并发缺陷暴露、检测与规避研究综述

苏小红 禹 振 王甜甜 马培军
(哈尔滨工业大学计算机科学与技术学院 哈尔滨 150001)

摘 要 当今普遍流行的多核架构使得硬并发成为现实.为了从硬件的并发能力获益,并发程序设计正越来越流行.然而由于内在的并发性和不确定性,并发程序易于遭遇并发缺陷,并且它们难以检测、调试和修复.文中指出软件开发正从顺序模式转向并发模式的趋势,揭示并发程序和并发缺陷各自的三大特点,剖析并发缺陷面临的三大挑战,然后将并发缺陷分为死锁、数据竞争、原子性违背和顺序违背 4 类,并讨论 4 类并发缺陷的相互关系,接着就如何尽快暴露、及时检测和高效规避各类并发缺陷对已有研究作出分析、比较和归纳,最后从智能快速的缺陷暴露、通用准确的缺陷检测、确定性重放支持、软硬件协同设计和新的并发编程模型等 5 个方面展望了未来的研究重点.

关键词 并发缺陷;死锁;数据竞争;原子性违背;顺序违背; 程序分析;软件测试
中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2015.02215

A Survey on Exposing, Detecting and Avoiding Concurrency Bugs

SU Xiao-Hong YU Zhen WANG Tian-Tian MA Pei-Jun
(*School of Computer Science and Technology, Harbin Institute of Technology, Harbin* 150001)

Abstract The prevalent multi-core architecture today makes real concurrency come true. In order to benefit from the concurrency power of hardware, concurrent programming is becoming more and more popular. However, out of inherent concurrency and non-determinism, concurrent programs are more likely to suffer from concurrency bugs, which are hard to detect, debug and repair. In this paper, we point out the trend that software development is turning toward concurrent model from sequential model. We reveal three characteristics respectively for concurrent programs and concurrency bugs and explore three challenges confronted by concurrency bug analysis. We then divide concurrency bugs into four categories, i.e. deadlock, data race, atomicity violation and order violation and investigate relations among them. For each category of concurrency bugs, we make analyses, comparisons and summarizations on previous researches about how to quickly expose, in time detect and efficiently avoid them. At last, we look into the research priorities in future from the following five aspects: smart and quick concurrency bugs exposure, common and accurate concurrency bug detection, deterministic replay support, collaborative design involving software and hardware and new concurrent programming model.

Keywords concurrency bug; deadlock; data race; atomicity violation; order violation; program analysis; software testing

1 引 言

过去的几十年间,软件一直从处理器(CPU)性能的不断提升中获益.计算机工业界中存在一个有趣的现象:“安迪送,比尔取”.无论 CPU 性能提升多少,软件都有办法迅速吞噬.CPU 性能十倍于前,软件就能在同样时间段内处理十倍于前的工作量(或者运行速

收稿日期:2014-03-03;最终修改稿收到日期:2014-11-17.本课题得到国家自然科学基金(61173021,61202092)资助.**苏小红**,女,1966 年生,博士,教授,中国计算机学会(CCF)高级会员,主要研究领域为软件缺陷检测、程序分析、信息融合、目标检测与跟踪等.E-mail:sxh@hit.edu.cn.**禹 振**,男,1987 年生,博士研究生,中国计算机学会(CCF)会员,主要研究方向为隐式规则挖掘、并发缺陷检测和规避等.**王甜甜**,女,1980 年生,博士,副教授,主要研究方向为程序分析、软件缺陷检测、软件测试等.**马培军**,男,1963 年生,博士,教授,主要研究领域为信息融合、软件工程、目标检测与跟踪等.

度十倍于前).大多数情况下,软件受益于 CPU 和内存、硬盘等外围设备的持续不断升级,其不作任何改变就能免费获得性能提升.但是这种免费午餐已经结束.

由于受制于一些物理学问题,如功耗、发热及电子泄露,CPU 时钟频率的提升越来越难,几乎已达极限.图 1 反映了 Intel 处理器的时钟频率与晶体管规模关系的演化历史.大约在 2003 年左右,一直快速攀升的时钟频率突然陷入了停滞.即使大幅增加晶体管数量,也无济于事:时钟频率仍旧不能提升,甚至会有所下降.最终 Intel 的单核 CPU 时钟频率止步于 3.8GHz.因此软件在保持单执行流的体系结构下,将再也不能从 CPU 性能提升中获益.Intel 于 2006 年 6 月发布革命性的"酷睿"双核/多核架构处理器[1],从此个人电脑领域进入硬并发时代.在当今多核时代,为提高运行速度,软件必须转向并发模式.然而相对于传统的结构化程序设计而言,并发程序设计更加困难且容易出错.

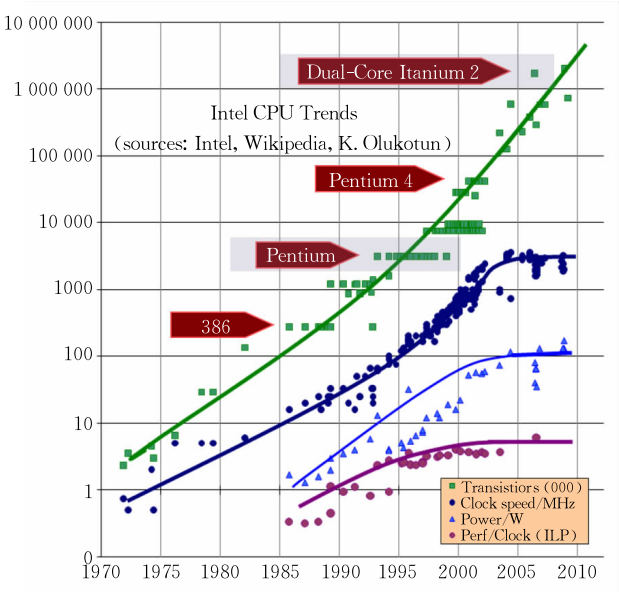

图 1 Intel 处理器趋势图[2]

1.1 并发程序的特点

由于内在的并发性,并发程序的执行具有不确定性且执行交错空间庞大.与顺序程序相比,并发程序的特点主要体现在以下几个方面:

(1)执行交错空间庞大.并发程序的复杂度随并发执行流数目及其长度增长而指数级增长[3].例如一个具有 5 个线程、每个线程执行 5 个动作的并发程序,其所有可能执行路径数为 25!/(5!)^5(大于 100 亿).

(2)开发需要并发的思维模式.并发程序要求程序员按照并发/并行的思维模式思考和处理问题,而人类习惯和擅长于串行的思维模式.对于若干有序操作序列的简单排列组合,即使最聪明和最有经验的程序员也可能遗漏掉某些执行交错.

(3)运行不确定性.并发程序内在的并发性造成其运行具有不确定性.并发程序中的数个独立执行流受到调度器的随机调度,即使给定同一输入,其前后两次运行的执行交错和执行结果可能完全不同.

1.2 并发缺陷的特点

由于并发思维模式与人类思维习惯相悖,并发缺陷常常被程序员无意识地引入到并发程序中.相对于传统的程序缺陷,并发缺陷具有以下特点:

(1)难以检测.并发缺陷仅在罕见的、特定的执行交错下才暴露出来,而并发程序执行交错空间庞大,传统的压力测试即使运行数天甚至数月也不能触发某些并发缺陷[3-6],更无法检测到它们.

(2)难以调试.并发缺陷从产生到暴露需要一定的传播过程,具有延时性[7].另外并发程序的运行不确定性使得传统的通过迭代运行以定位缺陷原因的调试方式不再适用,加剧了并发缺陷的调试难度.

(3)难以修复.并发缺陷即使被检测到,也难以修复[8-9]:补丁代码往往不是修复了缺陷,而只是降低了其发生的概率;并且补丁代码常常会导致新的并发缺陷.

在传统的测试方法下,许多并发缺陷没有被检测出来,而继续潜伏在软件中,直至在生产性运行中暴露出来,造成重大事故和财产损失.例如由于数据竞争缺陷,纳斯达克 OMX 系统发生故障,延迟为脸谱公司 IPO 提供服务达 30 分钟,造成 1300 万美元损失①.

1.3 关键技术挑战

并发程序执行交错空间庞大且运行具有不确定性,而并发缺陷又仅在罕见的执行交错下才发生,传统以测试用例生成为中心的测试技术不能有效暴露和检测并发缺陷,更不能规避并发缺陷.并发缺陷的动态测试和静态分析主要面临三大挑战:

(1)尽快暴露.并发程序的执行交错空间庞大,而其中只有极少数的执行交错能够触发并发缺陷.传统测试技术即使能够生成触发并发缺陷的测试用例,也不能保证并发缺陷在测试用例的每次执行中都暴露.因此在并发程序运行能够触发并发缺陷的测试用例时,并发缺陷测试技术应设法控制或者影响并发程序的执行,以测试尽可能多的执行交错或尽快测试那些包含并发缺陷的执行交错.

(2)准确、及时检测.对并发缺陷检测报告进行

① Nasdaq's Facebook glitch came from race conditions. http://www.computerworlduk.com/news/it-business/3359088/nasdaqs-facebook-glitch-came-from-race-conditions/, 2012.5.22

确认通常费时费力，并需要丰富的并发调试经验，因此检测方法应当只报告真正的并发缺陷.在保证误报率为零的情况下，检测方法可以有一定的漏报率，因为并发缺陷的发生概率极低.实际中，对并发缺陷的检测以动态检测为主.动态检测时，由于并发缺陷从产生到暴露可能经历成百上千条指令和多次上下文切换，检测方法应能及时监测到并发缺陷的产生并检测出其构成要素.

(3)高效规避.并发缺陷难以修复而又仅在罕见的执行交错下才暴露出来，但只要合理调度程序执行，使其避开那些包含并发缺陷的执行交错，则即使程序中存在并发缺陷，也不会对程序的正确性造成影响.对并发缺陷的规避方法有两种：静态源码分析和动态执行控制.静态规避方法容易添加过多的同步设施，可能降低并发执行的并发度；动态规避方法全程监视并控制并发程序的执行，可能降低并发执行的执行效率.

近年来，众多学者在构建并发程序软件支撑环境(如并发库、支持并发的程序设计语言等)、并发缺陷暴露、检测与调试工具、并发缺陷预防、规避与重演等方面做了大量工作.在参阅以往研究工作的基础上，本文第 2 节将并发缺陷分成 4 类，并逐类给出定义和示例；第 3 节总结和评价各类并发缺陷的暴露技术；第 4、5、6 节分别就如何及时检测和在线规避死锁、数据竞争和原子性违背，对已有研究作出分析、比较和归纳.由于顺序违背这种缺陷模式被提出和认可的时间较短，只有少量研究，故不单列一节讨论它，只在讨论其他种类并发缺陷的检测和规避技术时捎带提及；第 7 节从 5 个方面展望未来的研究重点；第 8 节总结全文.

2　并发缺陷分类及其相互之间的关系

本文考虑共享内存并发系统中的常见并发缺陷，先将其分为死锁、数据竞争、原子性违背和顺序违背四大类(本文不讨论活锁)，然后讨论它们之间的相互关系.当然并发缺陷包括但不限于这 4 种模式，新的缺陷模式总是随着人们对并发缺陷的深入认识而被提取出来.例如，Lu 等人[8]在 2008 年提出的顺序违背这种新的并发缺陷模式，并且被广泛认可.

