ConSeq: Detecting Concurrency Bugs

through Sequential Errors

夏亦谦

一、背景

Concurrency bugs是由不同线程对共享内存的交叉存取导致共享内存中的内容在错误的时间出现了错误的内容造成的。一般来说，Bug的生命周期包含三个过程：(1) 触发：即某个操作触发bug，使得某个数据产生错误；(2) 传播：错误的数据被传播出去；(3) 失败：当某个操作使用这个数据时，导致程序崩溃、中止或产生错误输出等。目前已有的检测这类Bug的方法是通过分析程序源代码来发现潜在的Bug，然后分析出代码中的Bug Patterns（指根据实验发现的Bug Pattern）。但是这一类方法有三种局限性：

1. 漏报（False negatives），由于这类方法是利用已知的Bug Patterns来寻找Concurrency Bug，因此是否能找的更多的Bug取决于Bug Patterns是否更多；
2. 误报（False positives），传统的方法只是根据Bug Patterns找到了可能的Concurrency Bug，至于这些Bug是否会造成系统的Crash并不能给出确定的结论（2-10%的Bug Patterns会造成failure），因此会存在误报的可能性；
3. 用户不友好 ，有时候这类方法检测出来的类交叉存取导致的Bug很难向用户解释成因。用户很难通过有限的代码片段去判断这到底是不是一个Bug，所以已有的一些商业软件通常会忽略掉这类Bug。

作者对70个非死锁造成的concurrency bugs进行了研究分析，得出了三个结论：

1. 59/70的非死锁Bug的传播距离是很短的，ConSeq通常会检查之前四个与共享内存访问有关的代码切片；
2. 66/70的bugs导致的crash是出现在一个线程中的，也就是说系统crash基本是Sequential error，这也是标题detecting concurrency bugs through sequential errors的依据）；
3. failure patterns of concurrency bugs主要有五种：assertion-violation，error-massage（eg: printf、log函数），incorrect outputs，infinite loop，memory-bugs，占了65/70；并且和Sequential failure非常像。

基于以上背景，ConSeq提出了基于结果的、从发现错误（failure）入手的一种反向分析的方法，先找程序产生错误的地方，然后分析传播路径，最后找到触发Bug的地方。

