

程序优化

Program Optimization

课 程 名 : 计算机系统

第 7 讲 (2025 年 5 月 7 日)

主 讲 人 : 杜海文

本课内容

■ 概述

■ 通用的优化方法

- 代码移动（预先计算）
- 复杂运算简化
- 共享公用子表达式
- 去除多余的过程调用

■ 优化障碍（妨碍优化的因素）

- 过程调用
- 存储器别名使用Memory aliasing（不同名字指向相同内存）

■ 利用指令级并行

关于程序性能，需要认清几个现实

- 性能，不只是时间复杂度的问题
- 常数项同样重要！
 - 不同的代码写法，动辄导致 10 倍的性能差距
 - 优化必须在多个层次上开展：
 - 算法、数据表示（结构）、过程、循环
- 要优化性能，必先理解系统
 - 程序是怎样编译和执行的
 - 要明确地知道编译器能力的边界
 - 现代处理器 + 存储系统是怎么运作的
 - 怎样衡量程序的性能，找出限制性能的瓶颈
 - 如何在不破坏代码模块性和通用性（可移植性）的前提下提高性能

编译器优化——编译器友好型代码

- 构建从程序到机器的高效映射
 - 寄存器如何分配
 - 代码的选取，排序（调度）
 - 死代码dead code如何去除
 - 代码中效率不高的部分如何解决
- 不要只想着复杂度asymptotic efficiency
 - 由程序员决定**总体上**采用哪一种算法
 - 大 O 级别的优化固然比常数项重要
 - 然而常数项也很重要
- 优化障碍
 - 潜在的内存别名使用问题（例见教材 5.1: twiddle1/2）
 - 潜在的函数副作用问题（例见教材 5.1: func1/2）

编译器优化的局限性

■ 基本限制

■ 一定不能造成程序的行为发生任何改变

- 有一个例外：程序有可能会使用非标准的语言特性
- 例如：&& 标号、间址 goto

■ 常常因为考虑到一些极端条件，导致编译器不作优化

■ 有些行为在程序员眼中再明显不过，在编译器看来却有些不太确定

■ 例：数据的取值范围可能远小于变量类型所规定的范围

■ 代码分析通常只限定在过程内部

■ 全程序分析成本太高

■ 新版 GCC 能在单个文件中进行过程间分析

- 但仍做不到跨文件的分析

编译器优化的局限性（续）

- 大多数分析都是基于静态信息的
 - 编译器很难预测程序运行期间的实时输入
- 在不确定的情况下，编译器只能采取最保守的策略
- 编译后的指令执行顺序可能与源程序不同
- 反汇编并分析代码是理解编译器运作的有效手段
- 修改源代码，引导编译器生成更高效的实现
- 我们假定编译器很 low：
 - 编写高效的源程序代码
 - 从而引导编译器也产生高效代码
 - 保证程序的可读性、模块化、可移植性等
 - 这样做虽不一定能达到最高性能，但好过直接用汇编编程

本课内容

■ 概述

■ 通用的优化方法

- 代码移动（预先计算）
- 复杂运算简化
- 共享公用子表达式
- 去除多余的过程调用

■ 优化障碍（妨碍优化的因素）


- 过程调用
- 存储器别名使用memory aliasing（不同名字指向相同内存）

■ 利用指令级并行

通用优化

- 指不考虑处理器型号与编译器版本，所有程序员（编译器）都应该做的优化
- 删减不必要的工作：函数调用、条件测试、内存引用等
- 代码移动
 - 减少计算的频率，满足以下前提：
 - 该计算总是产生相同的结果
 - 尤其注意可否将相应代码从循环中移出

```
void set_row(double *a, double *b,  
            long i, long n)  
{  
    long j;  
    for (j = 0; j < n; j++)  
        a[n*i+j] = b[j];  
}
```



```
    long j;  
    int ni = n*i;  
    for (j = 0; j < n; j++)  
        a[ni+j] = b[j];
```


编译器所作的代码移动 (-O1)