2.1　死　锁

定义 1.　死锁.某线程集合中的每一个线程都在等待另一个线程占有的互斥性资源，由此造成的循环等待即为死锁.

图 2 中的代码片段展示了开源数据库 SQLite-3.3.3 中的一个死锁缺陷①.假设线程 $T1$ 先执行并在 $L1$ 处获得锁 $mutex1$，然后调度器暂停 $T1$ 并调度线程 $T2$ 执行.$T2$ 被调度执行前已获得锁 $mutex2$，它在 $L3$ 处请求锁 $mutex1$ 时将被阻塞，因为 $mutex1$ 正被 $T1$ 占据.当 $T1$ 执行时，它将在 $L2$ 处请求锁 $mutex2$ 时被阻塞，因为 $mutex2$ 正被 $T2$ 占据.这就构成了死锁.

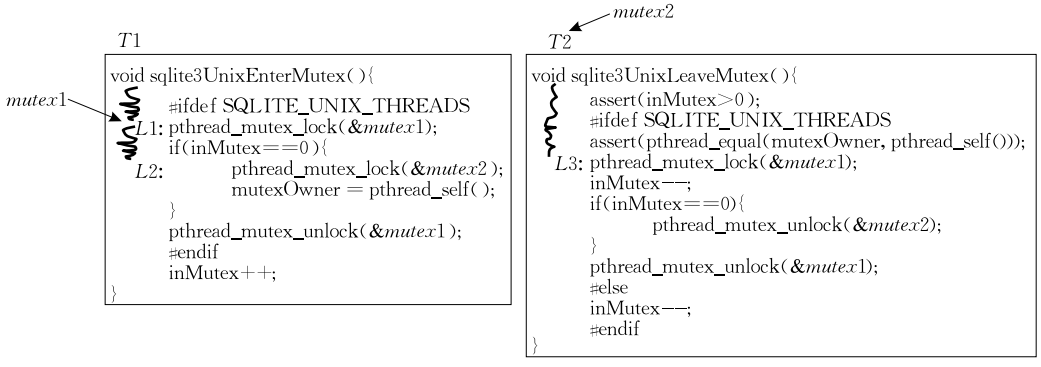

图 2　SQLite-3.3.3 中的死锁缺陷 Bug#1672

在 C/C++和 Java 中，死锁一般由同步设施如互斥锁、读写锁、条件变量和信号量造成[10].互斥锁和读写锁常导致有环死锁，而条件变量和信号量则能够导致无环死锁[11].消除有环死锁的措施是定义全局的锁占用顺序并使所有线程都按照该顺序来请求锁；消除无环死锁的必要措施是保证对条件变量和信号量发出的信号没有丢失.

2.2　数据竞争

定义 2.　数据竞争.对某一共享内存单元，存在来自不同线程的两个并发访问，且至少一个为写访问.

定义 2 中的并发意指这两个访问之间没有 happens-before 关系[5,12]，存在“同时”发生的可能.消除数据竞争的措施有两种：为共享内存单元加上一致性锁保护，或者利用条件变量等同步设施使两个访问之间形成一个固定的执行次序.两种措施的区别是，构成数据竞争的两个访问在前者中可以两

① SQLite bug #1672. http://www.sqlite.org/src/info/a6c30be214, 2006,2,10

种次序执行，而在后者中只能以一种次序执行. 图 3 中的线程 $T1$ 和 $T2$ 可能同时分别执行 $L1$ 和 $L2$，受调度器调度的影响，共享变量 *shared*[0]值要么增加 2 要么增加 1. 比如当线程 $T1$ 先读取 *shared*[0]的值，执行 $L1$，然后 $T2$ 读取 *shared*[0]的值，执行 $L2$ 时，*shared*[0]值正确地自增 2；而如果线程 $T1$ 和 $T2$ 先读取 *shared*[0]的值，然后分别执行 $L1$ 和 $L2$，则 *shared*[0]值错误地自增 1，$T1$ 或者 $T2$ 对 *shared*[0]的一个更新丢失. 然而由于没有同步机制保护，这两种情况在一次运行中都有可能发生，相互之间构成“竞争”关系.

```
T1=new Thread(){                T2=new Thread(){
  public void run(){              public void run(){
    while(T1 !=null){               while(T2 !=null){
      …                               …
    L1: shared[0]=shared[0]+1;      L2: shared[0]=shared[0]+1;
      …                               …
    }                               }
  …          T1                   …          T2
```

图 3 一个 Java 语言编写的数据竞争示例

数据竞争常被分为 3 类[13-14]：貌然数据竞争(apparent data races)、可行数据竞争(feasible data races)和实际数据竞争(actual data races). 貌然数据竞争指在不考虑同步设施的语义的情况下，被检测出来的数据竞争. 可行数据竞争指在考虑同步设施的语义的情况下，有可能被检测出来的数据竞争. 实际数据竞争指在某次实际运行中被检测出来的数据竞争. 图 4 中，存在两个貌然数据竞争和一个可行数据竞争. 两个貌然数据竞争，一个在 *result* 上，一个在 *done* 上. 然而在考虑程序语义后，发现 *done* 被用作同步设施：$T2$ 只有当 *done* 为 TRUE 时才访问 *result*. 因此 *result* 上的数据竞争不可能发生，只有 *done* 上的数据竞争是可行的，即可行数据竞争其才可发生.

```
T1                    T2
result=x;             while(done== FALSE);
done=TRUE;            y=result;
```

图 4 貌然数据竞争与可行数据竞争[13]

2.3 原子性违背

定义 3. 原子性违背. 对某一为保证正确性必须原子性执行的指令序列，存在一个执行交错，其执行效果不与任何该指令序列原子性执行时的执行交错的执行效果相同.

图 5 展示了开源 Web 服务器 Apache Httpd 中的一个原子性违背缺陷. 线程 $T1$ 中，$L1$ 首先检查缓冲区 *buf* 是否有足够容量，如果是的话，$L2$ 从 *log* 向 *buf* 中拷贝 *len* 个字节数据. 程序员假设 $L1$ 和 $L2$ 应原子性执行，但没有使用同步设施加以实施. $T1$ 在执行 $L1$ 和 $L2$ 的中间，可能被来自 $T2$ 的 $L3$ 打断，造成缓冲区溢出. 消除原子性违背的措施一般是为相互之间应原子性执行的区域加上共同的锁保护.

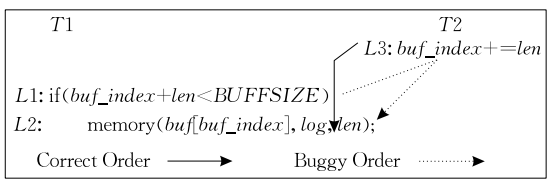

图 5 Apache Httpd 中的原子性违背缺陷[6,15]

原子性违背分为单变量原子性违背和多变量原子性违背两类. 对于单个共享变量，如图 6 中的 x，3 个读写该共享变量的访问(两个来自同一线程，第

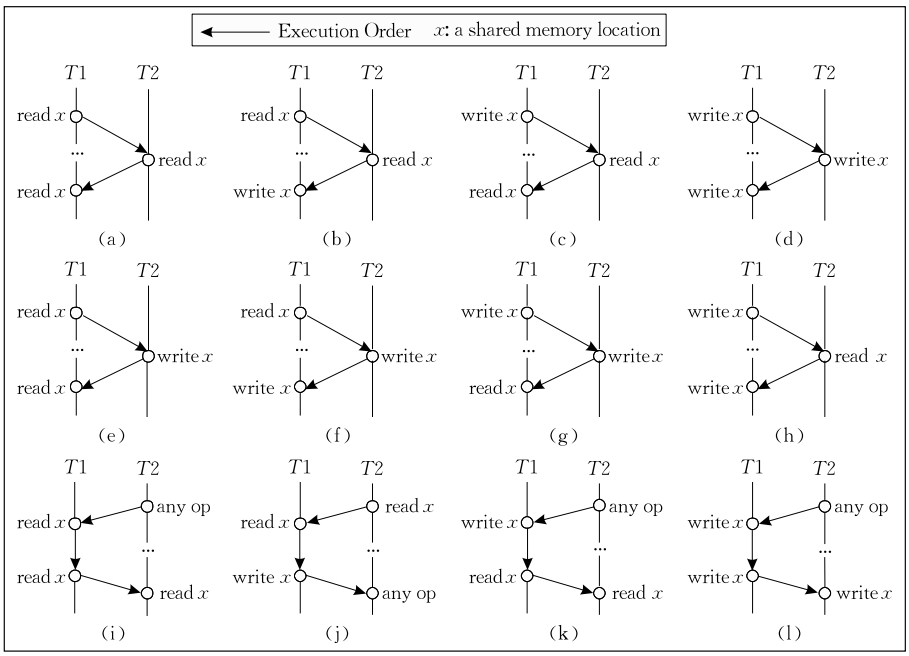

图 6 可序列化与非可序列化的执行交错[6,16-18]

3 个来自另一线程)共有 8 种执行交错((a)～(h)),其中 4 种((a)～(d))存在等价的原子执行交错,分别为(i)～(l),另外 4 种((e)～(h))是不可序列化的.例如,如果两个本地读操作被一个远程写操作交错(图 6(e)),则两个本地读操作将读取到不同的值,这是一个不可序列化的执行交错.图 5 给出了该类原子性违背的一个例子.

2.4　顺序违背

定义 4.　顺序违背.某一指令(组)没有按照预期,总是在另一指令(组)之前或者之后执行.

图 7 展示了开源 Web 浏览器 Mozilla 中的一个顺序违背缺陷.线程 $T1$ 调用 PR_CreateThread() 创建一个以 mMain 为入口函数的子线程,并将该线程的句柄存储在 *mThread* 中.然而由于 PR_CreateThread() 不是原子性地执行,线程 $T2$ 可能在 PR_CreateThread() 还没返回时就对 *mThread* 解引用,从而造成空指针解引用或者读取错误的值.此例中,程序员的意图是 $T2$ 对 *mThread* 的解引用应发生在 $T1$ 对 PR_CreateThread() 的调用返回之后,但此意图并没有得到实施和保证.消除顺序违背的措施一般是使用条件变量在指令(组)之间形成固定的执行次序.

