**并发BUG检测技术研究**

刘宁 李菁菁 夏亦谦 高策 张坚鑫

摘要

关键词 并发，Bug检测

Researches on Concurrent Bug Detection

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

Abstract

Keywords Cocurrency, Bug detection

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，包括（）。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。

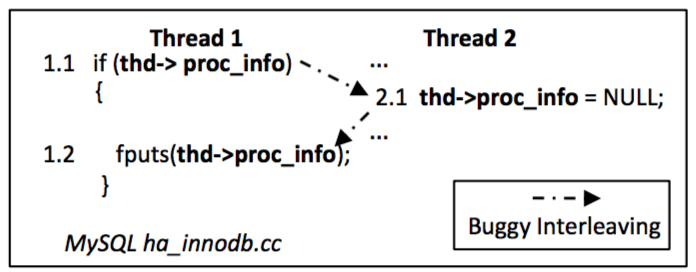
* + 1. 数据竞争(Data race)

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。

//举个栗子

* + 1. 违反原子性操作(Atomicity violation)

如图所示，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\* MySQL中的Atomicity violation bug

* + 1. //更多地并发Bug
  1. 并发Bug特征

//有哪些哪些特征，而且在接下来的论文当中有用到其中的某些特征。

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(加上引用)，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(全称是什么)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
  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：

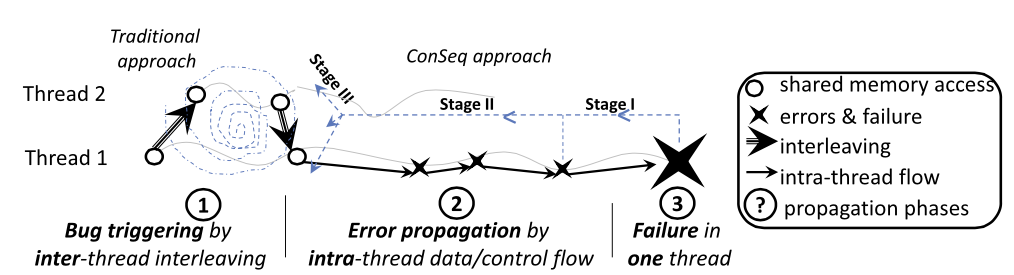


图3.2-1 Bug生命周期

* 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，这是触发阶段；然后S4中读入了runningURL错误数据，这是传播阶段；最后不满足断言，这是失败阶段。

