**Concurrency Bugs检测技术研究**

刘宁 李菁菁 夏亦谦 高策 张坚鑫

**摘 要** 在多核时代，如何在多线程软件发布之前对其有效地测试并检测到concurrency bug很重要。但Concurrency bug是最难检测和调试的一类Bug。当前对concurrency bug的检测和验证已经有不少成果。本文对几种不同的concurrency bug检测技术进行了描述和比较。

**关键词** 并发，Bug检测，数据竞争

**Researches on Concurrency Bug Detection**

LIU Ning LI Jing-jing XIA Yi-qian GAO Ce ZHANG Jian-xin

**Abstract** In the multi-core era, it is critical to efficiently test multi-threaded software and expose concurrency bugs before software release. However, concurrency bugs are among the most difficult to detect and diagnose of all software bugs. Previous work has made significant progress in detection and validation. In this paper, we describe and compare several different technologies to detect concurrency bugs.

**Keywords** Concurrency, Bug detection, Data-race

1. **引言**

随着硬件技术的发展，计算机步入了多核时代，软件并发执行以更充分地利用资源，提高运行效率。

但是并发执行的不确定性可能造成一些Concurrency bug。例如在输入相同的情况下，如果线程没有适当地同步，也会导致输出结果不同。还有不同线程对共享内存的交叉存取导致内存状态错误。

针对以上问题，产生了基于对程序的静态分析和动态分析的Concurrency bug检测技术。

本文总结了近年来学术界为检测Concurrency bug而做出的尝试，例如Goldilocks[1], MUVI[5], ConSeq[6], CFP[7]等。本文列举了这些技术对应的系统，分析并比较系统间的差异，以及各自的特点与缺陷。

1. **Concurrency Bug**
   1. Concurrency Bug分类
      1. 数据竞争(Data race)

数据竞争是指冲突的内存访问，即多个线程同时访问相同的地址，且至少有一个为写操作。如图2-1所示，如果没有同步机制，可能存在任意顺序的内存访问，从而导致内存状态不同。

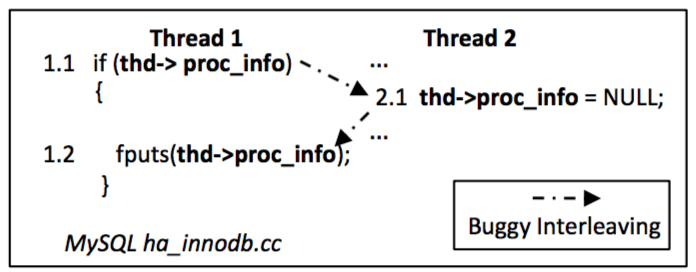


图2-1 数据竞争

* + 1. 非数据竞争(Non-Data race)

通常来说，解决数据竞争的方法是对其加锁，但这不一定能够解决问题。如图2-2所示，即使对gCurrentScript的访问被锁保护起来，由于执行顺序的不同，还是会引发错误。

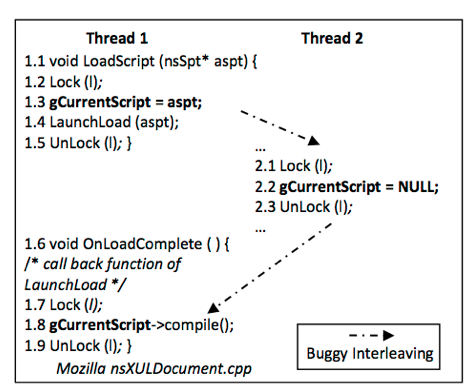


图2-2 非数据竞争

* 1. Concurrency Bug特征

对现实世界concurrency bug特征的研究发现：

1. 非死锁concurrency bug里的三分之一是因为原子性被破坏导致的。究其根本，这类错误是程序员的串行编程思维所导致的，不能通过简单的同步语义，例如锁和事务表达出来。[11]
2. 34%的非死锁concurrency bug均存在多变量问题，不能被已有bug detection工具发现。MUVI重点解决了多变量导致的concurrency bug[5]。
3. 92%的concurrency bug都可以在不多于4次的内存读写后被触发。在ConSeq里面也有用到这个特征[6]。
4. 73%的concurrency bug都不能通过简单地添加或者改变锁来进行修复，而且许多修复第一次都是失败的。
5. Concurrency bug通常发生在一个线程里。即错误是由多线程引发的，但通常仅有一个线程产生了可见错误。可以将分析阶段分为并发分析和顺序分析两个阶段。[6]
6. Concurrency bug的Failure pattern和sequential bug相似。可以参照sequential bug的failure pattern找到可能由从currency bug导致failure的地方。[6]
7. **Concurrency Bug检测技术**
   1. MUVI

之前已有不少检测concurrency bug的方案，但它们都无法处理多变量问题。因为它们没有检测变量间的一致性关系。MUVI通过代码分析和数据挖掘技术检测多个变量间的相关关系，并检测其在交互过程中产生的concurrency bug。主要因为两个原因引起多变量concurrency bug：

1. 关联变量没有被同时更新(inconsistent updates)
2. 在对关联变量进行存取操作时，没有同时对它们进行保护(multi-variable concurrency)
   * 1. MUVI实现