%rdi: 数组 a 首地址
 %rsi: 数组 b 首地址
 %rdx: 开始时为行号 i
 %rcx: 列数 n
 %rax: 列号 j

```

void set_row(double *a, double *b,
             long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
  
```

```

long j;
long ni = n*i;
double *rowp = a+ni;
for (j = 0; j < n; j++)
    *rowp++ = b[j];
  
```

```

set_row:
    testq    %rcx, %rcx
    jle      .L1
    imulq    %rcx, %rdx
    leaq     (%rdi,%rdx,8), %rdx
    movl     $0, %eax

.L3:
    movsd    (%rsi,%rax,8), %xmm0
    movsd    %xmm0, (%rdx,%rax,8)
    addq     $1, %rax
    cmpq     %rcx, %rax
    jne      .L3

.L1:
  
```

开始%rdx里存的是行号i
 # 检测n的值
 # 若为零, 结束
 # 计算ni = n*i存于%rdx
 # 计算行首地址, 存于%rdx
 # %eax清0, 则%rax也为0
 # movsd为浮点数传送指令,
 # 将double型数
 # 送至a[n*i+j]
 # j++
 # j与n比较
 # 不相等则循环
 # 结束循环
 # 返回, 见3.6.4旁注

rep ; ret

复杂运算简化reduction in strength

- 将复杂操作作用简单操作替换
- 用 shift 和 add 指令替代乘除法指令
 - $16 * x \rightarrow x \ll 4$
 - 实际效果依赖于机器
 - 取决于乘法或除法指令的时间成本
 - Intel Nehalem 机，一次整数乘法需要 3 个 CPU 周期
- 识别乘积的顺序

```
for (i = 0; i < n; i++) {  
    int ni = n*i;  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
}
```



```
int ni = 0;  
for (i = 0; i < n; i++) {  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
    ni += n;  
}
```

共享公用子表达式

- 重用表达式的一部分
- GCC 使用 -O1 选项实现此优化

```
%rsi: 开始 i → i*n
%rax: 开始 i+1 → (i+1)*n
%r8: 开始 i-1 → (i-1)*n
%rcx: 列数 n
%rdx: 列数 j
```

```
/* 将val[i,j]的四邻元素相加 */
up      = val[(i-1)*n + j];
down    = val[(i+1)*n + j];
left    = val[i*n      + j-1];
right   = val[i*n      + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up      = val[inj - n];
down    = val[inj + n];
left    = val[inj - 1];
right   = val[inj + 1];
sum = up + down + left + right;
```

3次乘法: $i*n$, $(i-1)*n$, $(i+1)*n$

```
leaq 1(%rsi), %rax      #
i+1
leaq -1(%rsi), %r8      # i-1
1
imulq %rcx, %rsi        # i*n
imulq %rcx, %rax        # (i+1)*n
imulq %rcx, %r8         # (i-1)*n
addq  %rdx, %rsi        # i*n+j
addq  %rdx, %rax        # (i+1)*n+j
```

1次乘法: $i*n$

```
imulq %rcx, %rsi        # i*n
addq  %rdx, %rsi        # i*n+j
movq  %rsi, %rax        # i*n+j
subq  %rcx, %rax        # i*n+j-n
leaq  (%rsi,%rcx), %rcx  # i*n+j+n
```

