软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(8):3726−3744 [doi: 10.13328/j.cnki.jos.006618]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 Petri 网展开的多线程程序数据竞争检测与重演*

鲁法明[1], 黄 莹[1,2], 曾庆田[1], 包云霞[1], 唐梦凡[1]

[1](山东科技大学 计算机科学与工程学院, 山东 青岛 266590)

[2](中国科学院 深圳先进技术研究院, 广东 深圳 518055)

通信作者: 黄莹, E-mail: ds00hy@163.com; 包云霞, E-mail: baoyunxia98@163.com

摘　要: 数据竞争是多线程程序的常见漏洞之一, 传统的数据竞争分析方法在查全率和准确率方面难以两全, 而且所生成检测报告难以定位漏洞的根源. 鉴于 Petri 网在并发系统建模和分析方面具有行为描述精确、分析工具丰富的优点, 提出一种基于 Petri 网展开的新型数据竞争检测方法. 首先, 对程序的某一运行轨迹进行分析和挖掘, 构建程序的一个 Petri 网模型, 它由单一轨迹挖掘得到, 却可隐含程序的多个不同运行轨迹, 由此可在保证效率的同时降低传统动态分析方法的漏报率; 其次, 提出基于 Petri 网展开的潜在数据竞争检测方法, 相比静态分析方法在有效性上有较大提升, 而且能明确给出数据竞争的产生路径; 最后, 对上一阶段检测到的潜在数据竞争, 给出基于 CalFuzzer 平台的潜在死锁重演调度方法, 可剔除误报, 保证数据竞争检测结果的真实性. 开发相应的原型系统, 结合公开的程序实例验证了所提方法的有效性.

关键词: 数据竞争; Petri 网; 网展开; 动态程序分析

中图法分类号: TP311

中文引用格式: 鲁法明, 黄莹, 曾庆田, 包云霞, 唐梦凡. 基于Petri网展开的多线程程序数据竞争检测与重演. 软件学报, 2023, 34(8): 3726–3744. http://www.jos.org.cn/1000-9825/6618.htm

英文引用格式: Lu FM, Huang Y, Zeng QT, Bao YX, Tang MF. Data Race Detection and Replay of Multi-threaded Programs Based on Petri Net Unfolding. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3726–3744 (in Chinese). http://www.jos.org.cn/1000-9825/6618.htm

Data Race Detection and Replay of Multi-threaded Programs Based on Petri Net Unfolding

LU Fa-Ming[1], HUANG Ying[1,2], ZENG Qing-Tian[1], BAO Yun-Xia[1], TANG Meng-Fan[1]

[1](College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China)

[2](Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China)

Abstract: Data races are common defects in multi-threaded programs. Traditional data race analysis methods fail to achieve both recall and precision, and their detection reports cannot locate the root cause of defects. Due to the advantages of Petri nets in terms of accurate behavior description and rich analysis tools in the modeling and analysis of concurrent systems, this study proposes a new data race detection method based on Petri net unfolding technology. First, a Petri net model of the program is established by analyzing and mining a program running trace. The model implies different traces of the program even though it is mined from a single trace, which can reduce the false negative rate of traditional dynamic analysis methods while ensuring performance. After that, a Petri net unfolding-based detection method of program potential data races is proposed, which improves the efficiency significantly compared with static analysis methods and can clearly show the triggering path of data race defects. Finally, for the potential data race detected in the previous stage, a scheduling schema is designed to replay the defect based on the CalFuzzer platform, which can eliminate false positives and ensure the authenticity of detection results. In addition, the corresponding prototype system is developed, and the effectiveness of the proposed

* 基金项目: 国家自然科学基金 (61602279); 山东省泰山学者工程专项基金 (ts20190936); 山东省高等学校青创科技支持计划 (2019KJN024); 山东省博士后创新专项基金 (201603056); 国家海洋局海洋遥测工程技术研究中心开放基金 (2018002); 山东科技大学教学名师培育计划 (MS20211102)
本文由“形式化方法与应用”专题特约编辑陈立前副教授、孙猛教授推荐.
收稿时间: 2021-09-05; 修改时间: 2021-10-14; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28
CNKI 网络首发时间: 2023-02-23

method is verified by open program instances.

Key words: data race; Petri net; net unfolding; dynamic program analysis

大数据时代, 随着对计算性能要求的不断提高, 云计算和多线程程序等并行计算技术越来越受到人们的重视[1,2]. 在多线程程序中, 由于线程调度的时序不确定性以及共享内存空间访问控制的复杂性, 如果设计不当, 程序会存在诸多并发缺陷, 进而导致运算结果出错甚至程序崩溃[3]. 这些并发缺陷的检测和排除是多线程程序设计的一大挑战, 而数据竞争[4]就是引起广泛关注的并发缺陷之一.

数据竞争是指对同一块内存空间存在并发访问, 并且至少有一个访问是写操作. 数据竞争通常会对程序的正确性产生恶劣影响. 但是, 数据竞争的产生场景往往难以检测, 不仅需要特定的输入, 还需要特定的线程调度顺序, 这造成了数据竞争检测的困难. 当前, 数据竞争检测方法一般分为静态检测、动态检测和动静结合的数据竞争检测方法 3 类[5].

静态检测方法针对程序源代码进行分析, 主要结合锁集对潜在的数据竞争缺陷进行检测, 常见的静态检测工具包括 RacerX[6]、RELAY[7]和 locksmith[8]等. 该类方法通过静态代码分析程序的完整行为, 存在查全率高的优点. 不过, 由于静态分析方法不实际执行程序, 它以源代码为输入直接进行程序分析, 难以准确获取程序的运行时信息, 从而导致了较高的误报率. 此外, 对程序全部行为进行分析使得静态分析方法时间效率偏低.

动态检测方法监视程序执行过程中的行为, 收集必要的信息来判断哪些操作构成数据竞争. 常见的动态数据竞争检测算法分为基于 lockset 的算法[9−11], 基于 happens-before 关系的算法[12−17], 以及两者混合的 hybrid 算法[18−20]这 3 种. 基于 lockset 的方法维护程序执行过程中每个线程的当前持有锁信息, 同时更新共享变量持有的锁信息, 当共享变量不再受到锁保护的时候, 报告数据竞争; 基于 happens-before 关系的方法借助逻辑时钟识别两个操作之间因为同属于一个线程、因为锁的释放和申请, 以及因为线程的 fork 或 join 而导致的因果关系, 将不具备上述因果关系的两个操作识别为并发. 如果同一个共享变量的读写/写写操作之间是并发的, 则认定两者构成数据竞争; hybrid 算法通常先基于 lockset 找到可疑的数据竞争, 然后借助逻辑时钟等方法对可疑竞争的真实性进行验证. 受限于线程执行交错的不确定性以及所收集信息的不完整性, 动态竞争方法会有较多的漏检, 但是数据竞争的动态分析方法一般比静态方法有更高的准确度.

