**Detecting concurrency bugs from the**

**perspectives of synchronization intensions**

夏亦谦

1. Motivation

Data race: 当两个以上的没有同步机制的线程访问同一个内存空间，至少有一个线程执行的是写操作时，就是data race。

有三种工具用来检测data race：lockset race detection, happens-before race detection和hybrid tools combining the lockset and happens-before。

* Lockset算法：检测发现访问统一内存的操作没有加锁的情况；
* Happens-before：检测当两个有冲突的访问与之前有过的行为不一致的情况；

即使对data race的检测工具已经做得很好了，但是检测concurrency bugs仍然是个难点，因为两种bug有很多不同点。

2. 关注重点

作者对随机采集的74个非死锁bug进行了分析，得出了两个结论：

* 51/74的bug属于违反原子性一类，一般都是由于程序员的线性编程思维导致的；
* 25/74的bug都涉及到了两个及以上语义上相关的变量。

因此，原子性问题和语义相关变量问题将是两大重点关注的问题。

3. 推断AI invariants及检测冲突的可行性

想让程序员提供哪些代码可能被interleaving所影响是有难度的，因为他们筒仓考虑得不够充分，会有各种疏忽遗漏。所以最好的方法就是通过不断地执行程序来学习程序的行为。

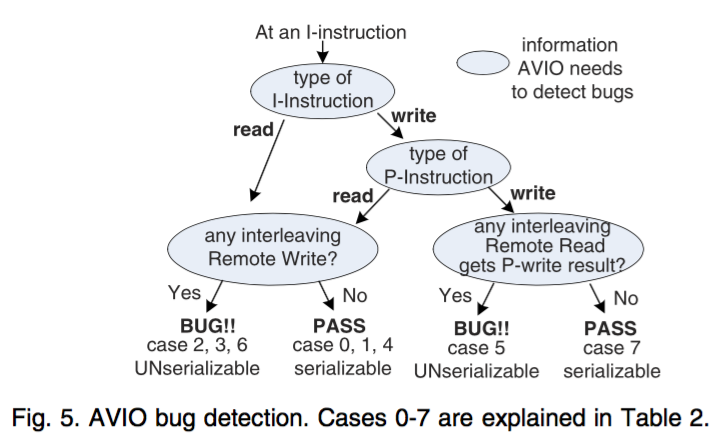
由于concurrency bugs的两个特点，使得AVIO中的training并不难：

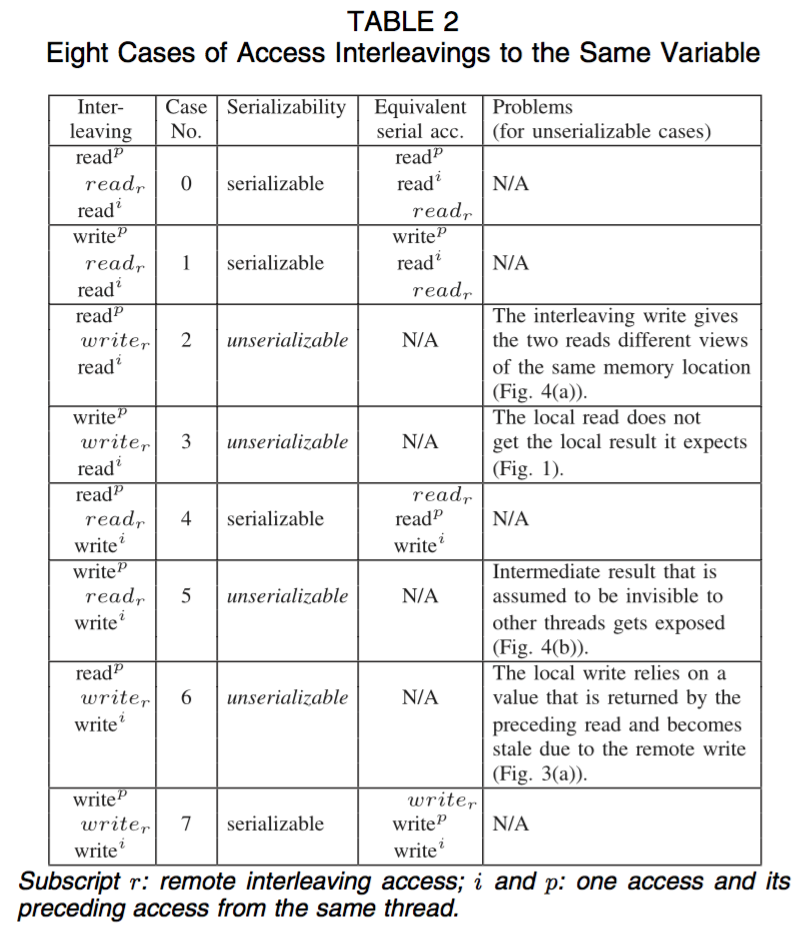
* 正确的执行结果可以通过少数的尝试被确定，因为concurrency bugs出现的次数并不多；
* 充分性问题很容易解决，应为对于有很多interleaving的程序，每一次执行都可以得到不同的执行路径。