优化障碍 1: 过程调用

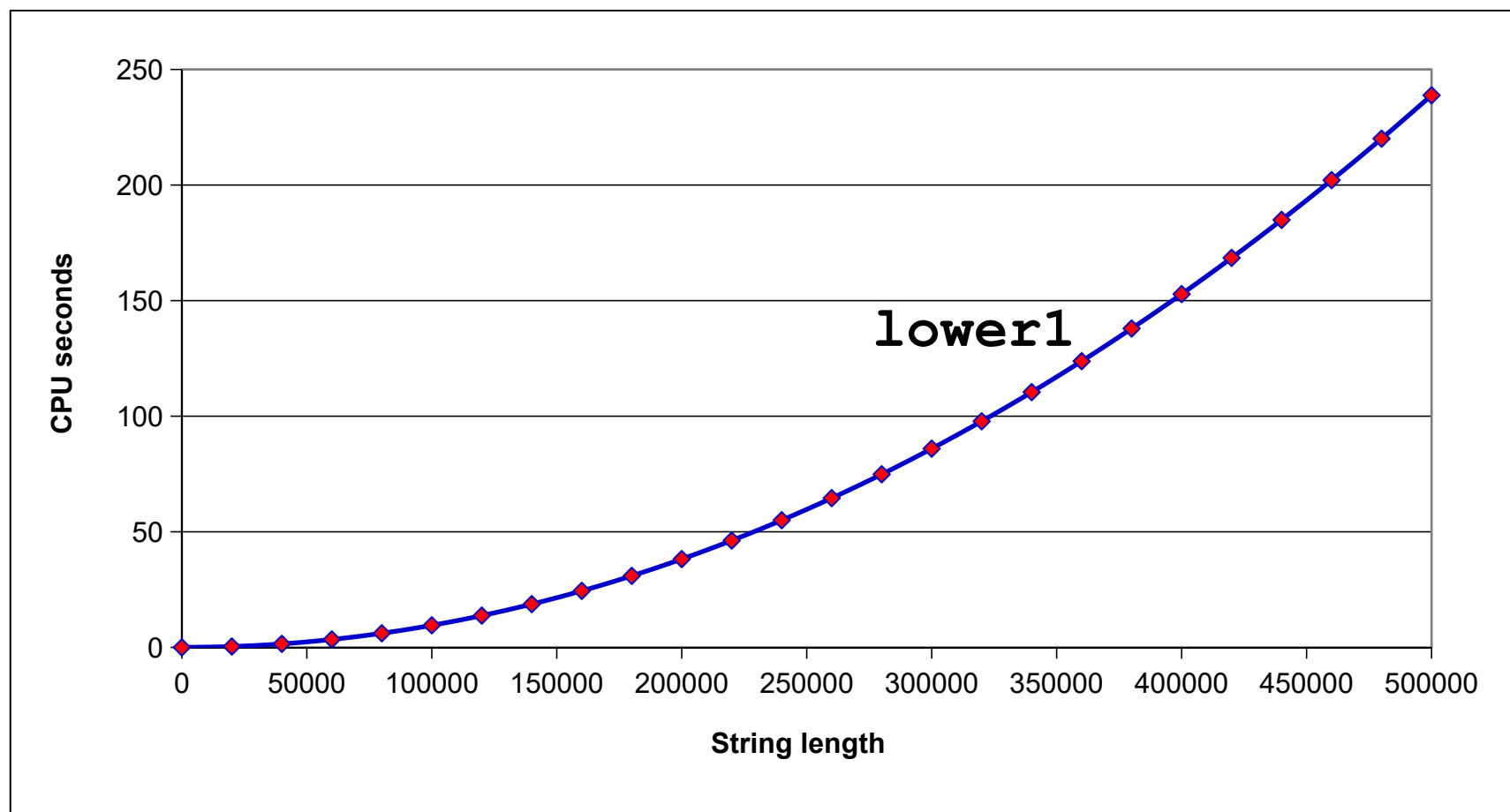
- 程序行为中严重依赖执行环境的方面，程序员要编写容易优化的代码，以帮助编译器
- 将字符串转换为小写的过程

```
void lower1(char *s)
{
    size_t i;           // size_t为无符号整型

    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

小写转换过程 `lower1` 的性能

- 当字符串长度翻倍时，时间翻两番（变为 4 倍）
- 其性能是串长度的二次幂



把循环变成 goto 形式——类汇编写法

```
void lower1(char *s)
{
    size_t i = 0;
    if (i >= strlen(s))
        goto done;
loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
done:
}
```

- **strlen** 每次循环都要重复执行

调用 strlen