动静结合的数据竞争检测方法通常先通过静态方法检测潜在的数据竞争, 然后, 在程序动态执行过程中对线程的调度进行干预, 以此增加数据竞争发生的概率, 重演成功的数据竞争均为真实的并发缺陷. 例如, 文献 [21] 从一个已有的动态分析技术中获得的潜在数据竞争的相关信息, 并据此控制线程的随机调度程序, 以提高真实数据竞争发生条件的出现概率, 以此提高数据竞争重演成功的概率; 文献 [22] 首先使用静态分析方法识别潜在的并发错误, 然后使用静态程序切片来针对潜在的并发错误获取较小的程序, 最后, 动态控制线程的调度以便多个线程同时访问同一内存位置, 以此验证并发错误是否会导致程序失败. 文献 [23] 以有害数据竞争的检测为目标, 首先综合考虑 happens-before 关系与 ad-hoc 类型的同步来精简需验证的数据竞争数量, 然后, 将相互之间不存在干扰的潜在数据竞争划分到同一组中, 再借助线程调度器以组为单位对潜在数据竞争进行重演. 动静结合的数据竞争检测方法综合两种方法的优点, 在数据竞争的查全率、准确度方面均有一定保证, 不过, 前期静态分析仍然耗时, 后期动态重演通常是一些随机性的调度方法, 重演成功率有待进一步提高.

综上所述, 目前的数据竞争检测方法在查全率、准确率方面难以两全, 而且数据竞争检测的效率也有待提高. 此外, 如文献 [5] 所述, 现有方法所产生的数据竞争报告使得开发者难以理解并据此追溯漏洞产生根源. 针对上述问题, 考虑到 Petri 网在并发系统建模和行为分析方面的优点, 本文拟提出一种基于 Petri 网展开[24−30]的新型数据竞争检测方法. 首先, 与动态检测方法类似, 所提方法也是从多线程程序某次具体的运行轨迹出发进行分析, 不过, 从该轨迹出发, 本文能构造出一个蕴含多种运行轨迹的程序 Petri 网模型, 从该模型出发进行数据竞争检测能降低传统动态分析方法的漏报率, 提高查全率; 然后, 与静态分析方法类似, 通过对前述程序 Petri 网模型的分析检测潜在的数据竞争缺陷, 本文将数据竞争检测问题转化为 Petri 网展开[30]中共享变量访问操作的并发关系识别, 可有效提高数据竞争的检测效率; 与此同时, 潜在数据竞争对应的出现网片段可以很好地展示数据竞争产生的根源和具体路径; 最后, 由于程序运行轨迹中捕获的程序行为信息不够完备, 由此挖掘到的 Petri 网模型可能与源程序在行

为上不一致, 从而导致数据竞争误报的现象, 为解决此问题, 本文给出一种数据竞争重演方法, 重演成功的数据竞争可保证真实性, 从而可提高检测准确率. 就性能而言, 本文从单一运行轨迹出发构造仅包含程序部分行为的 Petri 网模型, 相比静态分析方法构造反映程序全部行为的模型而言, 所提方法简化了模型构建和分析的复杂度, 在一定程度上可提升数据竞争检测方法的效率.

1 实例与动机分析

与经典的数据竞争动态分析方法类似, 本文假设程序由有限个线程构成, 这些线程通过共享对象的加锁/解锁以及读写进行同步和交互. 在程序的一次运行过程中, 将线程的启动/终止/阻塞、锁的获取和释放, 以及共享对象读写等操作所构成的操作序列称为一条多线程程序运行轨迹, 其形式化定义如下.

定义 1. 多线程程序运行轨迹. 多线程程序的运行轨迹, 记作 α, 是满足如下条件的一个操作序列:

$$\alpha \in Trace ::= Operation*$$

$$Operation ::= c:\text{fork}(u,v)|c:\text{join}(u,v)|c:\text{stop}(u)|c:\text{acq}(u,l)|c:\text{rel}(u,l)|c:\text{rd}(u,x)|c:\text{wr}(u,x)$$

其中,

- u, v 表示线程, l 表示锁, x 表示一个共享变量, c 表示程序语句的标签 (例如行号、列号等);
- $\text{fork}(u,v)$ 表示线程 u 启动线程 v;
- $\text{join}(u,v)$ 表示线程 u 阻塞直到线程 v 终止;
- $\text{stop}(u)$ 表示线程 u 停止;
- $\text{acq}(u,l)$ 表示线程 u 获取锁 l;
- $\text{rel}(u,l)$ 表示线程 u 释放锁 l;
- $\text{rd}(u,x)$ 表示线程 u 读共享变量 x;
- $\text{wr}(u,x)$ 表示线程 u 写共享变量 x.

以图 1 中的 Java 多线程程序为例, 它包括主线程、threadA 和 threadB 这 3 个线程、两个共享变量 x 与 *flag*、一个锁对象 *lock*. 主线程首先为共享变量 *flag* 指派一个随机值, 然后依次启动线程 threadA 和 theadB; 待两个线程都结束后, 主线程终止. 线程 threadA 启动后, 首先将共享变量 x 的值设置为 1, 然后尝试获取锁对象 *lock*; 获取 *lock* 成功后将共享变量 *flag* 的值设置为 true. 线程 threadB 启动后, 首先尝试获取锁对象 *lock*, 获取成功后读取共享变量 *flag* 并将其值赋予私有变量 *new_flag*; 若 *new_flag* 之后的取值为 true 则将 x 的取值设置为 2.

程序 Program 1 存在两处关于共享变量 x 的数据竞争. 具体而言, 若线程 threadB 首先获得 *lock* 的使用权, 当主线程为 *flag* 指派的初值为 true 时, 第 22 行 threadB 对 x 的写操作与 threadA 对 x 的写操作构成数据竞争; 当主线程为 *flag* 指派的初值为 false 时, 第 24 行 threadB 对 x 的写操作与 threadA 对 x 的写操作构成数据竞争.

然而, 程序 Program 1 也存在一些运行轨迹不会呈现上述并发缺陷, 例如表 1 中的操作序列. 当中, 线程 threadA 首先获得锁对象 *lock* 的使用权, 并将共享变量 *flag* 的值设为 true; 然后, 线程 threadB 获得 *lock* 使用权, 并将共享变量 x 的取值最终设置为 2. 若程序按照表 1 中的轨迹执行, 则前述数据竞争不会触发.

对于图 1 中的 Program 1, 首先, 基于 lockset 的数据竞争检测方法[19]不适用, 因为该方法要求所有的共享变量在执行读写操作时都必须通过锁来保护, 而 Program 1 对共享变量 x 的访问操作均未加锁保护; 其次, 基于 happens-before 关系的数据竞争检测方法, 若以表 1 中的运行轨迹为输入进行检测, 也无法检测到实际存在的竞争. 以得到广泛认可的 VerifiedFT[17]数据竞争检测算法为例, 基于 happens-before 关系的数据竞争方法认为运行轨迹中一个锁释放操作与其后首次执行的该锁的申请操作间存在必然的因果关系. 例如, 表 1 的执行轨迹中, 由于线程 threadA 释放锁对象 *lock* 的操作后, threadB 首次申请 *lock*, 故 VerifiedFT 等基于 happens-before 关系的方法会假设在锁对象 *lock* 的使用权上, 线程 threadA 一定先于 threadB, 从而在两个操作之间加上一个错误的时序约束关系. 而原本的程序行为并不存在这一约束, 线程 threadB 完全可以先获得 *lock* 的使用权, 然后对变量 x 进行写操作, 进而导致与 threadA 写 x 时的数据竞争. 由此可见, 传统的 happens-before 关系对程序行为人