MUVI(Multi-variable Inconsistency)的实现分为两个步骤，首先需要发现多变量的关联关系，然后基于关联关系，去发现（1）同时更新的bug（2）多变量的concurrency bug。

* + - 1. 变量关联性分析

在代码关联性分析中，需要进行两种关系的分析，一种是Access Together，一种是Access Correlation。Access Together是用对变量的访问在源代码中的距离来衡量两次访问之间的关系。Access Correlation是用两次访问同时出现的概率来衡量两次访问之间的关系。

在最后的Correlation Generation阶段，MUVI会根据以上两个评价标准来确定关联性。

* + - 1. Bug Detection

在bug detection阶段，MUVI会根据关联性分析得到的结果，来分析两类Bug。

对于没有同步更新的Bug，最初的算法是在结果中，找到Write(x) => AnyAcc(y)关系，将其列为同步更新Bug的候选集合，然后进行排序。以上过程是用静态分析来实现的。

对于两个变量的数据竞争Bug，MUVI主要扩展了目前的Lockset算法。因为MUVI使用静态分析来检测变量访问的关联性，而Lockset使用动态分析来确定是否有数据竞争问题，因此需要使用代码动态翻译技术来将MUVI与 Lockset算法结合起来。

如图3.1-1所示，在原本的Lockset算法中，它保证同一个变量x的读和写被一个公共的锁所保护，而在MUVI的实现中，如果x与y之间有关联性，那也必须要有一个公共的锁来保护x和y。

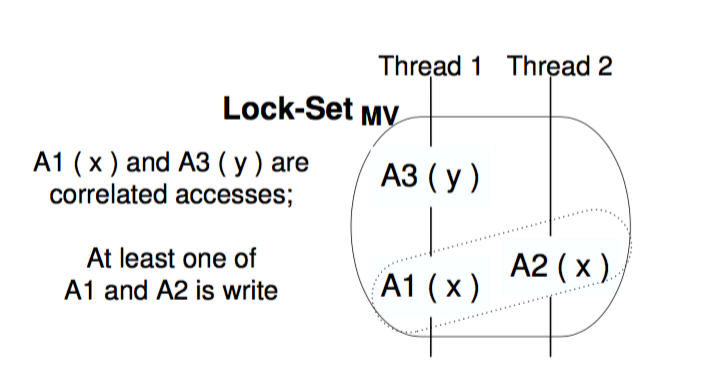


图3.1-1 MUVI扩展的Lockset算法

* 1. ConSeq

ConSeq取自Consequence，是一个面向结果的反向分析框架。该系统分析Bug的生命周期三阶段（分别为产生，传播和造成系统崩溃），利用Concurrency bug的特性C、E和F，反向找到可能存在的Concurrency bug。

* + 1. Bug生命周期

如图3.2-1所示，Bug的生命周期包含三个过程：

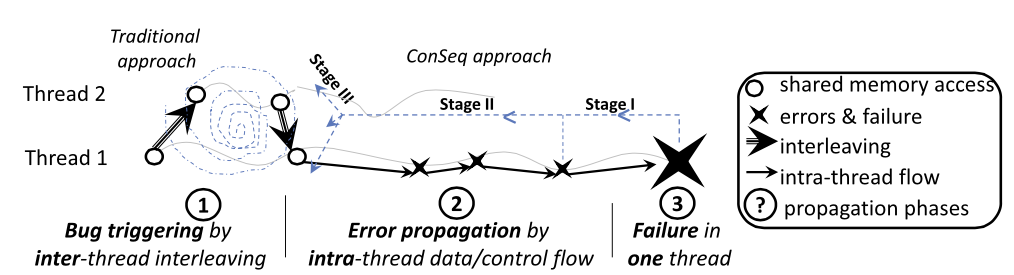


图3.2-1 Bug生命周期

* 触发：不同线程交叉访问共享内存，在某种执行顺序下产生错误数据；
* 传播：错误数据被读指令获取，程序执行时继续传播；
* 失败：某个操作使用错误数据，导致可见错误，例如程序崩溃、陷入死循环或产生错误输出。

例如在图3.2-2中，S3执行后S1执行，导致内存中被写入错误的runningURL值，是触发阶段；然后S4中读入了runningURL错误数据，是传播阶段；最后不满足断言，是失败阶段。

