wuhui_gdnt的专栏

个人资料

wuhui_gdnt

访问： 295374次
积分： 8541
等级： BLOG 6
排名： 第769名

原创： 549篇 转载： 0篇
译文： 30篇 评论： 43条

C、C++中未定义行为的指引， 第3部分（完）

分类： C++语言 2013-03-26 12:00 467人阅读 评论(0) 收藏 举报

目录(?) [+]

作者：John Regehr

原作：http://blog.regehr.org/archives/232

一个C或C++实现必须以编程序执行有副作用的操作。例如，如果一个程序执行这些代码行：

```
printf ("Hello\n")

printf ("world.\n");
```

该实现这样打印是不允许的：

```
world.
```

```
Hello
```

这是显然的，但记住编译器可以自由地重新安排你程序做其它事的次序。例如，如果程序以一个次序递增一些变量：

```
a++;

b++;
```

编译器可以产生以另一个次序递增它们的代码：

```
incl    b

incl    a
```

当这些变换被认为能提升性能，且不改变程序可观察的行为时，优化器执行像这样的变换。在C/C++中，可观察行为是那些正在进行的副作用。你可能争辩，说这个重新排序实际上是可观察的。当然，它是：如果a与b是全局变量、一个内存映射I/O设备、一个信号句柄、一个中断句柄，或另一个会看到b已经写入，而a尚未的内存状态的线程。不过，这个重排是合法的，因为在这些语言中，保存到全局变量不被定义为副作用。（事实是，让事情变得更混乱，根据标准，保存到全局变量是副作用，但没有编译器这样处理。参考本帖底部的注解）。

带着这个偏离议题的背景材料，我们可以问：在C及C++中，segfault或其它崩溃是一个副作用操作吗？更具体地说，当一个程序因为执行一个非法操作死去时，比如除0或解引用一个空指针，这被视为副作用吗？回答一定是“不”。

一个例子

因为引致崩溃的操作不是副作用，就其它操作而言，编译器可以重新安排它们，就像重排上面的例子中对a与b的储存。但实际的编译器会利用这个自由吗？它们会。考虑这个代码：

```
volatile int r;

int s;

void foo1 (unsigned y, unsigned z, unsigned w)

{

  r = 0;

  s = (y%z)/w;

}
```

foo1首先存入r；这个存入是一个副作用操作，因为f是volatile。然后foo1对一个表达式求值，并把结果存入s。第二个赋值的右手侧潜在执行一个带有未定义行为的操作：除0。在大多数系统上，程序以一个数学异常崩溃。

这是最近一个GCC版本的输出：

```
foo1:

  pushl   %ebp

  xorl    %edx, %edx

  movl    %esp, %ebp

  movl    8(%ebp), %eax

  divl    12(%ebp)          <-- might crash the program

  movl    $0, r             <-- side effecting operation
```

```
  movl    %edx, %eax
  xorl    %edx, %edx
  divl    16(%ebp)
  popl    %ebp
  movl    %eax, s
  ret
```

注意divl指令——如果除数是0，这将杀死该程序——在保存到r之前。Clang的输出是类似的：

```
foo1:
  pushl   %ebp
  movl    %esp, %ebp
  movl    8(%ebp), %eax
  xorl    %edx, %edx
  divl    12(%ebp)        <-- might crash the program
  movl    %edx, %eax
  xorl    %edx, %edx
  divl    16(%ebp)
  movl    $0, r           <-- side effecting operation
```

```
  movl    %eax, s

  popl    %ebp

  ret
```

而Intel CC:

```
fool:

  movl      8(%esp), %ecx

  movl      4(%esp), %eax

  xorl      %edx, %edx

  divl      %ecx              <-- might crash the program

  movl      12(%esp), %ecx

  movl      $0, r             <-- side effecting operation

  movl      %edx, %eax

  xorl      %edx, %edx

  divl      %ecx

  movl      %eax, s

  ret
```

在编译器的话语中，这里的问题是，在fool的代码中第一行与第二行之间没有依赖关系。因此，编译器可以进行任

何看起来像个好主意的重排。当然，在更大的场景中，存在依赖性，因为设计r的计算将永远得不到运行，如果向s赋值的RHS（译注：右手侧）使程序崩溃——如常——C/C++标准创建了这些规则，编译器只是遵守游戏规则。

另一个例子

这个C代码调用一个外部函数，然后执行少量数学计算：

```
void bar (void);

int a;

void foo3 (unsigned y, unsigned z)

{

  bar();

  a = y%z;

}
```

Clang在调用函数前进行数学计算：

```
foo3:

  pushl   %ebp

  movl    %esp, %ebp

  pushl   %esi

  subl    $4, %esp
```

```
  movl    8(%ebp), %eax

  xorl    %edx, %edx

  divl    12(%ebp)        <-- might crash the program

  movl    %edx, %esi

  call    bar             <-- might be side effecting

  movl    %esi, a

  addl    $4, %esp

  popl    %esi

  popl    %ebp

  ret
```

我们可以添加这个代码完成这个程序:

```
void bar (void)

{

  setlinebuf(stdout);

  printf ("hello!\n");

}
```

```
int main (void)

{

  foo3(1,0);

  return 0;

}
```

现在我们可以编译且运行它：

```
regehr@john-home:~$ clang -O0 biz_main.c biz.c -o biz

regehr@john-home:~$ ./biz

hello!

Floating point exception

regehr@john-home:~$ clang -O1 biz_main.c biz.c -o biz

regehr@john-home:~$ ./biz

Floating point exception
```

正如我们对foo3()的汇编代码所预期的，当打开优化时，除法指令在无缓存printf()发生前，使程序崩溃。

为什么这很重要?