为添加了更多的约束, 从而导致数据竞争的漏报. lockset 与 happens-before 两者混合的 hybrid 算法同时存在上述两个问题.

```
1   public class DataRaceTest {
2     static int x = 0;
3     static boolean flag = true;
4     static final Object lock = new Object ();
5     public static void  main (String [] args) throws InterruptedException {
6       flag = (new Random ()).nextBoolean ();
7       Thread threadA = new Thread () {
8           public void  run () {
9               x = 1;
10              synchronized (lock) {
11                  flag = true  ;
12              }
13          }
14      };
15      Thread threadB = new Thread () {
16          public void  run () {
17              boolean new_flag = false;
18              synchronized (lock) {
19                  new_flag = flag;
20              }
21              if (new_flag)
22                  x = 2;
23              else
24                  x = 3;
25          }
26      };
27      threadA.start ();
28      threadB.start ();
29      threadA.join ();
30      threadB.join ();
31      System.out.println ("The value of x is "+ x);
32    }
33  }
```

图 1　多线程程序实例 Program 1

表 1　程序 Program 1 的一个可能的操作执行序列

序号	主线程mainThread	线程threadA	线程threadB
1	6: wr(mainThread, *flag*)		
2	27: fork(mainThread, threadA)		
3	28: fork(mainThread, threadB)		
4		9: wr(threadA, *x*)	
5		10: acq(threadA, *lock*)	
6		11: wr(threadA, *flag*)	
7		12: rel(threadA, *lock*)	
8			18: acq(threadB, *lock*)
9			19: rd(threadB, *flag*)
10			20: rel(threadB, *lock*)
11			22: wr(threadB, *x*)
12	29: join(mainThread, threadA)		
13	30: join(mainThread, threadB)		
14	31: rd(mainThread, *x*)		
15	32: stop(mainThread)		

为解决上述问题, 本文不对共享变量的访问做锁保护的限定, 并从给定的程序运行轨迹出发挖掘能尽量准确反映程序行为的 Petri 网模型. 所挖掘模型中, 仅在如下情况下建立操作语句之间的因果关系: (1) 同一线程内相继执行的两个操作语句; (2) 线程的启动操作 fork 与被启动线程的第 1 个操作语句; (3) 线程的 join 操作与等待线程

的最后一个语句. 此外, 对于锁对象竞争导致的运行轨迹多样化的问题, 本文通过 Petri 网中的冲突结构来刻画这种行为的多样性. 如此一来, 前述动态分析方法中错误添加的因果关系便可以消除, 而且, 所挖掘模型中蕴含了多种潜在的程序运行轨迹, 这为本文方法更准确地、检测出更多的潜在数据竞争缺陷提供了可能.

此外, 由于定义 1 中的程序运行轨迹捕获的程序行为有限, 由此导致由其挖掘得到的 Petri 网模型对程序行为的刻画并不完全准确, 而且 Petri 网的运行原理与并发程序的执行语义不完全一致, 这导致网模型可能蕴含一些虚假的可执行操作序列, 进而导致数据竞争的误报. 针对这一问题, 本文将给出潜在数据竞争的重演方法, 以此保证本文方法所检测数据竞争的真实性.

最后, 需要指出的是, 由于动态分析以程序的运行轨迹为漏洞检测依据, 而线程执行交错的不确定性和程序控制逻辑与数据流交互的复杂性使得程序各类不同的运行轨迹难以完整捕获, 丢失的运行轨迹可能会导致数据竞争动态检测的漏报. 以图 1 中的程序为例, 若仅以表 1 中的运行轨迹为检测依据, 包括本文方法在内的动态分析方法无法检测到第 9 行对 x 的写语句与第 24 行对 x 的写语句间存在的数据竞争, 因为该轨迹中第 24 行语句对应的操作并未出现. 解决这一问题的关键在于生成尽量多的测试用例以尽可能地覆盖程序执行路径, 文献 [31] 提出了一种用于多线程程序数据竞争检测的测试用例自动生成技术, 其生成的测试用例覆盖率经评估可达 94%, 虽然能覆盖大多数程序执行路径, 但丢失的路径信息都可能导致数据竞争的漏报.

2 基于运行轨迹的多线程程序 Petri 网模型挖掘

Petri 网[24,25]是一种广泛应用于并发系统建模和分析的工具, 其相关定义如下.

定义 2. Petri 网. Petri 网是一个四元组 $\Sigma=(P,T;F,M_0)$, 其中, $P=\{p_1,p_2,\ldots,p_n\}$ 为库所集, $T=\{t_1,t_2,\ldots,t_m\}$ 为变迁集, $P\cap T=\varnothing$ 并且 $P\cup T\neq\varnothing$, $F\subseteq(P\times T)\cup(T\times P)$ 称为网的流关系, $M_0:P\to\{0,1,2,\ldots\}$ 为 Σ 的初始标识.

定义 3. 变迁使能与执行规则. 对 $\forall x\in P\cup T$, 称 ${}^\bullet x=\{y\in P\cup T|(y,x)\in F\}$ 为 x 的前集, $x^\bullet=\{y\in P\cup T|(x,y)\in F\}$ 为 x 的后集. 在标识 M 下, 若变迁 t 的每个前驱中都含有至少一个标记, 则称 t 是使能的. 使能的变迁可以执行从而引发系统状态标识发生改变. T 执行后的标识 M'满足如下条件, 并记之为 $M[t>M'$.

$$M'(p)=\begin{cases}M(p)-1, & \text{if } p\in {}^\bullet t-t^\bullet\\ M(p)+1, & \text{if } p\in t^\bullet-{}^\bullet t\\ M(p), & \text{otherwise}\end{cases}.$$

对 $\forall\sigma\in T^*$, 若 $M_0[\sigma_1>M_1[\sigma_2>M_2[\sigma_3>\ldots>M_{k-1}[\sigma_k>M$, 则 σ 称为一个可引发变迁序列, 并称 M 为系统的一个可达标识, Σ 的所有可达标识的集合记为 $R(\Sigma)$.

进行系统建模时, Petri 网的库所通常表示某种状态或者资源, 变迁通常表示某种动作, 流关系通常用以刻画动作执行的前提条件和导致的后继状态, 标识用于建模网系统的状态. 本文使用 Petri 网对多线程程序的行为进行建模时, 每个变迁对应程序执行过程中的一个操作; 库所分为两类, 控制库所用于建模线程的控制流状态, 资源库所用于建模程序中的锁对象; Petri 网中的标识表示程序的运行状态. 程序运行之初, 仅有主线程的初始控制流库所和各个锁对象对应的资源库所含有一个标记, 其余库所标记数为 0.

图 2 就是根据表 1 的程序运行轨迹挖掘到的一个能反映图 1 程序部分行为的 Petri 网模型 (具体挖掘方法稍后给出). 模型中, 每个变迁 (用小矩形表示) 对应一个程序操作, 变迁同时标注了其变迁 ID 和对应的程序操作; 黑色边线的库所 (用圆表示) 为某个线程的控制库所, 建模控制流状态; 红色边线的库所为一个资源库所, 建模一个锁对象. 控制流库所中的 token (用黑点表示) 表示线程的当前控制流状态处于激活态, 资源库所中的 token 表示当前锁对象处于可用状态. 初始状态下主线程的就绪态被激活, 各个锁对象可用, 故它们对应的库所各自有一个 token, 相应地, 图 2 中的系统初始标识满足 $M_0(p_0)=1\ \wedge M_0(p_{lock})=1\ \wedge\ \forall p\in P-\{p_0,p_{lock}\}:M_0(p)=0$, 当中 p_0 是一个控制流库所, 对应主线程的就绪状态; p_{lock} 是一个资源库所, 对应程序中的锁对象 *lock*.

