Exploration1

从未定义行为出发，对编译器不等式优化的探索

学院：计算机科学与技术学院
专业： 计算机科学与技术
学号： 3210104169
姓名： 郑乔尹

2023年11月3日

一、背景说明

- 在一开始，我没有很好的选题目标，一直在纠结应该选什么切入点，耗费了不少时间，通过与老师的交流，我决定从C语言中的undefined behavior入手。其实这个问题当初在做C小程的题目时就有遇到——指的就是那些冗长的表达式，当我对他的输出结果摸不着头脑时，去问了一些同学，有时便得到了这是undefined behavior的回答，但是当时没有深究，就草草略过。这次刚好借此机会对undefined behavior做一个深入一些的探索。
- 在此过程中，我又从编译器对`n + 1 > n`的优化出发，探讨了编译器对于不等式的一些优化方法。

> 我的探索过程中可能会出现错误猜测，在后文中通过我的逐步探究会进行纠正。

二、探索过程

- 首先通过查阅C语言标准，我得知了以下几类***undefined behavior***:
  1. 符号数溢出 Signed overflow
  2. 数组越界 Access out of bounds
  3. 未初始化值的访问 Uninitialized scalar
  4. 无效值的访问 Invalid scalar
  5. 空指针解引用 Null pointer dereference
  6. 访问已传递给std::realloc的指针 Access to pointer passed to realloc
  7. 没有副作用的死循环 Infinite loop without side-effects

> 先来看看符号数溢出的反汇编结果

```
#include<stdio.h>
#include<limits.h>
int test(int n){
    return n + 1 > n;
}
int main() {
    test(INT_MAX);
```

```
    return 0;
}
```

- 编译器`x86-64 clang 17.0.1`，无额外编译选项：

```
test:                                   # @test
        push    rbp
        mov     rbp, rsp
        mov     dword ptr [rbp - 4], edi
        mov     eax, dword ptr [rbp - 4]
        add     eax, 1
        cmp     eax, dword ptr [rbp - 4]
        setg    al
        and     al, 1
        movzx   eax, al
        pop     rbp
        ret
main:                                   # @main
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], 0
        mov     edi, 2147483647
        call    test
        xor     eax, eax
        add     rsp, 16
        pop     rbp
        ret
```

- 通过打印test函数的返回值，可以看到当前输出为0，显然与`x+1>x`这个恒真的式子不相符
- 但是当我们开启优化，比如选择`-O2`，可以得到以下结果：

```
test(int):                              # @test(int)
        mov     eax, 1
        ret
main:                                   # @main
        xor     eax, eax
        ret
```

- 看来开了`-O2`级别优化编译器和我想的一样，它发现了这是个恒真的式子，通过查阅资料，我猜测，这一优化似乎是由编译器的常量折叠进行的。

> 这里穿插一则我在维基百科上查找到的常数折叠以及常数传播的简要说明：

- **常数折叠**（**Constant folding**）以及**常数传播**（**constant propagation**）都是编译器优化技术，他们被使用在现代的编译器中。进阶的常数传播形式，或称之为稀疏有条件的常数传播（sparse conditional constant propagation），可以更精确地传播常数及无缝的移除无用的程式码
- 但是我并不是很确定，因为我通过以上的查找搜索，认识到的常量折叠一般只是把显然为常量的部分进行计算，常数传播则是对于已确定值的变量，将其值代入后面的变量计算式中计算，所以我决定试试其他情况。
- 如果`n + 1 > n`这种两侧都有参数的不等式都能被常量折叠，那么改一下情况，一个单侧为常量的不等式应该很显然会被折叠，通过以下这段代码，我来尝试一下：

```
#include<stdio.h>

int test(int n) {
        if (n > 20 - sizeof(int))
                return 1;
        return 0;
}

int main () {
        int k = test(10);
        return 0;
}
```

- 以下是这段代码的汇编结果

```
test:                                   # @test
        push    rbp
        mov     rbp, rsp
        mov     dword ptr [rbp - 8], edi
        movsxd  rax, dword ptr [rbp - 8]
        cmp     rax, 16
        jbe     .LBB0_2
        mov     dword ptr [rbp - 4], 1
        jmp     .LBB0_3
.LBB0_2:
```

```
        mov     dword ptr [rbp - 4], 0
.LBB0_3:
        mov     eax, dword ptr [rbp - 4]
        pop     rbp
        ret
main:                                   # @main
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], 0
        mov     edi, 10
        call    test
        mov     dword ptr [rbp - 8], eax
        xor     eax, eax
        add     rsp, 16
        pop     rbp
        ret
```

