反汇编在常数因子优化中的应用

四川省成都七中 周以苏

绪言

- 程序优化是无止境的，其中常数因子也是决定程序运行快慢的关键之一。
- 然而在竞赛中，渐进时间复杂度是人们关注的重点，而同样能够决定程序运行快慢的常数因子优化问题却缺乏重视。
- 在 **Visual C++** 语言环境下，从特定编译器生成的汇编代码出发，我探讨了反汇编在常数因子优化中的应用，并提出了若干优化改进方案。

引例：关于 **memset** 函数的小实验

- 已知 memset 函数为 O(N) 复杂度的语句。

- 观看右边的 C++ 程序（假设计算机具有足够大的内存）

- 你能直接答出 Time 值与运行速度的关系？

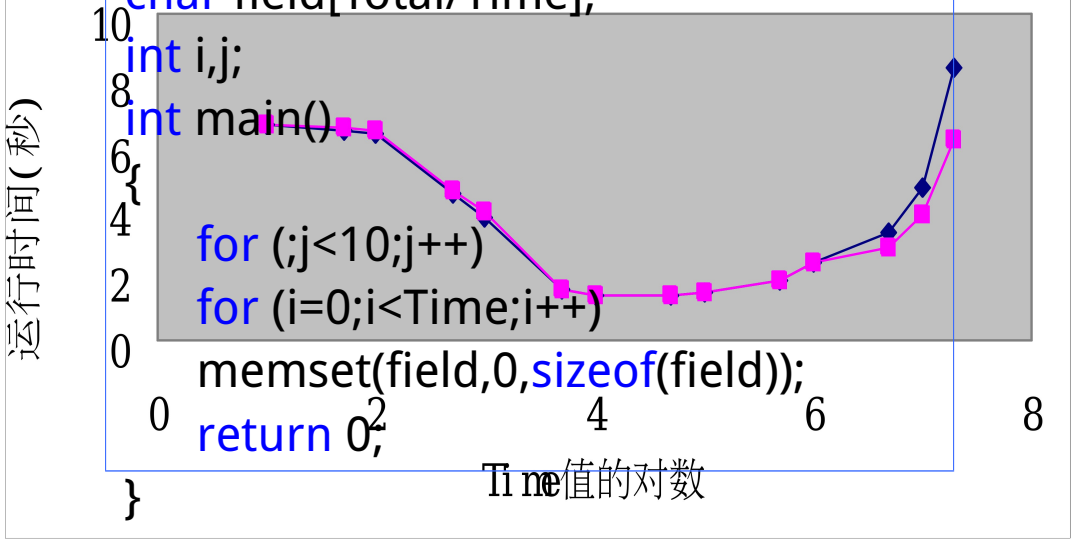
```
#include <string.h>
const int Total=1000000000;
const int Time= 你喜欢的合法的数值 ;
char field[Total/Time];
int i,j;
int main()
{
    for (;j<10;j++)
    for (i=0;i<Time;i++)
    memset(field,0,sizeof(field));
    return 0;
}
```

分析

Time 值与运行速度的关系

可能你认为 **Time** 值不影响程序时间复杂度，因此对程序的速度无影响。

```
#include <string.h>
const int Total=1000000000;
const int Time=你喜欢的合法的数值;
char field[Total/Time];
int i,j;
int main()
{
  for (;j<10;j++)
  for (i=0;i<Time;i++)
  memset(field,0,sizeof(field));
  return 0;
}
```



但是，当上机实验后，你会发现，**Time** 值较大或较小时运行速度会变慢，这是为什么呢?

分析

- Debug 模式下编译器对 memset 语句的处理如下

```
memset(field,0,sizeof(field));
```

```
00411A6B  push     2710h
00411A70  push     0
00411A72  push     offset field
00411A77  call   @ILT+350(_memset)
00411A7C  add  esp,0Ch
```

- Release 模式下编译器对 memset 语句的处理如下

```
memset(table,0,sizeof(table));
```

```
00401001  xor eax,eax
00401003  mov ecx,9C40h
00401008  mov edi,offset table
0040100D  rep stos dword ptr [edi]
```