就程序 Petri 网模型的挖掘方法而言, 当给定多线程程序的某个运行轨迹后, 可以按照表 2 所示的网模型构建规则构造程序本次运行对应的 Petri 网模型.

具体而言, 首先为主线程的就绪状态和各个锁对象分别构建一个含有 token 的库所; 然后, 针对运行轨迹中各个操作的种类, 按照表 2 中给出的规则添加相应的变迁及其输入、输出库所和流关系即可. 具体地说, 每个共享变

量的读/写操作所对应变迁仅有唯一的前驱库所和唯一的后继库所, 分别对应该操作所属线程的两个控制流状态; 每个线程 fork 操作所对应变迁有一个输入库所对应该操作所在线程的上一个控制流状态, 两个输出库所分别对应该操作所在线程的下一个控制流状态, 以及被启动线程的就绪状态; 每个线程 join 操作所对应变迁有两个输入库所分别对应该操作所在线程的上一个控制流状态, 以及被 join 线程的结束状态, 有唯一的输出库所对应该操作所在线程的下一个控制流状态; 对于锁对象的获取操作而言, 它对应的变迁有两个输入库所, 其一对应该操作的前驱控制流状态, 另一个对应申请访问的锁对象; 对于锁对象的释放操作而言, 它对应变迁有两个输出库所, 其一对应该操作的后继控制流状态, 另一个对应释放的锁对象. 此外, 为直观起见, 表 2 为每个共享变量的读写操作额外添加了共享对量对应的一个红色圆圈, 需要指出, 该元素并不属于 Petri 网的库所, 仅起到对共享变量读写的提示作用, 进行 Petri 网行为分析时完全可以忽略这些变量标记. 图 2 中的 Petri 网就省略了这些变量标记.

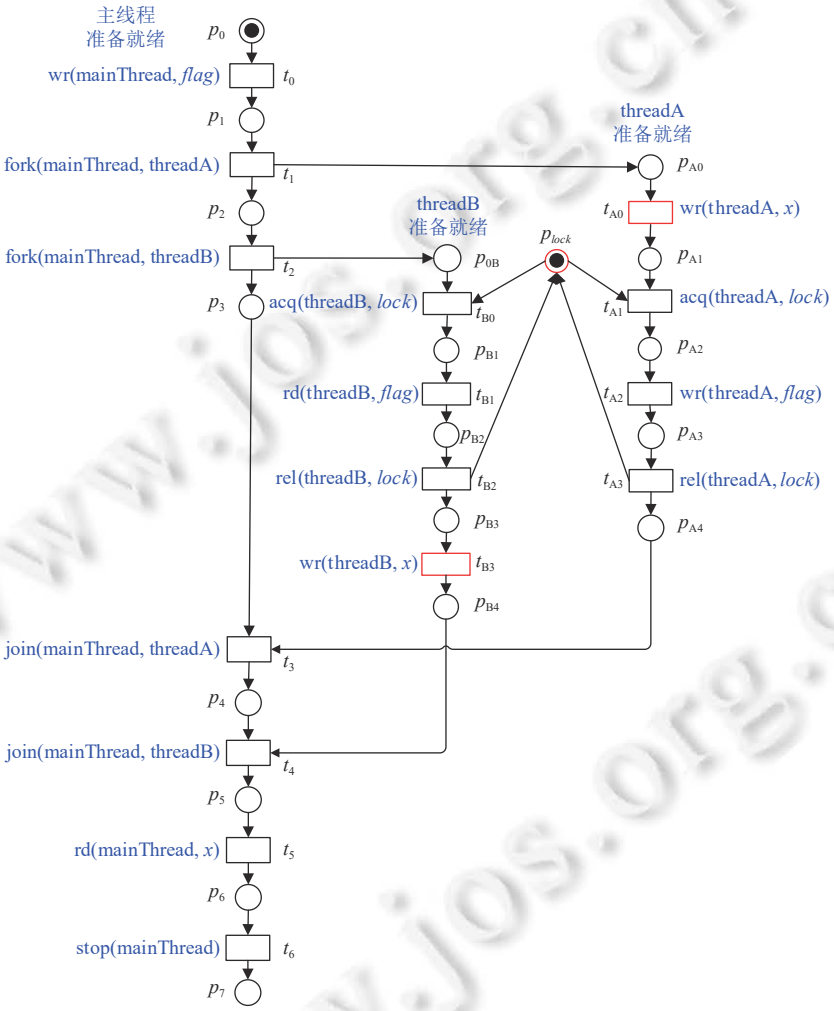
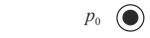
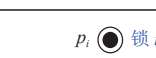
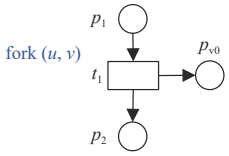
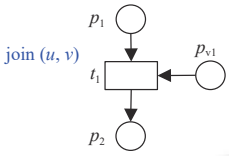
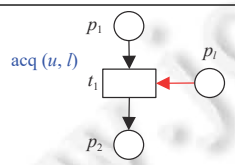
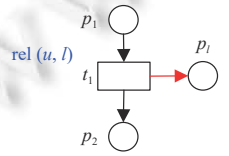
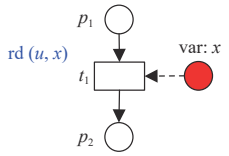
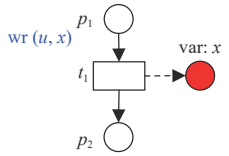
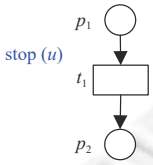


图 2 由表 1 运行轨迹挖掘得到的程序 Petri 网模型

以表 1 中程序 Program 1 的运行轨迹为例, 结合前述 Petri 网模型挖掘方法和表 2 中的 Petri 网模型构造规则, 可得图 2 所示的 Petri 网模型. 初始标识下, 该模型仅有变迁 t_0 可以执行, 它对应源程序中主线程的首条可执行语句, 即对共享变量 *flag* 的写操作; t_0 执行完毕后, 状态库所 p_1 被输入一个 token, 代表主线程的下一个控制流状态被激活; 在这一新的状态下, 变迁 t_1 使能, 代表启动线程 threadA 的操作可以执行; t_1 的执行将同时向库所 p_2 和 p_{A0} 各输入一个 token 以激活他们, 前者对应主线程该操作执行后的后继状态, 后者对应线程 threadA 的就绪状态,

之后的操作依次类推. 实际上, 表 1 中的程序运行轨迹对应着图 2 Petri 网的可执行变迁序列 $\sigma = t_0t_1t_2t_{A0}t_{A1}t_{A2}t_{A3}t_{B0}t_{B1}t_{B2}t_{B3}t_3t_4t_5t_6$, 该变迁序列执行完毕后, 主线程的终止状态库所和锁对象 *lock* 对应的库所各自含有一个 token, 这对应着程序 Program 1 的终止状态.

