编译优化导致运行结果不同的现象观察和提问

曹弈轩，黎浩然，叶茂林。

时间：2023年4月8日。

实验环境：

1. gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
2. gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
3. openEuler (4.19.90-2003.4.0.0036.oe1.aarch64) 20.03 (LTS) , gcc 7.3.0

如无特别说明，环境为1。openEuler上的实验结果也在一个单独小节中给出。

缘起：`gcc`的编译优化改变了代码的行为？

给定一个`c`代码`test.c`：

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

int isLessOrEqual(int x, int y) {
    int temp=(x+~(y+1)+1)>>31;
    int sign=!((x>>31)^(y>>31));
    int result=(sign&(!!temp))+(!sign&(!(y>>31)));
    return result;
}
int main() {
    int test = isLessOrEqual(0x80000000,0xffffffff);
    printf("test %d\n", test);
}
```

分别用`-O0`、`-O1`、`-O2`、`-O3`编译运行，结果不尽相同。其他环境中，如gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0，从`-O1`开始就输出`test 0`。

```
wuhui@wuhui-virtual-machine:~/compile_opt$ gcc -O0 -m32 -g
test.c -o test_O0
wuhui@wuhui-virtual-machine:~/compile_opt$ gcc -O1 -m32 -g
test.c -o test_O1
wuhui@wuhui-virtual-machine:~/compile_opt$ gcc -O2 -m32 -g
test.c -o test_O2
wuhui@wuhui-virtual-machine:~/compile_opt$ gcc -O3 -m32 -g
test.c -o test_O3
wuhui@wuhui-virtual-machine:~/compile_opt$ ./test_O0
test 1
wuhui@wuhui-virtual-machine:~/compile_opt$ ./test_O1
test 1
wuhui@wuhui-virtual-machine:~/compile_opt$ ./test_O2
test 0
wuhui@wuhui-virtual-machine:~/compile_opt$ ./test_O3
test 0
```

一些已有的实验现象

`printf`打印和`gdb`观察

不宜在运行过程中插入`printf`语句，否则影响编译优化结果。如试验结果可知：

```
wuhui@wuhui-virtual-machine:~/compile_opt$ gcc -O1 -m32 -g
test.c && ./a.out
temp: -1
sign: -1
result: -1
test 1
wuhui@wuhui-virtual-machine:~/compile_opt$ gcc -O2 -m32 -g
test.c && ./a.out
temp: -1
sign: -1
result: -1
test 1
wuhui@wuhui-virtual-machine:~/compile_opt$ gcc -O3 -m32 -g
test.c && ./a.out
temp: -1
sign: -1
result:
```

