《编译系统原理》预习作业 1 *

姓名：郭坤昌　　学号：2012522　　专业：计算机科学与技术

September 16, 2022

C 程序编程和优化

题目：对于如下两个程序，为了获得运行速度更快的程序，你会选择哪种编程方式?

```
/* LOOP1 */
for (int i = 0; i < N; i++)
{
    a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
}
```

```
/* LOOP2 */
b = a;
for (int i = 0; i < N; i++)
{
    *b = *b * 2000;
    *b = *b / 10000;
    b++;
}
```

答：两种编程方式的区别在于，循环 1 使用下标寻址，循环 2 使用指针寻址。对于下标寻址，需要计算偏移量（i«3，使用 double 类型，8 个字节），再与基址（a 的地址）相加，才能得到 a[i] 的地址；对于指针寻址，每次指针递增即可，i 作为计数使用。从运算量看，使用指针寻址似乎更快。

实验方案

实验平台：Ubuntu22.04，x86 Intel i7-8565U @ 1.80 GHz 4 核 8 线程，8GB RAM

*

N	100	10 000	1 000 000	100 000 000
loop1	3.16E-07	3.11E-05	0.0034241	0.378583
loop2	2.85E-07	2.53E-05	0.00292545	0.299193

Table 1: 不同规模测量结果（时间：s）

Opt-level	O0	O1	O2	O3
loop1	0.378583	0.187991	0.174203	0.150666
loop2	0.299193	0.172342	0.164162	0.132263

Table 2: 不同优化等级测量结果（时间：s）

测试方案：改变 N 的大小、编译器优化等级、编译器种类进行测试，探究对应改变对性能的影响；通过汇编观察编译器具体行为

测试代码及测量记录数据：C 程序编程和优化实验文件

实验结果

不同规模：使用 GNU 编译器，在不开启优化的条件下，改变 N 的大小，得到测量结果如表1所示。观察到，loop2 运算速度快于 loop1。在实验的最大规模下，约快 21%。

不同优化等级：在 N=100 000 000 条件下，改变编译器优化等级分别为 O0、O1、O2、O3 进行测量（还有其他优化选项，如 Os，这里暂不进行测试），结果如表2。观察到开启优化，对性能的提升是十分显著的。对 loop1，开启优化后，O1 取得了 201% 的加速比，随着优化等级提升，加速效果渐不显著，且两种方式的性能很接近。

不同编译器：在 N=100 000 000 条件下，使用 GNU 和 Clang 分别在优化等级为 O0 和 O3 时进行测量，结果如表3。Clang 编译器生成的可执行文件，相比 GNU，执行性能更好，在相同优化等级下均有不同的提升，尤其在 O0 和 O3 时，对第一种循环的性能提升较大。

汇编分析

以下根据 Intel 格式的汇编代码分析不同优化等级和编译器下时对应程序的具体执行行为。

Compiler	GNU		Clang	
Opt-level	O0	O3	O0	O3
Loop1	0.378583	0.150666	0.277596	0.115467
Loop2	0.299193	0.132263	0.253883	0.12565

Table 3: 不同编译器测量结果（时间：s）

GUN 编译器在优化等级为 O0 时，Loop1 产生的汇编代码主要部分如下。

```
        mov     DWORD PTR -4[rbp], 0
        jmp     .L12
.L13:
        mov     rdx, QWORD PTR a[rip]
        mov     eax, DWORD PTR -4[rbp]
        cdqe
        sal     rax, 3
        add     rax, rdx
        movsd   xmm1, QWORD PTR [rax]
        mov     rdx, QWORD PTR a[rip]
        mov     eax, DWORD PTR -4[rbp]
        cdqe
        sal     rax, 3
        add     rax, rdx
        movsd   xmm0, QWORD PTR .LC1[rip]
        mulsd   xmm0, xmm1
        movsd   QWORD PTR [rax], xmm0
        mov     rdx, QWORD PTR a[rip]
        mov     eax, DWORD PTR -4[rbp]
        cdqe
        sal     rax, 3
        add     rax, rdx
        movsd   xmm0, QWORD PTR [rax]
        mov     rdx, QWORD PTR a[rip]
        mov     eax, DWORD PTR -4[rbp]
        cdqe
        sal     rax, 3
        add     rax, rdx
        movsd   xmm1, QWORD PTR .LC2[rip]
        divsd   xmm0, xmm1
        movsd   QWORD PTR [rax], xmm0
        add     DWORD PTR -4[rbp], 1
.L12:
        cmp     DWORD PTR -4[rbp], 99999999
        jle     .L13
        nop
        nop
```

```
38          pop     rbp
39          .cfi_def_cfa 7, 8
40          ret
```