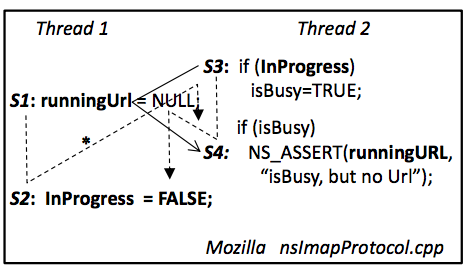


图3.2-2 Bug传播过程

* + 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。

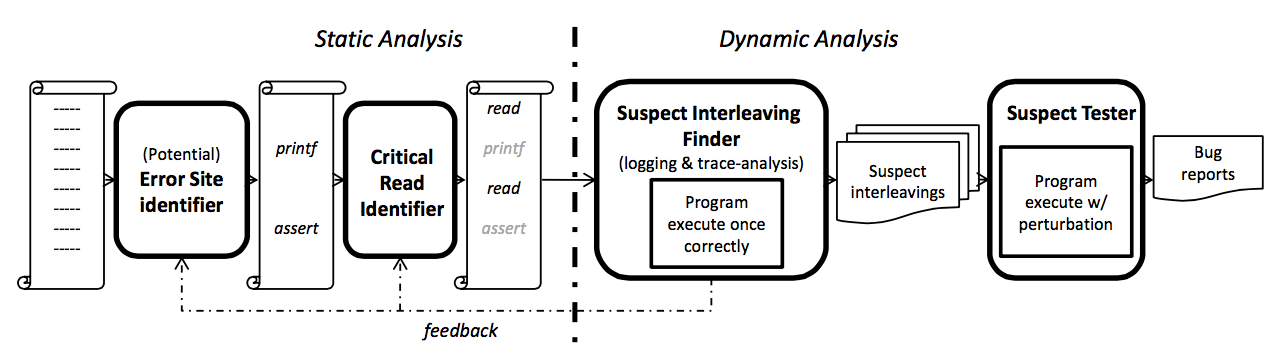


图3.2-3 Conseq结构

Error-site identifier，通过对binary的静态分析，找出五类可能的failure的位置，通过一些特征值来发现，比如assert，printf，fprintf，stderr等，还有一些工具，SCC，Daikon（用于发现implicit errors）；如图1的代码示例，通过静态分析，可以找出S4是可能的failure的位置。

Critical-read identifier，用静态分析找到影响error site的critical-read（short propagation distance）。Critical-read指的就是导致程序失败的读操作。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1的代码示例，在这个代码片段中，虚线代表不会产生 bug的执行路径，如S1->S2->S3->S4或者S3->S4->S1->S2；实线代表会产生bug的执行路径：S3->S1->S4。在这段代码中，Critical-read就是S4，即在进行断言操作时读到了个空值，导致程序中断。

Suspicious-interleaving finder，动态分析，运行一次程序，得到运行路径以及control/data依赖图，通过trace和依赖图来找到可能的关联变量；再对它们进行分析，排除掉不可能的，剩下的就是可能性较大的bug。当要判断R与W'是否有关联（数据依赖）时，它们必须要满足三个条件，（一）它们必须都访问相同内存；（二）W'必须在R之前；（三）被W'写入内存的值在被R读取之前不能被重写。图3就展示了三种不满足（三）的情况：在W’操作之后，总会有一次W操作先于R被执行，其中W、W’和R访问的都是同一块内存。图3(a)是线程内的一种执行顺序，W在W’和R之间被执行；图3(b)是由于一些栅栏同步机制强制规定了W’必须在W之前被执行，同时W必须在R之前被执行；图3(c)是互斥现象，如W’和W与R互斥，或者W’与W和R互斥。只要满足这三种情况的一种，R原本该读到的由W’写入的值就会被W覆盖掉，从而判断出关联变量为W。以上是Suspicious-interleaving identification的核心分析方法。在图3.2-4的代码示例中，可以分析出来，可能导致程序中断的共享内存访问操作是S1。

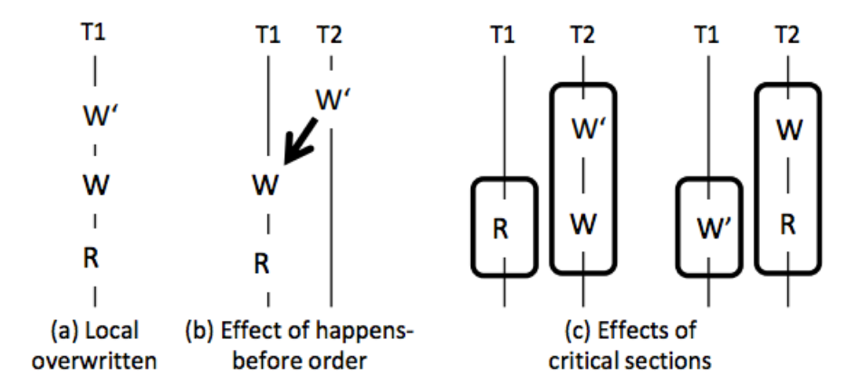


图3.2-4 W'总会先于R的执行顺序

Suspicious-interleaving tester，动态分析，找到那些真正会导致的failure的bug。在这最后一步中，ConSeq会在critical read指令之前插入一些条件延迟（time-out）来改变程序的执行顺序。在图1的示例代码中，程序本该以S3->S4->S1->S2的顺序运行，ConSeq在S4之前插入延迟，使得程序以S3->S1->S4的顺序执行，然后产生中断。这样就成功验证了这段代码是存在Bug的。