表 2 程序锁对象、初始状态以及操作所对应的 Petri 网模型

程序对象及其操作	对应的Petri网模型片段	说明
主线程就绪状态		该库所表示主线程的就绪状态, 初始标识下它包含一个token, 表示初始状态下主线程处于就绪状态
锁对象		该库所对应一个锁对象, 初始标识它包含一个token, 表示初始状态下锁对象处于可用状态
fork(u,v)		变迁t_1对应fork操作, 其前驱库所p_1表示线程u中fork操作的前驱控制流状态, 后继库所p_2表示线程u中fork操作的后继控制流状态, 后继库所p_{v0}表示线程v的就绪状态
join(u,v)		变迁t_1对应join操作, 其前驱库所p_1表示线程u中join操作的前驱控制流状态, 前驱库所p_{v1}表示线程v的结束状态; 后继库所p_2表示线程u中join操作的后继控制流状态
acq(u,l)		变迁t_1对应锁对象l的acq操作, 其前驱库所p_1表示线程u中acq操作的前驱控制流状态, 前驱库所p_l表示锁对象l; 后继库所p_2表示线程u中acq操作的后继控制流状态
rel(u,l)		变迁t_1对应锁对象l的rel操作, 其前驱库所p_1表示线程u中rel操作的前驱控制流状态; 后继库所p_2表示线程u中rel操作的后继控制流状态, 后继库所p_l表示锁对象l
rd(u,x)		变迁t_1对应共享变量x的读操作, 其前驱库所p_1表示线程u中该操作的前驱控制流状态; 后继库所p_2表示线程u中该操作的后继控制流状态. 红色圆圈表示的是一个共享变量, 它不属于Petri网的元素, 仅用来描述此操作执行了某个变量的读操作
wr(u,x)		变迁t_1对应共享变量x的写操作, 其前驱库所p_1表示线程u中该操作的前驱控制流状态; 后继库所p_2表示线程u中该操作的后继控制流状态. 红色圆圈表示的是一个共享变量, 它不属于Petri网的元素, 仅用来描述此操作执行了某个变量的写操作
stop(u)		变迁t_1对应线程u的终止操作, 其前驱库所p_1表示线程u中该操作的前驱控制流状态; 后继库所p_2表示线程u中该操作的后继控制流状态

需要强调的是, 虽然上述 Petri 网是由程序的单一运行轨迹挖掘得到, 但它实际隐含了原程序多个不同的可执行操作序列. 究其原因, 虽然某个具体的程序运行轨迹中, 不同线程的锁获取操作对同一个锁对象有固定的先后顺序, 但根据运行轨迹构造 Petri 网模型时并没有强加上这种因果依赖关系.

仍然以图 2 中的 Petri 网为例, 锁对象 *lock* 对应的资源库所 p_{lock} 有两个后继变迁 t_{A1} 与 t_{B0}, 它们构成一种冲突结构, 从而带来了多种不同的可执行变迁序列: 一种情景是变迁 t_{A0} 先引发, 对应着线程 threaA 首先获得锁对象 *lock* 的使用权; 另一种情景是变迁 t_{B1} 先引发, 对应着线程 threaB 首先获得锁对象 *lock* 的使用权. 具体来说, 当网系统在初始状态执行完毕变迁序列 $t_0t_1t_2$ 后, Petri 网的标识为 $M=\{p_3, p_{A0}, p_{B0}, p_{lock}\}$ 当前状态下, 若 t_{A1} 先执行则得到前述可执行变迁序列 σ, 它对应表 1 中的程序运行轨迹; 若 t_{B0} 先执行, 则得到另一个可执行变迁 $\sigma'=t_0t_1t_2t_{B0}t_{B1}t_{B2}t_{B3}t_{A0}t_{A1}t_{A2}t_{A3}t_3t_4t_5t_6$, 它对应着表 3 中的程序运行轨迹. 显然, 表 1 的运行轨迹未触发数据竞争, 而表 3 中第 7、8 行的两个操作则触发了数据竞争. 接下来借助网展开等 Petri 网分析工具对不同执行情景下共享变量读写操作之间可能的并发关系进行检测, 每个可能并发的共享变量读写操作都对应着源程序的一个潜在数据竞争.

表 3 程序 Program 1 的另一个操作执行序列

序号	主线程	线程threadA	线程threadB
1	6: wr(mainThread, flag)		
2	27: fork(mainThread, threadA)		
3	28: fork(mainThread, threadB)		
4			18: acq(threadB, *lock*)
5			19: rd(threadB, *flag*)
6			20: rel(threadB, *lock*)
7			22: wr(threadB, *x*)
8		9: wr(threadA, *x*)	
9		10: acq(threadA, *lock*)	
10		11: wr(threadA, *flag*)	
11		12: rel(threadA, *lock*)	
12	29: join(mainThread, threadA)		
13	30: join(mainThread, threadB)		
14	31: rd(mainThread, *x*)		
15	32: stop(mainThread)		

3 基于 Petri 网展开的潜在数据竞争检测

要检测源程序中某共享变量的读写/写写操作间是否存在数据竞争, 实际只需分析前述所挖掘 Petri 网模型中这两个操作对应的变迁间是否可并发即可. 在诸多 Petri 网分析工具中, Petri 网的展开技术[24,30]可以对变迁之间是否存在并发作出有效判断, 为此, 本文将基于 Petri 网的展开进行程序潜在数据竞争的检测. Petri 网展开技术基于出现网对系统行为进行分析, 下面简单介绍相关概念, 更多内容见文献 [24,26,30].

首先, 对于无标识网 $N=(P,T;F)$ 中的任意节点 $y,y'\in P\cup T$, 若存在变迁 $t,t'\in T$ 使得 ${}^\bullet t\cap {}^\bullet t'\neq\phi\wedge(t,y)\in F^*\wedge(t',y')\in F^*$ (当中, F^* 指流关系 F 的自反传递闭包), 则称节点 y 与 y' 冲突, 记作 $y\#y'$; 若 $(y,y')\in F^+$ 则称 y 与 y' 具有因果关系, 记作 $y\prec y'$; 若 $(y,y')\notin F^*\wedge(y,y')\notin(F^{-1})^*\wedge\neg(y\#y')$, 则称 y 与 y' 并发, 记作 $y\|y'$. 例如, 若忽略图 2 中的 token, 将其视作一个网, 则我们有 $t_{B0}\#t_{A1}$、$t_2\|t_{A0}$ 特殊的, 节点自身与自身冲突的情况称为自冲突, 如 $t_4\#t_4$.

出现网 (本文出现网的定义与文献 [25] 不同. 文献 [25] 要求 $\forall b\in B:|{}^*b|\leqslant 1$, 这意味着网中不允许有冲突, 本文允许不同节点间存在冲突) 是一个三元组 $ON=(B,E;G)$, 当中 B 为库所集, 又称条件集; E 为变迁集, 又称事件集, 它们满足 (1) $\forall b\in B:|{}^*b|\leqslant 1$; (2) $G^+\cap(G^{-1})^+=\phi$, 即网中不含有向圈; (3) $\forall y\in B\cup E:\neg(y\#y)$, 即网中不存在自冲突; (4) $\forall y\in B\cup E$, 集合 $\{x|x\in B\cup E\wedge(x,y)\in G^+\}$ 是有限的, 即任意节点的前驱节点都是有限的.

