data race不是导致concurrency bug的根本原因，concurrency bug的根本原因是程序的实现不符合程序员的同步意图，程序员的同步意图没有得到实现。

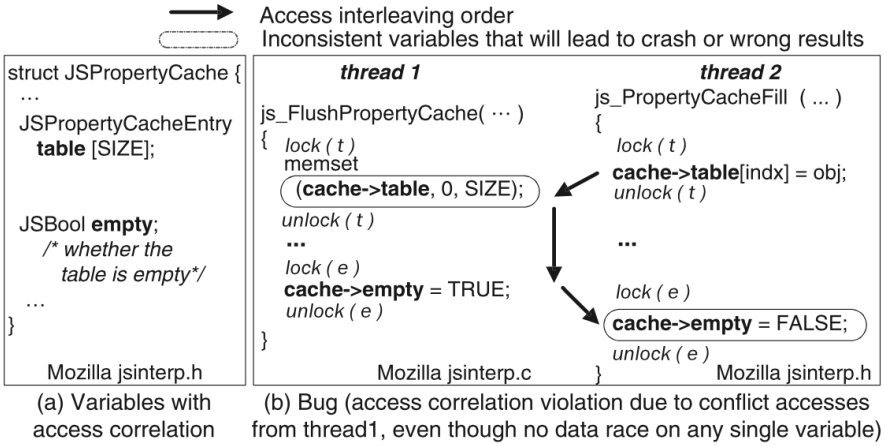
根据对现实世界中的concurrency bug的研究，发现有两类同步意图被频繁的违背：

（1）原子性（atomicity,also called serializability）．如果并发的执行S操作和按一定的顺序执行一系列的S操作的结果等价，那么就说操作是原子性的。

原子性冲突bug（在现实世界中占非死锁bug的70%）——程序员期望一些代码区域拥有原子性，但是在实现过程中忘记或者没有成功实现原子性，从而引起了原子性冲突bug。下面的例子中，thread1和thread2没有data race,但是违反了原子性。race free不能保证原子性。

```
thread 1                                        thread 2
1.1  void LoadScript (nsSpt* aspt) {
1.2     Lock (l);
1.3     gCurrentScript = aspt;
1.4     LaunchLoad (aspt);
1.5     UnLock (l);
1.6  }                                          2.1  Lock (l);
                                                2.2  gCurrentScript = NULL;
1.7  void OnLoadComplete ( ) {                  2.3  UnLock (l);
     /* call back function of LaunchLoad */
1.8     Lock (l);
1.9     gCurrentScript->compile();
1.10    UnLock (l);
1.11 }                                          Mozilla nsXULDocument.cpp
```

（2）变量访问相关性（variable access correlation）。关联的变量在访问的时候要保持一致性。



(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)

多变量并发bug(在现实世界中占非死锁bug的30%),data race detector 和 单变量原子性冲突检测不能检测出多变量并发bug.他们在单变量上是race free的，但是相关联的变量之间没有得到同步。

如何推测程序员的同步意图，哪一段代码区域需要原子性保护，哪些变量是相关联的。论文提出了一种自动推测出同步意图，并发现并发bug的方法。

论文贡献：

（1）现实世界中的并发bug研究，发现大多数并发bug产生的根本原因是违反了原子性，另外，在研究的并发bug里，有三分之一涉及到多变量。

（2）设计两个简单的程序不变量来捕获常见的同步意图。交叉存取不变量（AI invariants）来捕获原子性意图，变量存取相关性来捕获变量之间的存取一致性关系。

（3）自动提取出潜在的不变量来进行同步意图推测。设计AVIO根据训练时收集的信息来自动推测可能的AI不变量；设计MUVI根据源代码和数据挖掘结果来推测变量相关性。

（4）实现两个新的bug detection工具，AVIO和MUVI，可以分别高效的检测出原子性冲突bug和多变量并发bug。

AVIO在运行时检测内存存取，自动检测原子性冲突，两种实现，硬件实现AVIO-H和软件实现AVIO-S。

MUVI扩展两种经典的race detection方法来检测多变量并发bugs。

（5）利用现实世界中的应用来评估AVIO和MUVI，发现之前没有没有发现的bug.

AVIO算法：

通过运行程序,根据trace，提取出AI不变量（trainning），之后在被监测的运行中，通过提取出的AI不变量检测出潜在的冲突。

AI 不变量推断：从多个正确的执行中，找到AI不变量。（在执行过程中，多个线程之间的操作顺序不变）

如果一段代码在正确运行的过程中，都是按照一定的顺序交错执行，那么猜测程序员希望这段代码是原子性的。如果在正确的执行过程中，一段代码经常不按顺序交错执行，那么猜测这段代码不需要保持原子性。

检测算法：利用上面推断的结果，根据下图进行原子性冲突检测。

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.

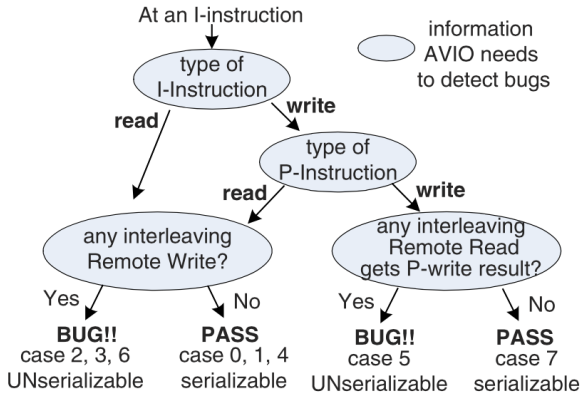


Fig. 5. AVIO bug detection. Cases 0-7 are explained in Table 2.

分析：

AVIO-H： AVIO的硬件实现，适合产品上线运行时检测；AVIO-S：AVIO的软件实现，适合线下检测和诊断。由于AVIO-H利用的是cache-line,所以AVIO-S的准确率比AVIO-H高，AVIO-S的准确率依靠它的训练结果。

AVIO只能检测出涉及一个变量的并发bug，且集中在原子性冲突方面

AVIO不能通过静态分析得到AI不变量，但是变量存取关联性可以通过静态分析或者基于训练的trace分析获得，MUVI选择静态分析因为训练的输入集合有限制，可能覆盖不到整个程序。

在检测bug之前，MUVI要对代码进行静态分析来进行存取关联推测，但是对所有输入和运行检测只需要进行一次分析；AVIO需要进行训练来收集AI不变量的信息，输入改变后，新的输入如果覆盖了新的代码区域，AVIO的AI不变量集合需要进行进一步的训练。