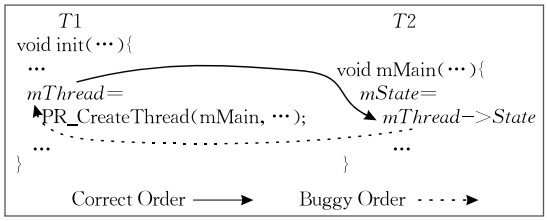

图 7　Mozilla 中的顺序违背缺陷[8]

2.5　4 类并发缺陷的相互关系

通过深入分析,我们发现 4 类并发缺陷的关系如图 8 所示.从图 8 可看出:

(1)并发缺陷包括但并不限于死锁、数据竞争、原子性违背和顺序违背这 4 种缺陷模式;

(2)死锁与其他 3 类并发缺陷不相关;

(3)数据竞争在所有并发缺陷中所占比例较大,且大部分情况下是原子性违背和顺序违背的本质原因;

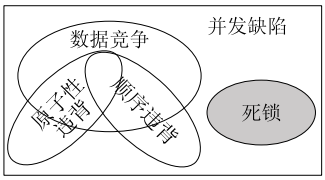

图 8　4 类并发缺陷的关系

(4)数据竞争常常导致原子性违背和顺序违背,但原子性违背和顺序违背并不完全由数据竞争造成.在图 5 所示的原子性违背中,$L1$ 和 $L3$ 以及 $L2$ 和 $L3$ 之间构成数据竞争;在图 7 所示的顺序违背中,两线程对 *mThread* 的写操作构成数据竞争.对于上述两例中的共享变量 buf_index 和 *mThread*,如果线程在每次访问前都持有一致性锁,并在访问完后立即释放该锁,则数据竞争被消除,而原子性违背和顺序违背仍然存在;

(5)数据竞争、原子性违背和顺序违背互不相同.顺序违背与数据竞争的区别在于:顺序违背的相关指令(组)在任何执行交错中只允许存在一种固定的顺序关系,而数据竞争的相关指令在执行时只需要具有有序关系即可.原子性违背与顺序违背的区别在于(针对单共享变量):原子性违背涉及 3 个操作,而顺序违背只涉及两个操作;另外原子性违背一定由至少两个顺序违背构成,但顺序违背可独立存在,不一定是原子性违背的构成部分.原子性违背与数据竞争的区别类似于其与顺序违背的区别.

3　并发缺陷暴露技术

并发缺陷只有首先出现,才能被观察、检测和验证.某些研究[10,19-20]能通过分析程序的一次运行而预测"潜在的"并发缺陷,并设法使这些缺陷在第 2 次运行中暴露出来.通过观察缺陷能否暴露,这些研究检验预测的缺陷是否是真正的缺陷.并发缺陷暴露技术考虑并发缺陷的共同特点,能暴露各类并发缺陷.

传统内控测试中并发缺陷的暴露概率很低,这主要是由于:并发程序的执行交错空间庞大;并发缺陷只隐藏于某些罕见的、特殊的执行交错;测试技术缺乏对底层调度器的控制,并发程序常常在多次不同执行中按照同一执行交错执行.针对这些问题,学者们提出了各种各样的尽快暴露并发缺陷的方案,本文将其总结为 3 类.

3.1　随机延时扰动

该技术[4,6,16-17]在并发程序进行共享内存访问和同步控制时,插入随机延时.对于死锁,随机延时扰动在线程成功获取锁后插入延时,从而本线程挂起而其他线程得到执行机会,增大了其他线程获取锁的概率.对于原子性违背,该技术在两个本地操作之间插入延时,从而增大远程操作交错在两个本地操作之间的概率.对于数据竞争和顺序违背,在预知

正确执行次序的前提下，该技术将试图执行错误的执行次序，从而使两者的暴露概率接近于1(限时等待)，否则，两者暴露概率接近于0.5.

3.2 线程调度/切换

该技术[3-4,21-22]对底层调度器施加直接或者间接影响，当线程进行共享内存访问或者同步控制前后，挂起该线程，调度另一线程执行. 比如ConTest[4]在线程访问共享变量后，调用yield()强制挂起本线程同时调度优先级同级的其他线程，或者调用priority()改变本线程的优先级，从而间接控制调度器调度执行其他线程. CHESS[21]通过覆盖200多个操纵线程和同步设施的Win32 API，完全、彻底地控制调度器的线程调度和异步事件的发送与接收；在此基础上，CHESS又使用执行枚举技术，使得并发程序的每次运行都按照不同的执行交错执行，从而能极大地提高有限测试用例的执行交错覆盖率.

3.3 Fuzzing技术

该技术[5-7,23-25]首先使用并发缺陷检测技术检测出可能的并发缺陷，然后根据被检测出的并发缺陷的信息，控制线程调度和执行，设法使程序按照能够暴露并发缺陷的执行交错执行. 例如RaceFuzzer[5]使用hybrid技术[12]检测出可能构成数据竞争的一个访问对{$s1$,$s2$}，当某个线程如$T1$将要访问$s1$时，RaceFuzzer暂停$T1$执行，让其等待直到另外一个线程如$T2$将要执行$s2$，这时它从$T1$和$T2$中随机选择一个，令其执行下一条语句，如图9所示. RaceFuzzer的数据竞争暴露概率为0.5. CTrigger[6]使用pcr① 技术检测出可能构成原子性违背的3个访问{p,c,r}，然后使用随机延时扰动来增大原子性违背的暴露概率. ConLock[25]使用magliclock技术[25-26]检测可能导致死锁的约束条件，然后控制线程调度使得约束条件满足以观察死锁能否被暴露.

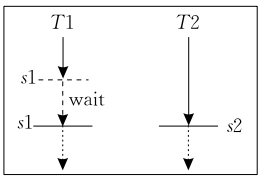

图9 RaceFuzzer[5]数据竞争暴露机制

随机延时扰动技术通过插入延时的方式间接影响线程的调度；而线程调度/切换技术则直接控制线程的调度. 这两种技术在对线程调度施加影响时都比较盲目，没有具体的目标. 而Fuzzing技术虽能有意识地调度线程以观察某些并发缺陷能否暴露，但它依赖于其他工具的检测结果.

4 死锁检测与规避

在操作系统和数据库领域，无论在进程层面还是事务层面，死锁的预防、检测、规避和解除已得到充分完善的研究，这里不予涉及. 本节讨论进程内部线程层面上的死锁检测与规避.

4.1 死锁检测

表1将近年来死锁检测方面的研究分为定理证明与模型检验、数据流分析和动态分析3类，并从人工干预度、扩展性、误检率、漏检率和执行开销5个维度对这3类研究进行对比.

表1 死锁检测研究分类与对比

类别		概述	人工干预度	扩展性	误检率	漏检率	执行开销
定理证明与模型检验[27-29]		证明一个程序是否是免于死锁的	高	差	低	低	低
数据流分析[30-32]		组合使用多种静态分析技术，计算程序的静态锁顺序图，并把其上的环报告为可能的死锁	低	一般	高	低	低
动态分析	在线[34-40]	监视目标程序运行，建立同步状态的抽象表示，在其上检测死锁	无	良好	低	高	高
	离线[10,23,26,33,41-43]	获取并分析程序执行轨迹，建立锁占用顺序图，在其上检测死锁					

定理证明与模型检验[27-29]试图验证一个程序是免于死锁的. 如果程序中存在死锁，则该方法将检测并报告一条从初始状态到死锁状态的路径. 定理证明与模型检验不能直接对程序源码进行分析，需要人工构建用于证明与检验的模型. 但如何保证抽象模型与程序源码在语义上的等价性是一个悬而未解的问题. 而且现实世界中的并发程序通常十分复杂，很难构建模型以供证明和检验.

为降低定理证明与模型检验的人工干预度并提高其可扩展性，数据流分析技术[30-32]直接分析程序源码，组合使用调用图分析、指向分析和逃逸分析等静态分析技术，计算静态锁占用约束或者锁占用顺序图[19,33]，使用约束求解和环检测算法在其上检测

① pcr代表对某一共享变量的3个访问，其中p,c来自同一线程，r来自另一线程. c是当前访问，p是前继访问，r是远程访问.

环，将环作为可能的死锁报告出来.数据流分析缺乏精确的运行时信息，一般对变量值作保守估计，因此其误检率高而漏检率低.

动态分析[10,23,26,33-43]是进行死锁检测的主流方法，一般分为在线[34-40]和离线[10,23,26,33,41-43]两类.在线方法监视程序的运行，实时获取感兴趣的信息，建立和更新目标程序同步状态的抽象表示，并在其上检测死锁.离线方法先对程序源码静态插桩，然后执行程序并获取执行轨迹，最后分析轨迹，建立锁占用顺序图[35]，将其上检测到的环作为死锁报告出来.在线方法只能检测到本次执行中实际出现的死锁，而离线方法则还能“预测”在其他执行中可能出现的死锁.例如 MagicFuzzer[26]、DeadlockFuzzer[23]和 MagicLock[42]根据预测到的死锁信息，在程序的下一次执行中，控制线程调度，试图使死锁暴露出来.相对于在线方法，离线方法需要存储执行轨迹，对于长时间运行的并发程序不可行.动态分析不需要人工干预，扩展性好，误报率低.但其漏报率高，且执行开销较大，如 Valgrind[39]使目标程序的运行时间增加到原来的 22 倍以上.

4.2　死锁规避

死锁经常在多个线程按照相反的顺序请求或者等待某些资源的情况下发生.只要合理安排线程请求资源的顺序，死锁是可以被规避的.表 2 将近年来的死锁规避研究分为 4 类，并从人工干预度、扩展性和执行开销 3 个方面对它们进行对比.

表 2　死锁规避研究分类与对比

类别	概述	人工干预度	扩展性	执行开销
类型系统[44-45]	检查程序是否按照某个全局锁占用顺序申请锁	高	差	低
效应系统[46-48]	静态分析计算加锁解锁操作的连续效应，将获得的锁效应信息静态插桩到加锁语句之后，运行时根据效应信息决定加锁操作是否执行	无	良好	低
Petri 网控制论[49-54]	对整个程序进行分析，建立 Petri 网模型，利用离散控制论寻找可能的死锁，插入额外的控制逻辑来规避死锁发生	无	差	低
动态分析[36-37,40,55-57]	根据死锁的特征规避死锁，或者遇到死锁后回退到无死锁状态重新执行	无	良好	高

类型系统[44-45]检查程序中各个线程是否按照某个全局锁占用顺序申请锁.如果所有线程都按照相同的锁占用顺序请求锁的话，就不会出现两个或多个线程按照相反的顺序占用锁的情况，从而避免死锁的发生.如果某个线程请求锁的顺序与全局顺序不符，则该并发程序不会通过类型检查.虽然类型系统支持类型自动推断，但是仍然需要为某些锁添加锁级(lock level)注释.当并发程序规模较大时，为其添加注释将是一项繁重的工作.

类型系统试图在编译阶段保证并发程序是没有死锁的，而效应系统、Petri 网方法和动态分析方法则在并发程序可能存在死锁的情况下，控制线程调度来避开包含死锁的执行交错.

效应系统[46-48]静态地分析和计算每一个加锁解锁操作对其他加锁解锁操作的连续效应(continuation effect)，然后把获得的锁效应信息插桩到相应的语句之前.程序在动态执行的时候，会根据效应信息计算出每个加锁操作对应的未来锁集(future lockset)，只有当锁集中的所有锁都没有被占据的时候，当前加锁操作才会被执行.这样就动态规避了死锁的发生.效应系统动静分析结合，在死锁规避方面效率较高，但它依赖于程序源码.如果目标程序中的死锁由源码不可见的第三方库造成，则不能规避这种死锁.

基于 Petri 网控制论的方法[49-54]，如 Gadara[49]，将整个程序转化为并发语义等价的 Petri 网模型，利用离散控制理论在模型上寻找可能导致死锁的转换(transition)，并插入额外的控制逻辑避免死锁在运行时发生.该方法因在程序运行前插入控制逻辑，对目标程序的性能影响较小，能高效地规避死锁，但扩展性差，且对指针别名非常敏感.在最坏情况下，该方法可能插入过多的控制逻辑以致于降低目标程序的并发度.

