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：

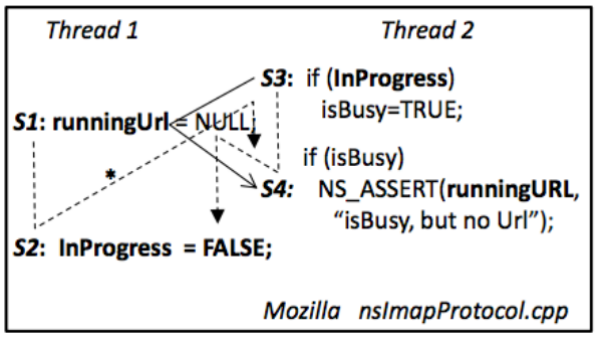

图 1. 由共享内存访问导致的 bug 示例

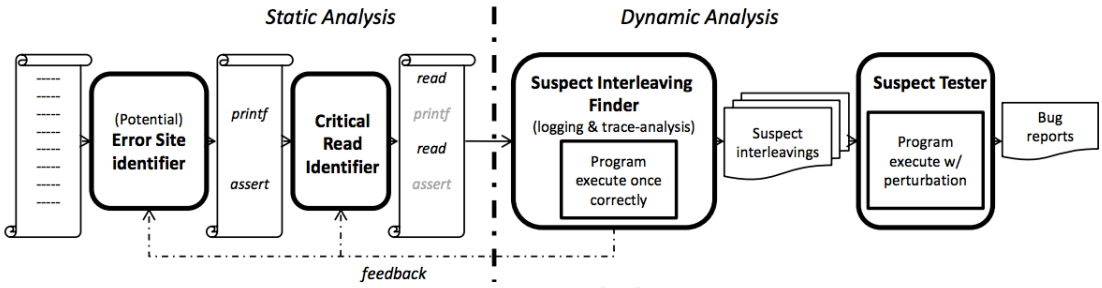

图 2. ConSeq 分析过程

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。

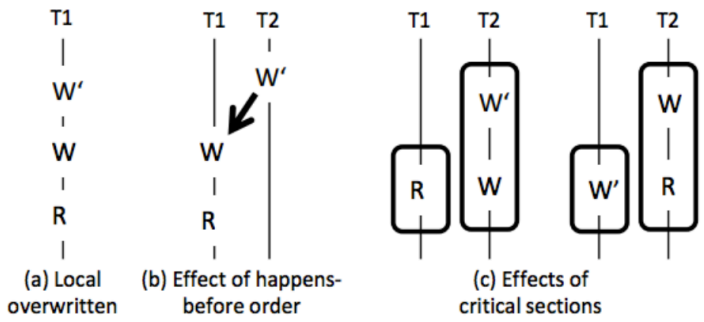

图 3 W'总会先于 R 的执行顺序

4) 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。