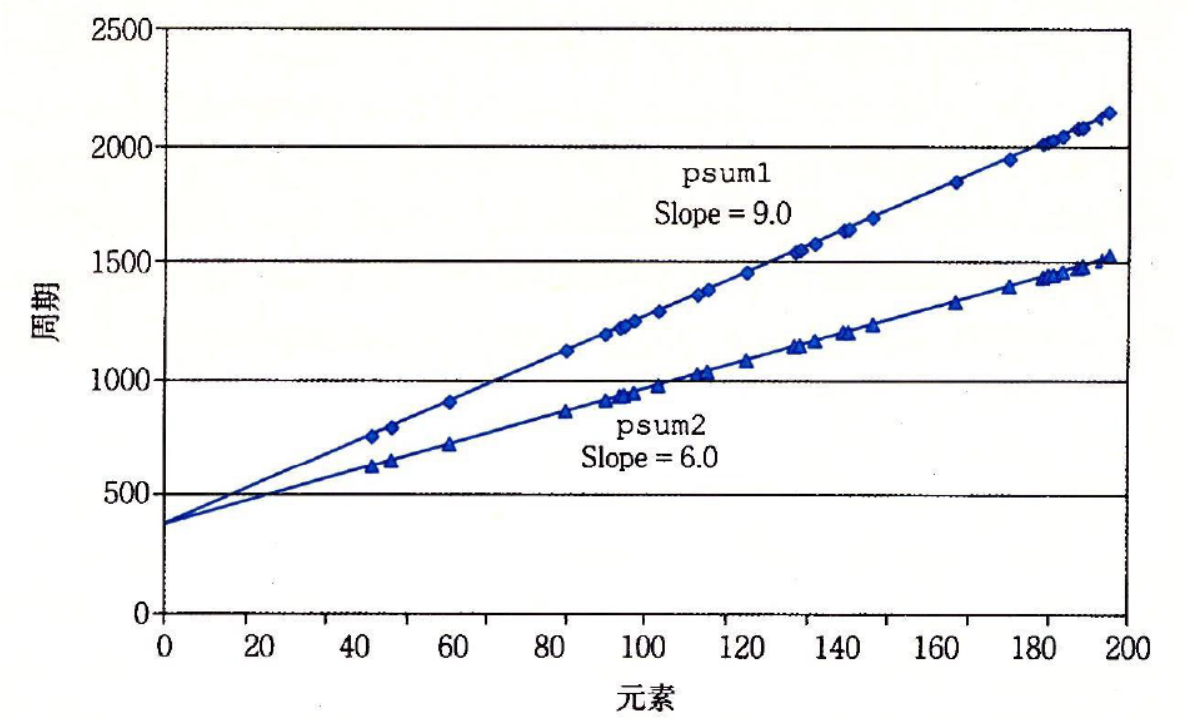
5.2 表示程序的性能 CPE

我们引入CPE作为一种表示程序性能并指导我们改进代码的方法。

CPE：cycles per element , ***每元素的周期数***。

```
/* Compute prefix sum of vector a */
void psum1(float a[], float p[], long n){
    long i;
    p[0] = a[0];
    for (i = i; i < n; i++){
        p[i] = p[i-1] + a[i];
    }
}

void psum2(float a[], float p[], long n){ //循环展开技术
    long i;
    p[0] = a[0];
    for (i = 1; i < n-1; i+=2){
        float mid_val = p[i-1] + a[i];
        p[i] = mid_val;
        p[i+1] = mid_val + a[i+1];
    }
    /*For even n,finish remaining element */
    if (i < n)
        p[i] = p[i-1] + a[i];
}
```



运行时间 （单位：时钟周期 比如 4GHz处理器 ， 运行频率为每秒4*10^9个周期 ， 一个时钟周期为0.25纳秒）

psum1：368 + 9.0n

psum2：368 + 6.0n

对于较大的n，函数的运行时间就会主要由线性因子决定，即每个元素的周期数（Cycles Per Element，CPE）

5.3 程序示例

```
// 这里使用宏不是故弄玄虚，是为了便于测试加法和乘法对比，后续的效率对比都会对比两种操作
typrdef struct {
    long len;
    data_t *data;
}vec_rec, *vec_ptr;

int vec_length(vec_ptr v) {
    return v->len;
}
int get_vec_element(vec_ptr v, long index, data_t *dest) {
    if (index < 0 || index >= v->len)
        return 0;
    *dest = v->data[index];
    return 1;
}

vec_ptr new_vec(long len){
   ```malloc`````
}