基于动态分析的方法[36-37,40,55-57]，如 Dimmunix[36-37]、Rx[56]和 Sammati[40]，不需要预先获知目标程序的任何先验信息(如程序源码)，就能在运行时动态规避死锁.Dimmunix 先检测死锁，并记录导致死锁的线程的栈帧作为死锁的特征，然后在并发程序下次运行时，有意识地控制程序执行，阻止相关线程再次进入相应的栈帧，从而成功规避已遇到的死锁.Rx 在目标程序执行时每隔一段时间设置一个检查点，一旦检测到死锁发生，就将目标程序恢复到最近一次检查点所对应的状态，并尝试新的执行交错来规避死锁.Sammati 认为设置全局检查点开销太大，因此只在每个线程尝试获取锁时设置线程局部的恢复点.Dimmunix 不能在检测死锁后即时规避死锁，因

此是离线的规避方案. 而 Rx 和 Sammati 则是在线规避死锁,无需重新执行程序. Rx 和 Sammati 周期性地设置检查点,时空开销很大,并且不是所有的操作都能回退,如用户输入、输出、文件系统操作等. 动态分析无需人工干扰,能处理大规模、长时间运行的并发程序,但会大幅降低目标程序的运行速度.

5 数据竞争检测与规避

数据竞争在所有并发缺陷中占有较大比例[8],并且常常是其他非死锁并发缺陷的根本原因. 本节分析和归纳数据竞争的检测与规避研究.

5.1 数据竞争检测

学者们已在数据竞争检测领域做出了大量研究,图 10 从基本思想和方法手段两个方面对已有工作进行分类和对比. 现有研究的检测思路可分为4 类:锁集法(lockset)、先发生于法(happens-before)、混合法(hybrid)和面向后果法(conseqence-oriented)其中前 3 种为竞争检测的传统方法,后一种为新近提出方法. 现有研究的检测手段可分为 4 类:类型系统、模型检验、数据流分析和动态分析,并以动态分析为主. 对检测思想,我们从复杂度、误检率和漏检率 3 个方面进行评价;对检测手段,我们从人工干预度、扩展性、误检率、漏检率和执行开销 5 个方面进行评价.

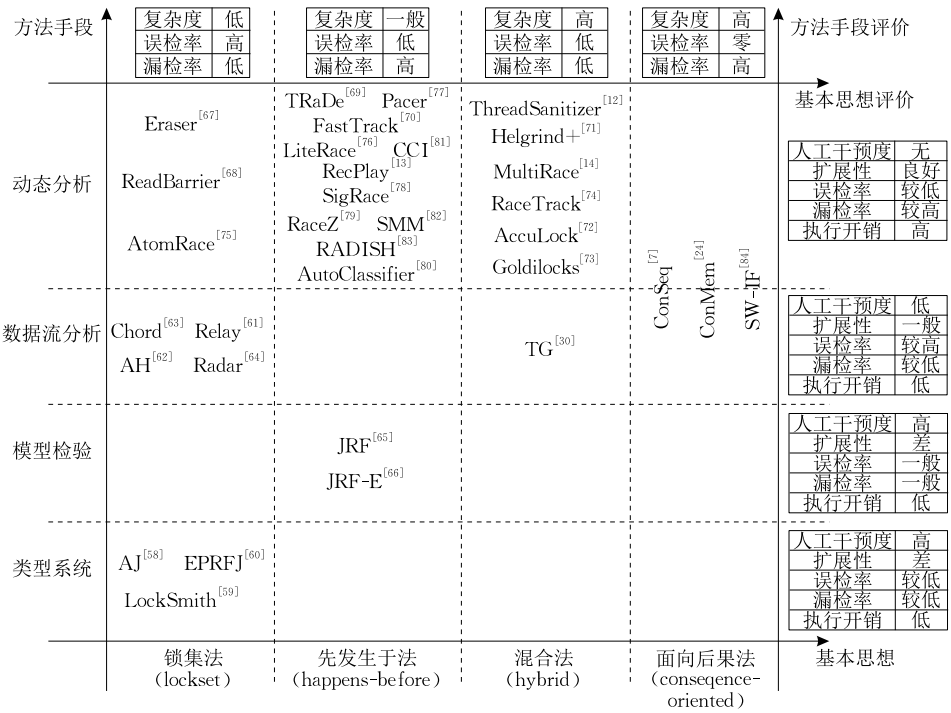

图 10 数据竞争检测研究分类与对比

锁集法检查两个线程对同一共享变量的访问是否具有共同的锁. 如果不具有共同锁,则这两个访问构成一个可能的数据竞争. 在给定精确信息下,锁集法能检测所有数据竞争,但因忽略除锁之外的其他同步原语,如 signal/wait、fork/join 等,它的误检率较高. 锁集法的实现手段一般有类型系统、数据流分析和动态分析. 类型系统将竞争检测问题转化为类型检查问题,检查是否所有对共享数据的访问都被一致性锁保护. 如果某个访问没有被一致性锁保护,则其类型为非法类型,该访问被检测为构成潜在数据竞争的一个元素. 类型系统需要用户添加注释,为共享变量关联一致性锁或者标注原子集标签. 数据流分析则计算每个读写共享变量的访问所持有的锁集,如果两个读写同一共享变量的访问(至少有一个为写访问)的锁集的交集为空,则这两个访问构成一个数据竞争. 数据流分析使用指向分析和逃逸分析,计算共享变量集和判断不同锁集中的元素是否指向同一个锁对象. 数据流分析能处理较大规模程序,能静态检测所有可能发生的数据竞争,其缺点是步骤较多,每步所采用的分析都是不精确的,因此误检很多. RELAY[61]在不使用过滤措施的情况下,准确率仅为 11%. 动态分析监视程序运行,在监视过程中识别共享变量和进行锁集求交运算,如果结果为空,则报告一个数据竞争. 动态分析能准确识别共享变量集和共享变量访问的锁集,因此误检率低,但因动态监视运行,会对目标程序造成较大执行开销.

为克服锁集法由于只考虑锁这一种同步设施而造成的高误检率问题，先发生于法考虑所有同步设施，利用线程内顺序和线程间顺序为程序中的所有操作建立偏序关系，如果对同一共享变量的并发访问不具有偏序关系，则它们构成一个数据竞争. 先发生于法没有误检，然而它对程序的执行交错敏感，在不同的执行交错中检测到的数据竞争也不同. 先发生于法的竞争检测能力弱于锁集法，在同一执行交错中，前者可能漏检后者能检测出来的数据竞争. 另外，先发生于法实现起来也比锁集法困难，开销较大，扩展性差. 先发生于法的实现手段有模型检验和动态分析两种. 模型检验先建立表征并发程序操作语义的抽象模型，然后按照某种优化策略(DFS、BFS或者深度受限搜索)搜索其中的执行交错，使得包含数据竞争的执行交错被尽早搜索到. 模型检验扩展性差，JRF 和 JRF-E 都只针对小规模程序适用. 动态分析直接监视某个执行交错，并实时计算、更新和比较共享变量访问的时间戳，检测实际发生的数据竞争. 动态分析法误报率低，但因既监视每一个同步原语，又监视每一个共享变量访问，执行开销较大. 为降低开销，LiteRace[76]、PACER[77] 和 RACEZ[79] 对程序执行进行动态采样，在监视部分同步原语和共享变量访问的情况下，检测数据竞争. 例如对于一个在当前执行中发生的数据竞争，PACER 保证以近似于采样率的概率检测到该数据竞争，采样率越高，检测概率越大. AutoClassifer[80] 和 RecPlay[13] 则离线检测数据竞争，将执行监视与竞争检测分离. 它们在程序第 1 次运行时记录其执行轨迹，然后根据执行轨迹严格重演程序第 1 次的运行，并在该执行重演中检测数据竞争.

为克服锁集法的高误检率和先发生于法的高漏检率问题，混合法综合使用锁集法和先发生于法. 它首先使用锁集法分析查找可能的数据竞争，然后使用先发生于法去除误检. 混合法在实践中取得了较好的检测效果，误报率和漏检率较低，然而由于需要将两种方法有机结合起来使用，其复杂性较高. 混合法的主要实现手段是动态分析，近来 TG[30] 通过精确的静态分析建立事务图(transaction graph)来抽象表示程序的并发执行行为，在事务图上用数据流分析实现了混合法.

传统方法正向检测数据竞争，即查找可能构成数据竞争的共享变量访问转变. 而面向后果法则动静结合地逆向检测数据竞争. 它先静态分析数据竞争发生后可能导致程序崩溃的程序点，比如读取未赋值变量、空指针引用和内存溢出等；然后逆向查找跟该程序点存在数据和控制依赖的程序点，如果这两个程序点来自不同线程，则检测到一个可能的数据竞争；最后使用随机延时扰动使可能的数据竞争暴露出来. 不同并发缺陷的构成要素虽然不同，但造成的后果却几乎相同. 面向后果法，如 ConSeq[7] 和 ConMem[24]，从并发缺陷可能造成的后果出发逆向推断构成并发缺陷的元素，能检测所有非死锁并发缺陷，包括数据竞争、原子性违背和顺序违背.

5.2　数据竞争规避

规避某一共享变量上的数据竞争的方法有两种：(1)为该变量的所有访问加上一致性锁保护，或者(2)在该变量的任何两个访问之间形成固定执行次序. 软件事务内存[56-57,84-85] 和 AtomRace[75,86] 属于第 1 种，PSet[87] 属于第 2 种，而 CFix[88-89] 和 Loom[90] 则综合使用这两种方法. 我们从规避开销和规避能力角度评价这两种规避方法.

软件事务内存[56-57,84-85] (Software Transaction Memory，STM)把对共享变量的访问视为事务. 对共享变量的一系列操作要么原子性地全部执行，要么一个也不执行. 如果两个由共享变量访问构成的事务在提交时存在冲突，STM 会回退其中一个事务，抛弃其更新，并重新执行被回退的事务，从而保证共享变量的完整性和一致性，避免数据竞争的发生. Grace[57] 使用进程模拟线程执行，这样“线程”对共享变量的更新都被隔离在各自的进程空间内，仅当“线程”终结时才提交这些更新. 如果“线程”之间的更新存在冲突，则最早提交的那个“线程”的更新得以提交，其他“线程”重新执行. 通过以“线程”为单位原子性地执行，Grace 不仅能规避数据竞争，也能规避原子性违背和顺序违背. STM 规避开销小，但因为有些操作，如外围设备 I/O 和网络 I/O，不可回退，故其规避能力有限.

AtomRace[75,86] 检测到某个变量上存在数据竞争时，采取 3 种规避措施：插入随机延时或者改变线程优先级以影响执行调度；为后继对该共享变量的访问加锁；用正确的执行交错的值代替错误的执行交错的值. 其中第 1 种措施只是降低了数据竞争的发生概率，没有完全消除数据竞争；第 2 种措施可能引入新的并发缺陷如死锁；第 3 种措施实现复杂开销大. AtomRace 与 STM 的区别是：针对某共享变量，前者不能规避该变量上的第 1 个数据竞争，而

STM 能规避该变量上的所有数据竞争. AtomRace 控制线程调度,规避开销比 STM 高.