```
wuhui@wuhui-virtual-machine:~/compile_opt$ gcc -O0 -m32 -g
test.c && ./a.out
temp: -1
sign: -1
result: -1
test 1
```

因此不使用`printf()`输出，而应尝试使用`gdb`观察。

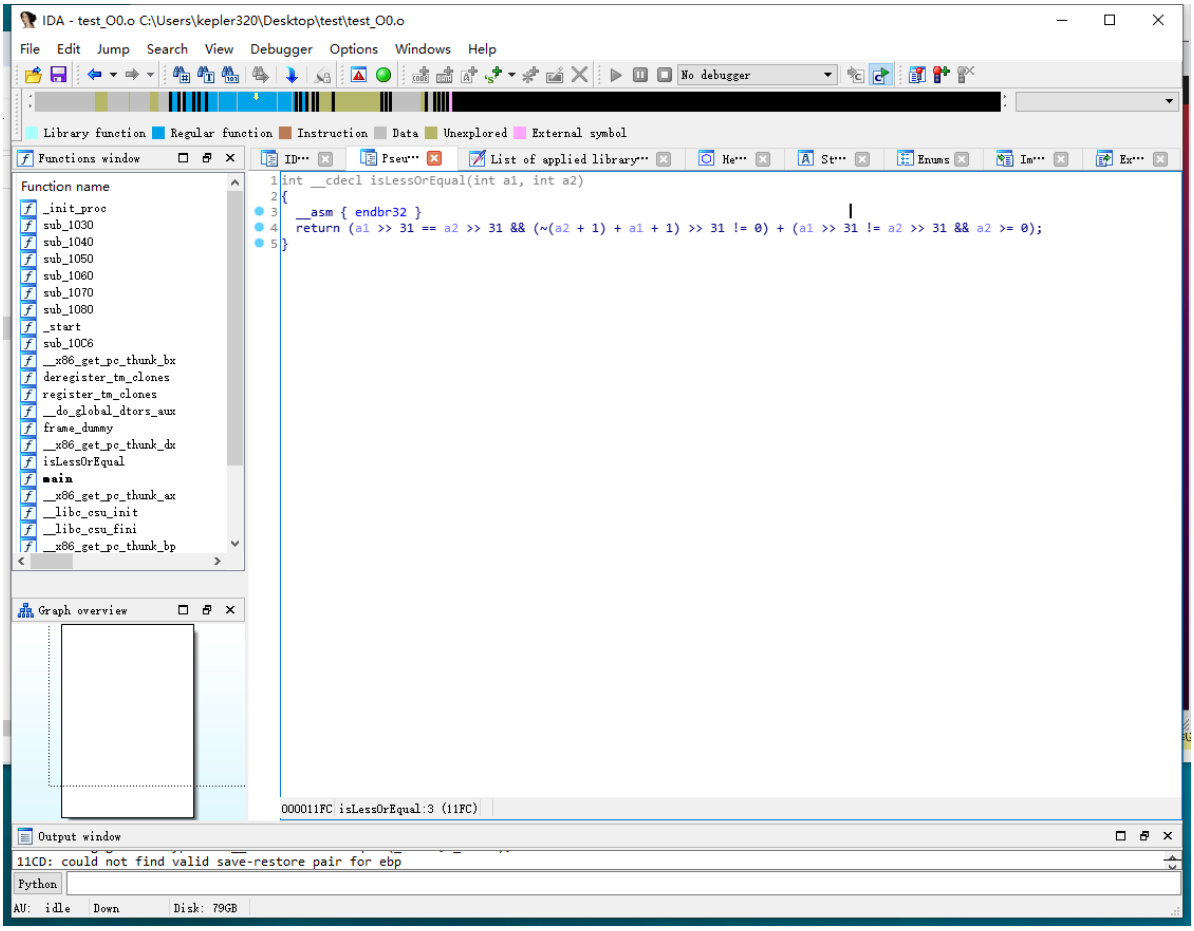
- 对于`-O0`选项，易知`temp=-1, sign=1, result=1`.
- 使用优化选项后，经过繁杂的优化，可执行文件已经被优化得面目全非，使用`gdb`无法完成源码中变量粒度的追踪。

  反汇编观察也是如此，编译优化开启后，汇编代码可读性差，无法提供更多细节。

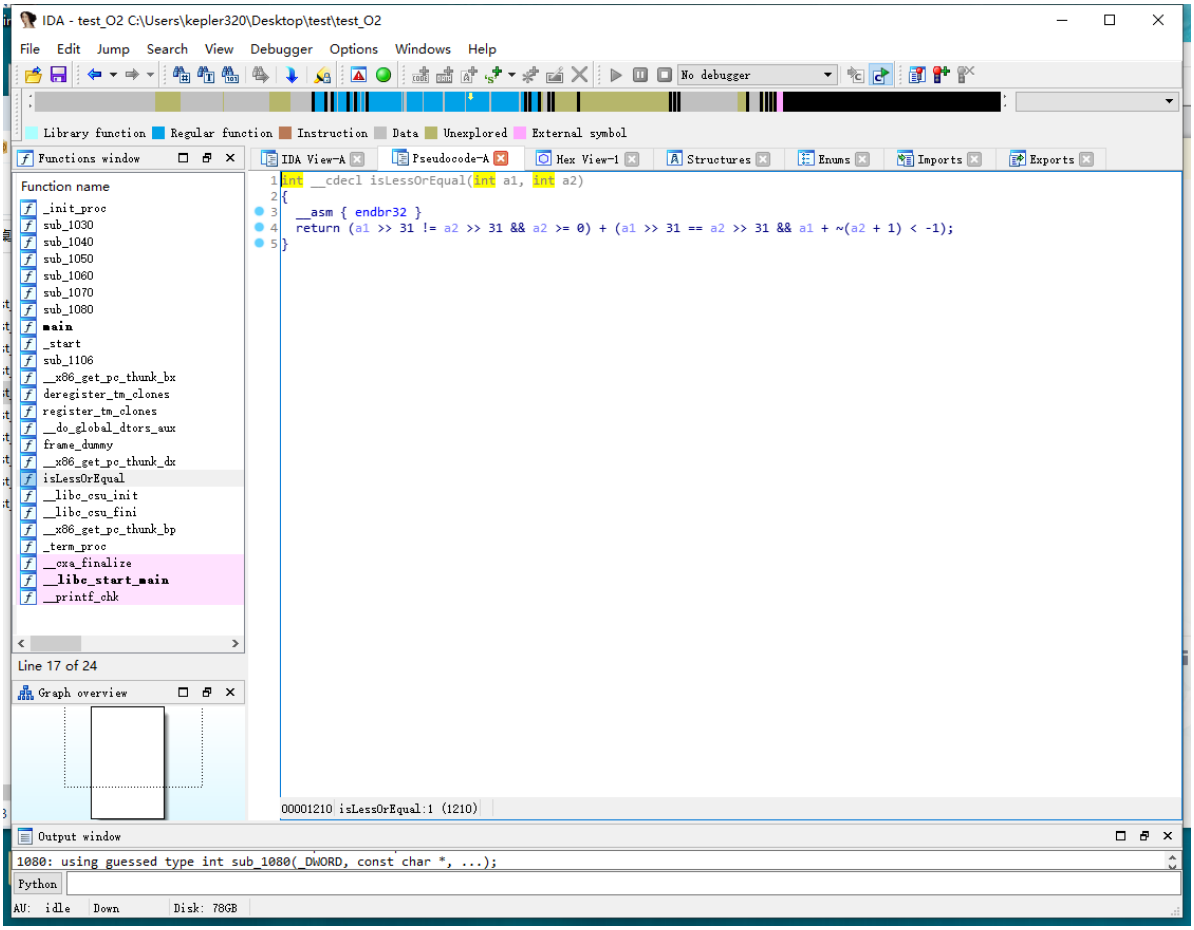
利用IDA逆向分析

利用`IDA`逆向分析，得到结果如下：

1. `-O0`：`isLessOrEqual()`逆向分析等价为`(a1 >> 31 == a2 >> 31 && (~(a2 + 1) + a1 + 1) >> 31 != 0) + (a1 >> 31 != a2 >> 31 && a2 >= 0)`。



2. `-O2`：`isLessOrEqual()`逆向分析结果等价为`(a1 >> 31 != a2 >> 31 && a2 >= 0) + (a1 >> 31 == a2 >> 31 && a1 + ~(a2 + 1) < -1)`。



我们将优化前后逆向分析得到的等价写法，改写成一个新的C程序`test1.c`，分别输出。