* + 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。

//有没有例子

* 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。

针对上述步骤，已有很多研究对此进行改进和优化。对于第一步，无论是对但进行还是多进程程序，很多技术已经能够自动生成高代码覆盖率的输入。对于第二步，很多工具已经能够检测出不少类型的interleaving bug，例如data race, atomicity-violation等。对于第三步，也有各种schemes能够高效的验证suspicious buggy interleaving。

然而这些方法都没有改变第二步和第三步的基础实现，这导致原本就存在的性能损耗没有得到解决，通常会带来10倍到100倍的性能下降，因此需要新的方法来解决性能问题。

主要原因是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。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。真实bug的report也是相似的，对data race平均有4-7个输入检测到，对atomicity violation平均有3.5-7个输入检测到。

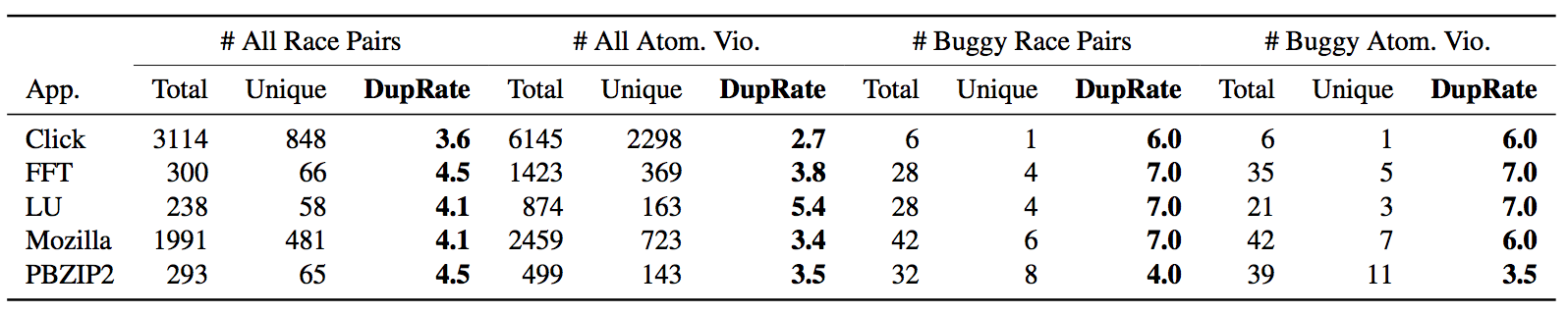


表3.3-1 检测重复率

* + 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。

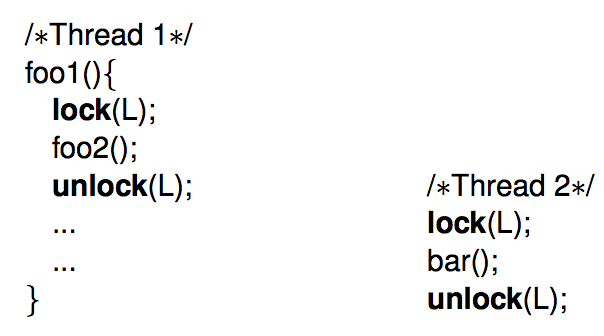