分析

```
memset(field,0,sizeof(field));
```

```
00411A6B  push    2710h
00411A70  push    0
00411A72  push    offset field (4284E8h)
00411A77  call    @ILT+350(_memset)
00411A7C  add     esp,0Ch
```

```
memset(table,0,sizeof(table));
```

```
00401001  xor eax,eax
00401003  mov ecx,9C40h
00401008  mov edi,offset table
0040100D  rep stos dword ptr [edi]
```

Windows 分配内存

第一层循环

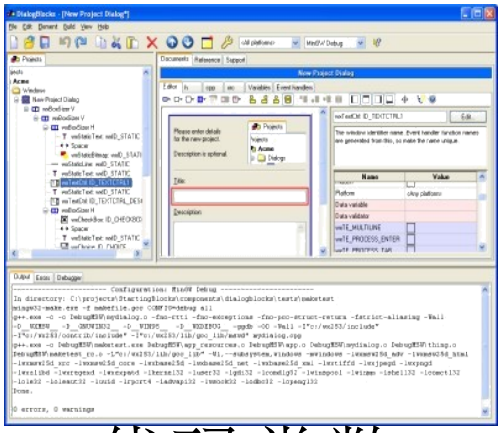
第二层循环

额外汇编语句
(push call ret mov xor)

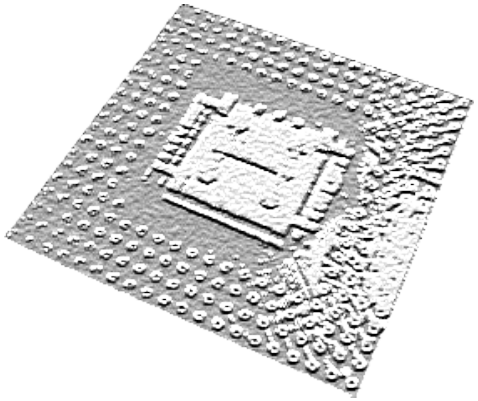
真正作业
(rep stos)

思路

常数因子优化



代码常数因子优化



本质常数因子优化

高级语言 → 机器代码

应用思想 ← 反汇编

时间常数归类

层次	运算	比例时间
1.	**mov**， **lea** 数据移动运算	1
	and or xor not 逻辑运算	
	add， **sub** 加减法运算， **test** 运算	
2.	**shl**， **shr**， **sal**， **sar** 位运算	1.5~2
3.	**ptr** 取地址值， **push+pop** 堆栈运算 ***2**， **jmp** 跳转运算	4
4.	**mul**， **imul** 乘法	5
5.	**div**， **idiv** 除法	25
6.	**call+ret** 调用子函数 + 返回	27

一、关于调用常数因子的优化

- 调用常数因子是指在函数调用过程中 push pop （有的编译器如 VS.NET 的 cl 用 mov 实现）和 call ret 等汇编伪代码在调用过程中的耗费。
- 虽然调用过程在 Release 模式下会被自动优化，但是在某些只提供 Debug 模式的竞赛环境中，我们该如何优化？所以本文主要阐述在 Debug 模式下的调用常数因子优化。

1、Debug 模式

- 我们常使用 inline 关键字对代码进行优化，但是， inline 关键字对编译器的作用是提示性质的而不是强制性质的。

测试调用的函数原形：

```
inline void swap (int&a,int&b) { int t=a; a=b; b=t;}
```

测试代码：

```
swap(a,b);
004133AD  lea     eax,[b]
004133B0  push    eax
004133B1  lea     ecx,[a]
004133B4  push    ecx
004133B5  call    swap (41158Ch)
004133BA  add     esp,8
```

1、Debug 模式

- 所以，在竞赛中应针对这个问题进行优化，这里本文提供了两种替代方案：

```
int tmp;
#define swap(A,B)
    tmp=A,A=B,B=tmp
int main()
{
    int a=3,b=4;
    swap(a,b);
    return 0;
}
```

1、不使用子函数

2、使用宏定义

2 、 Release 模式

- 与 Debug 模式不同的是，在 Release 模式下，任何函数会被优先尝试作为 inline 函数，即使没有显式指定 inline 关键字，inline 关键字仍然没有实际的作用。

正因为这样，在 Debug 模式和 Release 模式下，stl 库的运行效率才会有巨大的差别。

测试

```
void swap(int&a,int&b){int t=a; a=b; b=t;}
```

尽管没有 inline 关键字，在反汇编中已经看不到对 swap 的调用了

```
                a++;
0040105B  add     eax,1
                swap(a,b);
0040105E  mov     dword ptr [esp+8],eax
                a*=b;
                printf("%d%d\n",a,i);
                return 0;
00401062  imul    eax,ecx
```