#define IDENT 0 // 1
#define OPER +  // *

void combin1(vec_ptr v, data_t *dest) {
    int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OPER val;
    }
}
```

函数	方法	整数		浮点数	
		+	*	+	*
combine1	抽象的未优化的	22.68	20.02	19.98	20.18
combine1	抽象的-O1	10.12	10.12	10.17	11.14

5.4 消除循环的低效率

观察到上面代码循环条件调用`vec_length`，实际上该列表在循环中并不会改变长度，所以将这个函数的返回值作为临时变量，减少函数调用。
```

```
void combine2(vec_ptr v, data_t *dest) {
    int i;
    *dest = IDENT;
    int len = vec_length(v);
    for (i = 0; i < len; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OPER val;
    }
}
```

这是一类常见的、被称为***代码移动***的优化方法。编译器通常会非常小心，它们无法可靠的知道一个函数挪动位置是否有副作用。所以这种优化只能由程序员来处理。通过上述的改动，得到程序的运行时间的改变对比

函数	方法	整数		浮点数	
		+	*	+	*
combine1	抽象的-O1	10.12	10.12	10.17	11.14
combine2	移动 vec_length	7.02	9.03	9.02	11.03

所谓循环低效率，就是说循环里面做了一些不必的事，效率自然低了。道理很简单，减少不必要的调用。
我认为任何优化都是在两个方向上出力，一个是操作数量，一个是单个操作的时间。可以简单的认为 消耗时间 = 单个时间 * 操作数量。
什么叫`渐进效率`：即随着循环次数的增加，指数级的增加运行时间。书中举了个在循环判断中使用strlen函数的例子，随着字符串长度的增加，循环的耗时急剧增长。应当注意避免，少在循环条件中使用一些耗时的函数。

5.5 减少过程调用

减少过程调用即减少***函数调用***。例如上面combine1函数循环中会调用的get_vec_element函数，而这个函数实际上是很容易优化掉的。例如，我们使用指针的思路来改造为：

```
data_t *get_vec_start(vec_ptr v) {
    return v->data;
}
void conbine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

看起来代码简洁简单了许多，然而令人遗憾的是，性能基本没有发生变化：

函数	方法	整数		浮点数	
		+	*	+	*
combine2	移动 vec_length	7.02	9.03	9.02	11.03
combine3	直接数据访问	7.17	9.02	9.02	11.03

但是应该注意，不是说这个优化方法、方向不对，而是对于这个函数，`get_vec_element`不是函数的瓶颈，有其他部分的性能限制了它的性能界限。

可能有人奇怪，就算其他地方有性能限制，这里说到底也是减少了函数调用和判断的操作，怎么性能就没有任何变化，完全说不通啊。

不要着急，优化到了这里，仿佛已经开始往奇诡的方向发展，但是要相信科学，马上我们就能打开不一样的世界

5.6 消除不必要的存储器引用

通过查看`combine3`的汇编代码：

```
    Inner loop of combine3.  data_t = double, OP = *
    dest in %rbx, data+i in %rdx, data+length in %rax
1   .L17:                                   loop:
2     vmovsd  (%rbx), %xmm0                   Read product from dest
3     vmulsd  (%rdx), %xmm0, %xmm0            Multiply product by data[i]
4     vmovsd  %xmm0, (%rbx)                   Store product at dest
5     addq    $8, %rdx                        Increment data+i
6     cmpq    %rax, %rdx                      Compare to data+length
7     jne     .L17                            If !=, goto loop
```

可以看到`(%rbx)`是表示dest，在2、3、4行中，被反复的读取和写入内存。

内存的读写比寄存器要慢不少，消除这种内存的读写，可以提高上面函数的循环效率。

怎么消除？我们知道临时变量在寄存器够用的情况下都是在寄存器中的，这里可以通过一个临时变量来实现：