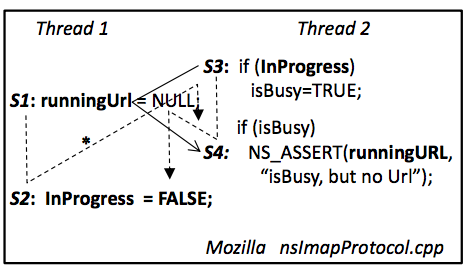


图3.2-2 Bug传播过程

* + 1. ConSeq实现

ConSeq结合静态分析与动态分析技术，使用图3.2-3所示模型，沿着bug传播链反向分析可能存在的Bug。

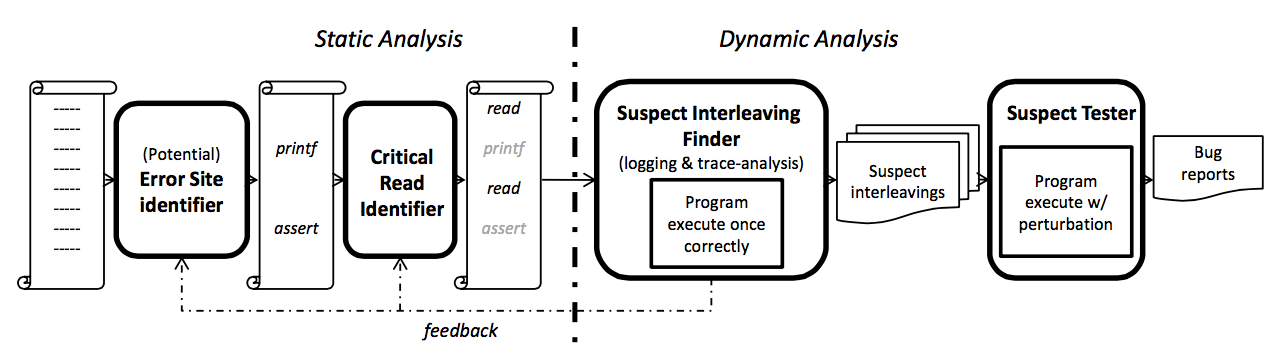


图3.2-3 ConSeq结构

* Error-site identifier，通过对二进制文件的静态分析，找出可能的failure的位置，比如Assertion。
* Critical-read identifier，找到影响error site的critical read。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3.3-2，在这个代码片段中，S4对runningURL的读操作就是critical read。
* Suspicious-interleaving finder，执行一次程序，得到相应的trace。分析trace文件，得到suspicious-interleaving。
* Suspicious-interleaving tester，执行上一步产生的的suspicious-interleaving, 动态分析，找到那些真正会导致的failure的bug。
  1. CFP

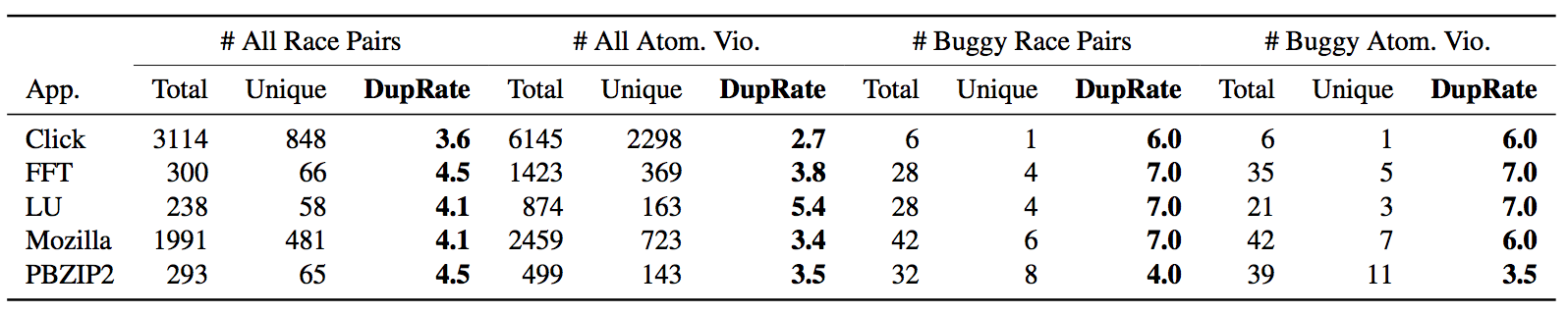
根据每个输入所覆盖的并行函数对(CFP)选择输入集合，从而减少对同一Bug的重复检测。

* + 1. Bug检测存在冗余

现有的技术通常包括以下几个步骤：（1）设计Input, 产生测试输入来保证代码的覆盖率（2）检测Bug， 使用动态Bug检测工具找到可能的Interleaving（3）验证Bug，对每个输入执行多次，来排除掉一些疑似的Bug。

但输入之间存在大量的重合，即同一个bug可以被多个输入检测到。因此会带来10倍到100倍的性能下降。如表3.3-1所示，每个检测出的data race或atomicity violation都平均被2.7-4.5个输入检测到。

表3.3-1 检测重复率



* + 1. CFP

CFP是Concurrent function pair的缩写，是能够并行执行的函数对。如图3.3-1所示，由于锁的存在，foo1()和bar()可以并行执行，而foo2()和bar()不可以。

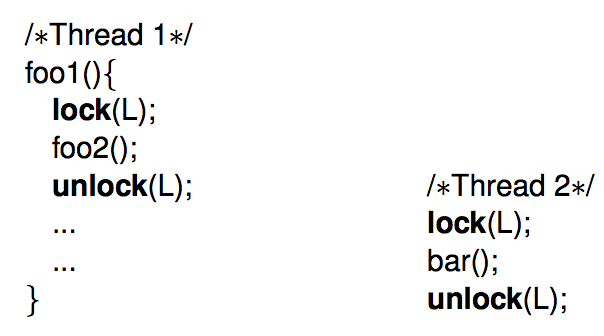


图3.3-1 并行函数对

可以使用同步信息来判断函数是否能够并行执行。如图3.3-2伪代码所示。

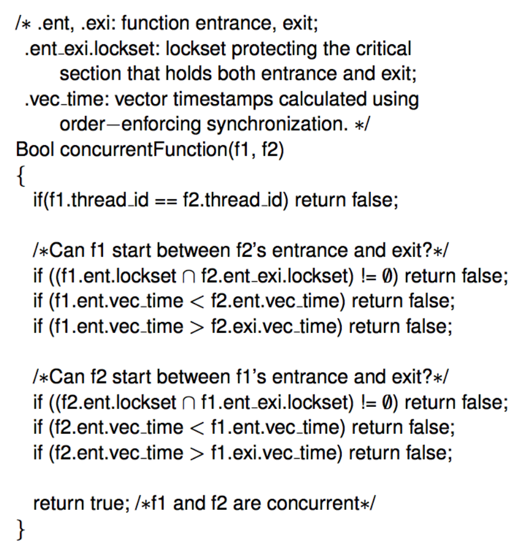


图3.3-2 判断并行函数对伪代码

* + 1. 基于CFP的检测技术实现

第一步对每个输入，分析其能覆盖的并行函数对，并生成并行函数对集合(aggregated CFP)。第二步，选择最小输入集合，使其能够覆盖并行函数对集合。但这是一个NP问题，所以采用贪心算法来得到近似最优解。如图3.3-3所示。第三步，使用已有的data race和atomicity violation检测器对选择的输入进行检测。

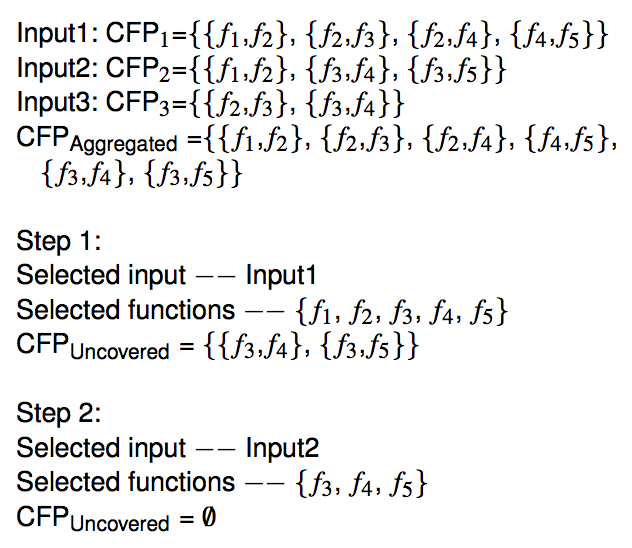


图3.3-3 选择输入

* 1. OSCS

OCSC通过找到order-sensitive critical sections, 来检测非数据竞争引起的concurrency bug。

* + 1. 检测OSCS

如图3.4-2所示，临界区有不同类型。其中真正引发bug的是order-sensitive临界区。

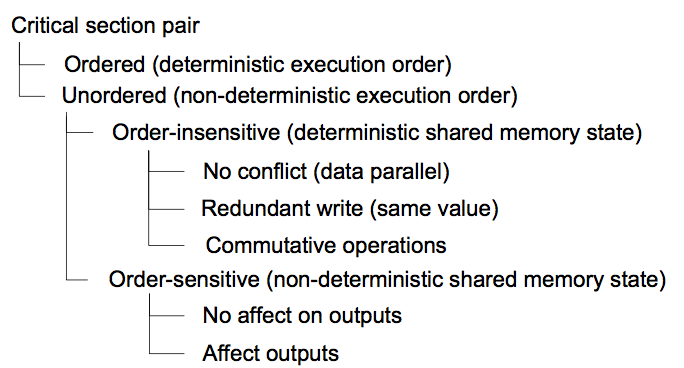


图3.4-2 临界区类型

因此OSCS利用不同临界区的特征，来识别出有序的临界区，无序但是对顺序不敏感的临界区。如果不是上面所述两种临界区，就会被认定为order-sensitive critical section。

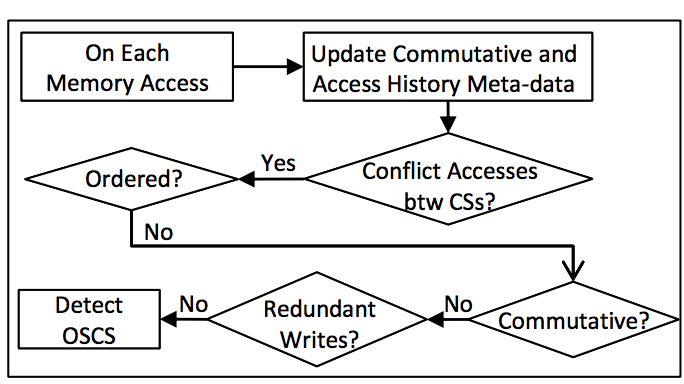


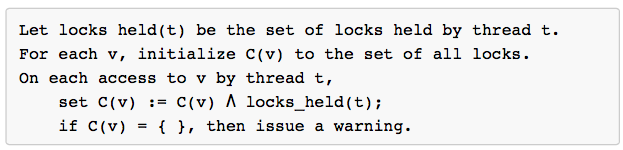
图3.4-3 OSCS检测方法

* 1. Goldilocks

Java以共享内存作为其内存模型[2]，因此存在多线程环境下的数据竞争问题。Goldilocks[1]是在Java虚拟机上实现的，用于数据竞争动态检测的算法，它引入了一种新的运行时错误，被称为DataRaceException。每当程序中出现数据竞争的情况时，虚拟机就会抛出DataRaceException的异常，避免问题继续扩散。

* + 1. Lockset算法

Goldilocks基于Eraser[3]提出的Lockset算法扩展得到。Lockset算法的思想是：如果一个共享变量在程序多线程执行过程中能够始终被一个或多个锁保护的话，那么在该共享变量上不会发生数据竞争；反之，则有可能发生数据竞争。基于这样的观察，提出了Lockset的算法：



算法3.5-1 Lockset

其思想有两个阶段，第一阶段初始化，对于每个线程t，维护一个locks\_held(t)表明当前获得的锁集；对于每一个共享变量v，这个变量在初始化的时候获得程序执行过程中的所有可能锁集C(v)。第二阶段更新，对于当前线程的每一次读写操作，更新当前被访问变量的候选集合C(v)。

Lockset算法是一种动态检测的算法，需要动态地对程序代码进行改写，用以统计读写、加锁释放锁、内存声明还有线程创建等操作。基于对这些操作的Trace，可以用Lockset得到可能出现数据竞争的变量。

* + 1. Goldilocks形式化描述

Goldilocks提出了一种形式化的描述方法，来对读写、加锁释放锁操作等等进行更好的描述与刻画。它将所有需要关注的操作抽象成了三种kind，分别是SyncKind，DataKind和AllocKind。对于每一种Kind，都是一些抽象操作的集合，这些集合在Goldilocks中被定义为：

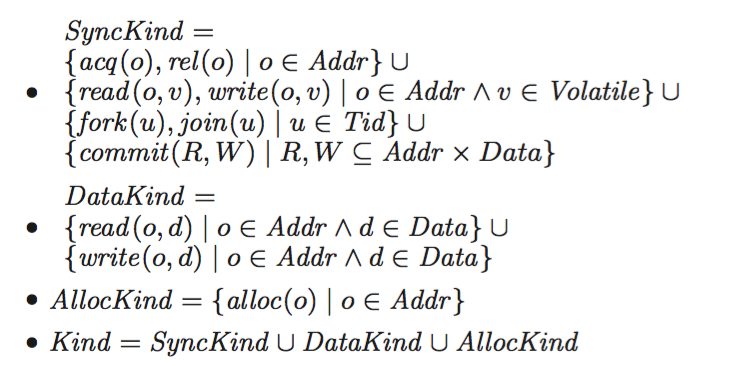


图3.5-1 Goldilocks形式化定义

Lockset的更新规则如下：

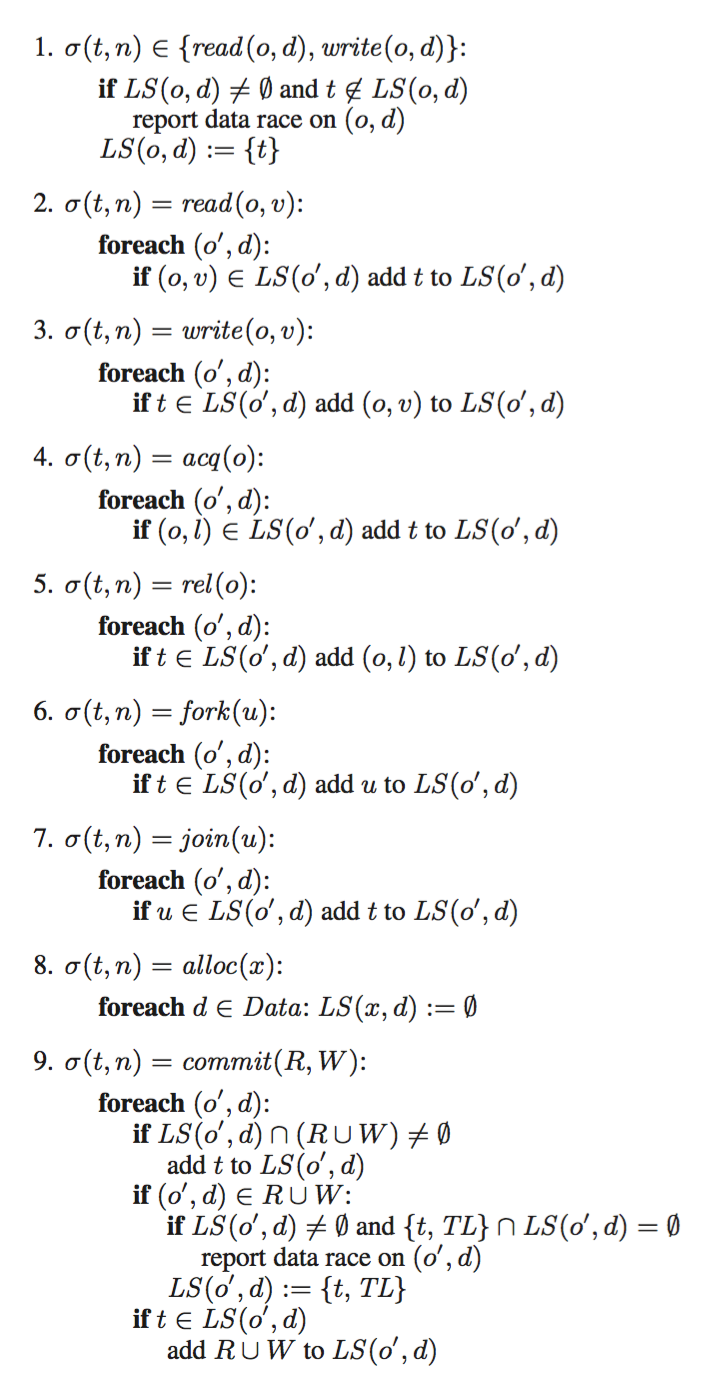


图3.5-2 更新规则

在这样的抽象下，使得Goldilocks对于事务，以及原子变量等等其他的同步操作有更好的支持。

* + 1. 不同实现方式比较

Goldilocks有两种实现方式。在较早发表的论文中，Goldilocks实现在Kaffe（C语言实现的Java虚拟机）的运行时引擎解释器模式下。这样实现的优势在于能够得到源代码在虚拟机内部，内存的布局，以及可以使用虚拟机内部实现的诸多算法。

除了上述方法，Flanagan和Freund在ROADRUNNER上也实现了Goldilocks算法[4]。ROADRUNNER是一个动态分析工具，在这样的实现中，Goldilocks算法是通过在加载的时候，动态地插入Java字节码指令来实现的。

* + 1. 与Eraser比较

之前的Lockset算法比较保守，因此常会在一些没有数据竞争的情况下抛出异常。Goldilocks对于数据竞争的检测更加准确。

除此之外，Goldilocks的形式化抽象，使得它可以支持更多的同步方法。比如对于内存事务的方法进行同步的操作，以及volatile变量的读写等等。这些同步方法在传统的算法中都是没有支持的， Goldilocks可以支持。

* 1. AVIO

AVIO针对原子性问题进行检测。

* + 1. 程序员的原子性思维

AI invariants，交错访问不变性。如果一条指令（invariant指令）与之前的一条指令（preceding指令）访问的是相同内存并且具有原子性，那这条指令就具有AI invariants。

对于不同的程序员来说，对于原子性的考虑是不一样的。同时，想让程序员提供哪些变量可能被线程之间的交叉存取所影响是有难度的，因为他们通常考虑得不够充分，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中的学习过程并不难：

* 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，因为对于有很多交叉存取的程序，每一次执行都可能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，会得到高质量的训练结果。

* + 1. AVIO的算法

检测算法：通过对程序的分析，遵循状态机，如图3-6.1，检测出违反AI invariants的部分。图3-6.1的路径从表3.6-1中列举的交叉存储的八种情况得到。

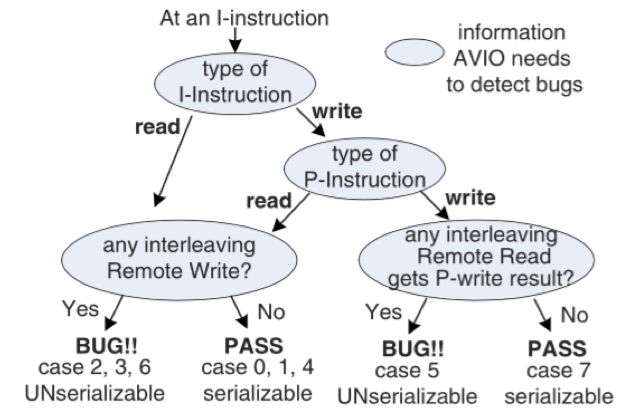
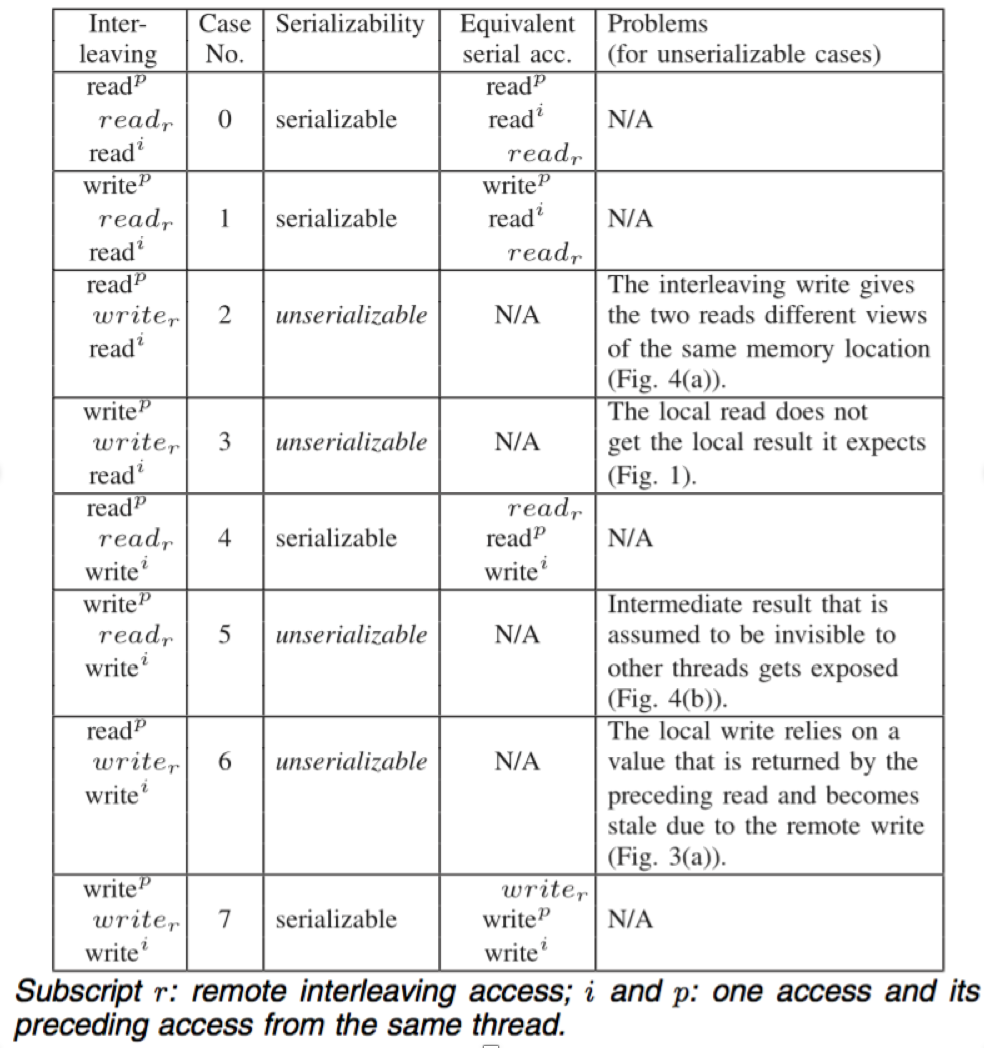


图 3.6-1 AVIO程序分析状态机

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。迭代直至AI集收敛。

表3.6-1 对同一变量的交叉存储的八种情况

1. 分析与比较
   1. 系统优缺点分析
      1. MUVI

MUVI可以发现多变量引发的concurrency bug，对lock-set与happens before算法的改进带来较小的性能负荷。

MUVI中有许多阈值需要用户根据实际情况进行调整，即使提供了灵活性，但给用户带来较大负担。

MUVI可以被已有工具(例如Autolocker 和Colorama)扩展，具有较高的可扩展性。

同时，由于MUVI对程序进行静态分析，没有考虑动态运行过程输入对变量相关性的影响，所以得到的变量相关关系和实际情况存在差异，导致漏报误报等错误。

* + 1. AVIO

AVIO的算法相对简单，可以通过硬件和软件分别进行实现。//优点需要扩充

检测结果依赖输入，不断调整输入来找到AI Invariants，整个过程需要接近100次的程序运行，却仍不能保证所有的AI Invariants都被找到。

* + 1. ConSeq

ConSeq从产生bug的error site开始，一步步反向查找出错路径，所以可以给用户提供更具有解释性的bug report。

由于ConSeq利用了Concurrency bug传播路径较短的特点，无法找到有较长传播路径的Bug。

* + 1. CFP

CFP大大减少检测bug所需要的程序运行次数，提高检测效率。同时，它利用已有的bug detector，具有较高可扩展性。

选择部分输入可能会导致False positive比较高。这是因为，某些并行函数对在特定输入下执行时，才会产生Bug。而本文使用CFP选择了部分输入，可能并不包括这个特定输入，所以Bug并未被检测到。

* + 1. OSCS

OSCS提出order-sensitive critical section的概念。它可以检测多种bug，包括原子性冲突，顺序冲突和多变量引发的bug，而不是局限于一种。通过排除法来检测非数据竞争bug。

它除了软件实现，还有硬件实现。硬件实现只有0.23%的延迟。

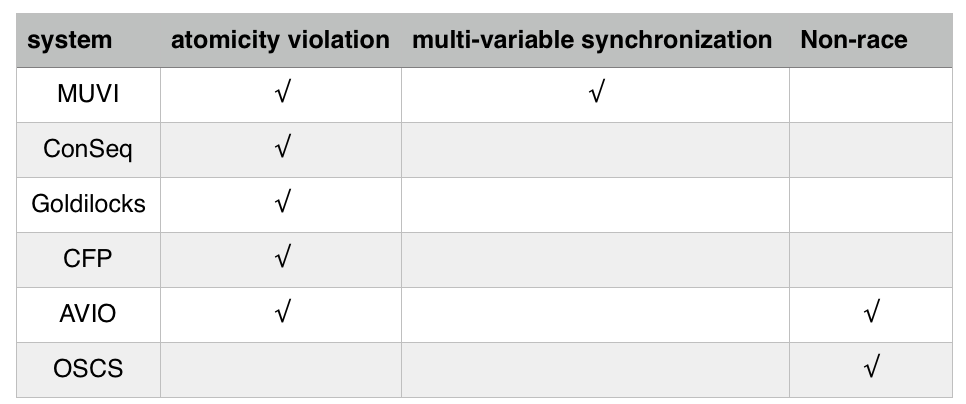
* + 1. Goldilocks

Goldilocks拓展了原本Eraser的Lockset算法，其最大的优势在于，因为给出了形式化的算法描述，使得算法的扩展变得简单。Goldilocks不仅可以检测传统的同步操作，加锁与释放锁造成的数据竞争的Bug，同时也可以支持Transactin，volatile变量等等新的同步方式。而且论文提出了将Goldilocks实现在Java VM层的想法，这样可以复用虚拟机中的代码，使得Goldilocks实现起来也要相对而言更加方便。