Petri 网展开利用出现网描述系统行为的基本思想如下: 用出现网中的一个事件表示 Petri 网系统中某个变迁的一次执行, 用出现网中的一个条件表示网系统运行中涉及的某个 token, 出现网中的流关系用以刻画原始 Petri 网中变迁执行对 token 的消耗和产生情况, Petri 网初始标识下的各个 token 分别对应出现网中一个没有前驱的条件. 例如, 图 3 中的出现网就是图 2 Petri 网系统的一个有限展开. 当中, 各节点旁标有两个标签, 节点内部的标签是出现网中该节点的唯一 ID, 节点外的标签是该节点对应的原 Petri 网系统中库所或者变迁节点的 ID, 双边框表示

的事件为截断事件. Petri 网的展开中, 一个事件 e 的局部配置定义为 $[e]-\{e' \mid (e'_2 e) \in G^*\}$, 局部配置对应的 Petri 网可达标识是指从初始状态执行 $[e]$ 中各个事件后到达的原 Petri 网系统标识. 若事件 e 局部配置对应的可达标识与某已经存在之事件局部配置对应的可达标识相等, 则称 e 为截断事件. 例如, 图 3 给出的就是一个出现网, 它用以描述图 2 中 Petri 网的行为.

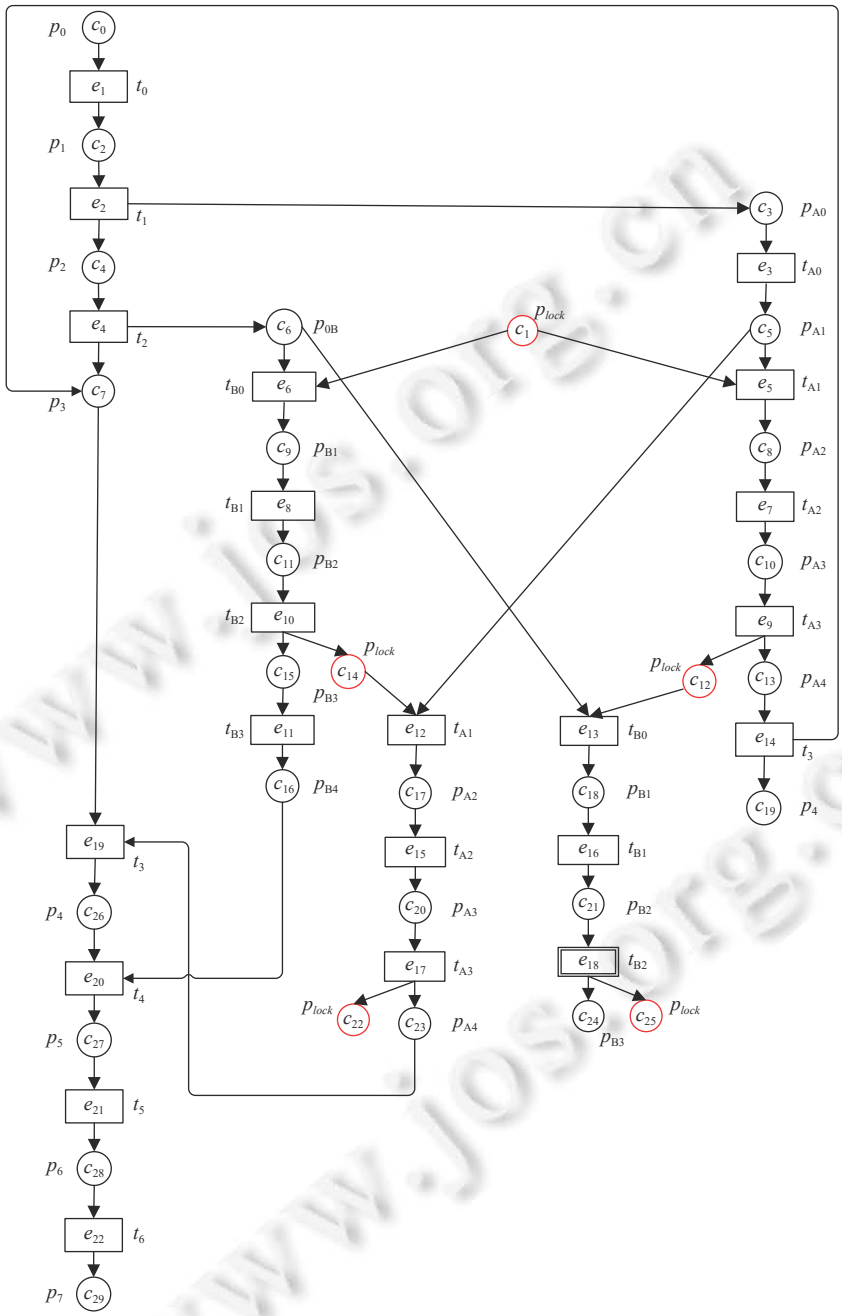


图 3 图 2 中程序 Petri 网模型的展开

文献 [30] 已指出, 对于任意一个有界的 Petri 网系统, 可以构造一个有限的出现网来刻画网系统的行为, 称该出现网为原 Petri 网的有限完全展开. 例如, 图 3 所示的 Petri 网展开中, 前驱为空的条件集合为 $\{c_0, c_1\}$, 它对应原

Petri 网的初始标识$\{p_0, p_{lock}\}$. 事件 e_6 和 e_{13} 对应原 Petri 网中变迁 t_{B0} 两次不同场景下的执行. 具体而言, e_6 对应锁对象 *lock* 先授权给线程 threadB、后授权给线程 threadA 这一场景下 t_{B0} 的一次执行, 而 e_{13} 对应 *lock* 先授权给线程 threadA、后授权给线程 threadB 这一场景下 t_{B0} 的一次执行. 条件 c_1、c_{12}、c_{14}、c_{22} 和 c_{25} 均对应原 Petri 网库所 p_{lock} 中的一个 token. c_1 对应的是 Petri 网初始标识下 p_{lock} 中的 token, c_{12} 对应的是 *lock* 先授权给线程 threadB、后授权给线程 threadA 这一场景下锁释放操作 t_{B2} 执行完毕后产生的 p_{lock} 中的 token, 其余类似可得. 事件 e_{11} 对应的是 *lock* 先授权给线程 threadB、后授权给线程 threadA 这一场景下, 线程 threadB 对变量 x 的写操作 t_{B3} 的一次执行. 而 *lock* 先授权给线程 threadA、后授权给线程 threadB 这一场景下 t_{B3} 的执行则因为截断事件 e_{18} 的存在而被省略. e_{18} 之所以是截断事件, 是因为该事件的局部配置$\{c_{24}, c_{25}, c_{13}, c_7\}$与事件 e_{17} 的局部配置$\{c_{22}, c_{23}, c_{15}, c_7\}$对应着原 Petri 网的同一个标识$\{p_{B3}, p_{lock}, p_{A4}, p_3\}$.

事件 e_{11} 对应线程 threadB 对变量 x 的写操作 t_{B3} 的一次执行, 事件 e_3 对应线程 threadA 对变量 x 的写操作 t_{A0} 的一次执行. 在图 3 的展开中, 两者满足$e_3 \| e_{11}$, 两个事件具有并发关系, 这意味着原 Petri 网中变迁 t_{B3} 与 t_{A0} 存在并发的可能, 这就得到了关于共享变量 x 的一个写冲突, 亦即一个潜在的数据竞争缺陷.

生成 Petri 网展开的算法有多种, 本文基于 Esparza 等人在文献 [30] 中提出的展开方法进行前述多线程程序 Petri 网模型的展开, 具体展开算法见算法 1. 在算法实现过程中, 鉴于本文得到的 Petri 网模型满足 1-safe 性质, 而且本文目的是进行共享变量读写 (或者写写) 操作之间的并发关系检测, 本文使用优先队列和哈希函数的方法对文献 [30] 给出的展开算法进行了性能优化, 具体优化措施包括: (1) 使用优先队列按照 $\prec$ 关系动态维护展开中产生的事件; (2) 按照序关系 $\prec$ 弹出优先队列中最小的 e; (3) 根据哈希函数判断 e 是否为截断事件.