二、除法（求余）的优化（预备）

- 预备知识：
- 求余运算 **c=a%b** 等效于 **c=a-a/b*b** 但是，其内部实现直接使用除法的第二个返回值：

```
    a%=b;
00411B53  mov   eax,dword ptr [a]
00411B56  cdq
00411B57  idiv  eax,dword ptr [b]
00411B5A  mov   dword ptr [a],edx
```

二、除法（求余）的优化

- 除法指令 idiv 是一种比例时间很大的指令。编译器的设计者也知道这一点。所以大多数情况下编译器都能将常数除法转化为快得多的位运算。

（注：编译器同样也会把特定的乘法转化为位运算，比如乘以 2 等）

二、除法（求余）的优化

- 比如，对于 **a/=2** （**a** 为 32 位整数）这句语句在 **Debug** 模式下的解释：
- 这里对编译器的智能作了很大的局限☹，比如除法需要额外的特殊情况判断，无法判断数量的特殊性这在代码中直接反映为了余数的运算代码。所以这样运算直接可用移位运算代替，潜在的性能优化就没有被用到。

  因为，对于除法 **idiv** 操作要比其他操作慢很多。但是，[illegible] 量除数是时，编译器根本无法知道除数的特殊性，使任何编译器直接将语句翻译为 **div(idiv)** 操作。如果除数是常量的特殊性，推荐使用优化后 [illegible]

a/2

```
00411B4F  mov    eax,dword ptr [a]
00411B52  cdq
00411B53  sub    eax,edx
00411B55  sar    eax,1
00411B57  mov    dword ptr [a],eax
```

二、除法（求余）的优化

正确的方法是，
判断出特殊性，
使用手工的优化方式，
如：

原始代码：

```
const
a[]={1,2,4,8,16,32,64,128,256,512,1024,2048,4096};
    c=b%a[i];
    d=e/a[i];
```

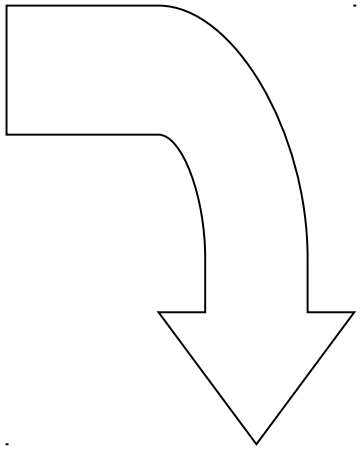
优化后的代码：

```
const
a[]={1,2,4,8,16,32,64,128,256,512,1024,2048,4096};
    c=b&(a[i]-1);
    d=e>>(i-1);
```

三、关于多维数组的性能优化

- 由于计算机内存是线性的，多维数组的元素在排列为线性序列后存入存储器，如下所示：

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3



0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3	3,0	3,1	3,2	3,3

三、关于多维数组的性能优化

- 由于在结构上需要进行转换，多维数组的引用操作被翻译成乘法操作，
  - 数组定义： a[10][10]
  - Debug 模式下，对数组的取值操作使用了 imul 运算（ Release 模式编译时会进行和乘法运算相同的优化）：

```
            return a[i][j];
00411B6F  mov     eax,dword ptr [i]
00411B75  imul    eax,eax,28h
00411B78  lea     ecx,a[eax]
00411B7F  mov     edx,dword ptr [j]
00411B85  mov     eax,dword ptr [ecx+edx*4]
```

三、关于多维数组的性能优化

- 由于 imul 是一种比例时间较大的指令，如果能消去这一指令，便能够产生较大幅度的优化。

```
                return a[i][j];
00411B6F  mov   eax,dword ptr [i]
00411B75  imul  eax,eax,28h
00411B78  lea   ecx,a[eax]
00411B7F  mov   edx,dword ptr [j]
00411B85  mov   eax,dword ptr
                [ecx+edx*4]
```