```
// test1.c
#include<stdio.h>

int main(){
        int a1 = 0x80000000;
        int a2 = 0xffffffff;
        printf("test %d\n", (a1 >> 31 == a2 >> 31 && (~(a2 +
1) + a1 + 1) >> 31 != 0) + (a1 >> 31 != a2 >> 31 && a2 >=
0));
        printf("test %d\n", (a1 >> 31 != a2 >> 31 && a2 >= 0)
+ (a1 >> 31 == a2 >> 31 && a1 + ~(a2 + 1) < -1));
}
```

运行结果为：

```
wuhui@wuhui-virtual-machine:~/compile_opt$ gcc test1.c &&
./a.out
test 1
test 0
```

说明编译器优化选项确实改变了代码的行为。

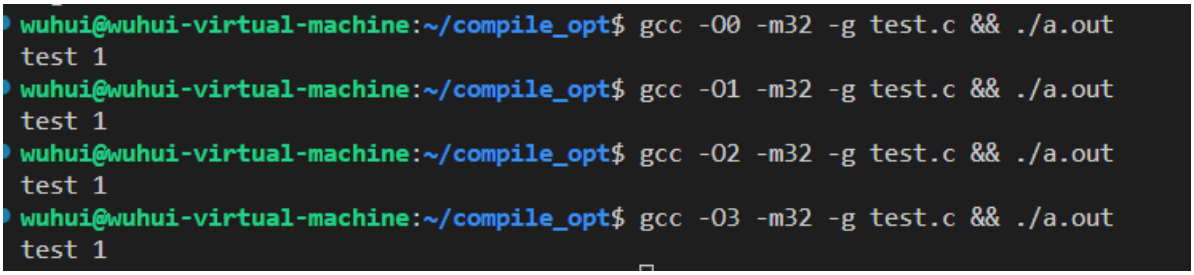
将临时变量改为无符号整数

函数中用到了多个临时变量，将这几个临时变量声明为无符号整数。

```
#include <stdio.h>

int isLessOrEqual(int x, int y) {
    unsigned temp=(x+~(y+1)+1)>>31;
    unsigned sign=!((x>>31)^(y>>31));
    unsigned result=(sign&(!!temp))+(!sign&(!(y>>31)));
    return result;
}
int main() {
    int test = isLessOrEqual(0x80000000,0xffffffff);
    printf("test %d\n", test);
}
```

最终都可以得到一致的运行结果。



openEuler上的实验

由于`-m32`在我们使用的泰山服务器/openEuler操作系统并上不支持，不妨暂时先不考虑该选项。我们再进行`test.c`和`test1.c`的2组实验。这一现象在泰山服务器/openEuler上依然存在。

1. `test.c`观察

