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。

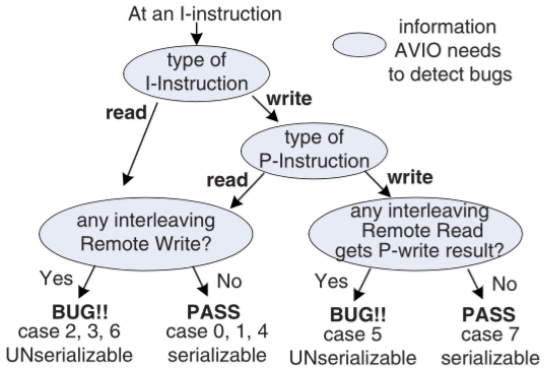


Fig. 5. AVIO bug detection. Cases 0-7 are explained in Table 2.

TABLE 2
Eight Cases of Access Interleavings to the Same Variable

Inter-leaving	Case No.	Serializability	Equivalent serial acc.	Problems (for unserializable cases)
read^p $read_r$ read^i	0	serializable	read^p read^i $read_r$	N/A
write^p $read_r$ read^i	1	serializable	write^p read^i $read_r$	N/A
read^p $write_r$ read^i	2	*unserializable*	N/A	The interleaving write gives the two reads different views of the same memory location (Fig. 4(a)).
write^p $write_r$ read^i	3	*unserializable*	N/A	The local read does not get the local result it expects (Fig. 1).
read^p $read_r$ write^i	4	serializable	$read_r$ read^p write^i	N/A
write^p $read_r$ write^i	5	*unserializable*	N/A	Intermediate result that is assumed to be invisible to other threads gets exposed (Fig. 4(b)).
read^p $write_r$ write^i	6	*unserializable*	N/A	The local write relies on a value that is returned by the preceding read and becomes stale due to the remote write (Fig. 3(a)).
write^p $write_r$ write^i	7	serializable	$write_r$ write^p write^i	N/A

Subscript r: remote interleaving access; i and p: one access and its preceding access from the same thread.

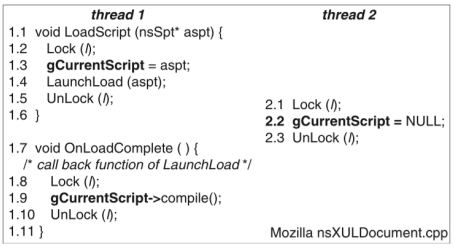
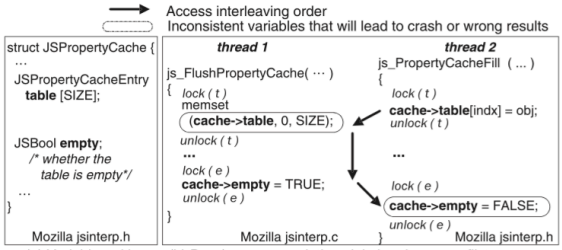


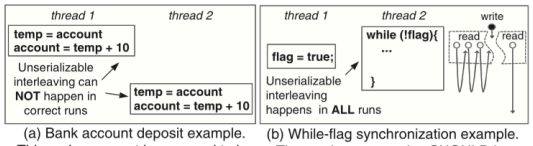
Fig. 1. Race-free does not guarantee correct synchronization. This is a real bug in the Mozilla Application Suite. Although there is no race, *thread 2* can still violate the atomicity of *thread 1*'s accesses to `gCurrentScript` and make the program crash.



(a) Variables with access correlation

(b) Bug (access correlation violation due to conflict accesses from thread1, even though no data race on any single variable)

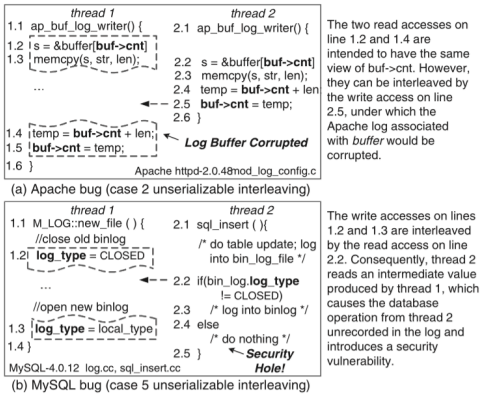
Fig. 2. A multivariable access correlation and the related concurrency bug in Mozilla. Data-race detectors or single-variable atomicity violation detectors cannot correctly detect this bug.



(a) Bank account deposit example. This code segment is assumed to be serializable.

(b) While-flag synchronization example. The read access series SHOULD be unserializably interleaved.

Fig. 3. (a) An example with AI invariant; (b) An example without the AI invariant.



(a) Apache bug (case 2 unserializable interleaving)

(b) MySQL bug (case 5 unserializable interleaving)

Fig. 4. Real bugs from Apache and MySQL caused by unserializable interleavings.

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。