图3.3-1 并行函数对

可以使用同步信息来判断函数是否能够并行执行。//这里具体的同步信息还需要扩展

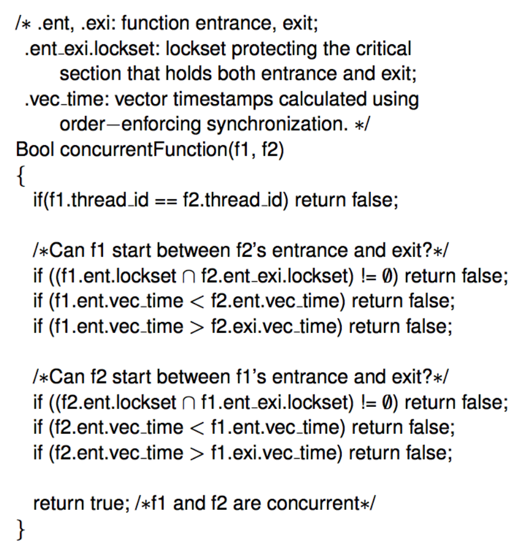


图3.3-2 判断并行函数对伪代码

* + 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，在3个输入中，Input1可以覆盖的CFP最多，因此先选择Input1；此时只有两个CFP没有被覆盖，在Input2和Input3中选择Input2，因为它可以覆盖这两个函数对；最后发现没有被覆盖的CFP集合为空，算法结束。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。（这里可以把不同detector的引用加进来）

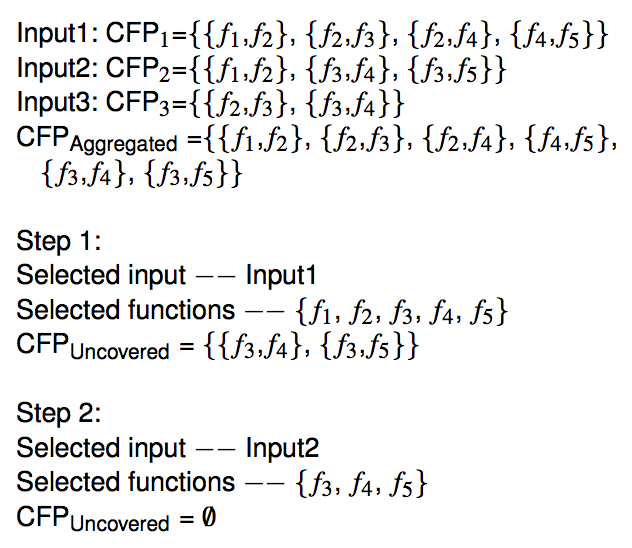


图3.3-3 选择输入

* + 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。(false positive的百分比是多少)

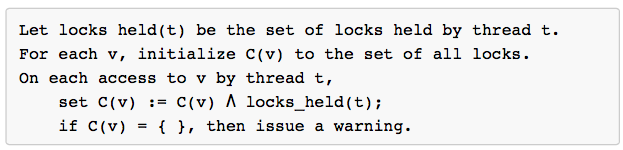
还有目前只针对data-race和atomicity-violation两种类型的bug进行检测。

* 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3.4-1 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：

