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。

2. Concurrency Bug

2.1. Concurrency Bug 分类

2.1.1. 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。

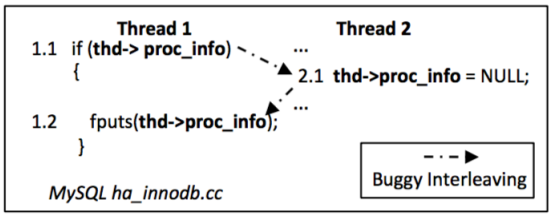

图 2-1 数据竞争

2.1.2. 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。

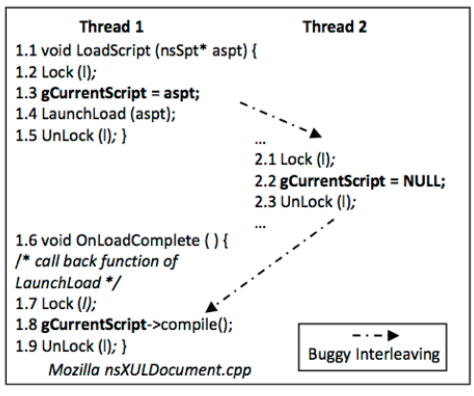

图 2-2 非数据竞争

2.2. Concurrency Bug 特征

对现实世界 concurrency bug 特征的研究发现：

A. 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]

B. 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]。

C. 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]。

D. 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。

E. 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]

F. 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]

3. Concurrency Bug 检测技术

3.1. 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)

3.1.1. 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。

3.1.1.1. 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。

3.1.1.2. 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。

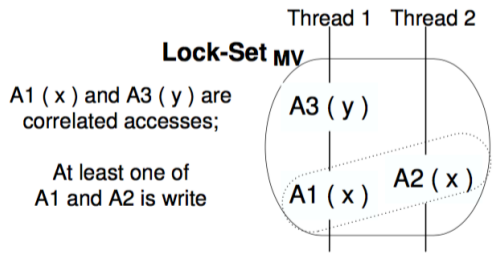

图 3.1-1 MUVI 扩展的 Lockset 算法

3.2. 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（分别有产生，传播和造成系统崩溃三个

阶段），利用 Concurrency bug 的特性 C、E 和 F，反向找到可能存在的 Concurrency bug。

3.2.1. Bug 生命周期

如图 3.2-1 所示，Bug 的生命周期包含三个过程：

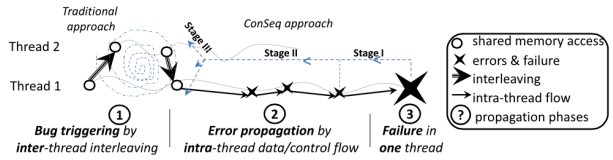

图 3.2-1 Bug 生命周期

- 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。

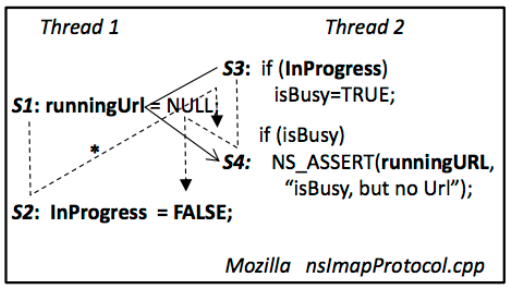

图 3.2-2 Bug 传播过程

3.2.2. 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。

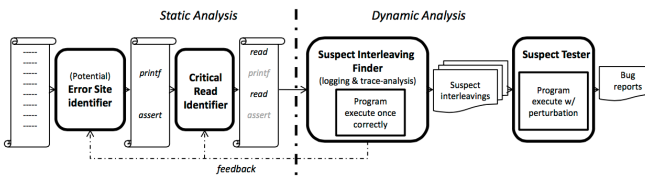

图 3.2-3 ConSeq 结构

- 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。

3.3. 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。

3.3.1. 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

	# All Race Pairs			# All Atom. Vio.			# Buggy Race Pairs			# Buggy Atom. Vio.		
App.	Total	Unique	**DupRate**	Total	Unique	**DupRate**	Total	Unique	**DupRate**	Total	Unique	**DupRate**
Click	3114	848	**3.6**	6145	2298	**2.7**	6	1	**6.0**	6	1	**6.0**
FFT	300	66	**4.5**	1423	369	**3.8**	28	4	**7.0**	35	5	**7.0**
LU	238	58	**4.1**	874	163	**5.4**	28	4	**7.0**	21	3	**7.0**
Mozilla	1991	481	**4.1**	2459	723	**3.4**	42	6	**7.0**	42	7	**6.0**
PBZIP2	293	65	**4.5**	499	143	**3.5**	32	8	**4.0**	39	11	**3.5**