PSet[87] 使用第 2 种方法规避数据竞争. PSet 为每个共享变量操纵指令都关联一个指令集,其中的元素表示该指令的合法前驱,在生产性运行中,只允许前驱集中的指令刚好在该指令之前执行. 指令的前驱集通过内控测试自动学习得到,表示那些经过良好测试、不包含并发缺陷的执行交错. PSet 能规避所有非死锁并发缺陷,但其实现需要修改现有硬件和指令体系结构,并控制线程调度,故规避开销大;由于训练不足和偶然正确性等原因,前驱集对应的执行交错可能仍然包含数据竞争,因此 PSet 规避能力有限.

CFix[88-89] 和 Loom[90] 同时使用两种方法规避数据竞争. CFix 自动为非死锁并发缺陷生成补丁代码. 它以并发缺陷检测工具(如 CTrigger[6] 和 ConSeq[7] 等)的输出作为输入,通过添加互斥锁和条件变量,在相关代码段之间实施符合程序员期望的互斥关系和顺序关系. CFix 能在不引入死锁缺陷的情况下,修复所有非死锁并发缺陷. 但其缺点是只能离线规避并发缺陷. Loom 则能在线规避数据竞争. 在某个数据竞争发生后,Loom 允许用户在线添加一个"执行过滤",在其中指出正确的执行交错顺序,以避免此数据竞争再次发生. Loom 使用互斥锁和信号量来实施"执行过滤",提供即时的竞争规避功能. 其缺点是不能完全规避数据竞争,并且引入死锁的风险较高.

6 原子性违背检测与规避

并发程序中没有数据竞争,并不意味着其中没有原子性违背[7]. 对并发程序来说,原子性是比免于数据竞争更强的正确性保证[91]. 本节讨论原子性违背的检测与规避.

6.1 原子性违背检测

原子性违背一般分为单变量原子性违背和多变量原子性违背两类[92-94],目前针对前者的检测研究较多. 检测原子性违背的关键在于:确定原子性区域和检查原子性区域是否得到原子性执行.

确定原子性区域的方法有 3 种:(1)模式定义. 即人工注释应原子性执行的区域或者经验主义地自动定义原子性区域;(2)统计学习. 即根据预定义的原子性区域模式和多个正确执行的轨迹,自动推断应原子性执行的区域;(3)规则提取. 即提取共享变量访问,组成数据库,然后从数据库中挖掘规则,将规则视为原子性区域. 模式定义和统计学习都需要预先定义原子性区域模式,而规则提取则无任何先验知识地从源码中挖掘原子性区域. 模式定义和统计学习虽然都依赖经验知识,但后者能从训练性执行中学习真正的原子性区域,因此对原子性区域的识别效果好于前者.

针对一个原子性区域,检查其是否得到原子性执行的方法有 5 种:(1)动态监视. 即监视是否有不可序列化的访问交错到该原子性区域中;(2)执行控制. 即控制线程调度以监视不可序列化的访问能否交错到该原子性区域中;(3)轨迹分析. 即记录程序执行,分析执行轨迹,检查是否有不可序列化的访问曾交错到该原子性区域;(4)执行规约. 即检查当前执行交错是否可等价规约为这样的执行交错:在其中,原子性区域被顺序执行;(5)反例检测. 即根据被提取出来的规则(原子性区域)的置信度和支持集,检查是否有不支持该规则的代码段.

根据如何确定原子性区域和如何检查原子性执行,图 11 对现有研究进行分类,并从人工干预度、扩展性、误检率、漏检率和执行开销 5 个维度进行对比. 原子性违背的检测研究以动态分析或者动静结合分析为主,纯静态的研究较少.

"统计学习+动态监视"方法[15,18],分析并发程序多次正确执行后得到的多个执行轨迹,离线学习应原子性执行的代码区域,并在后续的执行中在线检查这些原子性区域是否得到原子性执行. AVIO[18] 从正确的训练性执行中学习"访问交错不变量",即线程内未被其他线程的指令交错的指令对. 与 AVIO 类似,DefUse[15] 也需要训练性执行,并从中学习"定义-使用不变量". 对于图 6 所示的原子性违背,AVIO 能检测(e)~(h)所有 4 种类型,而 DefUse 只能检测(g)一种类型. 虽然 DefUse 原子性违背检测能力较弱,但它能检测顺序违背. 该方法不需要或者只需要很少的人工干预,扩展性好,动态监视程序执行并检测原子性违背,因此误检率低,但漏检率高,对目标程序的执行性能影响较大.

"模式定义+动态监视"(动态分析)方法[95-99],监视程序运行,根据预先定义的原子性区域模式,在线推断符合预定义模式的区域为原子性区域,使用针对预定义模式的专用算法检测原子性违背. SVD[95] 将一个共享变量的读指令,以及数据依赖和

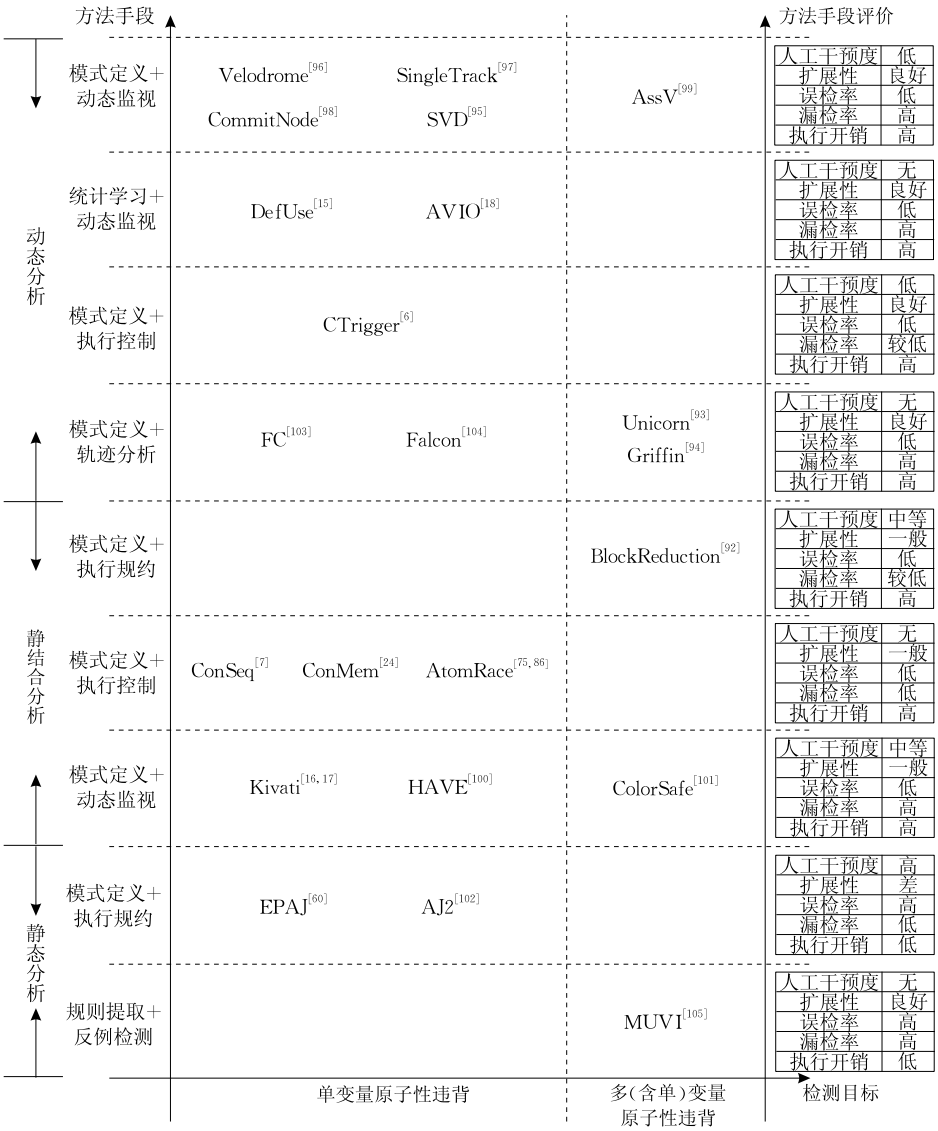

图 11　原子性违背检测研究分类与对比

控制依赖该共享变量的指令，定义为一个“计算单元”，然后监视程序执行，检测是否有其他线程的写指令交错到该“计算单元”. Velodrome[96] 和 SingleTrack[97] 在线将程序划分为互不相交的多个原子性区域，根据指令间的 hb 关系，建立区域间的偏序图，并实时检测其上是否有环存在. 如果发现偏序图上有环存在，则报告检测到的原子性违背以及被违背的原子性区域. CommitNode[98] 在线监视对共享变量的读写操作、加锁解锁操作和条件控制操作，建立访问树和访问森林，其中一棵访问树表示一个原子性区域. CommitNode 检查每棵访问树是否至多只拥有一个“提交节点”，如果否，则检测到一个原子性违背. AssV[99] 从反面定义原子性区域，即定义能够导致原子性违背的内存访问序列模式. AssV 通过动态检查目标程序对共享内存的访问序列是否符合预定义模式来检测原子性违背. 该方法需要少量人工干预，即经验主义地定义原子性区域的模式，扩展性较好. 但由于动态监视执行，因此其具有误检率低的优点和漏检率高、执行开销高的缺点.

“模式定义＋动态监视”(动静结合分析)方法[16-17,100-101] 通过人工指定或者静态分析，标注原子性区域或者原子变量集，然后通过动态监视检查是否有不可序列化的访问交错到原子性区域或者对原子变量集的多个访问构成的执行交错是否不可序列化. Kivati[16-17] 静态分析并转换程序源码，在原子性区域的开始和结尾插入插桩代码，以便在运行时检测该区域的原子性是否被违背. ColorSafe[101] 将属于同一类原子变量集的变量着上相同的颜色，并将所有对变量的访问转变为对某一种颜色的访问. ColorSafe 在颜色层面上，定义五种不可序列化的执

行交错,据此在线检测多变量(含单)原子性违背.该方法需要人工指定或者静态分析来标注原子性区域或者原子变量集,因此人工干预度较高,扩展性较差.同时由于动态监视执行,故具有误检率低的优点和漏检率高、执行开销高的缺点.

"模式定义+执行控制"(动态分析)方法[6]根据预定义模式推断原子性区域,控制线程调度,检查不可序列化访问能否交错到原子性区域中.CTrigger[6]使用 pcr 技术检测原子性违背,然后根据一条可能触发原子性违背的测试用例,多次运行目标程序,利用随机延时扰动影响线程调度,试图增大原子性违背暴露(检测)的概率.该方法人工干预度低,扩展性良好,由于控制线程调度以增大原子性违背的暴露概率,该方法的漏检率较低.

"模式定义+执行控制"(动静结合分析)方法[7,24,75,86]通过静态分析,经验主义地标注原子性区域,然后在运行时控制线程调度,试图使原子性违背暴露出来.ConSeq[7]和 ConMem[24]检测原子性违背的方式与检测数据竞争类似(5.1 节),这里不再赘述.AtomRace[75,86]静态在每个共享变量访问的前后插桩,在运行时根据插桩信息检测原子性违背.该方法无需人工干预,但由于使用静态分析确定原子性区域,其扩展性一般,漏检率低,同时由于控制线程调度,其执行开销较大.