希望前面的例子使得相关性明显，但在这里我将详细说明。假设你正在调试一个困难的segfault：

1. 在调用foo()之前你添加了一个printf()，其中foo()是一段包含了一些可疑指针操作的、令人讨厌逻辑的代码。

当然——就像我们在这个例子中做的——你关闭了stdout上的缓存，在程序继续执行前，这些行被实际输出到终端。

2. 运行代码，看到在程序segfaults之前，没有触及printf()。
3. 你总结到崩溃不是由于foo()：它必定发生在更早的代码中。

在第3步的推论是错的，如果在foo()中某些危险的操作被移到printf()前面，并触发一个segfault。当你在调试时做出了一个不正确的推论，你开始走在错误的道路上，并在那里浪费大量的时间。

解决方案

如果你正在经历这类问题，或仅对此有偏执，你应该怎么做？最显而易见的尝试是关闭优化或单步通过你代码的指令。对于嵌入式系统的开发者，这都不可行。

对一个以令人惊讶的方式失败的一个解决方案是添加一个编译器屏障。即修改foo1为：

```
void foo2 (unsigned y, unsigned z, unsigned w)

{

  r = 0;

  asm volatile ("" : : : "memory");

  s = (y%z)/w;

}
```

新加的代码行是一个GCC习语（也为Intel CC及Clang理解），它的含义是“这行汇编码，虽然没有包含指令，可能触及所有的内存”。其效果基本上是在代码中人为插入大量的依赖，在这行汇编码前强制把所有的寄存器值存入RAM，之后重新载入。这行新代码使得Intel CC与GCC在对s的任一RHS求值前保存r，但Clang走得更远，产生这

个代码：

```
foo1:

  pushl   %ebp

  movl    %esp, %ebp

  movl    8(%ebp), %eax

  xorl    %edx, %edx

  divl    12(%ebp)         <-- might crash the program

  movl    %edx, %eax

  xorl    %edx, %edx

  divl    16(%ebp)

  movl    $0, r            <-- side effecting operation

  movl    %eax, s

  popl    %ebp

  ret
```

我相信Clang是正确的。编译器屏障添加了一些人为依赖性，但它们不足以阻止divide指令移到保存volatile之前。

这个问题真正的解决方案——它有助于理解，尽管只要我们无法摆脱C与C++，它对我们没有好处——是向异常情形分配一个语义，比如除0或解引用一个空指针。一旦语言这样做，就具有潜在危险的重排操作而言，可以约束优化器

的行为。这方面Java做得不错。

C与C++中副作用的一个技术注解

C99标准规定：

访问一个volatile对象，修改一个对象，修改一个文件，或调用一个进行任何这些操作的函数都是副作用，它们是执行环境状态的改变。

在C标准中，“对象”即是“变量”。C++0x草案（译注：C++0x在11年正式发布）包含非常类似的语言。条款“修改一个对象（modifyingan object）”是无意义的，因为——结合顺序点（sequencepoint）语义——当程序包含像这样的代码时，它禁止编译器消除重复的保存：

```
x=0;
```

```
x=0;
```

没有一个优化的C或C++编译器认为“修改一个对象”是一个具有副作用的行为。

尽管我们在挑剔，“调用一个进行任何这些操作的函数”部分也被普遍被忽视了，因为函数内联。在内联的函数里的代码可能有副作用，但调用本身永远不会被这样处理。

自2010年8月19日的更新

CompCert是迄今为止最实用的、经过验证的C编译器，它是一件很棒的作品。我花了一些时间尝试让它产生我在上面展示的目标代码——其中一个危险性操作被移到了一个副作用操作之前——我没有成功。XavierLeroy，CompCert项目的领导者，是这样说的：

事实上，你触及了一个相对深刻的语义问题。

我们都同意C编译器不必要保留未定义行为：越界访问一个数组的程序在运行时不是每次都崩溃。你可以对此感到惋

惜（如果这样，许多安全漏洞可以避免），但我们有点同意保留未定义行为的代价将会太高（数组边界检查等）。

现在，一个保留已定义行为的编译器有两个选择：

1- 如果源程序展示了一个未定义行为，编译后的代码可以随心所欲

2- 如果源程序展示了一个未定义行为，编译后的代码像直到该未定义行为点的源程序那样执行，有相同的可观察的效果，然后随心所欲（崩溃或以任意的后果继续）。

如果我没有理解错的你帖子，你观察到了选项1并指出选项2不仅是更直观，而且更有助于调试。我想我同意你，但某些编译器作者真的喜欢选项1……

至于CompCert，我证明的高级正确性法则都是属于上面的第一种类型：如果源程序出错了，没有什么可以保证编译后代码的行为。不过，我依赖的单步模拟论点实际上证明了上面的特性2：编译后的代码不能比源代码“更早”崩溃，并产生相同的崩溃前可观察行为，此外可能更多。因此也许我应该加强我的高级正确性法则……

考虑什么是可观察的，我同意你，C标准说的是荒谬的：写入nonvolatile对象当然是不可观察的，它们调用的函数也不是（仅该函数可以调用的系统调用是）。

因此我们拥有了它：CompCert可证明对这些问题免疫，但以一个可能是侵入的方式。在后续的email中，Xavier提到他正在考虑加强顶层的CompCert法则，以显示这不会发生。太酷了！

上一篇 C、C++中未定义行为的指引， 第2部分
下一篇 每个C程序员应该知道的未定义行为#1/3

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap