**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。Concurrency bug有很多种，例如数据竞争和原子性冲突。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。本文对几个不同的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. There are many types of Concurrency bugs, such as data race and atomicity violation. But 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并发Bug。例如即使有相同的输入，如果线程没有适当地同步，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。

因此产生了基于对程序的静态分析和动态分析的并发Bug检测技术，。

本文总结了近年来学术界为检测并发Bug的尝试，例如MUVI[5], ConSeq[6], CFP[7], Goldilocks[1]。本文列举了这些技术对应的系统，分析比较了不同系统的特点与缺陷。

1. **并发Bug**
   1. 并发Bug分类

常见的并发Bug包括以下几种类型。

* + 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。如果没有同步机制，内存访问可能存在任意地顺序导致内存状态不同。

* + 1. 违反原子性操作(Atomicity violation)

如图2-1所示，当thd->proc\_info没有被同步机制保护时，可能在1.1和1.2之间，2.1将它的值设置为NULL, 从而导致错误。所以需要对1.1-1.2以及2.1用临界区保护起来，保证原子执行。

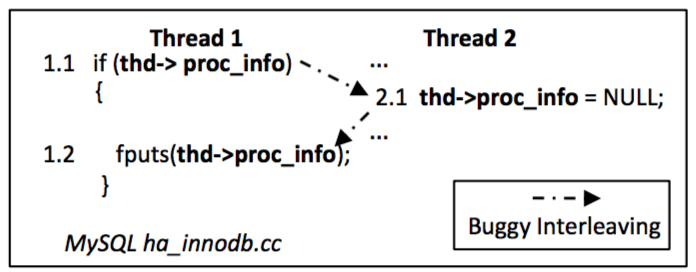


图2-1 MySQL中的Atomicity violation bug

* 1. 并发Bug特征

对现实世界concurrency bug特征的研究发现：

* 非死锁concurrency bugs里的三分之一bugs是由于程序员所安排的时序被破坏而引发的，这些concurrency bugs不能通过简单的同步语义，例如锁和事务表达出来。
* 被检测的非死锁concurrency bugs里的34%的bugs，均涉及到不能被已有bug detection工具发现的多变量问题。MUVI重点解决了多变量导致的concurrency bug[5]。
* 被检测的concurrency bugs里的大约92%的bugs，都可以以不多于4次的内存读写在确定的顺序下触发。在ConSeq里面也有用到这个特征[6]。
* 被检测的非死锁concurrency bugs里的73%的bugs,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。

1. **并发BUG检测技术**
   1. MUVI

之前已有不少检测并发错误的方案，但它们都无法处理多变量的问题。这是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并发错误。多变量主要因为两个原因引起并发错误：

* 相关变量没有被同时更新(inconsistent updates)
* 相关变量在存取操作时，没有被同时保护(multi-variable concurrency)
  + 1. MUVI实现

MUVI(Multi-variable Inconsistency)的实现分为两个环节，首先需要发现代码中变量的关联关系，然后基于变量的关联关系，去发现（1）同时更新的bug（2）多变量的concurrency bugs。

* + - 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是指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是指通过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会根据两个评价的标准来确定最后的访问的关联性，分别是Support和Confidence。

* + - 1. Bug Detection

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

对于没有进行同步更新的Bug，最初的算法就是在结果中，对于Write(x) => AnyAcc(y)这样的关系，都去进行分析，找到代码中所有违背这样的关系的存在。所有违背这样的关系的函数，都会被列为同步更新Bug的候选集合。然后进行排序。这样的检测是可以被静态分析来实现的。

对于两个变量的数据竞争Bug，MUVI最主要的实现是扩展了目前的算法Lockset。因为MUVI所关注的变量访问的关联性是使用静态分析，而Lockset是使用动态分析来确定是否有数据竞争的问题的，因此需要使用代码动态翻译的技术来将MUVI对变量的关联性分析结合Lockset算法来进行执行。

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。

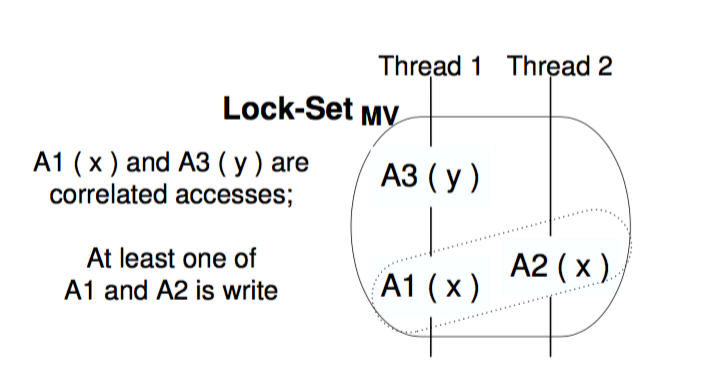


图3.1-1 MUVI扩展的Lockset算法

* 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），利用并发Bug的特性，反向找到可能存在的并发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：

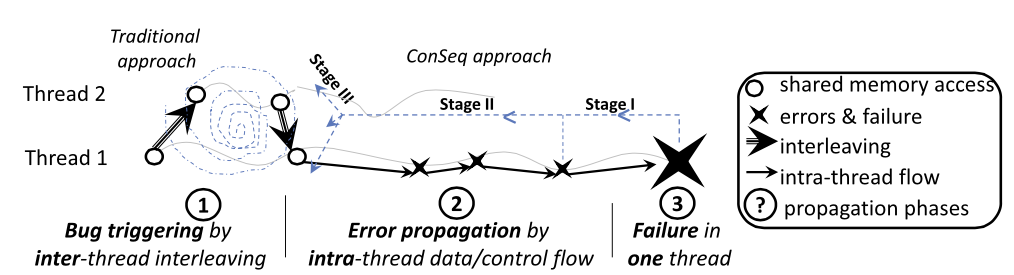


图3.2-1 Bug生命周期

* 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。

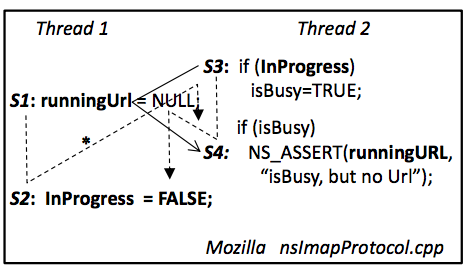


图3.2-2 Bug传播过程

* + 1. 并发Bug的特性

通过对已有的70个非死锁并发错误的分析，发现以下3个并发Bug的特性：

* 并发错误通常发生在一个线程里。即错误的产生是由于多线程，但结果通常仅有一个线程产生了可见错误。说明可以将分析阶段分为并发分析和顺序分析两个阶段。
* 并发Bug的Failure pattern和顺序Bug相似。说明可以根据顺序Bug的failure pattern找到可能由并发Bug导致failure的地方。
* 传播路径通常很短。即读入错误数据之后，很快会产生Failure。说明很容易找到产生Bug的代码。
  + 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。

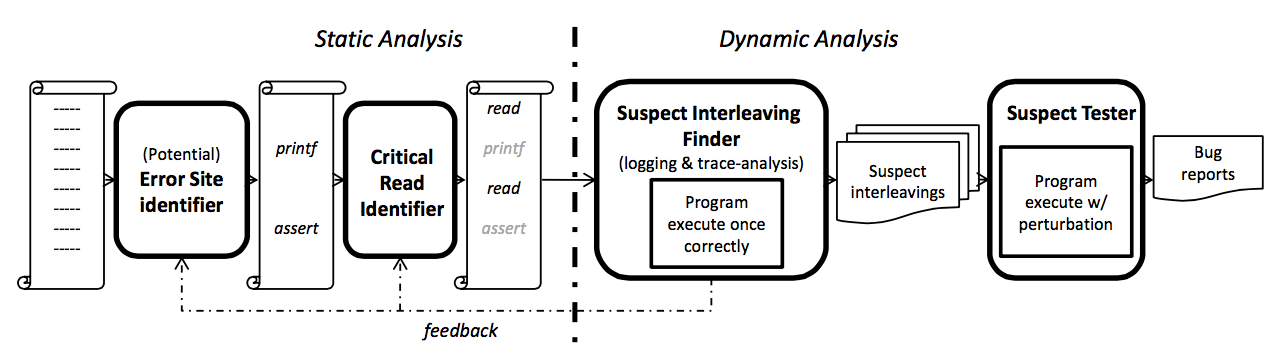


图3.2-3 ConSeq结构

* 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. 缺点

由于ConSeq利用了并发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

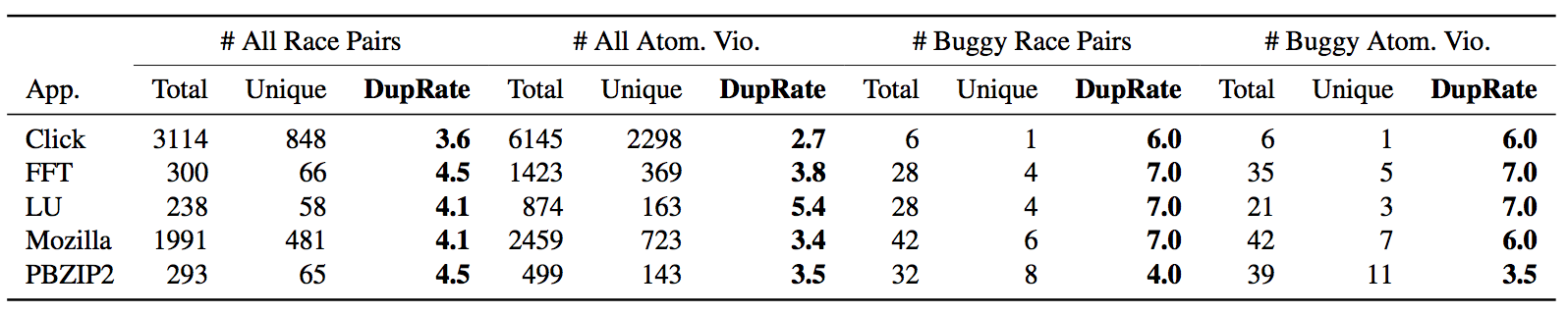
根据每个输入所覆盖的并行函数对(CFP)选择输入集合，从而减少对同一Bug的重复检测。

* + 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倍道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。

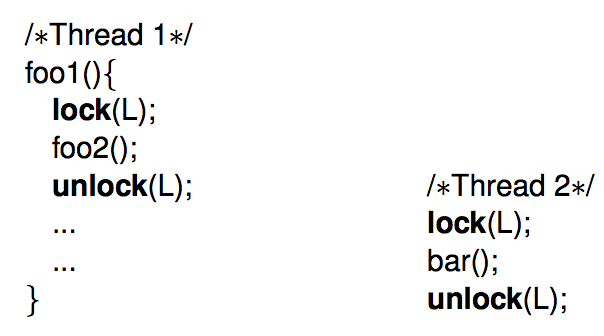


图3.3-1 并行函数对

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。

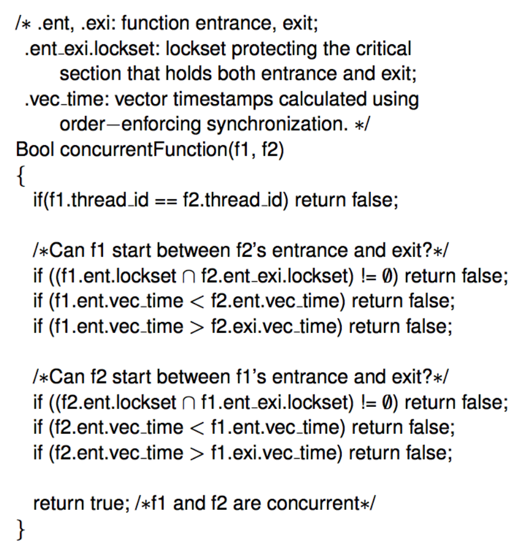


图3.3-2 判断并行函数对伪代码

* + 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。

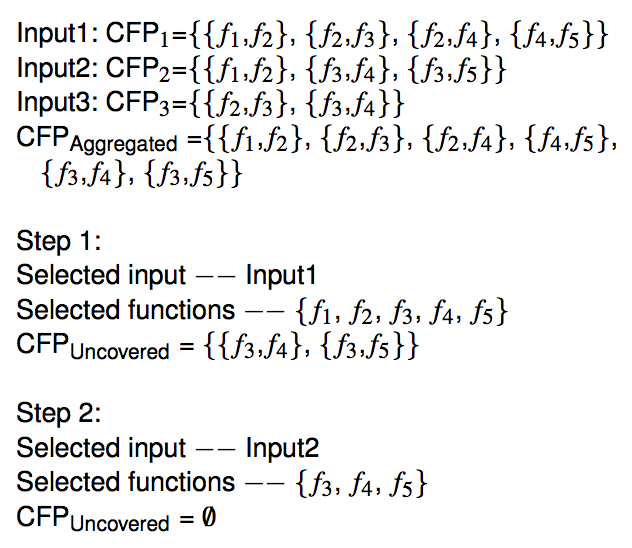


图3.3-3 选择输入

* + 1. 缺点

采用这种方法选择部分输入，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。

还有目前只针对data-race和atomicity-violation两种类型的bug进行检测。

* 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. 非数据竞争Bug

数据竞争问题是由对相同内存的并行读写导致的，通常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3.4-1所示，即使对gCurrentScript的访问被锁保护起来了，由于执行顺序的不同，还是会引发错误。这样的代码叫做order-sensitive critical section。

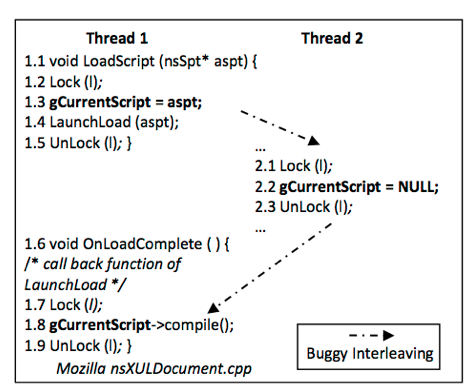


图 3.4-1 order-sensitive 临界区

* + 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。

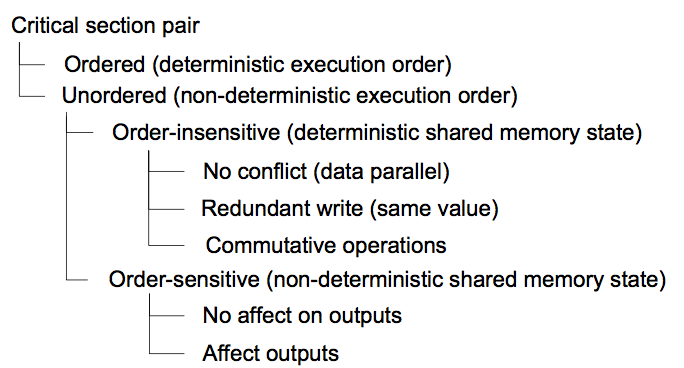


图3.4-2 临界区类型

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。

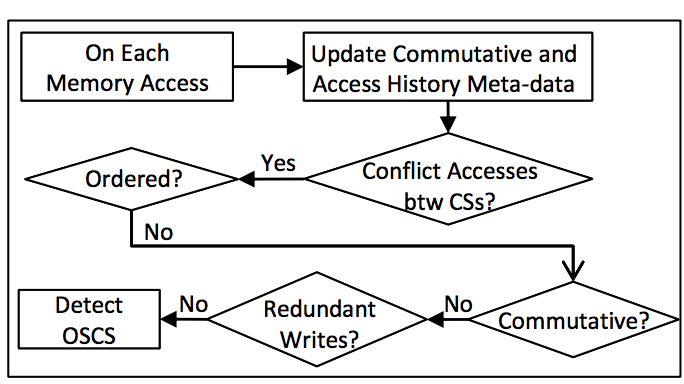


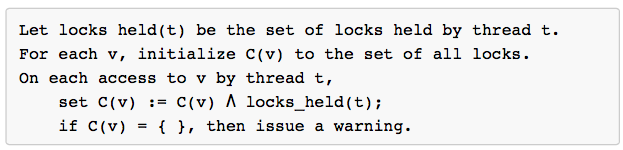
图3.4-3 OSCS检测方法

* 1. Goldilocks

Java是目前而言最受欢迎的编程语言之一，在业界中，有越来越多的公司选择Java作为开发语言。Java是以共享内存作为其内存模型的语言[2]，因此也会遇到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，可以及时处理，以免问题继续扩散。而如果在执行过程中，没有任何的DataRaceException抛出，则可以认为在程序的执行流中没有数据竞争。

* + 1. 相关工作

Goldilocks是一个基于Eraser[3]提出的Lockset算法扩展而来的算法，Lockset算法的思想就是，在多线程程序中，一般程序员都会使用锁对临界区进行保护，临界区中一般都是共享变量的访问操作，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肯定不会发生数据竞争。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

其思想非常简练，一共有两个阶段，第一个阶段为初始化阶段，对于每个线程t，都会维护一个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)。之后是更新阶段，对于当前线程的每一次读写操作，都会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得到会出现数据竞争的变量。

* + 1. Goldilocks实现
       1. 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：

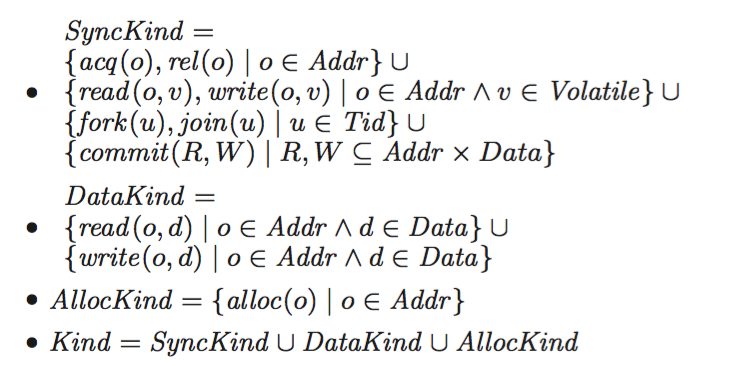


图3.5-1 Goldilocks形式化定义

而关于Lockset的更新规则，也进行了相应的改变：

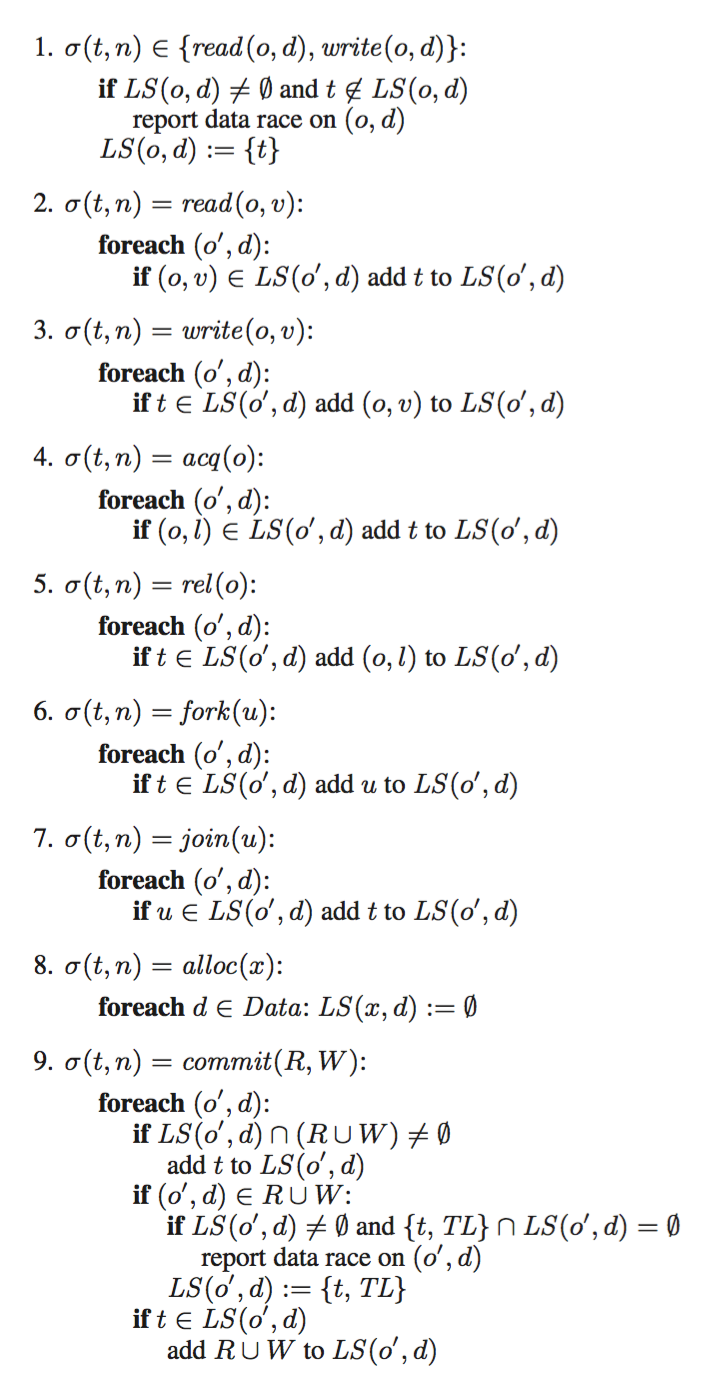


图3.5-2 更新规则

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. 例证

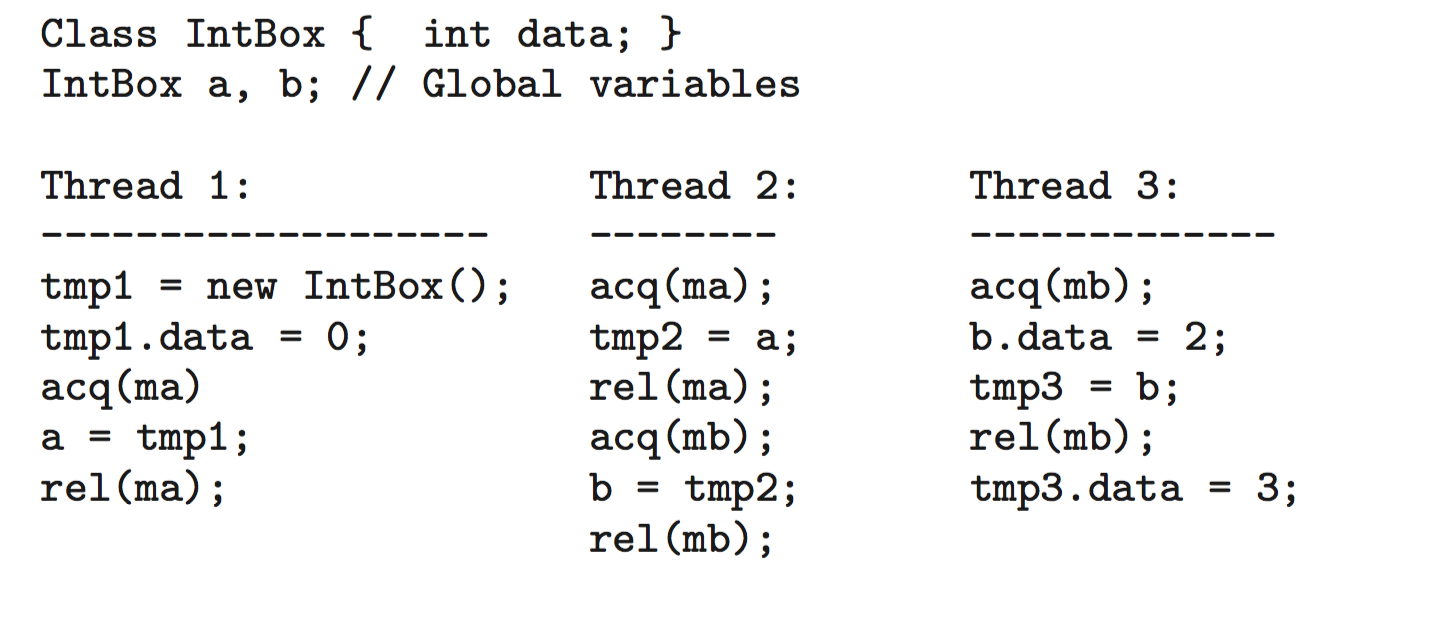


图3.5-3 示例代码

对于示例中的代码，线程1中的代码先执行，之后再执行线程2中的代码，最后运行线程3。Goldilocks算法在其上的运行情况如图所示：

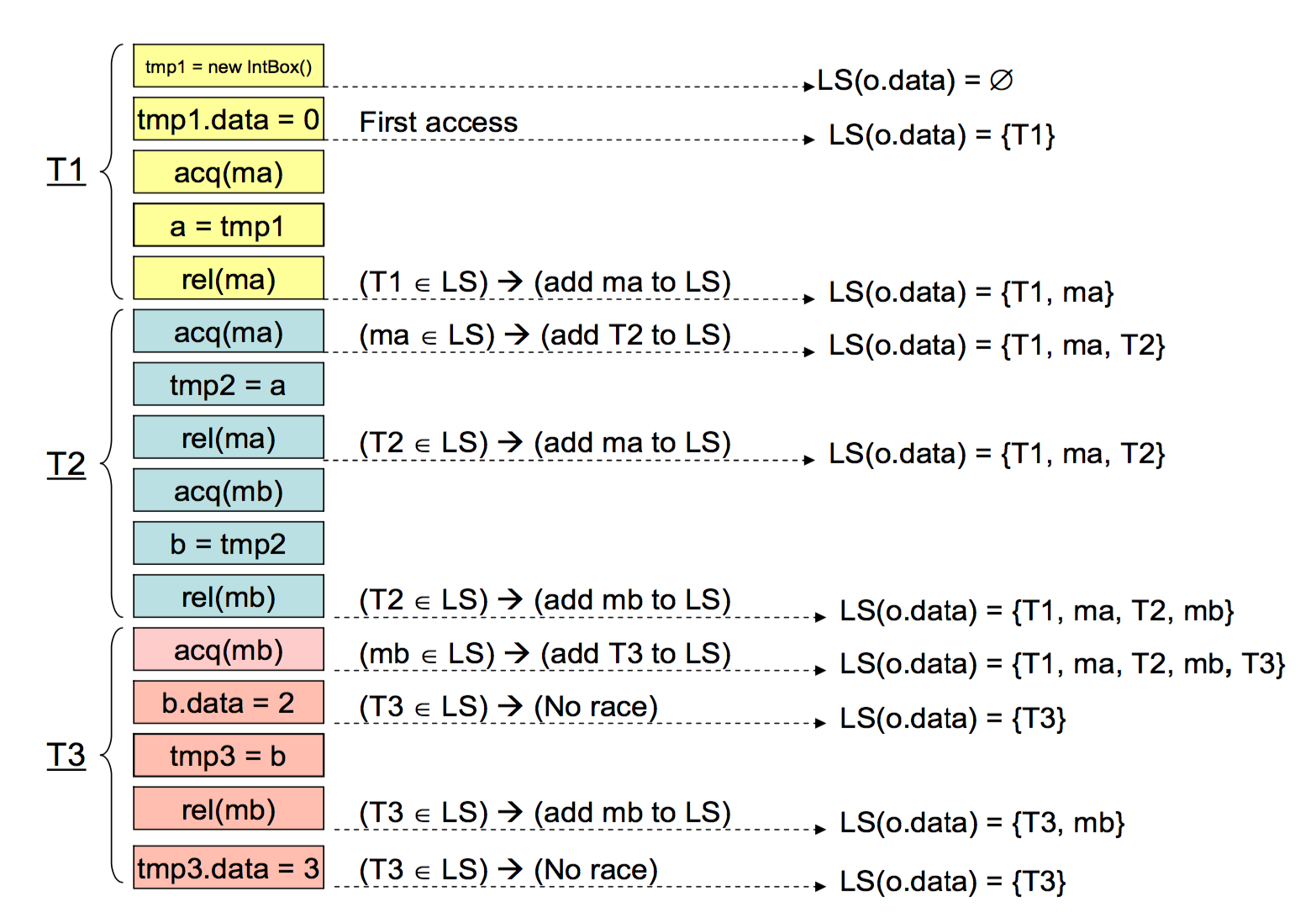


图3.5-4 更新规则

在Goldilocks的运行情况中可以得出，在这样的执行顺序中，不会出现数据竞争的情况。

* + 1. 不同实现方式比较

因为Goldilocks是基于Lockset的，所以也是一种动态的检测方法。Goldilocks会监控Java字节码的运行，在这样的等级上，每一次变量的读写访问，或者是同步操作的指令，都是一条字节码的指令。同时每一个字节码的指令都会对应一个变量或者源码中的一行。

在已经发表的论文中，Goldilocks的实现有两个。在较早时间发表的论文中，Goldilocks是实现在Kaffe的运行时引擎的解释器模式下，Kaffe是一个使用C语言实现的Java虚拟机。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，并发表了论文[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，是一种比较保守的算法，因此常常会在一些并没有数据竞争的情况下也会报出异常。举例来说明，在下面的实现中，Thread1中的对于data变量的初始化就会报出数据竞争的异常，因为没有任何一个锁保护该变量。

而在Goldilocks的实现中，对于数据竞争的检测是更加准确的。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，而在Goldilocks就可以得以支持。

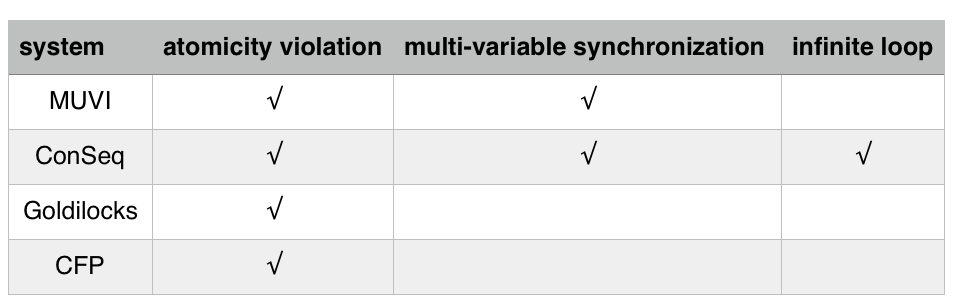
1. 比较

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。

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。

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4-1总结了不同系统检测的concurrency bug 种类。

表4-1检测Bug种类



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。

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并发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.