3.3.2. 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。

```
/*Thread 1*/
foo1(){
  lock(L);
  foo2();
  unlock(L);
  ...
  ...
}
                    /*Thread 2*/
                    lock(L);
                    bar();
                    unlock(L);
```

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。

```
/* .ent, .exi: function entrance, exit;
 .ent_exi.lockset: lockset protecting the critical
     section that holds both entrance and exit;
 .vec_time: vector timestamps calculated using
     order−enforcing synchronization. */
Bool concurrentFunction(f1, f2)
{
  if(f1.thread_id == f2.thread_id) return false;

  /*Can f1 start between f2's entrance and exit?*/
  if ((f1.ent.lockset ∩ f2.ent_exi.lockset) != ∅) return false;
  if (f1.ent.vec_time < f2.ent.vec_time) return false;
  if (f1.ent.vec_time > f2.exi.vec_time) return false;

  /*Can f2 start between f1's entrance and exit?*/
  if ((f2.ent.lockset ∩ f1.ent_exi.lockset) != ∅) return false;
  if (f2.ent.vec_time < f1.ent.vec_time) return false;
  if (f2.ent.vec_time > f1.exi.vec_time) return false;

  return true; /*f1 and f2 are concurrent*/
}
```

图 3.3-2 判断并行函数对伪代码

3.3.3. 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。

Input1: $CFP_1=\{\{f_1,f_2\},\{f_2,f_3\},\{f_2,f_4\},\{f_4,f_5\}\}$
Input2: $CFP_2=\{\{f_1,f_2\},\{f_3,f_4\},\{f_3,f_5\}\}$
Input3: $CFP_3=\{\{f_2,f_3\},\{f_3,f_4\}\}$
$CFP_{Aggregated}=\{\{f_1,f_2\},\{f_2,f_3\},\{f_2,f_4\},\{f_4,f_5\},\{f_3,f_4\},\{f_3,f_5\}\}$

Step 1:
Selected input −− Input1
Selected functions −− $\{f_1, f_2, f_3, f_4, f_5\}$
$CFP_{Uncovered}=\{\{f_3,f_4\},\{f_3,f_5\}\}$

Step 2:
Selected input −− Input2
Selected functions −− $\{f_3, f_4, f_5\}$
$CFP_{Uncovered}=\emptyset$

图 3.3-3 选择输入

3.4. OSCS

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

3.4.1. 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。

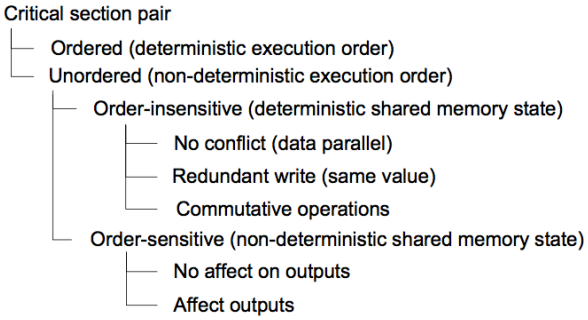

图 3.4-2 临界区类型

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。

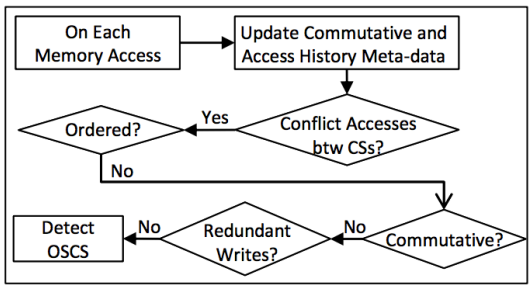

图 3.4-3 OSCS 检测方法

3.5. Goldilocks

Java 是以共享内存作为内存模型的[2]，因此存在多线程环境下的数据竞争问题。Goldilocks[1]在 Java 虚拟机上实现，能够动态检测数据竞争。它引入了一种新的运行时错误 - DataRaceException。每当程序中出现数据竞争，虚拟机就会抛出 DataRaceException，避免问题继续扩散。