算法 1. 满足 1-safe 性质的 Petri 网模型的展开算法.

输入: 由程序运行轨迹挖掘到的 Petri 网$\Sigma = (P, T; F, M_0)$;

输出: Petri 网模型的展开$\pi = (ON, h)$, 中$ON = (B, E; G)$是一个出现网, h 是将 ON 中的条件/事件映射到Σ中库所/变迁的函数.

步骤:

1. $B := \varnothing, E := \varnothing, G := \varnothing$
2. **FOREACH** ($p \in P$)
3. **IF**($M_0(p) \neq 0$)
4. 向 B 中添加一个新的条件 b, 并令$h(b) := p$;
5. 计算当前展开片段π的候选扩展集合 PE, 每个候选扩展是一个二元组$< t, B_t >$, 当中$t \in T$, $B_t \subseteq B$ 且$h(B_t) = {}^\bullet t$;
6. 初始化截断事件的集合 *Cur-Off* 为$\varnothing$;
7. **WHILE** ($PE \neq \varnothing$) {
8. 在 PE 中选择扩展$< t, B_t >$使得据其添加的事件$e :=< t, B_t >$按照第 3 节定义的关系$\prec$最小;
9. 在 E 中添加事件 e, 并令$h(e) := t$, $G := G \cup \{< b, e > | b \in B_t\}$;
10. **FOREACH** ($p \in t^\bullet$)
11. 向 B 中添加一个新的条件 b, 并令$h(b) := p$; $G := G \cup \{< e, b >\}$;
12. 从 PE 中删除$< t, B_t >$;
12. **IF** (e 不是截断事件)
13. 根据新添加的条件 b 重新计算并更新π的候选扩展集合 PE
14. **ELSE**
15. 向 *cut-off* 中添加事件 e
16. }//WHILE
17. **RETURN** $\pi = (ON, h)$, 当中$ON = (B, E; G)$

在上述 Petri 网的展开过程中, 为便于进行数据竞争检测, 对于每个共享变量的读写事件均会计算与其并发的所有其他读写事件, 然后判断这些并发的读写事件之间是否存在针对同一个共享变量的读写/写写冲突. 若存在,

则输出冲突事件及其全部前驱节点构成的 Petri 网展开的一个前缀, 该前缀就对应着原始程序一个潜在的数据竞争, 称其为该数据竞争伴随的 Petri 网展开前缀, 可记之为一个四元组 $DR\_ON=(B',E';G',h')$, 当中 $(B',E';F')$ 为该展开前缀对应的出现网, h' 是将各个条件库所和变迁映射到源程序 Petri 网模型中一个锁对象库所、控制流状态库所或者程序操作变迁的映射函数.

以图 3 中的 Petri 网展开为例, 当得到事件 e_{11} 并检测到 $e_3 \| e_{11}$ 时, 我们会导出图 4 所示的 Petri 网展开前缀 (为方便理解, 该图中给出了各个节点对应的程序元素, 同时加入了共享变量节点), 它对应的就是第 2 节所述的程序 Program 1 中存在的数据竞争.

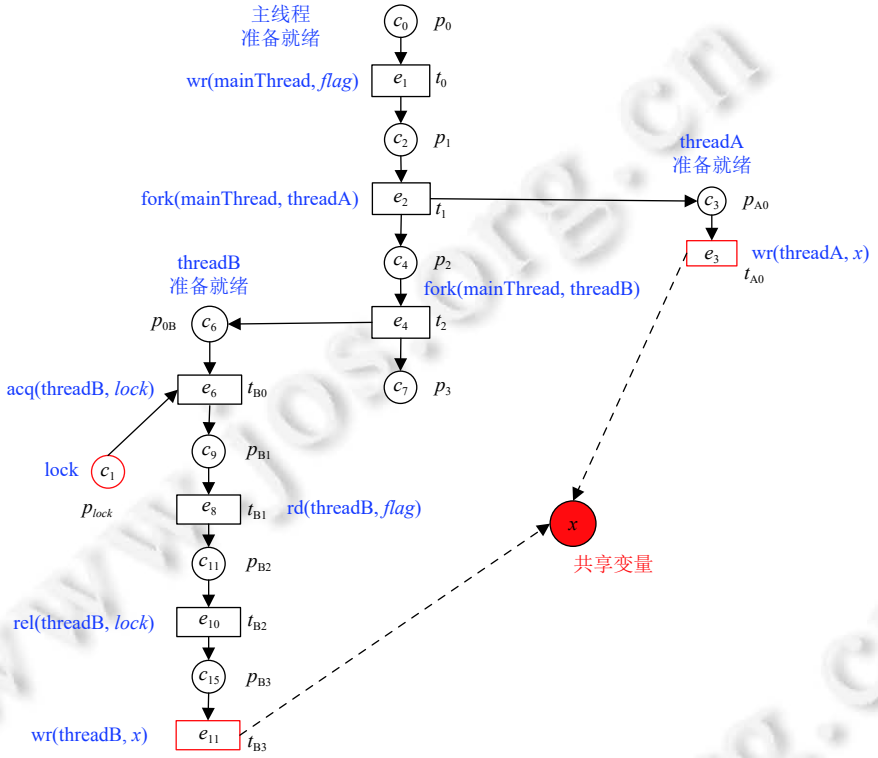


图 4 Program 1 中潜在数据竞争所伴随的 Petri 网展开前缀

4 基于网展开前缀的数据竞争重演

第 2 节所定义的程序运行轨迹忽略了很多可能影响数据竞争出现与否的同步原语, 比如 wait、notify、suspend、LockSupport.park 等, 这有可能导致前文所得 Petri 网模型的可执行变迁序列中隐含着原始程序中无法执行的操作序列, 进而会导致一些数据竞争的误报. 幸运的是, 上节所得 Petri 网展开前缀隐含了各个潜在数据竞争的产生根源和具体发生路径, 本节将据此给出控制源程序运行的一种确定性调度方法, 以此重演潜在的数据竞争, 重演成功的数据竞争方为真实的并发漏洞.

为此, 首先对上节所得潜在数据竞争对应的 Petri 网展开前缀进行分析, 计算其中出现的各个锁对象的授权线程序列. 具体见算法 2, 基本思想是, 展开前缀中对关联着一个锁对象且无前驱的条件库所必然对应着这个锁对象的初始可用状态, 找到这些条件库所后, 只需沿该条件库所的后继节点追踪这个锁对象的使用线程, 一旦持有该锁对象的线程发生一次改变则将该线程 ID 追加到锁对象的授权线程序列中即可, 具体过程见算法 1. 算法 2 的时间复杂度为 $O(|Locks|\times(|B'|+|E'|))$, 当中 $|Locks|$ 为数据竞争伴随 Petri 网展开前缀中出现的锁对象数量, $|B'|$ 为前缀中条件库所的个数, $|E'|$ 为前缀中事件的个数.

以图 4 所示的展开前缀为例, 当中仅仅出现了一个锁对象 *lock*, 而且该对象仅在执行变迁 t_{B0} 时被授权给了线程 threadB, 故 *lock* 的授权线程序列为 $\delta(lock) =<$ threadB $>$.