```
void combine4(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t add = IDENT;
    for (i = 0; i < length; i++) {
        acc = acc OP data[i];    // 通过局部变量来累加
    }
    *dest = acc;   // 最终才赋值给dest
}
```

```
    Inner loop of combine4.  data_t = double, OP = *
    acc in %xmm0, data+i in %rdx, data+length in %rax
1   .L25:                                   loop:
2     vmulsd  (%rdx), %xmm0, %xmm0            Multiply acc by data[i]
3     addq    $8, %rdx                        Increment data+i
4     cmpq    %rax, %rdx                      Compare to data+length
5     jne     .L25                            If !=, goto loop
```

对比CPE，可以看到停滞不前的CPE终于又有了动静。

函数	方法	整数		浮点数	
		+	*	+	*
combine3	直接数据访问	7.17	9.02	9.02	11.03
combine4	累积在临时变量中	1.27	3.01	3.01	5.01

可能有人会觉得好的编译器应该能自己做到使用临时变量来优化。但是对于这个函数，由于可能存在的内存别名的问题，编译器没法这么优化，例如dest如果是v列表中的某一项的地址呢？结果其实就会完全不一样了。只有程序员知道是否会出现这个情况。

现在我们还有两个问题：第一，为什么上一步减少`get_vec_element`的优化，看起来没有作用。第二，`combine4`是否还有进一步优化的空间？

且听下回分解。

5.7 理解现代处理器 *

5.8 循环展开 *

5.9 提高并行性 *

5.10 优化合并代码的结果小结

5.11 一些限制因素 *

5.12 理解内存性能*

5.13 应用：性能提高技术

优化程序的基本策略：

1. 高级设计：选择适当的算法和数据结构，避免指数级增加消耗的算法和编码技术
2. 基本编码原则：

   消除连续的函数调用。将可以移出循环的函数或者计算移出来

   消除不必要的内存引用。加载内存和写入内存都有消耗，使用中间变量来减少内存的访问
3. 低级优化

   循环展开，降低循环开销

   使用多个累积变量的方法，利用处理器指令并行处理能力

   使用功能性风格重写条件判断，减少判断，便于编译采用条件传送功能。

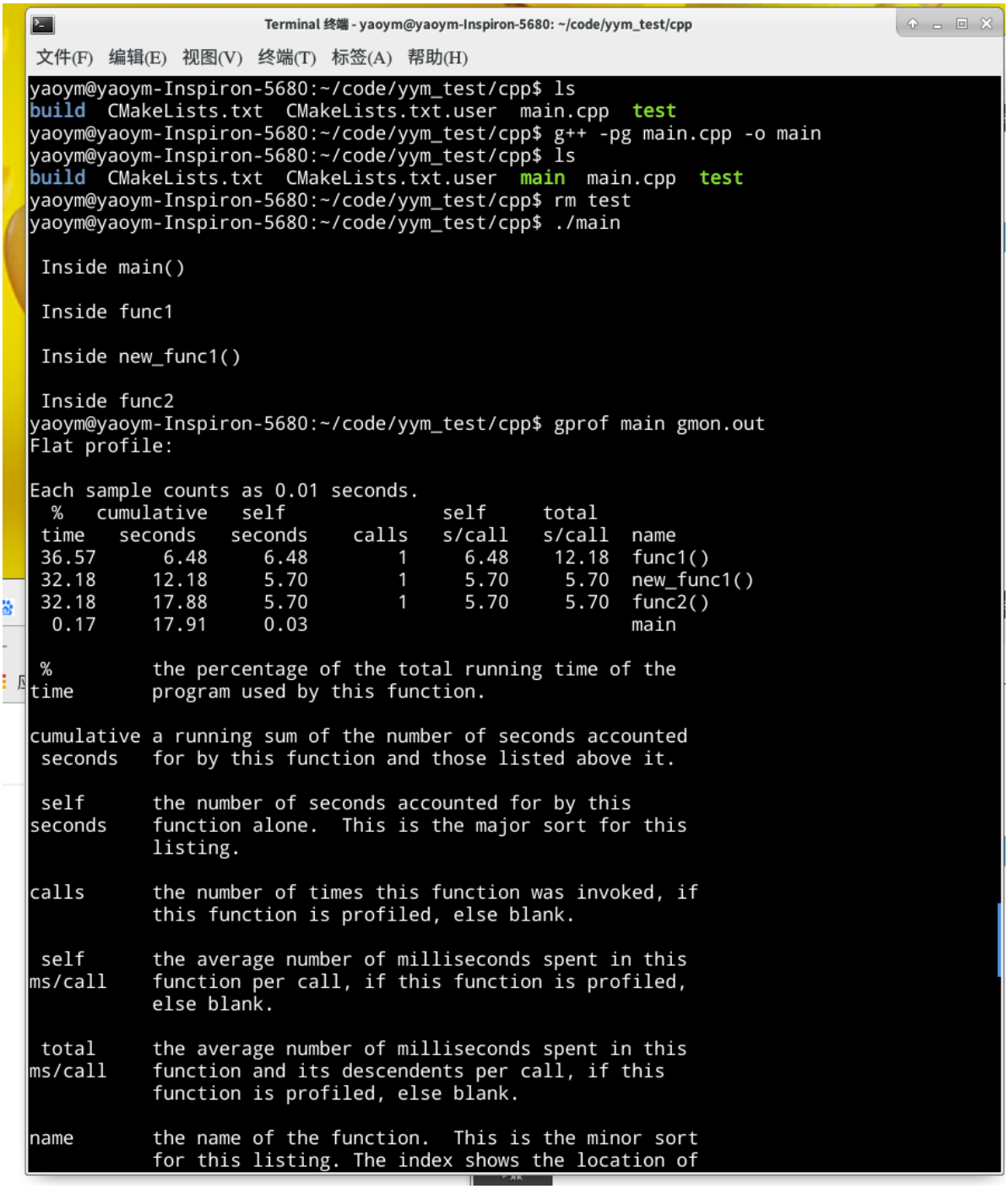
需要注意，优化代码不应该改变代码本身的逻辑，同样的输入应当产生同样的输出。

5.14 确认和消除性能瓶颈

介绍一个工具，用于分析时间