3.5.1. Lockset 算法

Goldilocks 扩展了 Eraser[3]提出的 Lockset 算法。Lockset 算法的思想是：如果一个共享变量在多线程执行过程中能够始终被至少一个锁保护的话，那么该共享变量不会发生数据竞争；反之，则有可能发生数据竞争。基于

这样的观察，Eraser 提出了 Lockset 的算法：

```
Let locks_held(t) be the set of locks held by thread t.
For each v, initialize C(v) to the set of all locks.
On each access to v by thread t,
    set C(v) := C(v) ∧ locks_held(t);
    if C(v) = { }, then issue a warning.
```

算法 3.5-1 Lockset

Lockset 算法分为两个阶段。第一阶段为初始化，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)。第二阶段为更新，对于当前线程的每一次读写操作，更新被访问变量的候选集合 C(v)。

Lockset 是一种动态检测算法，需要动态地对程序代码进行改写，来实现统计读写、加锁释放锁、内存声明还有线程创建等操作。基于这些操作产生的 Trace，可以用 Lockset 得到可能出现数据竞争的变量。

3.5.2. 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个集合，分别是 SyncKind，DataKind 和 AllocKind。这些集合在 Goldilocks 中被定义为：

- $SyncKind = \{acq(o), rel(o) \mid o \in Addr\} \cup \{read(o,v), write(o,v) \mid o \in Addr \wedge v \in Volatile\} \cup \{fork(u), join(u) \mid u \in Tid\} \cup \{commit(R,W) \mid R, W \subseteq Addr \times Data\}$
- $DataKind = \{read(o,d) \mid o \in Addr \wedge d \in Data\} \cup \{write(o,d) \mid o \in Addr \wedge d \in Data\}$
- $AllocKind = \{alloc(o) \mid o \in Addr\}$
- $Kind = SyncKind \cup DataKind \cup AllocKind$

图 3.5-1 Goldilocks 形式化定义

Lockset 的更新规则如下：

1. $\sigma(t,n) \in \{read(o,d), write(o,d)\}$:
   if $LS(o,d) \neq \emptyset$ and $t \notin LS(o,d)$
   report data race on (o,d)
   $LS(o,d) := \{t\}$
2. $\sigma(t,n) = read(o,v)$:
   foreach (o',d):
   if $(o,v) \in LS(o',d)$ add t to $LS(o',d)$
3. $\sigma(t,n) = write(o,v)$:
   foreach (o',d):
   if $t \in LS(o',d)$ add (o,v) to $LS(o',d)$
4. $\sigma(t,n) = acq(o)$:
   foreach (o',d):
   if $(o,l) \in LS(o',d)$ add t to $LS(o',d)$
5. $\sigma(t,n) = rel(o)$:
   foreach (o',d):
   if $t \in LS(o',d)$ add (o,l) to $LS(o',d)$
6. $\sigma(t,n) = fork(u)$:
   foreach (o',d):
   if $t \in LS(o',d)$ add u to $LS(o',d)$
7. $\sigma(t,n) = join(u)$:
   foreach (o',d):
   if $u \in LS(o',d)$ add t to $LS(o',d)$
8. $\sigma(t,n) = alloc(x)$:
   foreach $d \in Data$: $LS(x,d) := \emptyset$
9. $\sigma(t,n) = commit(R,W)$:
   foreach (o',d):
   if $LS(o',d) \cap (R \cup W) \neq \emptyset$
   add t to $LS(o',d)$
   if $(o',d) \in R \cup W$:
   if $LS(o',d) \neq \emptyset$ and $\{t, TL\} \cap LS(o',d) = \emptyset$
   report data race on (o',d)
   $LS(o',d) := \{t, TL\}$
   if $t \in LS(o',d)$
   add $R \cup W$ to $LS(o',d)$

图 3.5-2 更新规则

这样的抽象使得 Goldilocks 对事务，原子变量和其他的同步操作有更好的支持。

3.5.3. 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 是在 Kaffe（C 语言实现的 Java 虚拟机）的运行时引擎解释器模式下实现的。这样能够得到源代码在虚拟机内部，内存的布局，同时可以使用虚拟机内部实现的很多算法。