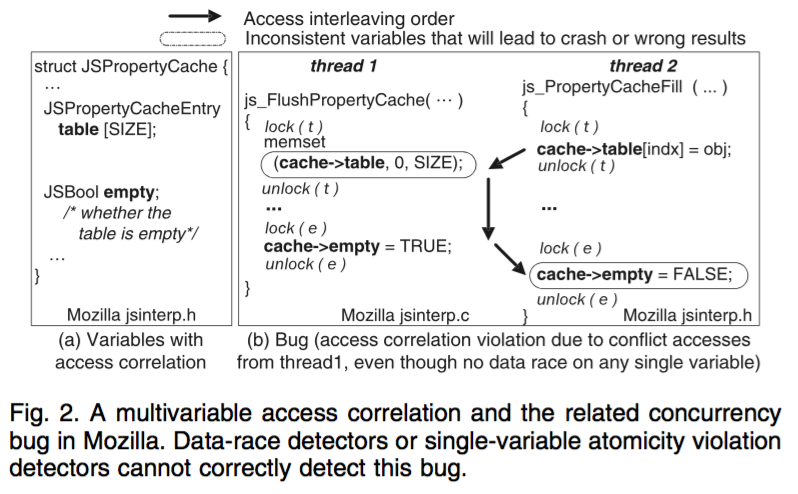
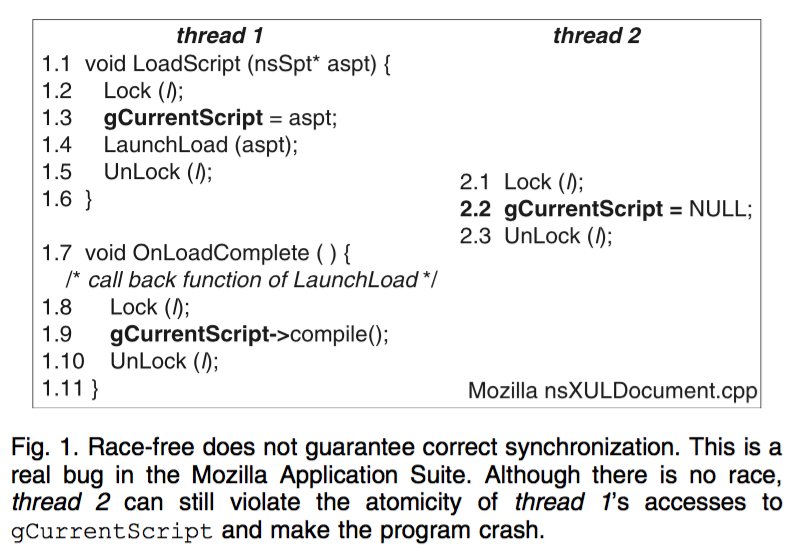
所以对于AVIO来说，concurrency bug的不确定性提供了独特的优势：用可以触发bug的输入执行程序若干次，会得到高质量的good-quality training.