Goldilocks也存在一些缺点，因为它是基于动态分析进行的，所以非常依赖程序的执行路径，所以不可能会找到所有的Bug。另外，目前的实现都是针对于Java语言的，对于不同的语言要进行不同的实现，通用性不是特别好。不过从研究的角度来讲，Goldilocks为在检测新的同步方法引入的数据竞争的Bug提供了一个方向。

* 1. 系统原则分析
* Hybrid：Goldilocks仅使用动态分析技术，而MUVI、CFP和ConSeq系统将静态分析和动态分析技术相结合，体现了Hybrid原则。-> G
* KISS：CFP系统在选择输入的过程中，并没有选取最优解，而是使用贪心算法得到近似最优解。它牺牲结果的准确性来简化复杂的NP问题。这体现了KISS(Keep It Simple and Stupid)原则。-> N
* Avoid excessive generality：每个系统试图解决某一类或某几类的concurrency bug，而不是一次性解决所有的问题。这体现了避免过度通用(Avoid Excessive generality)原则。表4.2-1总结了不同系统检测的concurrency bug 种类。

表4.2-1 不同系统检测bug种类



* Principle of least astonishment： user-friendly -> N
* Avoid rarely used components: MUVI用了happens before and lockset -> J
* Be explicit： ConSeq(4 writes) -> N
* Open design: 最后，很多系统都复用了已有的开源工具，而不是再次实现一遍。这体现了open design原则。-> X
* Unyielding foundations rule MUVI -> G
  1. 系统间的比较
     1. AVIO与MUVI

AIVO和MUVI都提供了通用、可扩展的框架。同时，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。[10]把MUVI与AVIO结合起来，实现多变量的原子性冲突检测。两者具体差异如下：

1. 推断concurrency bug的方法不同，MUVI使用了数据挖掘方法找到Bug候选集，主要使用静态分析；而AVIO只用到了一个简单的模式匹配算法，为动态分析。
2. 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要调整输入，并运行多次来收集AI invariants，。
3. MUVI关注多变量关系，而AVIO针对原子性冲突。
   * 1. 其他系统间的比较

MUVI检测多变量之间引发的concurrency bug，而其他系统仅针对单变量。AVIO和OSCS检测Non-Race concurrency bug，而其他系统偏重由data-race引发的bug。CFP跟其他系统不同在于，它重点放在提高检测效率上。和CFP相比，OSCS的检测粒度更细。CFP的粒度为函数，而OSCS的粒度为临界区。ConSeq则巧妙地利用反向分析的方法检测bug。Goldilocks仅针对Java语言，可以检测不同同步方法下的bug，例如transaction和barrier。

1. 总结与展望

本文提到的几个concurrency bug检测系统，构思精巧，能够有效针对不同的方面进行Concurrency bug的检测。

参考文献

1. Elmas T, Qadeer S, Tasiran S. Goldilocks: a race and transaction-aware java runtime[C]// Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2007, 42(6): 245-255.
2. Manson J, Pugh W, Adve S V. The Java memory model[M]. ACM, 2005.
3. Savage S, Burrows M, Nelson G, et al. Eraser: A dynamic data race detector for multithreaded programs[J]. ACM Transactions on Computer Systems (TOCS), 1997, 15(4): 391-411.
4. Flanagan C, Freund S N. The RoadRunner dynamic analysis framework for concurrent programs[C]//Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. ACM, 2010: 1-8.
5. Lu S, Park S, Hu C, et al. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs[C]//Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. ACM, 2007, 41(6): 103-116.
6. Zhang W, Lim J, Olichandran R, et al. ConSeq: detecting concurrency bugs through sequential errors[C]// Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems. ACM, 2011, 46(3): 251-264.
7. Deng D, Zhang W, Lu S. Efficient concurrency-bug detection across inputs[C]. Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, 2013, 48(10): 785-802.
8. Lu S, Park S, Seo E, et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics[C]// Proceedings of the 13th international conference on Architectural support for programming languages and operating systems. ACM, 2008, 43(3): 329-339.
9. Huang R, Halberg E, Suh G E. Non-race concurrency bug detection through order-sensitive critical sections[C]// Proceedings of the 40th Annual International Symposium on Computer Architecture. ACM, 2013, 41(3): 655-666.
10. Lu S, Park S, Zhou Y. Detecting concurrency bugs from the perspectives of synchronization intentions[J]. Parallel and Distributed Systems, IEEE Transactions on, 2012, 23(6): 1060-1072.
11. Lu S, Tucek J, Qin F, et al. AVIO: detecting atomicity violations via access interleaving invariants[C]// Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. ACM, 2006, 40(5): 37-48.