- 可以看到符号拓展后的参数n被加载到了RAX寄存器中。应该说不出所料，这条不等式变为了`cmp rax, 16`，不等式右侧被直接计算为16，用于比较。既然验证了简单的***常数折叠***，那我又有一个大胆的想法，把`sizeof(int)`移到左侧呢？这对人类来说应该是一眼等价的不等式，虽然现得重新减回来才能构造出单侧的常数，但根据之前的现象，这一不等式理论上来说这应该也能被编译器优化成n与常数比较吧，让我对测试代码做一个简单的修改：

```
#include<stdio.h>

int test(int n) {
        if (n + sizeof(int) > 20 )
                return 1;
        return 0;
}

int main () {
        int k = test(10);
        return 0;
}
```

- 现在if的条件变为了`n+constant1>constant2`的形式，通过反汇编工具得到汇编代码：

```
test:                                   # @test
        push    rbp
        mov     rbp, rsp
        mov     dword ptr [rbp - 8], edi
        movsxd  rax, dword ptr [rbp - 8]
        add     rax, 4
        cmp     rax, 20
        jbe     .LBB0_2
        mov     dword ptr [rbp - 4], 1
        jmp     .LBB0_3
.LBB0_2:
        mov     dword ptr [rbp - 4], 0
.LBB0_3:
        mov     eax, dword ptr [rbp - 4]
        pop     rbp
        ret
main:                                   # @main
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], 0
        mov     edi, 10
        call    test
        mov     dword ptr [rbp - 8], eax
        xor     eax, eax
        add     rsp, 16
        pop     rbp
        ret
```

- 很奇怪的是，这次没被折叠，编译器并没有移动左侧的常数到右侧一起做计算，而是先将`sizeof(int)`(4)与加载到rax寄存器中的n相加，再与右侧的20进行比较。
- 不过这还没有开启任何优化选项，给他开个`-O2`优化试试：

```
test:                                   # @test
        movsxd  rcx, edi
        add     rcx, -17
        xor     eax, eax
        cmp     rcx, -21
        setb    al
```

```
        ret
main:                                   # @main
        xor     eax, eax
        ret
```

- 又变成常数了，只不过看起来是先把`n=10`与`sizeof(int)`算了（虽然不明白为什么要-17）

> 后来又补充了对***x86-64 gcc 13.2***下的测试，倒是更直观

```
test:
        movsx   rdi, edi
        xor     eax, eax
        add     rdi, 4
        cmp     rdi, 20
        seta    al
        ret
.LC0:
        .string "%d"
main:
        sub     rsp, 8
        xor     esi, esi
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

- 不过两种编译器的行为逻辑大同小异，main函数里都不调用test函数了，可以看到根本没有出现`call`，发生什么事了？？？不过仔细一想，大概是因为测试代码写的太简单了，传入函数的是个常数10，更重要的是，test函数的返回值对于整个程序来说无关紧要，所以编译器认为压根不用调用，就把整个函数调用优化掉了，这个简单，加个对k的输出应该就行。

> 后来才反应回来，其实这段test的反汇编结果就是正常带参的test函数，只不过比较形式比较怪，先给rcx寄存器-17，又和-21比较

```
#include<stdio.h>

int test(int n) {
```

```
        if (n + sizeof(int) > 20 )
                return 1;
        return 0;
}

int main () {
        int k = test(10);
        printf("%d", k);
        return 0;
}
```

- 然而：

```
test:                                   # @test
        movsxd  rcx, edi
        add     rcx, -17
        xor     eax, eax
        cmp     rcx, -21
        setb    al
        ret
main:                                   # @main
        push    rax
        lea     rdi, [rip + .L.str]
        xor     esi, esi
        xor     eax, eax
        call    printf@PLT
        xor     eax, eax
        pop     rcx
        ret
.L.str:
        .asciz  "%d"
```

- 还是没有出现对test函数的调用，真怪。我想了好久，感觉编译器似乎直接看穿了整段代码，因为我在main函数中传入的是一个常数 `10`，这里看起来直接被整个"塞进"了 `test` 函数的汇编代码中，而且被计算了一遍，可以看到实际上有两次对eax寄存器的清零（`xor eax, eax`），因为根据源代码，这个函数在传入参数为10时应该返回0，于是体现在printf函数的传入参数获取上，编译器干脆直接用了 `xor eax, eax` 将eax寄存器清零，作为k的值。不过唯一能确定的是，即使如此，函数内部不等式的形式仍然是 `n(10) + sizeof(int) > 20`，就是不肯把左侧的 `sizeof(int)` 移动到右侧。

- 不过这似乎又回到了undefined behavior的问题上，一旦n是`INT_MAX - 1`这样的值，左侧的值会引发signed overflow；如果把`sizeof(int)`移到不等号右侧，这种情况将会消失，因为此时不等式`n > 20 - sizeof(int)`左侧没有发生加操作，不可能出现overflow，这就导致两种情况可能会因为n的取值不同而产生不一致的比较结果，我猜测这也许是编译器拒绝对此计算顺序进行优化的原因。
- 接下来就是四处查找资料求证，最终在clang的后端，也即llvm的官方文档中找到了似乎是这个问题的答案的描述

Poison Values

A poison value is a result of an erroneous operation. In order to facilitate speculative execution, many instructions do not invoke immediate undefined behavior when provided with illegal operands, and return a poison value instead. The string 'poison' can be used anywhere a constant is expected, and operations such as add with the nsw flag can produce a poison value.

Most instructions return 'poison' when one of their arguments is 'poison'. A notable exception is the select instruction. Propagation of poison can be stopped with the freeze instruction.

It is correct to replace a poison value with an undef value or any value of the type.

This means that immediate undefined behavior occurs if a poison value is used as an instruction operand that has any values that trigger undefined behavior. Notably this includes (but is not limited to):

- The pointer operand of a load, store or any other pointer dereferencing instruction (independent of address space).
- The divisor operand of a udiv, sdiv, urem or srem instruction.
- The condition operand of a br instruction.
- The callee operand of a call or invoke instruction.
- The parameter operand of a call or invoke instruction, when the function or invoking call site has a noundef attribute in the corresponding position.
- The operand of a ret instruction if the function or invoking call site has a *noundef* attribute in the return value position.

> 见此词条LLVM Language Reference Manual — LLVM 18.0.0git documentation

- 这里定义了一种名为Posion Value的东西。大意为Poison Value是一个错误操作的结果。为了方便预测执行（speculative execution），许多指令在遇到非法操作数时不会立即引发undefined behavior，而是返回一个poison value。
- 我个人的理解是，像我的测试代码中的两种情况:

1. `n > 20 - sizeof(int)`：标准的一侧变量，一侧常量的情形，编译器正常地使用常量折叠进行优化；
2. `n + sizeof(int) > 20`：左侧在某些情况下会出现***Signed Overflow***(具体到这个例子上就是$n \in [INT_MAX - sizeof(int) + 1, INT_MAX]$时，作为`int`类型的有符号数n会发生***Signed Overflow***)，此时这就是一个Poison Value，llvm将其保留以作为一种它处理undefined behavior的方式，为其优化留下空间。

> 不过《深入理解计算机系统》(以CSAPP为人熟知？)一书中似乎提到了一种优化——不安全优化，由于我记得不是很清楚，可能得后面再求证一下。

- 虽然gcc与llvm的实现有所不同，但是表现出来的行为却相同，大致可以确定gcc对这种潜在*undefined behavior*的处理方式是类似的。从这个简单（虽然我觉得寻找问题原因的过程并不简单）的问题中，再回头看看`n+1>n`，我敢以99%的确定性猜测这绝不是一个常量折叠可以做到的，毕竟它的左侧就是一个潜在的**Undefined Behavior**，然而右侧又带着`n`，说实话

做到这里，我不太有头绪这是怎么做到被编译器成功认定为恒真表达式的，因为从上面这个Poison Value的存在看来，编译器对于优化的条件"审核"是比较严格的。

- 不过好在有这么一个朴素的网站，简单的介绍了一些编译器的优化方式(Compiler Optimizations)，进去一条一条的遍历。
- 然而，很遗憾的是，并没有找到符合的。
- 我决定先试试常量折叠以及常量传播能够做到何种程度的优化，这里直接整了个大的，不过也非常简单——等差数列求和，这应该属于常量传播的范畴，代码如下:

```c
#include <stdio.h>
int f(int x) {
    int sum = 0;
    for (int i = 0; i < 50; ++i) {
        sum += i;
    }
    return sum;
}

int main() {
    int x = f(50);
    printf("%d", x);
}
```

- 只要打开`-O1`级别的优化，就可以得到编译器对此代码的优化结果：

```
f:                                      # @f
        mov     eax, 1225
        ret
main:                                   # @main
        push    rax
        lea     rdi, [rip + .L.str]
        mov     esi, 1225
        xor     eax, eax
        call    printf@PLT
        xor     eax, eax
        pop     rcx
        ret
.L.str:
        .asciz  "%d"
```

- 可以看到函数内部的循环直接被计算，现在这个函数里其实根本没有用到传入的参数，如果将i的上界改为这个参数，则可以得到以下结果：

```
f:                                      # @f
        test    edi, edi
        jle     .LBB0_1
        lea     eax, [rdi - 1]
        lea     ecx, [rdi - 2]
        imul    rcx, rax
        shr     rcx
        lea     eax, [rdi + rcx]
        dec     eax
        ret
.LBB0_1:
        xor     eax, eax
        ret
main:                                   # @main
        push    rax
        lea     rdi, [rip + .L.str]
        mov     esi, 1225
        xor     eax, eax
        call    printf@PLT
        xor     eax, eax
        pop     rcx
        ret
.L.str:
        .asciz  "%d"
```

- 可以看到函数体没有被常量传播优化，但是函数调用得到了优化，可以看到1225的值仍然出现在了main函数中，跳过了对f函数的调用，说明常量传播还是发生了。但是这和我想要的`n+1>n`还是相去甚远，到这里似乎就已经卡住了。
- 让我们回到`n + 1 > n`：

```
int test(int n) {
    return n + 1 > n;
}
```

- 使用**x86-64 gcc 13.2**，编译选项`-O0`，此时不发生常量折叠，但是请看以下的反汇编结果：

```
test:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, 1
        pop     rbp
        ret
```

- 可以看到出现了`mov eax, 1`一行，这表明了`n + 1 > n`在没有常量折叠优化的情况下，仍然被编译器计算，并获得了正确结果1，基本可以排除常量折叠的可能性。于是，我开始寻找其他可能应用在此的优化方法。

- 在不断的寻找之后，我最终找到了一种优化方法——窥孔优化。

> 这里引入对窥孔优化的介绍，参考中文维基百科

- **窥孔优化**是一种编译器优化技术，在编译的后面阶段，寻找编译器生成的特定的指令序列，将该序列替换成性能更好的等效指令序列。该算法的可视范围极小，往往只能同时窥看几条指令，故名曰窥孔优化（指其犹如通过窥孔观察程序）。被优化掉的这少量指令序列被称为窥视孔或窗口。
- `n + 1 > n`恰巧符合这种情形，编译器通过观察上下几条指令，便能发现其为恒真表达式，这并不属于常量折叠和常量传播的范畴，为了找到这一结论，我走了不少弯路。

三、效果分析

- 通过以上的探究过程，我从一个简单的undefined behavior出发，为了探究编译器是如何对`n + 1 > n`进行优化，尝试了一些不同的案例，但是由于对于编译器的了解还是太少，导致一开始的猜测有误，不过在探索过程中，我还是认识到了不少常量折叠、常量传播的应用场景，以及编译器对于避免C标准中的Undefined behavior带来的优化影响，做出的类似于Poison Value这样的特殊规定。
- 同时，我也认识到了对于一个我们眼中等价的式子，编译器要考虑的事其实很多，很多时候并不能直接用简单直观的人类思维直接对程序进行优化。

四、实验体会

- 在实验中，我深深体会到计算机学科的博大精深，其实上面这些看似很简单的问题——比如给不等式两侧的值移个位置，要究其原因也十分复杂，而且一旦涉及编译器层面，搜索引擎能提供的资料也就越来越少，必须要自己开动脑筋构造样例进行对比，才能真正明白编译器究竟在做什么，而要究其原因，最好还是要去到官方文档中寻找蛛丝马迹。

五、经验教训

- 直接使用搜索引擎并不能对解决问题提供什么特别大的帮助，如果要探究编译器的一些问题，还是需要进入到编译器的官方文档中进行翻阅查找，本次实验中，大部分结论都是通过自己上手尝试以及查阅官方文档得到的结果。而官方文档**最好不要使用机翻**，因为我一开始图省事直接使用浏览器自带的翻译，结果看到了更加晦涩难懂的描述，不如直接看原文来的清晰明了。

六、参考文献

---

1. Constant Folding (compileroptimizations.com)
2. Constant Propagation (compileroptimizations.com)
3. 常数折叠 - 维基百科，自由的百科全书 (wikipedia.org)
4. 窥孔优化 - 维基百科，自由的百科全书 (wikipedia.org)
5. Peephole optimization - Wikipedia
6. 4. Kaleidoscope: Adding JIT and Optimizer Support — LLVM 18.0.0git documentation