Loop1 所做行为正如之前所分析，首先计算 i 导致的偏移量，再与 a 的基地址相加得到 a[i] 地址。其中较为繁琐的部分，在与计算 a[i] = a[i] * 2000 和 a[i] = a[i] / 10000 时，均分别对 a[i] 地址进行了两次运算，即对每个表达式，都计算了操作数 a[i] 地址和结果 a[i] 地址。其中一些细节在于使用了 128 位寄存器 xmm 等将 double 型变量存在了高 64 位，且取 a 地址是 32 位，需要扩展到 64 位。这一部分对应于求 a[i] 地址时，基址与偏移量需要是同一类型（位数应当对齐），属于语义分析完成。

相比之下，此时的 Loop2 产生的汇编代码量小很多。相同点在于同样取了 4 次 a[i]，但每次迭代需要对用以遍历的指针加 1。因为省去了计算地址的代码，因此计算效率更快。

O1 时，Loop1 有了较大的变化，代码如下：

```
1  mov     eax, 0
2           movsd   xmm2, QWORD PTR .LC2[rip]
3           movsd   xmm1, QWORD PTR .LC3[rip]
4  .L18:
5           mov     rdx, rax
6           add     rdx, QWORD PTR a[rip]
7           movapd  xmm0, xmm2
8           mulsd   xmm0, QWORD PTR [rdx]
9           movsd   QWORD PTR [rdx], xmm0
10          mov     rdx, rax
11          add     rdx, QWORD PTR a[rip]
12          movsd   xmm0, QWORD PTR [rdx]
13          divsd   xmm0, xmm1
14          movsd   QWORD PTR [rdx], xmm0
15          add     rax, 8
16          cmp     rax, 800000000
17          jne     .L18
18          ret
```

改进部分在于，直接将 2000 和 10000 存进 128 位浮点寄存器的高位中，i 加 1 变为 i+8，最终直接与 800 000 000 比较，省去了计算真正偏移量的过程。在计算过程中，不将 a[i] 直接取出，而是隐含在了计算过程中（可能是 Intel 指令集的强大和复杂之处），这样似乎减少了直接访问的次数。

对于此时的 Loop2，则使用另一种思路。开始时即计算好最终停止的地址 (a[100 000 000])，这样当 a[i] 超过最终地址时，即退出循环。

O2 下，Loop1 和 Loop2 都使用相同的行为，相对于 O1 的改进在于计算的过程变为连续的，省去了重复取出 a[i] 的过程。退出循环的判断规则与 O1 下的 Loop2 相同，即确定最终退出地址。

O3 下，Loop1 和 Loop2 也使用相同的行为，相对于 O2 引入的改进在于使用了 sse 指令集，即一次取出了两个浮点数，这样 128 位寄存器就被充分利用起来，即 SIMD 的加速思路。

