搜狗实验室技术交流文档

Vol.1:2 乱序优化与GCC 的Bug

摘要

乱序优化是现代编译器非常重要的特性，本文介绍了什么是乱序优化，以及由此引发的一个 gcc bug，希望引起各位开发者的注意。

乱序优化

乱序优化和 cpu 的乱序执行很类似。

现代 CPU 都采用流水线结构，流水线的各级可以同时执行不同的指令，也只有用多条指令将流水线填满以后，CPU 的能力才能得到充分发挥。

乱序执行（out-of-orderexecution），是指 CPU 允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理的技术。这样将根据各个电路单元的状态和各指令能否提前执行的具体情况分析后，将能提前执行的指令立即发送给相应电路单元执行，在这期间不按规定顺序执行指令，然后由重新排列 单元将各执行单元结果按指令顺序重新排列。采用乱序执行技术的目的是为了使 CPU 内部电路满负荷运转并相应提高了 CPU 的运行程序的速度。

由于 CPU 流水线的指令预取范围有限，所以只能在很小的范围内判断指令是否能够并发，如果相隔比较远的指令才可以并发就无能为力了。编译器可以分析相当长的一段代码从而把能够并发的代码尽量靠近，这就是所谓的乱序优化。

乱序优化的关键在于编译器能够正确的识别哪些代码能够并发，如果发生误判就会导致不可预见的 bug。一般情况下都没有问题——但是如果你写出一些太过诡异的代码就很难讲了。

下面我们来看看一个优化错误实例。

实例程序

buggy.c

```
#include <stdio.h>
#include <stdlib.h>

typedef unsigned short UINT16;
typedef unsigned int UINT32;
```

```
struct EndPoint {
    UINT16 tcpPort_;
    UINT16 udpPort_;
    //UINT32 ipAddress_;
};

inline UINT32 EndPointToUInt32(struct EndPoint* ep) {
    return *(const UINT32*)(ep); // bug here
}

struct EndPoint endpoint = { 0x8080, 0x1080 };

int main() {
    //下句在inline+乱序优化时出错
    endpoint.udpPort_ = 0;
    UINT32 tmp2 = EndPointToUInt32(&endpoint);
    //UINT32 tmp2 = *(const UINT32*)(&endpoint); // 用这一句替换上一句同样出错!
    srand(tmp2); // for break the optimize
    printf("%08x %08x should be same as 00008080\n", tmp2, EndPointToUInt32(&endpoint));
}
```

运行结果如下：

[@66.209 ~]# gcc buggy.c
[@66.209 ~]# ./a.out
00008080 00008080 should be same as 00008080
[@66.209 ~]# gcc -O2 buggy.c
[@66.209 ~]# ./a.out
10808080 00008080 should be same as 00008080

可以看到打开优化之后 EndPointToUInt32 这个函数的第一次执行就不正常了。

分析

粗略的分析一下目标码

gcc 直接编译的结果		替换掉函数调用后的结果		gcc -O2编译的结果	
movw	$0, endpoint+2	movw	$0, endpoint+2	movl	endpoint, %ebx
pushl	$endpoint	movl	endpoint, %eax	subl	$28, %esp
call	EndPointToUInt32	movl	%eax, -4(%ebp)	pushl	%ebx
addl	$4, %esp	subl	$12, %esp	movw	$0, endpoint+2
movl	%eax, -4(%ebp)	pushl	-4(%ebp)	call	srand
subl	$12, %esp	call	srand	addl	$12, %esp
pushl	-4(%ebp)	addl	$16, %esp	pushl	endpoint
call	srand	subl	$4, %esp	pushl	%ebx
addl	$16, %esp	pushl	$endpoint	pushl	$.LC0

subl $4, %esp	call EndPointToUInt32	call printf
pushl $endpoint	addl $4, %esp	
call EndPointToUInt32	pushl %eax	
addl $4, %esp	pushl -4(%ebp)	
pushl %eax	pushl $.LC0	
pushl -4(%ebp)	call printf	
pushl $.LC0		
call printf		

左边是优化前的代码，然后 movw 置 endpoint 的一半为 0，然后取出 endpoint 的地址调用 EndPointToUInt32，并把结果放到 tmp2 也就是-4(%ebp)中。

中间的代码是将函数 inline 化以后的结果，注意到现在直接把 endpoint 的内容通过%eax 传给了 tmp2 也就是-4(%ebp)。

右边的代码经过了-O2 优化，首先做了一次 inline 操作，取消了对 EndPointToUInt32 的调用，也就是直接把 endpoint 的内容作为 EndPointToUInt32 的返回值来处理。其次取消了 tmp2 变量，用%ebx 来代替。至此都没有问题。

问题在于将 movw $0, endpoint+2 一句优化到了 movl endpoint, %ebx 的后面。这里做了一个错误的乱序优化。这是因为首先 gcc 没有能够正确地判断出*(const UINT32*)(&endpoint) 实际上和 endpoint.udpPort_是相关的，从而优化出错。本来这也是可以容忍人的，毕竟写法太变态。但是 gcc 又在处理 inline 时过于冒进，没有按照真正函数调用那样在函数调用处设置一个边界，阻止函数调用前后的代码混杂，而是像一个宏展开一样简单处理了。最后导致了和预想的结果不一致。

结论

GCC 除少数版本外，在-O2 乱序优化时都不够完善，不能正确判断代码的影响范围，从而作出错误的乱序。所以请不要引入一些编译器难以判断影响范围的语句，尤其是胡乱 cast。典型的如上面程序中的*(const UINT32*)(ep);

GCC 的乱序优化对 inline 函数是像宏展开一样处理的，这可能可能导致将函数和函数附近的代码乱序，需要小心。

常用的 FC3/FC5 上的 gcc 都有此问题。