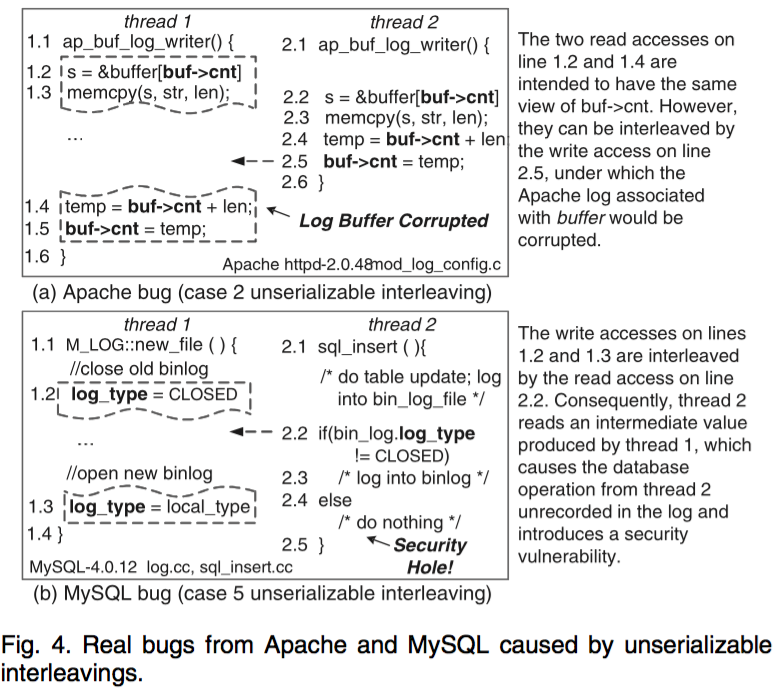
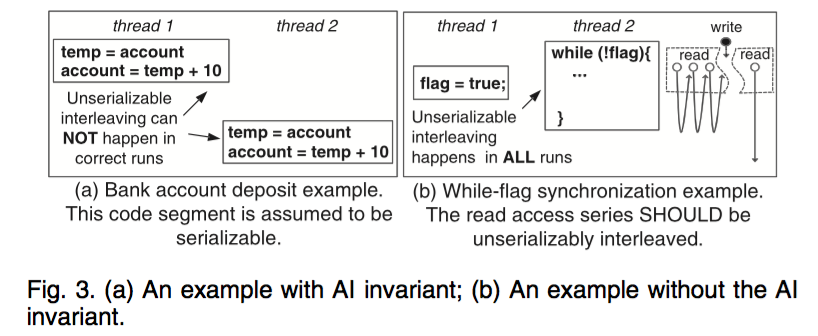
4. AVIO的算法：

* 检测算法：通过对程序的分析，遵循下面的决策图Fig. 5，可以检测出违反AI invariants的部分；Fig. 5的trace图是根据Table2中分析出来的对于其他线程的interleaving access、本线程的当前指令、前一条指令的所有可能执行结果画出来的，并且统计了哪些interleaving就算调换了执行顺序也不会有问题，哪些interleaving调换执行顺序后会出问题，并给出了会出问题的case的代码示例（Fig.1-Fig.4）。
* 推断算法：AVIO-IE通过若干次运行，抓取出一些AI invariants。把这些初始化为一个AI集，包含所有的全局内存访问部分。再运行程序，如果AVIO报出了有violation的指令i，那么i就会从AI集中移除。这样循环往复直到运行路径不再变化。









5. AVIO的实现：

* 硬件实现：利用高速缓存一致性，用开销很小的硬件扩展来实现。在L1缓存加两个bit，Preceding access Instrument bit（PI bit）标记preceding access的类型，DownGrade bit（DG bit）标记remote线程是否读了一个之前被preceding instrument写过的数据。L1 cache中已有的INV bit用来标记自从最近的内存访问后是否有interleaving remote write发生。所以说，AVIO-H只用检查INV来得知是否有一个remote write发生。L2是共享cache，因此并发访问的数据都是放在L2中的。所以，不能序列化的interleaving一定会在某一时刻去读取共享内存中的最新数据或者去对这个内存加锁，防止其它的写入。在这样的前提下，AVIO-H只需要对L2进行监控，并且只关注上述两种情况就行。
* 软件实现：每个线程维护一个access-table，记录最近一次对共享内存的访问信息。还有一个全局的access-owner-table，记录最近一次对共享内存进行写操作的线程。

软硬比较：软件实现无需硬件扩展，结果更精确，因为无需进行cache替换、上下文切换等；硬件实现开销更小。

6. AVIO与MUVI的比较：

不同点：

* 推断concurrency bug的方法不同，MUVI使用了数据挖掘方法，对象是源代码；而AVIO只用到了一个简单的算法（上文提到过），对象是执行路径。
* AI invarinats很难用静态分析方法分析出来，它需要基于运行结果来分析；而变量相关性的推断可以使用静态分析的方法。
* MUVI需要花很多时间来对变量相关性进行分析，但是只用分析一次；AVIO要花时间（运行很多次）来收集AI invariants，随着输入集的不同，AVIO还要进行持续的收集。

展望：

* 两者都提供了通用、可扩展的框架；
* 两者提出的方法可以在别的工具（Autolocker and Colorama）上使用；
* 以后可以尝试分析程序员的注释得到更多线索；
* MUVI倾向于对多变量关系的分析，AVIO倾向于对原子性冲突的分析。两者如果相结合，MUVI可以在AVIO上进行扩展，来实现多个变量的原子性冲突检测。