clang 编译器的代码行为与 GNU 编译器略有不同。GNU 的循环判断类似于 do-while 型，而 clang 的循环则类似于 while 类型。在 O0 时，Loop1 和 Loop2 对应与在 GNU 下的行为基本相同。而在 O3 下，clang 不仅使用了 sse 指令集，还使用了循环展开的技术，一次循环完成两次的运算量。代码如下：

```
# %bb.0:
        mov     eax, 2
        mov     rcx, qword ptr [rip + a]
        movapd  xmm0, xmmword ptr [rip + .LCPI3_0] # xmm0 = [2.0E+3,2.0E+
        movapd  xmm1, xmmword ptr [rip + .LCPI3_1] # xmm1 = [1.0E+4,1.0E+
        .p2align        4, 0x90
.LBB3_1:                                # =>This Inner Loop Header: Depth=
        movupd  xmm2, xmmword ptr [rcx + 8*rax - 16]
        movupd  xmm3, xmmword ptr [rcx + 8*rax]
        mulpd   xmm2, xmm0
        divpd   xmm2, xmm1
        movupd  xmmword ptr [rcx + 8*rax - 16], xmm2
        mulpd   xmm3, xmm0
        divpd   xmm3, xmm1
        movupd  xmmword ptr [rcx + 8*rax], xmm3
        add     rax, 4
        cmp     rax, 100000002
        jne     .LBB3_1
# %bb.2:
        ret
```

分词、构造语法树

题目： 对数据库查询表达式 Model = ”civic” AND Year = ”2001” 进行分词、构造语法树

答： 将 SQL 语句 Model = ”civic” AND Year = ”2001” 进行词法分析，得到如图1所示

分词结果。

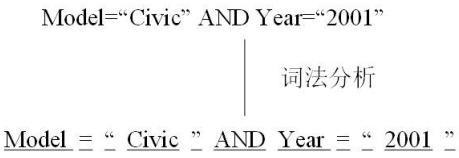

Figure 1: SQL 语句分词结果

进一步构造语法树，如图2

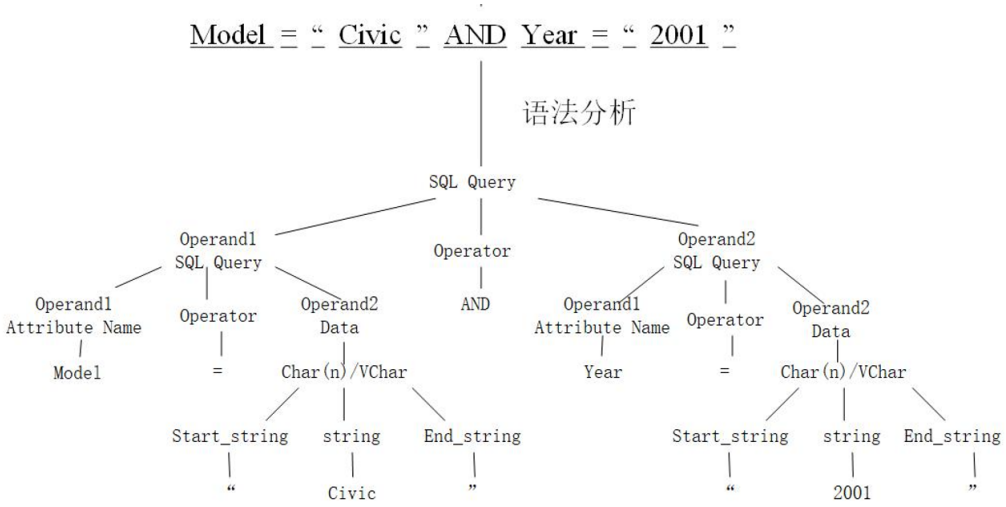

Figure 2: 构造 SQL 语句对应语法树

静态检查

题目 1：如下程序有什么错误？鼓励使用静态代码检查工具检查

```
1 char firstChar1(char* s)
2 {
3     return *s;
4 }
```

答：该程序没有对输入的字符指针是否为空进行检查。若 s 是空指针，则 *s 无法指向有效数据。

使用 splint 分析结果表示程序没有问题，如图3。只有当在 main 函数中，调用以空指针作为参数的 firstChar1 函数时，才检查出空指针的错误使用。

```
bill@linux-Ubuntu:~/Desktop/complile/compilation/pre-learning/work1/task3$ spli
nt 1.c
Splint 3.1.2 --- 21 Feb 2021

Finished checking --- no warnings
```