除了上述方法，Flanagan 和 Freund 在 ROADRUNNER 上也应用了 Goldilocks 算法[4]。ROADRUNNER 是一个动态分析工具，Goldilocks 算法协助其在加载时动态插入 Java 字节码指令。

3.5.4. 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（误报）。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。比如内存事务的同步的操作，volatile 变量的读写等。这些同步方法都是传统的算法无法支持的，而 Goldilocks 可以。

3.6. AVIO

AVIO 针对原子性问题进行检测。

3.6.1. 程序员的原子性思维

AI invariants(交错访问不变性)：如果一条指令与之前的一条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。

不同的程序员对原子性的考虑是不一样的。而程序员很难对复杂程序中可能存在的交叉存取考虑清楚。所以最好的方法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 的学习过程并不难：

- 由于 concurrency bug 出现的次数很少，几次尝试就可以得到正确的执行结果。
- 由于程序每一次执行都可能得到不同的执行路径，所以容易得到较高的覆盖率。

所以对于 AVIO 来说，concurrency bug 的不确定性提供了独特的思路：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。

3.6.2. AVIO 的算法

检测算法：通过对程序的分析得到如图 3-6.1 所示的状态机，检测到违反 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。

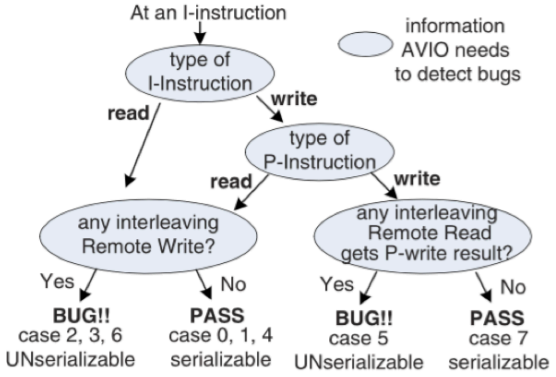

图 3.6-1 AVIO 程序分析状态机

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。迭代上述操作直至 AI 集收敛。

表 3.6-1 对同一变量的交叉存储的八种情况

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.

4. 分析与比较

4.1. 系统优缺点分析

4.1.1. MUVI

MUVI 可以发现由多变量引发的 concurrency bug，其对 lockset 与 happens before 算法的改进只增加了较小的性能负荷。

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

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。

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。

4.1.2. AVIO

AVIO 的算法使用了计算机系统中非常经典的模式匹配的思想，使得算法实现起来比较简单，除此之外，AVIO 提供了软件和硬件层面的两种实现方式，各自有着不同的使用场景，在软件层面上的实现更加简单，但是有着比较高的 overhead，而在硬件层面上的实现则需要通过添加或修改硬件结构，比较复杂，但是会有着更好的效率。

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。

4.1.3. 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详细更具有可读性的 bug report。

ConSeq 是基于 Concurrency bug 传播路径较短的假设而实现的，因而无法找到有较长传播路径的 Bug。

4.1.4. 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误报率比较高。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。

4.1.5. 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。

4.1.6. Goldilocks

Goldilocks 拓展了 Lockset 算法，其最大的优势在于，它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 Bug，同时也可以支持 Transaction，volatile 变量等新的同步方式。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 实现起来更加方便。

Goldilocks 也存在一些缺点，因为它使用动态分析，所以非常依赖程序的执行路径，也很难找到所有的 Bug。另外，目前的实现都是基于 Java 的，不适用于其他语言，通用性不是特别好。

4.2. 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。在计算机系统中，在一件事情上常常会有意见相左的两种设计理念。在 Kernel 的设计历史中，就存在 Monolithic kernel 和 Micro kernel 之争。而在两者的基础上，有研究将两者结合起来，提出了 Exo kernel 的概念，这样的设计理念可以总结为 Hybrid 原则：即在两种不同的设计方式可供选择时，将两者进行结合也可能是另外一种值得考虑的设计方式。在大多数的 concurrency bug 的检测技术中，大多数技术都是使用程序的动态分析或者是静态分析来实现的。而 MUVI 在变量的关联性分析时，使用了静态分析的技术，而在数据竞争 Bug 的检测时，对 Lockset 算法进行了改进，运用了基于动态分析的检测技术。将两种程序分析技术结合起来，使得 MUVI 在拥有合理的 overhead 的同时准确性也得以保证。由此可以看出，Hybrid 原则在系统的设计过程中很常见。不同的两种设计哲学，可以相互结合的。
- KISS：KISS(Keep it simple, stupid)原则强调设计的简单性。CFP 系统在计算完每个输入所能覆盖的并行函数对后，需要选择输入的子集以覆盖所有的并行函数对。在选择输入的过程中，它并没有试图找到最优解，因为这将是一个 NP 问题。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问题。这体现了 KISS 原则。
- Avoid excessive generality：本文提到的每个系统都是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

system	atomicity violation	multi-variable synchronization	Non-race
MUVI	√	√	
ConSeq	√		
Goldilocks	√		
CFP	√		
AVIO	√		√
OSCS			√