- 如果操作的变址方法固定（比如像宽度优先搜索，变址操作为 +1,-1,+N,-N），那么用指针加减操作以及辅助记录就能获得更快的速度（消去了乘法操作）。

三、关于多维数组的性能优化

定义表和指针：

```
int table[200][200];
int*ptr,*ptr2;
```

定义滑动常数：

```
//East,South,West,North
const go[]={1,200,-1,-200};
```

```
        // 假设 ptr 已赋值
        ptr2=ptr+go[0];
00411A4C  mov       eax,dword ptr [go]
00411A51  mov       ecx,dword ptr [ptr]
00411A57  lea       edx,[ecx+eax*4]
00411A5A  mov       dword ptr [ptr2],edx
        return *ptr2;
00411A60  mov       eax,dword ptr [ptr2]
00411A65  mov       eax,dword ptr [eax]
```

- 这样本来隐藏的乘法操作就被消去了。

三、关于多维数组的性能优化

- 这种操作被我称为指针的“行走”操作。使用这个优化有个条件，就是指针变化方式固定。
- 让我们通过一个例子来了解这种优化的作用。

例： adv1900 (NOI2005)

- 题意描述：

  在 N*M 的矩阵中，有一些障碍，有一个物体放在某个格子上。它会按照一个时间表向某一方向运动，一个时间单位移动 1 格。某一秒你可以让它运动，也可以让它静止。问物体最多能运动的长度。

  时间表由很多个时间片段构成，在每个时间片断中，物体将向同一方向运动。

- 数据规模：

  50% 的数据中， 1≤N, M≤200 ， **时间长度 (T) ≤ 200** ；

  100% 的数据中， 1≤N, M≤200 ， 时间片段个数 (K)≤200 ， **时间长度 (T) ≤ 40000** 。

例： adv1900 (NOI2005)

- 这道题有很多做法，其中最优做法是使用单调性降维。
- 无论用什么方法，都必经一个关键步骤，这就是在不同的时间点间进行状态转移，并且，都要将这一步“批处理”化。
- 最优做法的单调性降维，以及其他各式各样的优化，如堆和 RMQ 等，都是基于对这一步骤的渐进时间复杂度的优化。

例： adv1900 (NOI2005)

- 但是，利用“行走”操作，我们完全可以另辟蹊径。
- 基于此步骤具有的使用优化的典型特点：

  （1）位于循环最里层，直接影响运行速度；

  （2）大量使用对数组的变化方式固定的操作，可以用指针“行走”来优化。
- 虽然最终还是使用“批处理化”的思想，但是这种方法没有把精力用在渐进复杂度的优化上，而转向到了具体的实现上。

例： adv1900 (NOI2005)

- 本题的移动情况可以靠在移动前进行对变量的初始化实现。
- 在某个时间段中对前面位置的询问可以用反方向“行走”实现。
- 对于取址运算中的位运算，可以用强制转换指针的方法消去。
- 对障碍判断的实现可以用统一变量格式实现。

例： adv1900 (NOI2005)

- 下表展现了此方法与非“行走”优化方法的速度对比 (Debug 模式)

表 **3**： “行走”优化方法

选手名称： withWalk

试题： adv1900 文件名： adv1900

编号	评测结果	时间	内存
0	正确	0.016s	520KB
1	正确	0.016s	520KB
2	正确	0.016s	520KB
3	正确	0.047s	520KB
4	正确	0.234s	520KB
5	正确	0.703s	520KB
6	正确	0.547s	520KB
7	正确	0.734s	520KB
8	正确	0.484s	520KB
9	正确	0.797s	520KB

本题总得分 100， 有效用时 3.594s 。

表 **4**： 非“行走”优化方法

选手名称： withoutWalk

试题： adv1900 文件名： adv1900

编号	评测结果	时间	内存
0	正确	0.016s	520KB
1	正确	0.016s	520KB
2	正确	0.016s	520KB
3	正确	0.063s	520KB
4	正确	0.344s	520KB
5	超时	**1.016s**	**520KB**
6	正确	0.781s	520KB
7	超时	**1.031s**	**520KB**
8	正确	0.625s	520KB
9	超时	**1.156s**	**520KB**

本题总得分 70， 有效用时 5.064s 。

总结

- 汇编语言具有高速、高效的特点，并且它的细微差异，都会导致程序运行速度的一定的变化。
- 上述几个实例展现了反汇编在常数因子优化中的应用。
- 我相信，对汇编程序的分析与比较，能够使程序运行速度进一步提高，从而更快更好的解决实际问题。

Thank you