"模式定义+执行规约"(静态分析)方法[60,102]通过人工注释来标注原子性区域,静态检查并发程序的所有执行交错是否可规约为某个所有原子性区域顺序执行的执行交错.EPAJ[60]和 AJ2[102]定义五种基本的原子性类型及其偏序关系.每个原子性区域根据类型推理规则被赋予一个原子性类型.如果某个原子性区域的类型不为 atomic,则报告该区域可能得不到原子性执行.该方法静态检查原子性,对目标程序的性能影响小;但需要人工注释,人工干预度高,扩展性差;由于依赖指向分析等不精确的静态分析技术,其误检率较高,但漏检率低.

"模式定义+执行规约"(动静结合分析)方法[92]依赖经验知识,静态标注原子性区域,然后动态检查当前执行交错是否可规约为如下的执行交错:其中的原子性区域都顺序执行.BlockReduction[92]监视程序执行,在线进行执行规约,并使用动态逃逸分析、hb 分析和多重锁集分析等技术提高静态执行规约方法[60,102]的准确度和检测能力.由于使用锁集分析,BlockReduction 不仅能检测当前执行交错中的原子性违背,而且能预测其他未被执行的执行交错中的原子性违背.该方法需要静态标注原子性区域,人工干预度较高,扩展性较差,但由于能预测原子性违背,其漏检率较低.

"模式定义+轨迹分析"方法[93-94,103-104]分析多个正确或者错误执行的执行轨迹,根据预定模式识别不可序列化的执行交错,并将这些执行交错与执行结果(正确或者错误)关联起来,利用错误定位的方法检测导致并发程序执行出错的原子性违背.FC[103]和 Falcon[104]定义 5 种不可序列化模式,据此检测单变量原子性违背,而 Unicorn[93]和 Griffin[94]定义 14 种不可序列化模式,能检测双(和单)变量原子性违背.该方法无需人工干预,扩展性良好;由于通过分析执行轨迹检测原子性违背,其误检率低,漏检率高;且由于监视程序运行以生成执行轨迹,其执行开销大.

"规则提取+反例检测"方法[105]使用数据挖掘静态挖掘规则(即原子性区域),然后根据规则的置信度和支持集,检查是否有违反该规则的代码段.MUVI[105]从程序源码中挖掘"多变量访问关联"规则.这些规则表示多个变量应该被一致性更新和读取.通过检查那些置信度不为 1 的关联规则,MUVI 检测违反关联规则的反例,它们所对应的程序片段即为原子性违背.该方法不需要人工干预,扩展性好;但由于数据挖掘结果的偶然相关性,被挖掘出来的规则可能并不表示一个原子性区域,与此规则相关的原子性违背都是误检,因此该方法误检率高;同时由于只考虑检测违反"多变量访问关联"规则的原子性违背,该方法漏检率高.

6.2 原子性违背规避

原子性违背的规避研究根据规避措施分成 3 类:控制线程调度[16-17,75,86-87,91,106],添加同步设施[75,86,88-89,107-109]或者实施事务内存[57,101,110-111].控制线程调度可能造成活锁,而添加同步设施和实施事务内存则可能导致死锁.我们从规避开销和规避能力两方面评价这 3 类措施.

第 1 类措施控制线程调度以使得不可序列化的访问在原子性区域之外执行.Kivati[16-17]借助处理器硬件的帮助,消除交错到原子性区域中的远程指令的影响,重新排列本地指令和远程指令的执行,实现对原子性违背的在线规避.AtomRace[75,86]在不可序列化访问即将发生时插入随机延时,以降低其交错到原子性区域的概率.PSet[87]根据指令的前驱

集调度线程，只允许前驱集中的指令在本指令之前执行. AtomAid[91] 和 BulkSC[106] 将所有被执行的指令划分为指令块，线程调度以指令块为单位进行. 如果某一原子性区域落在某一指令块内，则其一定会被原子性执行. 该类措施直接控制或者间接影响线程调度，规避开销较大. 如果间接影响线程调度，则其只能降低原子性违背的发生概率，而不能完全规避原子性违背.

第 2 类措施为原子性区域添加锁等同步设施，使有可能冲突的原子性区域互斥执行. AtomRace[75,86] 在线检测原子性违背，并为原子性违背涉及到的原子性区域添加锁保护，这样原子性违背一旦被检测到就会在后续的执行中被规避掉. JAS[107] 根据“原子变量集”注释，使用静态分析，为读取和更新“原子变量集”中的变量的操作加上一致性锁保护. CFix[88-89] 和 Axis[108] 以其他原子性违背检测工具(如 CTrigger[6]) 的检测结果为输入，静态分析源码，为被违背的原子性区域添加锁保护. Autolocker[109] 与 JAS 类似，不过前者允许用户指定原子性区域，因而支持细粒度的原子执行. 该类措施通常通过分析程序源码为原子性区域添加锁保护，能在编译阶段规避原子性违背，对目标程序的性能影响小，故规避开销小，但其规避能力受限于用户注释或者其他检测工具的输出.

第 3 类措施将原子性区域置于事务中执行. Grace[57] 使用 5.2 节的方法规避原子性违背. ColorSafe[101] 能在某次没有发生原子性违背的执行中预测可能发生的原子性违背，并监视与此相关的多个变量，在后续执行中一旦发现对这些变量的访问，就自动添加瞬时事务内存，使得这些访问在事务内存的保护中进行，从而动态规避原子性违背. 事务内存乐观地看待并发程序的执行，认为事务之间发生冲突的可能性很低，如果发生冲突，就提交先执行的那个事务，重新执行另一个事务. 因此，第 3 类措施对目标程序的执行影响小且规避能力强.

7　研究展望

在当前多核架构充分普及的硬并发时代，并发程序设计已经成为软件开发行业的主流. 可以预见，今后导致软件不能正常工作的缺陷将不再是传统结构化程序设计中的 bug，而主要是并发程序设计中的 cug(concurrency bug). 目前，针对并发缺陷的暴露、检测和规避研究正方兴未艾，我们预计在今后一段时间内，研究工作的重点将集中在以下几个方面：

(1) 智能快速的缺陷暴露. 当前的缺陷暴露技术要么盲目控制线程调度，不能保证一定触发并发缺陷；要么依赖其他检测工具的输出作为输入，限制了自身的暴露能力. 将来的缺陷暴露技术应不依赖于外界输入，而是根据经验知识有意识地调度线程，快速定位到那些包含并发缺陷的执行交错.

(2) 通用准确的缺陷检测. 当前的缺陷检测研究的通用性较差，大多数只能检测一种并发缺陷，少数研究(如 ConMem[24]、ConSeq[7] 和 Falcon[104]) 虽能检测多种，但却不能检测所有并发缺陷. 这就导致测试人员不得不使用多种工具对同一软件进行缺陷检测. 另一个问题是误检率或者漏检率偏高，基于静态分析的检测技术误检率高，而基于动态分析和动静结合分析的检测技术漏检率高. 将来的缺陷检测方案应能检测所有并发缺陷，同时具有较低的误检率和漏检率.

(3) 确定性重放支持. 并发程序执行的不确定性导致并发程序的执行交错数量巨大，并发缺陷不容易暴露和检测. 目前基于动态分析的并发缺陷暴露和检测技术对执行交错敏感，即如果在当前执行交错暴露和检测到某个并发缺陷，则下次执行交错中几乎不可能再次暴露和检测到该并发缺陷. 这就大大降低了暴露和检测技术的有效性，增加了并发缺陷调试的难度. 确定性重放能使并发程序的某次执行具有可重复性，从而并发缺陷在新的执行中仍然能被暴露和检测出来. 将来的缺陷暴露和检测技术应具有确定性重放支持.

(4) 软硬件协同. 基于动态分析的并发缺陷检测和规避技术，如果以纯软件方法实现则对现有的硬件平台没有特别的要求，因而有较强的适用性. 但在软件层面监视运行并实施检测和轨迹技术会严重降低并发程序的执行速度. 为降低执行开销，部分研究使用纯硬件的方式检测和规避并发缺陷，如 HW-IF[84] 和 RaceSMM[82] 添加新硬件以高效检测和规避数据竞争，AVIO-H[18] 修改缓存一致性协议来检测原子性违背. 有硬件支持的检测和规避方案需要修改当前体系架构，可用性较差. 另外，基于硬件的解决方案针对不同种类的并发缺陷需要开发设计不同的硬件和缓存协议，缺乏通用性. 将来的并发缺陷检测和规避研究应提出一种软硬结合的解决方案，设计通用的硬件体系结构以支持多种并发缺陷

的检测和规避,将检测和规避功能合理划分到软件和硬件中,以充分利用各自的优势.

(5)新的并发编程模型.现有并发编程模型为实现计算任务的并发/并行化,既需要程序员划分任务的输入数据集,又需要程序员熟悉底层的同步设施和实现应用层面的同步机制.这就导致在现有模型下,并发缺陷几乎不可能得到完全彻底的解决.为彻底消除并发缺陷,人们已尝试新的并发编程模型,如 MapReduce[112-115] 系列. MapReduce,特别是 Phoenix[114],为从根本上消除并发缺陷提供了一种思路,但并不是万能药. MapReduce 需要程序员将待处理的数据集划分成多个独立的数据块,然而有些计算任务(如 LU 分解)的数据集中元素相互关联,无法划分成独立的数据块.另外,MapReduce 不适合规模较小的计算任务,并且对于输入数据集是随时间变化的计算任务(如在线视频解析,其输入是流式数据),MapReduce 也不适用.将来的并发编程模型应借鉴 MapReduce 的基本原理,同时克服其缺点,即允许数据块之间存在关联,能处理各种规模的计算任务,允许计算任务的输入数据集是流式数据.

8 总 结

本文综述了近 10 年来共享内存系统中并发缺陷的暴露、检测与规避机制与策略.我们首先指出软件开发必须从顺序设计模式转向并发设计模式,然后讨论并发程序的特点以及由此导致的并发缺陷为何难以暴露、检测与修复的问题.本文将常见并发缺陷分成四大类,逐类给出定义和示例,讨论 4 类并发缺陷相互之间的关系,并就如何快速暴露、及时检测和在线规避这 4 类并发缺陷对已有研究作出分析、比较和归纳.最后从 5 个方面展望了未来的研究重点.