Figure 3: splint 静态检查结果

题目 2：如下程序有什么错误？鼓励使用静态代码检查工具检查

```
1  int* glob;
2  int* f(int** x)
3  {
4      int sa[2] = { 0, 1 };
5      int loc = 3;
6      glob = &loc;
7      *x = &sa[0];
8      return &loc;
9  }
10 void h(void)
11 {
12     unsigned int i;
13     if (i >= 0)
14         printf(">=0\n");
15     else printf("<0");
16 }
```

答：首先分析，函数 f 中 sa 数组、loc 是临时变量，函数调用完毕后，返回值和 glob 都指向原栈帧中的位置，因此将被悬挂；另外，x 也必须是非空指针。

使用 splint 分析，总共得到了 5 条警告，如图4：

1. 返回的悬挂指针并不指向一个有效的数据地址
2. 指向立即数的地址被隐式返回，不满足一致性。可能是因为函数执行情况不同，使得 loc 初始化的地址不同，因此不是每次都返回相同地址
3. *x 被函数栈帧分配存储空间，它所指向的，也是一个栈帧中的地址
4. 全局变量 global 指针同样被悬挂
5. glob 作为全局变量，在不被其他外部模块调用时，应当用 static 进行限定

对于函数 h，i 在使用前没有被定义，另外无符号数与 0 进行大于或等于比较是恒成立的，分支是唯一确定的。

使用 splint 分析，得到 2 条警告，如图4：

1. 无符号数与 0 比较（这里应该指的就是大于或等于 0 的比较）是迷惑操作或是 bug
2. i 在使用前没有被定义

可以看出，splint 的静态检查结果基本准确，对变量（全局、临时）进行了地址、类型、使用的检查，其警告有助于构建更安全的程序。

```
2.c: (in function f)
2.c:11:12: Stack-allocated storage &loc reachable from return value: &loc
  A stack reference is pointed to by an external reference when the function
  returns. The stack-allocated storage is destroyed after the call, leaving a
  dangling reference. (Use -stackref to inhibit warning)
2.c:11:12: Immediate address &loc returned as implicitly only: &loc
  An immediate address (result of & operator) is transferred inconsistently.
  (Use -immediatetrans to inhibit warning)
2.c:11:17: Stack-allocated storage *x reachable from parameter x
    2.c:10:5: Storage *x becomes stack-allocated storage
2.c:11:17: Function returns with global glob referencing released storage
  A global variable does not satisfy its annotations when control is
  transferred. (Use -globstate to inhibit warning)
    2.c:11:12: Storage glob released
2.c: (in function h)
2.c:16:9: Comparison of unsigned value involving zero: i >= 0
  An unsigned value is used in a comparison with zero in a way that is either a
  bug or confusing. (Use -unsignedcompare to inhibit warning)
2.c:16:9: Variable i used before definition
  An rvalue is used that may not be initialized to a value on some execution
  path. (Use -usedef to inhibit warning)
2.c:4:6: Variable exported but not used outside 2: glob
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (Use -exportlocal to inhibit warning)

Finished checking --- 7 code warnings
```

Figure 4: splint 静态检查结果

语法描述

题目：在变量声明中，允许任意多个标识符形成的标识符列表，如 int a, b, ···, x;
用自然语言描述标识符列表的递归定义**答：**参照表达式的递归定义：
1. 标识符是表达式
2. 数是表达式
3. 若 expression1 和 expression2 是表达式，则 expression1+expression2, expression1*expression2, (expression1) 是表达式

其中，1，2 是基本规则，3 是递归定义，这样使得表达式能够从小到大地被构造出来。
因此，可以定义如下标识符列表的递归定义：
1. 如下的形式是标识符列表

$$< \text{类型说明符} > \text{标识符};$$

2. 若 list1 是标识符列表，list2 也是标识符列表，则如下形式也是标识符列表

$$\text{list1(去掉分号),list2(去掉类型说明符)}$$

其中，list1 和 list2 需具有相同类型说明符