```
[szu@taishan02-vm-2022-11 ~]$ gcc -O0 -g test.c && ./a.out
test 1
[szu@taishan02-vm-2022-11 ~]$ gcc -O1 -g test.c && ./a.out
test 1
[szu@taishan02-vm-2022-11 ~]$ gcc -O2 -g test.c && ./a.out
test 0
[szu@taishan02-vm-2022-11 ~]$ gcc -O3 -g test.c && ./a.out
test 0
```

2. `test1.c`观察

```
[szu@taishan02-vm-2022-11 ~]$ gcc -O0 -g test1.c && ./a.out
test 1
[szu@taishan02-vm-2022-11 ~]$ gcc -O1 -g test1.c && ./a.out
test 1
[szu@taishan02-vm-2022-11 ~]$ gcc -O2 -g test1.c && ./a.out
test 1
[szu@taishan02-vm-2022-11 ~]$ gcc -O3 -g test1.c && ./a.out
test 1
```

问题

1. 编译器选项的不同导致代码行为的不同，是编译器的bug（例如某个优化用的`PASS`出了问题），还是C语言标准的设计者和编译器的开发者允许该情况存在？如果是编译器bug，会不会有因此造成的安全漏洞？如果是后者，我们应该遵循何种依据和规范，才能防止出现这样的问题？
2. 上文实验中通过将临时变量声明为无符号数解决了问题。C语言中文网中的一篇文章（http://c.biancheng.net/view/362.html）提倡程序员在位操作的时候尽可能使用无符号数，并称“对有符号数执行位操作是不可移植的”。“不可移植”的依据何在？是否包括gcc的不同选项之间的可移植性？

当然，如果将变量 y 声明成为无符号类型，即将：

```c
int y=0x80000000;
```

修改为：

```c
unsigned int y = 0x80000000;
```

那么这种缓冲区溢出错误将不会发生。

在右移中合理地选择 0 或符号位来填充空出的位

在右移运算中，空出的位用 0 还是符号位进行填充呢？

其实答案由具体的 C 语言编译器实现来决定。在通常情况下，如果要进行移位的操作数是无符号类型的，那么空出的位将用 0 进行填充；如果要进行移位的操作数是有符号类型的，则 C 语言编译器实现既可选择 0 来进行填充，也可选择符号位进行填充。

因此，如果很关心一个右移运算中的空位，那么可以使用 unsigned 修饰符来声明变量，这样空位都会被设置为 0。同时，如果一个程序采用了有符号数的右移位操作，那么它就是不可移植的。

移位的数量必须大于等于 0 且小于操作数的位数

如果被移位的操作数的长度为 n，那么移位的数量必须大于等于 0 且小于 n。因此，在一次单独的操作中不可能将所有的位从变量中移出。例如，一个 int 型的整数是 32 位，并且 n 是一个 int 型整数，那么 n<<31 和 n<<0 是合法的，但 n<<32 和 n<<-1 是不合法的。因此，我们在进行移位运算的时候必须做相关测试。示例代码如下所示：

```c
unsigned int x;
unsigned int y;
unsigned int result;
/*初始x, y, result*/
if(y>=sizeof(unsigned int) * CHAR_BIT)
{
    // 错误处理
}
else
{
    result=x>>y;
}
```

这里还需要说明的是，对于变量 x 与 y，C99 规定：

- 对于 x<y 可以用结果类型表示，那么这个表达式就是结果值，否则，其行为是未定义的；如果 x 是无符号类型，则 x< y，是根据“结果类型可以表达的最大值加 1”进行求模运算得到的结果。需要注意的是，尽管在 C99 中指定了无符号整数的取模行为，无符号整数溢出还是常常导致出乎意料的值以及因此产生的潜在安全风险。
- 对于 x>>y，如果 x 是无符号类型或非负值的有符号类型，那么 x>>y 的移位结果为 x/2y 的商的整数部分；如果 x 是有符号类型的负值，那么 x>>y 的移位结果是由编译器所定义的。因此，对一个带符号整数进行右移运算和将它除以 2 的其次幂不一定是等价的。要证明这一点很容易，考虑 (-1)>>1 的值，它的执行结果不可能为 0，而在大多数 C 语言编译器