二、ConSeq的介绍：

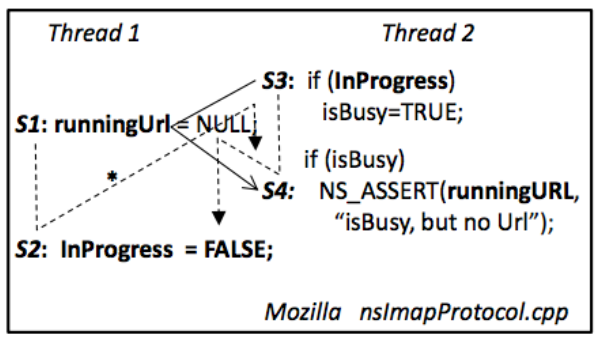


图1. 由共享内存访问导致的bug示例

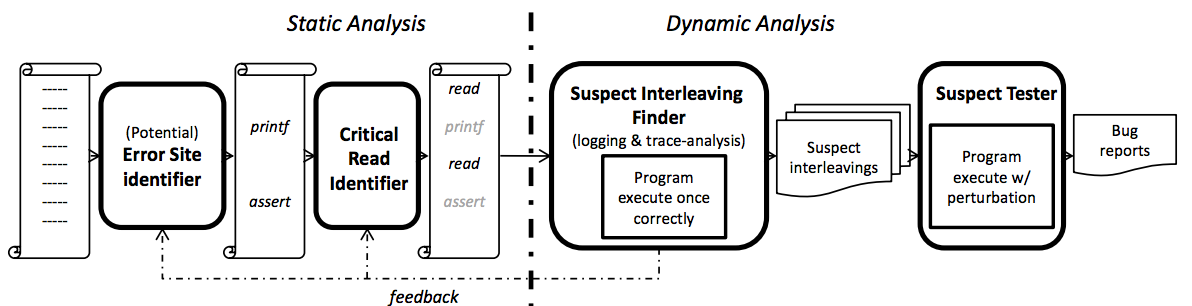


图2. ConSeq分析过程

ConSeq主要通过以下几个步骤来发现concurrency bugs：

1. Error-site identifier，通过对binary的静态分析，找出五类可能的failure的位置，通过一些特征值来发现，比如assert，printf，fprintf，stderr等，还有一些工具，SCC，Daikon（用于发现implicit errors）；如图1的代码示例，通过静态分析，可以找出S4是可能的failure的位置。
2. Critical-read identifier，用静态分析找到影响error site的critical-read（short propagation distance）。Critical-read指的就是导致程序失败的读操作。因为本文指出，能够导致程序崩溃的原因一定是由读操作引起的。如图1的代码示例，在这个代码片段中，虚线代表不会产生 bug的执行路径，如S1->S2->S3->S4或者S3->S4->S1->S2；实线代表会产生bug的执行路径：S3->S1->S4。在这段代码中，Critical-read就是S4，即在进行断言操作时读到了个空值，导致程序中断。
3. Suspicious-interleaving finder，动态分析，运行一次程序，得到运行路径以及control/data依赖图，通过trace和依赖图来找到可能的关联变量；再对它们进行分析，排除掉不可能的，剩下的就是可能性较大的bug。当要判断R与W'是否有关联（数据依赖）时，它们必须要满足三个条件，（一）它们必须都访问相同内存；（二）W'必须在R之前；（三）被W'写入内存的值在被R读取之前不能被重写。图3就展示了三种不满足（三）的情况：在W’操作之后，总会有一次W操作先于R被执行，其中W、W’和R访问的都是同一块内存。图3(a)是线程内的一种执行顺序，W在W’和R之间被执行；图3(b)是由于一些栅栏同步机制强制规定了W’必须在W之前被执行，同时W必须在R之前被执行；图3(c)是互斥现象，如W’和W与R互斥，或者W’与W和R互斥。只要满足这三种情况的一种，R原本该读到的由W’写入的值就会被W覆盖掉，从而判断出关联变量为W。以上是Suspicious-interleaving identification的核心分析方法。在图1的代码示例中，可以分析出来，可能导致程序中断的共享内存访问操作是S1。

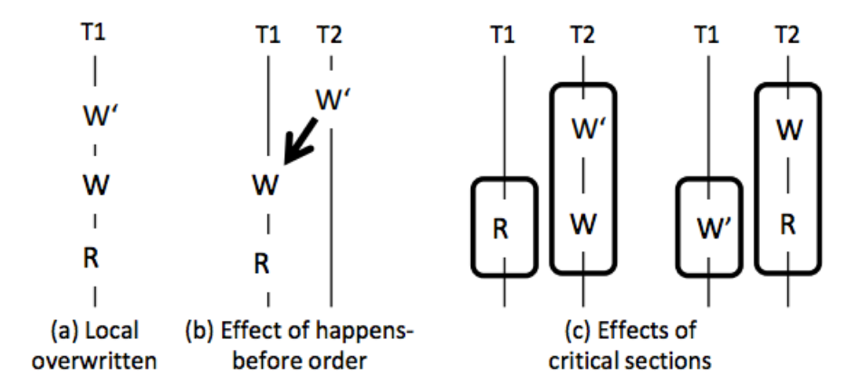


图3 W'总会先于R的执行顺序

1. 4. Suspicious-interleaving tester，动态分析，找到那些真正会导致的failure的bug。在这最后一步中，ConSeq会在critical read指令之前插入一些条件延迟（time-out）来改变程序的执行顺序。在图1的示例代码中，程序本该以S3->S4->S1->S2的顺序运行，ConSeq在S4之前插入延迟，使得程序以S3->S1->S4的顺序执行，然后产生中断。这样就成功验证了这段代码是存在Bug的。

三、ConSeq 与MUVI的比较

1. MUVI只基于源码分析，ConSeq可以基于Binaries分析；
2. MUVI是正向分析，找到源码中可能的bugs，但并不确定这些bug会导致failure，同时漏报的可能性大；ConSeq是反向分析，从failure入手，找到的Bugs肯定会导致failure，同时漏报的可能性小。

四、ConSeq的不足之处：

1. 由于存在动态分析，因此需要依赖输入，来保证高的代码覆盖率，以保证找到更多的bugs;
2. 目前对于failure只是关注五种，虽然基本覆盖了90%的failure，但还是有提高的空间；
3. 由于在方法中用到了传播路径短这一前提条件，因此对那些有较长传播路径的bugs是无法发现的；
4. ConSeq的实现主要基于failure和propagation发生在一个线程中这一研究成果。