```
//test_gprof.c
#include<stdio.h>
```

```c
void new_func1(void)
{
    printf("\n Inside new_func1()\n");
    int i = 0;

    for(;i<0xfffffee;i++);

    return;
}

void func1(void)
{
    printf("\n Inside func1 \n");
    int i = 0;

    for(;i<0xffffffff;i++);
    new_func1();

    return;
}

static void func2(void)
{
    printf("\n Inside func2 \n");
    int i = 0;

    for(;i<0xfffffaa;i++);
    return;
}

int main(void)
{
    printf("\n Inside main()\n");
    int i = 0;

    for(;i<0xffffff;i++);
    func1();
    func2();

    return 0;
}
```


Terminal 终端 - yaoym@yaoym-Inspiron-5680: ~/code/yym_test/cpp
文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)
yaoym@yaoym-Inspiron-5680:~/code/yym_test/cpp$ ls
build CMakeLists.txt CMakeLists.txt.user main.cpp test
yaoym@yaoym-Inspiron-5680:~/code/yym_test/cpp$ g++ -pg main.cpp -o main
yaoym@yaoym-Inspiron-5680:~/code/yym_test/cpp$ ls
build CMakeLists.txt CMakeLists.txt.user main main.cpp test
yaoym@yaoym-Inspiron-5680:~/code/yym_test/cpp$ rm test
yaoym@yaoym-Inspiron-5680:~/code/yym_test/cpp$ ./main
Inside main()
Inside func1
Inside new_func1()
Inside func2
yaoym@yaoym-Inspiron-5680:~/code/yym_test/cpp$ gprof main gmon.out
Flat profile:
Each sample counts as 0.01 seconds.
% cumulative self self total
time seconds seconds calls s/call s/call name
36.57 6.48 6.48 1 6.48 12.18 func1()
32.18 12.18 5.70 1 5.70 5.70 new_func1()
32.18 17.88 5.70 1 5.70 5.70 func2()
0.17 17.91 0.03 main
% the percentage of the total running time of the
time program used by this function.
cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.
self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
listing.
calls the number of times this function was invoked, if
this function is profiled, else blank.
self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
else blank.
total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
function is profiled, else blank.
name the name of the function. This is the minor sort
for this listing. The index shows the location of

5.15 小结