```
/* strlen 的 CS:APP 版 */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

■ strlen 性能

- 确定字符串长度的唯一方法是扫描它的整个长度，查找 '\0' 字符

■ 整体性能，长度为 N 的字符串

- N 次调用 strlen
- 整体 $O(N^2)$ 性能

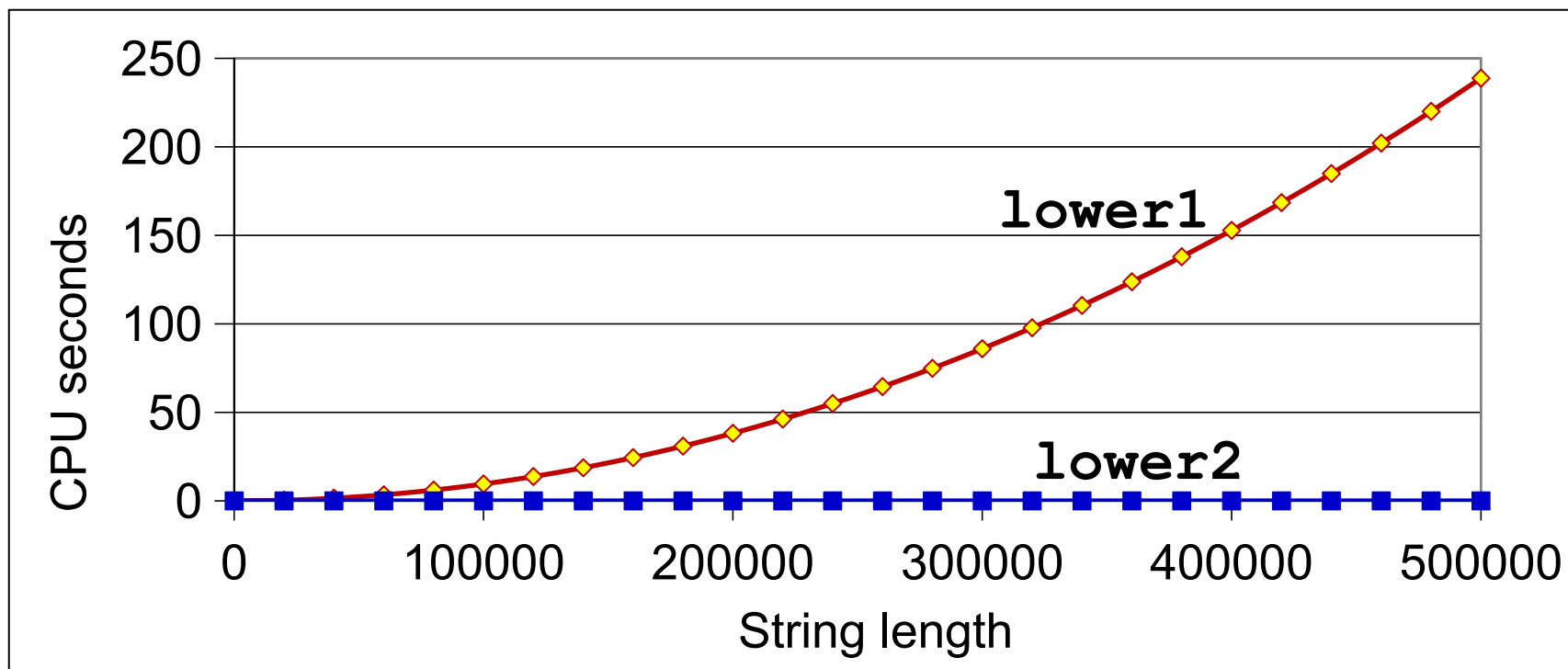
改进性能

```
void lower2(char *s)
{
    size_t i;
    size_t len = strlen(s);

    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- 把对 `strlen` 的调用移到循环外
- 依据：对于循环的每次迭代，结果都不会变化
- 此为代码移动的一种形式

lower2 的性能



- 字符串长度翻倍时，`lower2` 的时间也翻倍
- `lower2` 的性能为串长度的线性函数

本课内容

■ 概述

■ 通用的优化方法

- 代码移动（预先计算）
- 复杂运算简化
- 共享公用子表达式
- 去除多余的过程调用

■ 优化障碍（妨碍优化的因素）

- 过程调用
- 存储器别名使用Memory aliasing（不同名字指向相同内存）

■ 利用指令级并行

优化障碍 1: 过程调用

- 编译器为何不将 `strlen` 从内层循环中移出?
 - 过程可能会有副作用
 - 例如: 每次调用时都会改变全局变量 (状态)
 - 对于给定的参数, 同一函数也可能返回不同的值
 - 依赖于某些全局变量 (状态)
 - 谁能保证 `lower` 过程不会与 `strlen` 存在某种关联?
- **特别注意:**
 - 编译器将过程调用视为黑盒
 - 涉及过程调用只进行弱优化
- 补救措施:
 - 使用内联函数
 - GCC 的 `-O1` 参数实现该操作但仅局限于单一文件之内
 - 程序员自己完成代码移动

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

内存的重要性