参 考 文 献

[1] Gochman S, Mendelson A, Naveh A, et al. Introduction to Intel core duo processor architecture. Intel Technology Journal, 2006, 10(2): 89-97

[2] Sutter H. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal, 2005, 30(3): 202-210

[3] Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. ACM SIGPLAN Notices, 2007, 42(6): 446-455

[4] Bron A, Earchi E, Magid Y, et al. Applications of synchronization coverage//Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Chicago, USA, 2005: 206-212

[5] Sen K. Race directed random testing of concurrent programs. ACM SIGPLAN Notices, 2008, 43(6): 11-21

[6] Park S, Lu S, Zhou Y. CTrigger: Exposing atomicity violation bugs from their hiding places. ACM SIGPLAN Notices, 2009, 44(3): 25-36

[7] Zhang W, Lim J, Olichandran R, et al. ConSeq: Detecting concurrency bugs through sequential errors. ACM SIGPLAN Notices, 2011, 47(4): 251-264

[8] Lu S, Park S, Seo E, et al. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. ACM SIGARCH Computer Architecture News, 2008, 36(1): 329-339

[9] Yin Z N, Yuan D, Zhou Y Y, et al. How do fixes become bugs?—A comprehensive characteristic study on incorrect fixes in commercial and open source operating systems//Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Szeged, Hungary, 2011: 26-36

[10] Agarwal R, Stoller S D. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables//Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging. Portland, USA, 2006: 51-60

[11] Shimomura T, Ikeda K. Two types of deadlock detection: Cyclic and acyclic. Intelligent Systems for Science and Information, 2014, 54(2): 233-259

[12] Serebryany K, Iskhodzhanov T. ThreadSanitizer: Data race detection in practice//Proceedings of the Workshop on Binary Instrumentation and Applications. New York, USA, 2009: 62-71

[13] Ronsse M, De Bosschere K. RecPlay: A fully integrated practical record/replay system. ACM Transactions on Computer Systems, 1999, 17(2): 133-152

[14] Pozniansky E, Schuster A. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. Concurrency and Computation: Practice and Experience, 2007, 19(3): 327-340

[15] Shi Y, Park S, Yin Z, et al. Do I use the wrong definition?: DefUse: Definition-use invariants for detecting concurrency and sequential bugs. ACM SIGPLAN Notices, 2010, 45(10): 160-174

[16] Chew L, Lie D. Kivati: Fast detection and prevention of atomicity violations//Proceedings of the 5th European Conference on Computer Systems. Paris, France, 2010: 307-320

[17] Chew L. A System for Detecting, Preventing and Exposing Atomicity Violations in Multithreaded Programs [Ph. D. dissertation]. University of Toronto, Toronto, 2009

[18] Lu S, Tucek J, Qin F, et al. AVIO: Detecting atomicity violations via access interleaving invariants. ACM SIGARCH Computer Architecture News, 2006, 34(5): 37-48

[19] Agarwal R, Wang L, Stoller S D. Detecting potential deadlocks with static analysis and run-time monitoring// Proceedings of the 1st International on Hardware and Software, Verification and Testing. Haifa, Israel, 2006: 191-207

[20] Agarwal R, Bensalem S, Farchi E, et al. Detection of deadlock potentials in multithreaded programs. IBM Journal of Research and Development, 2010, 54(5): 3:1-3:15

[21] Musuvathi M, Qadeer S, Ball T, et al. Finding and reproducing heisenbugs in concurrent programs//Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. San Diego, USA, 2008: 267-280

[22] Burckhardt S, Kothari P, Musuvathi M, et al. A randomized scheduler with probabilistic guarantees of finding bugs. ACM SIGPLAN Notices, 2010, 45(3): 167-178

[23] Joshi P, Park C S, Sen K, et al. A randomized dynamic program analysis technique for detecting real deadlocks. ACM SIGPLAN Notices, 2009, 44(6): 110-120

[24] Zhang W, Sun C, Lim J, et al. ConMem: Detecting crash-triggering concurrency bugs through an effect-oriented approach. ACM Transactions on Software Engineering and Methodology, 2013, 22(2): 10:1-10:33

[25] Cai Y, Wu S R, Chan W K. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs//Proceedings of the 36th International Conference on Software Engineering. Hyderabad, India, 2014: 491-502

[26] Cai Y, Chan W K. MagicFuzzer: Scalable deadlock detection for large-scale applications//Proceedings of the 2012 International Conference on Software Engineering. Zurich, Switzerland, 2012: 606-616

[27] Bensalem S, Bozga M, Nguyen T H, et al. D-finder: A tool for compositional deadlock detection and verification// Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009). Grenoble, France, 2009: 614-619

[28] Bensalem S, Griesmayer A, Legay A, et al. Efficient deadlock detection for concurrent systems//Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Co-Design. Cambridge, UK, 2011: 119-129

[29] Zeng H, Miao H. Deadlock detection for parallel composition of components//Proceedings of the 9th International Conference on Computer and Information Science 2010. Kaminoyama, Japan, 2010: 23-34

[30] Kahlon V, Sankaranarayanan, S, Gupta A. Static analysis for concurrent programs with applications to data race detection. International Journal on Software Tools for Technology Transfer, 2013, 15: 321-336

[31] Naik M, Park C S, Sen K, et al. Effective static deadlock detection//Proceedings of the 31st International Conference on Software Engineering. Vancouver, Canada, 2009: 386-396

[32] Williams A, Thies W, Ernst M D. Static deadlock detection for Java libraries//Proceedings of the 19th European Conference on Object-Oriented Programming. Glasgow, UK, 2005: 602-629

[33] Bensalem S, Havelund K. Scalable dynamic deadlock analysis of multi-threaded programs//Proceedings of the 3rd International Workshop on Parallel and Distributed Systems: Testing and Debugging. Haifa, Israel, 2005

[34] Li T, Ellis C S, Lebeck A R, et al. Pulse: A dynamic deadlock detection mechanism using speculative execution// Proceedings of the 2005 USENIX Annual Technical Conference. Anaheim, USA, 2005: 31-44

[35] Pradel M, Gross T R. Fully automatic and precise detection of thread safety violations. ACM SIGPLAN Notices, 2012, 47(6): 521-530

[36] Jula H, Candea G. A scalable, sound, eventually-complete algorithm for deadlock immunity//Proceedings of the 8th International on Runtime Verification. Budapest, Hungary, 2008: 119-136

[37] Jula H, Tralamazza D M, Zamfir C, et al. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks//Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation. San Diego, USA, 2008: 295-308

[38] Koskinen E, Herlihy M. Dreadlocks: Efficient deadlock detection//Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures. Munich, Germany, 2008: 297-303

[39] Nethercote N, Seward J. Valgrind: A framework for heavy-weight dynamic binary instrumentation. ACM SIGPLAN Notices, 2007, 42(6): 89-100

[40] Pyla H K, Varadarajan S. Avoiding deadlock avoidance// Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. Vienna, Austria, 2010: 75-86

[41] Luo Z D, Das R, Qi Y. MulticoreSDK: A practical and efficient deadlock detector for real-world applications// Proceedings of the 4th International Conference on Software Testing, Verification and Validation. Toronto, Canada, 2011: 309-318

[42] Cai Y, Chan W K. Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs. IEEE Transactions on Software Engineering, 2014, 40(3): 266-281

[43] Cai Y, Ke Z, Wu S R, et al. TeamWork: Synchronizing threads globally to detect real deadlocks for multithreaded programs//Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Shenzhen, China, 2013: 311-312

[44] Marino D, Hammer C, Dolby J, et al. Detecting deadlock in programs with data-centric synchronization//Proceedings of

the 35th International Conference on Software Engineering. San Francisco, USA, 2013: 322-331

[45] Boudol G. A deadlock-free semantics for shared memory concurrency//Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing. Kuala Lumpur, Malaysia, 2009: 140-154

[46] Gordon C S, Ernst M D, Grossman D. Static lock capabilities for deadlock freedom//Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation. Philadelphia, USA, 2012: 67-78

[47] Gerakios P, Papaspyrou N, Sagonas K. A type and effect system for deadlock avoidance in low-level languages// Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation. Austin, USA, 2011: 15-28

[48] Gerakios P, Papaspyrou N, Sagonas K, et al. Dynamic deadlock avoidance in systems code using statically inferred effects//Proceedings of the 6th Workshop on Programming Languages and Operating Systems. Cascais, Portugal, 2011: 1-5

[49] Wang Y, Kelly T, Kudlur M, et al. Gadara: Dynamic deadlock avoidance for multithreaded programs//Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation. San Diego, USA, 2008: 281-294

[50] Kelly T, Wang Y, Lafortune S, et al. Eliminating concurrency bugs with control engineering. Computer, 2009, 42(12): 52-60

[51] Liao H, Wang Y, Stanley J, et al. Eliminating concurrency bugs in multithreaded software: A new approach based on discrete-event control. IEEE Transactions on Control System Technology, 2013, 21(6): 2067-2082

[52] Wang Y, Liao H, Reveliotis S, et al. Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software//Proceedings of the 48th IEEE Conference on Decision and Control. Shanghai, China, 2009: 4971-4976

[53] Liao H, Stanley J, Wang Y, et al. Deadlock-avoidance control of multithreaded software: An efficient siphon-based algorithm for Gadara Petri nets//Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference. Orlando, USA, 2011: 1142-1148

[54] Lafortune S, Wang Y, Reveliotis S. Eliminating concurrency bugs in multithreaded software: An approach based on control of Petri nets//Proceedings of the 34th International Conference on Application and Theory of Petri Nets and Concurrency. Milano, Italy, 2013: 21-28

[55] Nir-Buchbinder Y, Tzoref R, Ur S. Deadlocks: From exhibiting to healing//Proceedings of the 8th International on Runtime Verification. Budapest, Hungary, 2008: 104-118

[56] Qin F, Tucek J, Sundaresan J, et al. Rx: treating bugs as allergies—A safe method to survive software failures. ACM SIGOPS Operating Systems Review, 2005, 39(5): 235-248

[57] Berger E D, Yang T, Liu T, et al. Grace: Safe multithreaded programming for C/C++. ACM Sigplan Notices, 2009, 44(10): 81-96

[58] Vaziri M, Tip F, Dolby J, et al. A type system for data-centric synchronization//Proceedings of the 24th European Conference on Object-Oriented Programming. Maribor, Slovenia, 2010: 304-328

[59] Pratikakis P, Foster J S, Hicks M. LOCKSMITH: Context-sensitive correlation analysis for race detection. ACM SIGPLAN Notices, 2006, 41(6): 320-331

[60] Sasturkar A, Agarwal R, Wang L, et al. Automated type-based analysis of data races and atomicity//Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Chicago, USA, 2005: 83-94

[61] Voung J W, Jhala R, Lerner S. RELAY: Static race detection on millions of lines of code//Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. Dubrovnik, Croatia, 2007: 205-214

[62] Kahlon V, Yang Y, Sankaranarayanan S, et al. Fast and accurate static data-race detection for concurrent programs// Proceedings of the 19th International Conference on Computer Aided Verification. Berlin, Germany, 2007: 226-239

[63] Naik M, Aiken A, Whaley J. Effective static race detection for Java//Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. Ottawa, Canada, 2006: 308-319

[64] Chugh R, Voung J W, Jhala R, et al. Dataflow analysis for concurrent programs using datarace detection. ACM SIGPLAN Notices, 2008, 43(6): 316-326

[65] Kim K H, Kahveci T Y, Sanders B A. Precise data race detection in a relaxed memory model using heuristic based model checking//Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering. Auckland, USA, 2009: 495-499

[66] Kim K H, Kahveci T Y, Sanders B A. JRF-E: Using model checking to give advice on eliminating memory model-related bugs//Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering. New York, USA, 2010: 215-224

[67] Savage S, Burrows M, Nelson G, et al. Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems, 1997, 15(4): 391-411

[68] Nishiyama H. Detecting data races using dynamic escape analysis based on read barrier//Proceedings of the 3rd Conference on Virtual Machine Research and Technology Symposium. San Jose, USA, 2004: 127-138