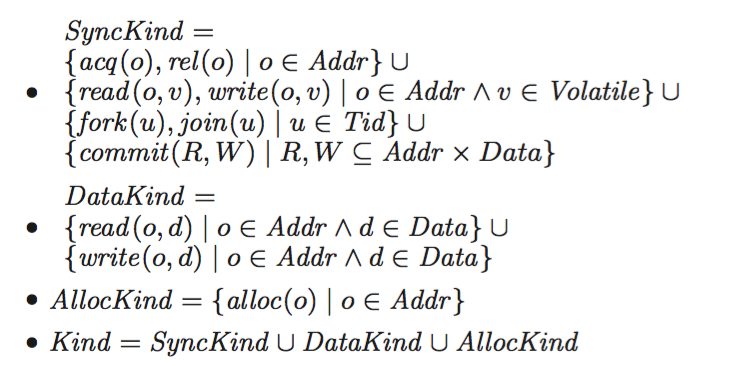


图3.4-1 Goldilocks形式化定义

而关于Lockset的更新规则，也进行了相应的改变：

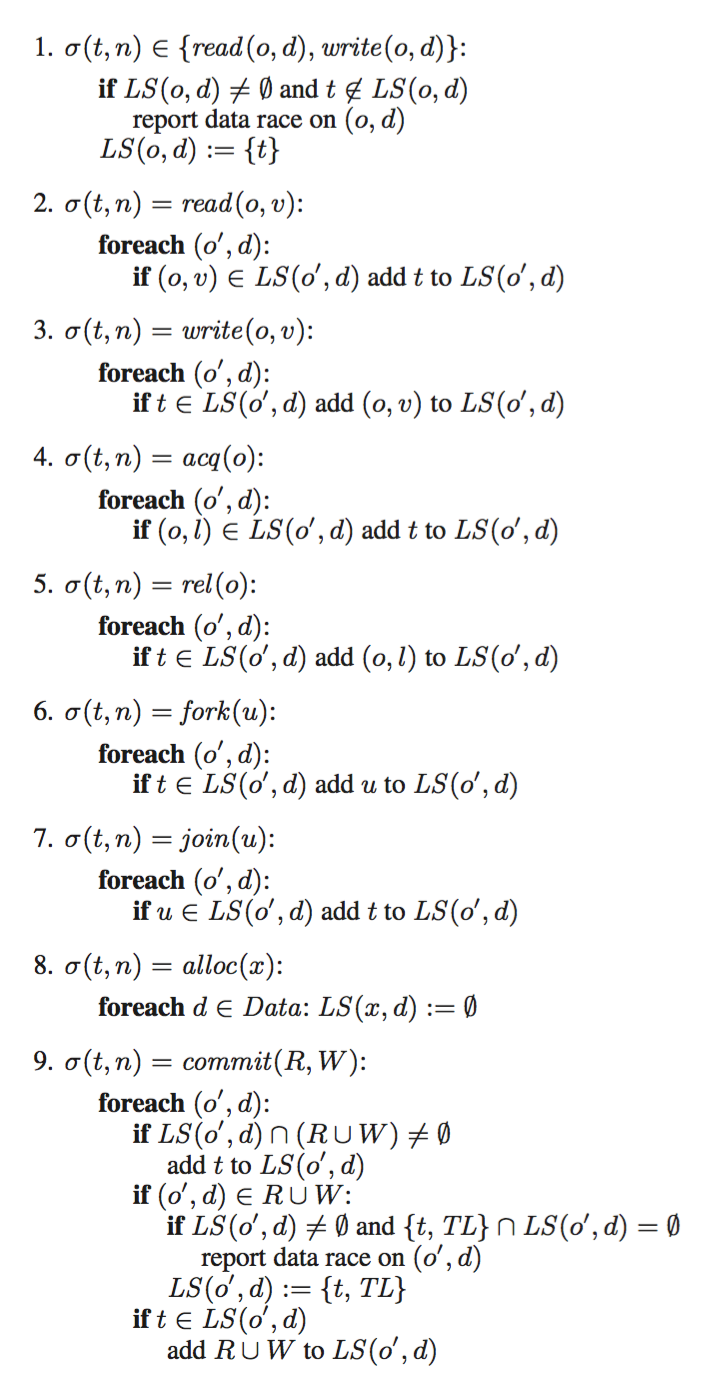


图3.4-2 更新规则

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

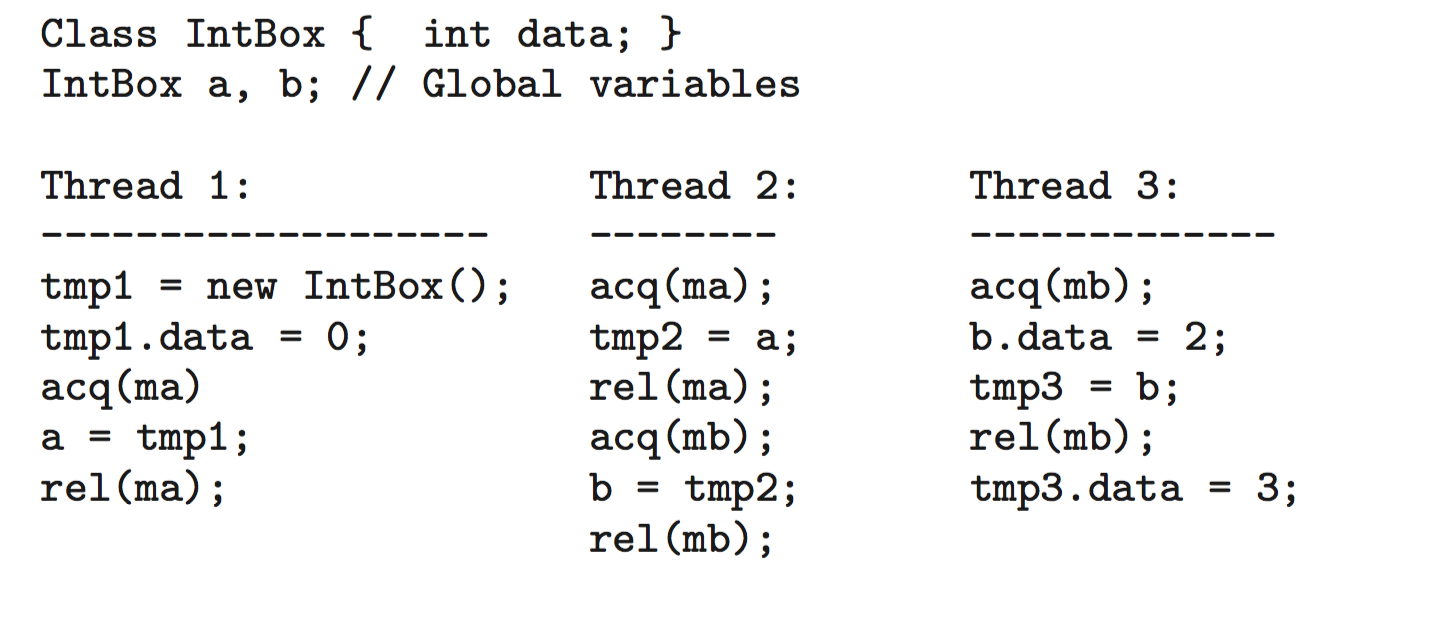


图3.4-3 示例代码

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：

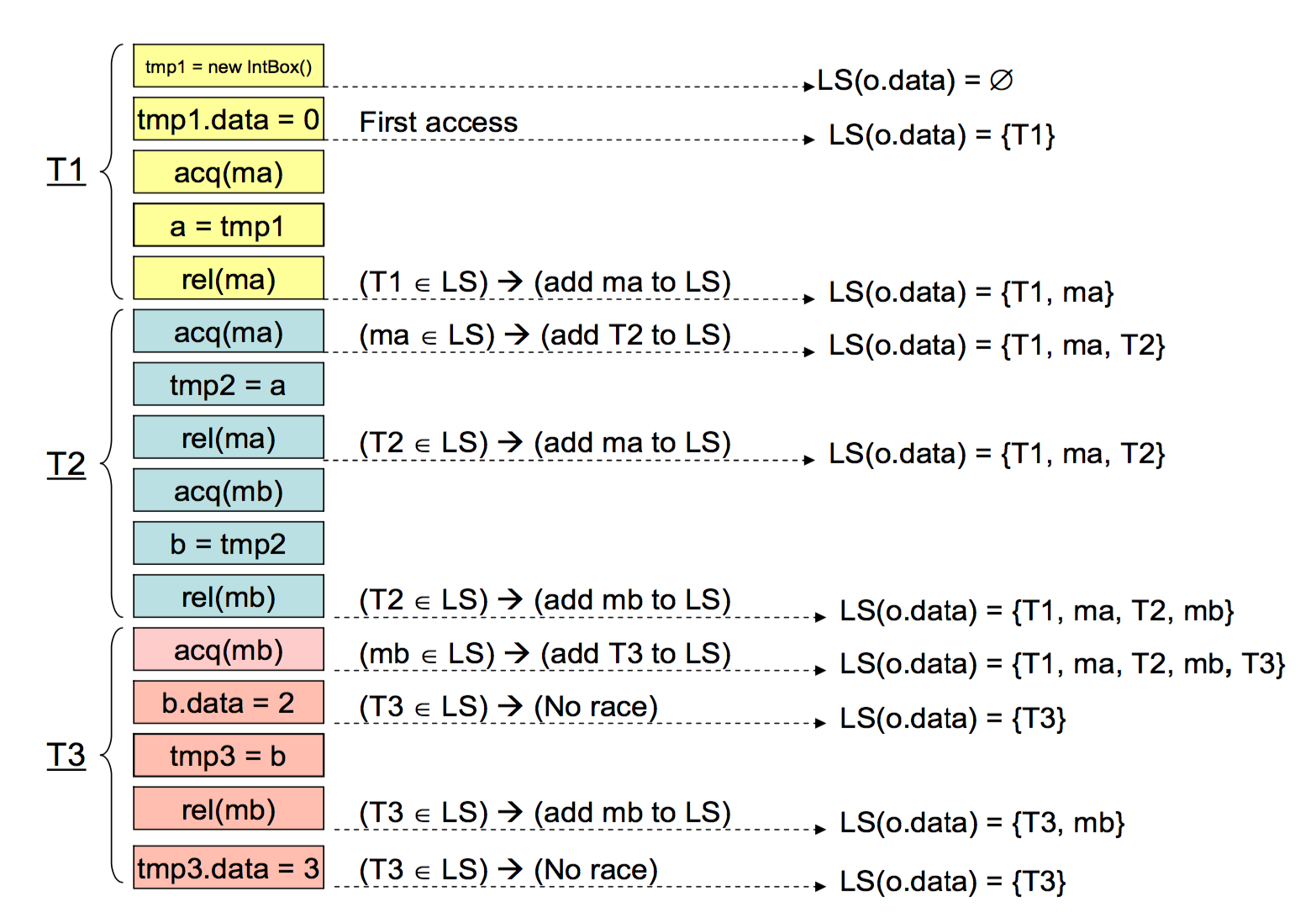


图3.4-4 更新规则

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. OSCS

1. 比较//加CSP
   1. 检测Bug种类
   2. 误报(False positive)
   3. 漏报(False negative)的比例
2. 总结与展望

参考文献