```
/* 本函数将 n X n 矩阵 a 的各行求和，并存至向量 b */
void sum_rows1(double *a, double *b, long n)
{
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

sum_rows1 过程的内循环

.L4:

```
movsd    (%rsi,%rax,8), %xmm0
addsd    (%rdi), %xmm0
movsd    %xmm0, (%rsi,%rax,8)
addq     $8, %rdi
cmpq     %rcx, %rdi
jne      .L4
```

```
# %rsi: b
# %rax: i
# 浮点读内存
# 浮点加
# 浮点写内存
```

move 双精度浮点数 (XMM 寄存器的低 64 位)
addsd 类似

- 每次循环都要更新 `b[i]` — 不停读出、写回
- 编译器为何不进行精简？原因：**内存别名使用**

内存别名使用——试考虑如下调用

```
/* 本函数将 n X n 矩阵 a 的各行求和，并存至向量 b */
void sum_rows1(double *a, double *b, long n)
{
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
{ 0,    1,    2,
  4,    8,   16,
 32,   64,  128 };

sum_rows1(A, A+3, 3); // B = A+3
```

B 数组的值:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

- 代码每次循环都会更新 b[i]
- 必须想到这些更新可能会对程序行为造成影响

移除内存别名使用

```
/* 本函数将 n x n 矩阵 a 的各行求和，并存至向量 b */
void sum_rows2(double *a, double *b, long n)
{
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

sum_rows2 过程的内循环

.L10:

```
    addsd    (%rdi), %xmm0
    addq     $8, %rdi
    cmpq     %rax, %rdi
    jne      .L10
```

%xmm0: val
浮点读 + 加

- 无需将中间结果存入内存

B 的结果: [3, 28, 224]

优化障碍 2：内存别名使用

- 别名使用aliasing
 - 两个不同的内存引用指向同一位置
 - 在 C 程序中很常见：
 - 因为可以做地址运算
 - 可以直接访问存储结构
 - 编译器不知道函数何时被调用，会不会在别处修改了内存，特别是当程序并行化，或者改变了执行顺序之后
 - 编译器的保守方法是不断地读和写内存（虽然效率不高）
 - 养成引入局部变量的习惯
 - 在循环中进行累加——用寄存器实现
 - 程序员借此告知编译器无需担心内存别名使用问题

本课内容

■ 概述

■ 通用的优化方法

- 代码移动（预先计算）
- 复杂运算简化
- 共享公用子表达式
- 去除多余的过程调用

■ 优化障碍（妨碍优化的因素）

- 过程调用
- 存储器别名使用Memory aliasing（不同名字指向相同内存）

■ 利用指令级并行

利用指令级并行ILP

- 需要对现代处理器的设计有一般性的了解
 - 硬件可以并行执行多条指令
- 其性能受到数据相关的限制
- 对代码简单的转换便可显著地提高性能
 - 但是，经常看到编译器无法做这样的调整
 - 比如：由于浮点运算不满足结合率和分配率
 - 例： $3.14 + (1e20 - 1e20) \neq (3.14 + 1e20) - 1e20$
 - 例： $1e300 * (1e300 - 1e300) \neq 1e300 * 1e300 - 1e300 * 1e300$
- 因此，编译器不会做以下优化：