[69] Christiaens M, De Bosschere K. TRaDe, a topological approach to on-the-fly race detection in java programs// Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium. Monterey, USA, 2001: 1-12

[70] Flanagan C, Freund S N. FastTrack: Efficient and precise dynamic race detection. ACM SIGPLAN Notices, 2009, 44(6): 121-133

[71] Jannesari A, Tichy W F. Library-independent data race detection. IEEE Transactions on Parallel and Distributed Systems, 2013, 20(2): 401-411

[72] Xie X W, Xue J L. AccuLock: Accurate and efficient detection of data races. Software: Practice and Experience, 2013, 43(5): 543-576

[73] Elmas T, Qadeer S, Tasiran S. Goldilocks: A race and transaction-aware java runtime. ACM SIGPLAN Notices, 2007, 42(6): 245-255

[74] Yu Y, Rodeheffer T, Chen W. Racetrack: Efficient detection of data race conditions via adaptive tracking. ACM SIGOPS Operating Systems Review, 2005, 39(5): 221-234

[75] Letko Z, Vojnar T, Krena B. AtomRace: Data race and atomicity violation detector and healer//Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. Seattle, USA, 2008: 1-10

[76] Marino D, Musuvathi M, Narayanasamy S. LiteRace: Effective sampling for lightweight data-race detection. ACM SIGPLAN Notices, 2009, 44(6): 134-143

[77] Bond M D, Coons K E, McKinley K S. PACER: Proportional detection of data races. ACM SIGPLAN Notices, 2010, 45(6): 255-268

[78] Muzahid A, Suárez D, Qi S, et al. SigRace: Signature-based data race detection. ACM SIGARCH Computer Architecture News, 2009, 37(3): 337-348

[79] Sheng T, Vachharajani N, Eranian S, et al. RACEZ: A lightweight and non-invasive race detection tool for production applications//Proceedings of the 33rd International Conference on Software Engineering. Honolulu, USA, 2011: 401-410

[80] Narayanasamy S, Wang Z, Tigani J, et al. Automatically classifying benign and harmful data races using replay analysis. ACM SIGPLAN Notices, 2007, 42(6): 22-31

[81] Jin G L, Thakur A, Liblit B, et al. Instrumentation and sampling strategies for cooperative concurrency bug isolation//Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications. Reno, USA, 2010: 241-255

[82] Huang R R, Halberg E, Ferraiuolo A, et al. Low-overhead and high coverage run-time race detection through selective meta-data management//Proceedings of the 20th International Symposium on High Performance Computer Architecture. Orlando, USA, 2014: 96-107

[83] Devietti J, Wood B P, Strauss K, et al. RADISH: Always-on sound and complete race detection in software and hardware//Proceedings of the 39th Annual International Symposium on Computer Architecture. New York, USA, 2012: 201-212

[84] Qi S X, Muzahid A A, Ahn W, et al. Dynamically detecting and tolerating IF-Condition data races//Proceedings of the 20th International Symposium on High Performance Computer Architecture. Orlando, USA, 2014: 120-131

[85] Herlihy M, Luchangco V, Moir M. A flexible framework for implementing software transactional memory. ACM SIGPLAN Notices, 2006, 41(10): 253-262

[86] Krena B, Letko Z, Tzoref R, et al. Healing data races on-the-fly//Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging. London, UK, 2007: 54-64

[87] Yu J, Narayanasamy S. A case for an interleaving constrained shared-memory multi-processor. ACM SIGARCH Computer Architecture News, 2009, 37(3): 325-336

[88] Jin G, Zhang W, Deng D, et al. Automated concurrency-bug fixing//Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. Hollywood, USA, 2012: 221-236

[89] Jin G, Song L, Zhang W, et al. Automated atomicity-violation fixing. ACM SIGPLAN Notices, 2011, 46(6): 389-400

[90] Wu J, Cui H, Yang J. Bypassing races in live applications with execution filters//Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation. Vancouver, Canada, 2010: 135-149

[91] Lucia B, Devietti J, Strauss K, et al. Atom-Aid: Detecting and surviving atomicity violations//Proceedings of the 35th International Symposium on Computer Architecture. Beijing, China, 2008: 277-288

[92] Wang L, Stoller S D. Runtime analysis of atomicity for multithreaded programs. IEEE Transactions on Software Engineering, 2006, 32(2): 93-110

[93] Park S, Vuduc R, Harrold M J. Unicorn: A unified approach for localizing non-deadlock concurrency bugs. Journal of Software Testing, Verification and Reliability, 2014, 24(5): 101-123

[94] Park S, Harrold M J, Vuduc R. Griffin: Grouping suspicious memory-access patterns to improve understanding of concurrency bugs//Proceedings of the 2013 International Symposium on Software Testing and Analysis. Lugano, Switzerland, 2013: 134-144

[95] Xu M, Bodik R, Hill M D. A serializability violation detector for shared-memory server programs. ACM SIGPLAN Notices, 2005, 40(6): 1-14

[96] Flanagan C, Freund S N, Yi J. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. ACM SIGPLAN Notices, 2008, 43(6): 293-303

[97] Sadowski C, Freund S N, Flanagan C. SingleTrack: A dynamic determinism checker for multithreaded programs//Proceedings of the 7th Asian Symposium on Programming Languages and Systems. Seoul, Korea, 2009: 394-409

[98] Wang L, Stoller S D. Accurate and efficient runtime detection of atomicity errors in concurrent programs//Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, USA, 2006: 137-146

[99] Hammer C, Dolby J, Vaziri M, et al. Dynamic detection of atomic-set-serializability violations//Proceedings of the 30th International Conference on Software Engineering. Leipzig, Germany, 2008: 231-240

[100] Chen Q, Wang L, Yang Z, et al. HAVE: Detecting atomicity violations via integrated dynamic and static analysis//Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering. York, UK, 2009: 425-439

[101] Lucia B, Ceze L, Strauss K. ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. ACM SIGARCH Computer Architecture News, 2010, 38(3): 222-233

[102] Flanagan C, Freund S N, Lifshin M. Type inference for atomicity//Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. Long Beach, USA, 2005: 47-58

[103] Park S. Fault comprehension for concurrent programs//Proceedings of the 35th International Conference on Software Engineering. San Francisco, USA, 2013: 1444-1446

[104] Park S, Vuduc R W, Harrold M J. Falcon: Fault localization in concurrent programs//Proceedings of the 32nd International Conference on Software Engineering. Cape Town, South Africa, 2010: 245-254

[105] Lu S, Park S, Hu C, et al. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. ACM SIGOPS Operating Systems Review, 2007, 41(6): 103-116

[106] Ceze L, Tuck J, Montesinos P, et al. BulkSC: Bulk enforcement of sequential consistency. ACM SIGARCH Computer Architecture News, 2007, 35(2): 278-289

[107] Vaziri M, Tip F, Dolby J. Associating synchronization constraints with data in an object-oriented language. ACM SIGPLAN Notices, 2006, 41(1): 334-345

[108] Liu P, Zhang C. Axis: Automatically fixing atomicity violations through solving control constraints//Proceedings of the 2012 International Conference on Software Engineering. Zurich, Switzerland, 2012: 299-309

[109] McCloskey B, Zhou F, Gay D, et al. Autolocker: Synchronization inference for atomic sections. ACM SIGPLAN Notices, 2006, 41(1): 346-358

[110] Feng M, Gupta R, Neamtiu I. Programming support for speculative execution with software transactional memory//Proceedings of the 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum. Cambridge, UK, 2013: 394-403

[111] Zhang W, Dekruijf M, Li A, et al. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution//Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. Houston, USA, 2013: 113-126

[112] Wang Shan, Wang Hui-Ju, Qin Xiong-Pai, Zhou Xuan. Architecting big data: Challenges, studies and forecasts. Chinese Journal of Computers, 2011, 34(10): 1741-1752(in Chinese)
(王珊, 王会举, 覃雄派, 周烜. 架构大数据: 挑战, 现状与展望. 计算机学报, 2011, 34(10): 1741-1752)

[113] Li Jian-Jiang, Cui Jian, Wang Dan, et al. Survey of MapReduce parallel programming model. Acta Electronica Sinica, 2011, 39(11): 2635-2642(in Chinese)
(李建江, 崔健, 王聃等. MapReduce 并行编程模型研究综述. 电子学报, 2011, 39(11): 2635-2642)

[114] Ranger C, Raghuraman R, Penmetsa A, et al. Evaluating MapReduce for multi-core and multiprocessor systems//Proceedings of the 13th International Symposium on High Performance Computer Architecture. Phoenix, USA, 2007: 13-24

[115] Kang Li-Yun, Wang Xiao-Yue, Bai Ru-Jiang. Analysis of MapReduce principle and its main implementation platforms. New Technology of Library and Information Service, 2012, (2): 60-67(in Chinese)
(亢丽芸, 王效岳, 白如江. MapReduce 原理及其主要实现平台分析. 现代图书情报技术, 2012, (2): 60-67)

SU Xiao-Hong, born in 1966, Ph.D., professor. Her research interests include software bugs detection, software refactoring, information fusing, target detecting and tracking.

YU Zhen, born in 1987, Ph.D. candidate. His research interests include implicit programming rules mining, concurrency bugs detecting and avoiding.

WANG Tian-Tian, born in 1980, Ph.D., associate professor. Her research interests include program analysis, software bugs detecting, software testing.

MA Pei-Jun, born in 1963, Ph.D., professor. His research interests include information fusing, software engineering, target detecting and tracking.

Background

Multi-core architecture makes real concurrency come true. In order to benefit from concurrency power, concurrent programming is becoming increasingly popular. In the near future, we believe that concurrent programming will overwhelm sequential programming and become the mainstream programming model. However, out of inherent concurrency and non-determinism, concurrent programs may be prone to suffer from concurrency bugs, which are notoriously hard to detect, debug and repair.

Recently, there has been a lot of exploration work on how to quickly expose, in time detect and efficiently avoid/repair concurrency bugs. This paper overviews the current state of the art and provides a taxonomy of the latest researches. The authors divide the common concurrency bugs into four categories, namely deadlock, data race, atomicity violation and order violation. We then explore the relationship among the four categories. As far as we know, this paper is the first work to categorize concurrency bugs and probe into the correlation among categories. Additionally, we notice that, before being observed, detected and checked, concurrency bugs must manifest themselves first. There are common ways to expose concurrency bugs, however, no common ways to detect and avoid/repair them. We classify three techniques that are able to expose all categories of concurrency bugs, namely random time delaying, thread controlled scheduling and the fuzzing technique. For each category of concurrency bugs, we make analyses, comparisons and summarizations on previous research papers that try to expose, detect and avoid them. At last, we look into the research priorities in future from five aspects.

This research is partly supported by the National Natural Science Foundation of China under Grant Nos. 61173021, 61202092. The group has been working for several years on software bugs detecting, software faults understanding and locating, clone codes and their related defects detection, software refactoring, software testing, and so on. A number of research papers have been published in respectable international journals and conferences, such as *Journal of Systems and Software*, *Journal of Computer Languages*, *Systems and Structures*, *the 21st ACM International Symposium on the Foundation of Software Engineering*, etc.