1. Elmas T, Qadeer S, Tasiran S. Goldilocks: a race and transaction-aware java runtime[C]//ACM SIGPLAN Notices. 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]//ACM SIGOPS Operating Systems Review. ACM, 2007, 41(6): 103-116.
6. Zhang W, Lim J, Olichandran R, et al. ConSeq: detecting concurrency bugs through sequential errors[C]//ACM SIGPLAN Notices. ACM, 2011, 46(3): 251-264.
7. Deng D, Zhang W, Lu S. Efficient concurrency-bug detection across inputs[J]. ACM SIGPLAN Notices, 2013, 48(10): 785-802.
8. Elmas T, Qadeer S, Tasiran S. Goldilocks: a race and transaction-aware java runtime[C]//ACM SIGPLAN Notices. ACM, 2007, 42(6): 245-255.
9. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics(ASPLOS'08)
10. Huang R, Halberg E, Suh G E. Non-race concurrency bug detection through order-sensitive critical sections[C]//ACM SIGARCH Computer Architecture News. ACM, 2013, 41(3): 655-666.
11. 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.
12. Lu S, Tucek J, Qin F, et al. AVIO: detecting atomicity violations via access interleaving invariants[C]//ACM SIGOPS Operating Systems Review. ACM, 2006, 40(5): 37-48.