- Principle of least astonishment：最小惊讶原则要求系统在设计时，尽量符合用户已有的经验和习惯。ConSeq 系统沿着 bug 传播链反向寻找产生 bug 的代码，因此产生的 bug report 相较于其他系统，可以给出更多信息，同时也方便用户分析 bug 真实产生的原因和过程，用户友好度更高。
- Avoid rarely used components: Lockset 算法和 happens-before 算法是已有的两种具有代表性的 data race bug 检测算法，并且被广泛使用。MUVI 可以对这两种算法分别进行扩展，来检测多变量之间的 data race bug。
- Be explicit： 这条原则要求系统在设计时，尽量明确所有的假设。MUVI 在判断变量间的关联关系时，假设关联关系和内存访问之间的距离以及同时出现的概率相关。ConSeq 假设 bug 会在 4 个内存读写操作内引发错误，因此只找传播路径较短的 concurrency bug。
- Open design: 开放设计原则要求系统开放源代码，让更多的人可以对其进行评论和扩展；充分利用已有的开源工具。MUVI 一文中提到，多变量 concurrency bug 无法由针对单一变量的数据竞争算法检测出来，因此 MUVI 采用的做法是对已有的两种数据竞争检测算法，即 lockset 和 happens before 算法进行扩展。当检测算法发现某些变量存在冲突的访问时，会同时检查锁的情况以及变量是否存在关联关系。这种做法复用并扩充了 lockset 算法，从而实现多变量的数据竞争检测。MUVI 对 happens before 算法的扩充方式与 lockset 类似，在比较了访问相同内存的指令的时间戳之后，还会比较相关性变量访问内存的时间戳。MUVI 一文中还提出，MUVI 本身可以在其他的 concurrency bug 检测工具上进行扩展，如 RaceTrack、RacerX、Chord、AVIO 等。以上都说明了 MUVI 是本着开放原则设计实现的，目的是为了让更多的工具能够直接受益于它，这体现了 open design 原则。
- Unyielding foundations rule：在任何设计中，修改具体的实现，是代价比较小的，而修改整体的架构，是代价比较大的。因此比较常见的做法是使得整体设计符合良好的抽象，尽量把修改限制在一个模块中，而不会在每次进行修改时涉及到架构的变动，这种原则被称为 Unyielding foundations 原则。在 Goldilocks 的设计中，其中最重要的部分就是把所有的操作用形式化的方式描述出来，这样的做法的好的好处是，将同步方法抽象出来，每当需要支持新的同步方法的数据竞争 Bug 时，只需要将新的同步方法以形式化的方法描述出来，然后给予实现即可，不需要涉及整体的形式化描述的结构的改动。这样也符合软件工程思想，保证代码中的变动尽量体现在新加代码，而不是修改代码上。

4.3. 系统间的比较

4.3.1. 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。

4.3.2. 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

偏重由数据竞争引发的 bug。CFP 跟其他系统不同在于，它重点放在提高检测效率上，仅选择部分输入，利用已有的 bug detector 进行检测。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，能够给出更具有解释意义的 bug report。Goldilocks 仅适用于 Java，可以检测不同同步方法下的 bug，例如 transaction 和 barrier。

5. 总结

多线程程序执行时存在内存共享，因此可能产生对内存的错误读写，继而引发程序崩溃等问题。找到 concurrency bug 的根本就是找到可能的交叉读写(interleaving)。本文提到的系统有些仅使用动态分析技术[1,9]，也有系统将动态分析和静态分析相结合[5,6,7,11]。大部分采用从前向后的分析方式[1,5,7,9,11]，也有系统沿着反向传播链寻找 bug[6]。它们都有各自的切入点，检测程序中存在的 concurrency bug。

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.