算法 2. 由潜在数据竞争伴随的 Petri 网展开前缀计算各个锁对象的授权线程序列.

输入: 某潜在数据竞争伴随的 Petri 网展开前缀 $DR\_ON = (B', E'; G', h')$;
输出: 展开前缀中出现的锁对象的集合 *LockSet*, 以及一个为其中每个锁对象指定一个授权线程序列的函数 δ.

步骤:

1. 在 *DR_ON* 中寻找前驱节点为空且 $h'(x)$ 对应一个锁对象的条件库所 x, 计入集合 *C_Lock*;
2. **FOREACH** ($c \in C\_Lock$) {
3. 设 o 为 $h'(c)$ 所对应的锁对象的 ID;
4. 初始化 $\delta(o)$ 为一个空序列;
5. **WHILE** (c 在 *DR_ON* 中存在后继节点){
6. 记 e 为 c 的后继事件, 并记 $\#_{rmThreadID}(e)$ 为 $h'(e)$ 所对应操作隶属的线程 ID;
7. **IF**($h'(e)$ 是一个针对锁对象 o 的锁获取操作){
8. 令 $\delta(o) := \delta(o) \circ \#_{\text{ThreadID}}(e)$, 其中 $\circ$ 表示序列的拼接操作;
9. 重置 c 为 e 的后继条件库所 (该库所是唯一的, 因为每个锁获取操作后只有一个控制流库所为其后继);
10. }
11. **ELSEIF** ($h'(e)$ 是一个针对锁对象 o 的锁释放操作)
12. 此时 $h'(e)$ 会有一个控制流状态库所和一个资源库所为其后继条件, 令 c 为资源库所对应的条件;
13. **ELSEIF** ($h'(e)$ 是一个线程的 fork、join 或 stop 操作, 或者是某共享变量的读写操作)
14. 此时 $h'(e)$ 的后继库所中存在一个库所描述线程 $\#_{\text{ThreadID}}(e)$ 的控制流状态, 重置 c 为该库所对应的条件;
15. } //WHILE
16. 将锁对象 o 并入集合 *LockSet*, 上述过程得到的 $\delta(o)$ 即为其授权线程序列;
17. }//FOREACH
18. **RETURN** *LockSet* 与函数 δ.

此外, 基于数据竞争伴随的 Petri 网展开前缀, 从主线程就绪态对应的条件库所出发, 遍历整个前缀, 易得潜在数据竞争的发生位置 (即哪 2 个线程第几次读/写哪个共享变量时发生了冲突). 以图 4 为例, 潜在数据竞争对应的是事件 e_3 与 e_{11} 对共享变量 x 的写操作冲突, 遍历前缀不难发现它们分别对应线程 threadA 对 x 的第 m=1 次写操作、线程 threadB 对 x 的第 n=1 次写操作.

得到各个锁对象的授权线程序列 δ, 以及数据竞争的位置信息 (假设潜在数据竞争由线程 t_1 第 m 次对共享变量 x 的读/写操作、线程 t_2 第 n 次对 x 的读/写操作构成) 后, 基于 CalFuzzer[32]或其他的多线程程序主动调试平台[33,34]再次运行原程序, 运行过程中按如下调度规则干预其执行过程.

● 调度规则 Rule 1. 每当有线程 t 尝试执行锁对象 *lock* 的获取操作时, 分如下 3 种情况.

(1) 若 *lock* 的授权线程序列 $\delta(lock)$ 当前非空、且尝试获取 *lock* 的线程 t 不是 $\delta(lock)$ 的首元素, 则在 CalFuzzer 提供的 lockBefore 函数接口中阻塞线程 t, 将其加入 *lock* 关联的一个阻塞线程池.

(2) 若 *lock* 的授权线程序列 $\delta(lock)$ 当前非空、且尝试获取 lock 的线程 t 是 $\delta(lock)$ 的首元素, 则直接执行该锁授权操作, 并在 CalFuzzer 提供的 lockAfter 函数接口中删除 $\delta(lock)$ 的首元素, 再通过 Calfuzzer 的 unblock 函数唤醒之前因为访问 *lock* 访问而被阻塞的各个线程.

(3) 若 *lock* 的授权线程序列 $\delta(lock)$ 当前为空, 则在 CalFuzzer 提供的 lockBefore 函数接口中阻塞尝试获取锁 *lock* 的线程.

以程序 Program 1 为例, 按照图 4 中展开前缀导出的锁授权线程序列为 $\delta(lock) =<$ threadB $>$. 重演阶段调度

Program 1 的运行时, 若 threadA 首先尝试获取 *lock*, 则应通过 block 函数阻塞线程 threadA; 若首次尝试获取 *lock* 的线程是 threadB, 则将锁 *lock* 授权给 threadB, 并删除 $\delta(lock)$ 的首元素、更新其为空; 若 threadA 首先尝试获取 *lock* 被阻塞, 而 threadB 之后尝试获取锁 *lock*, 则将 *lock* 授权给 threadB 并删除 $\delta(lock)$ 的首元素后, 还应唤醒被阻塞的线程 threadA.

● 调度规则 Rule 2. 当线程 t_1 尝试执行潜在数据竞争对应的对共享变量 x 的第 m 次读/写时, 在 CalFuzzer 提供的 writeBefore 函数接口中阻塞该线程, 同时设置变量 t_1_x 为 true; 类似地, 当 t_2 尝试执行潜在数据竞争对应的对共享变量 x 的第 n 次读/写时, 在 CalFuzzer 提供的 writeBefore 函数接口中也阻塞该线程, 同时设置变量 t_2_x 为 true. 一旦 t_1_x 和 t_2_x 均为 true, 则说明这 2 个共享变量的读/写操作在 Calfuzzer 的调度下同时处于了可执行状态, 数据竞争重演成功.

以前文 Program 1 为例, 假设基于 Calfuzzer 对该程序进行调度运行时, 原始程序仍然尝试按照表 1 的轨迹运行, 按照上述调度策略则会执行表 4 所示的各种调度措施, 最终会重演出我们想要的数据竞争缺陷. 需要指出的是, 这一并发缺陷按照表 1 的原始轨迹运行时不会出现, 而且 FastTrack[15]、VerifiedFT[17]等传统的动态分析方法也无法检测到该缺陷.

表 4　程序 Program 1 的一个数据竞争重演过程

序号	主线程	线程A	线程B	操作执行前后CalFuzzer的干预措施
1	6: wr(mainThread, *flag*)			writeBefore: 无干预 writeAfter: 无干预
2	27: fork(mainThread, threadA)			startBefore: 无干预
3	28: fork(mainThread, threadB)			startBefore: 无干预
4		9: wr(threadA, *x*)		writeBefore: (1) 阻塞threadA; (2) 设置threadA_ *x*为true; writeAfter: 无干预
5			18: acq(threadB, *lock*)	lockBefore: 无干预 lockAfter: 删除 $\delta(lock)$ 的首元素
6			19: rd(threadB, *flag*)	readBefore: 无干预 readAfter: 无干预
7			20: rel(threadB, *lock*)	unlockAfter: 无干预
8			22: wr(threadB, *x*)	writeBefore: (1) 阻塞threadB; (2) 设置threadB_ *x*为true; (3) 因threadA_ *x*与threadB_ *x*同时为true, 输出e_{11}与e_3所对应的程序第9行与第22行代码存在数据竞争

最后, 需要说明的是, 本文所提数据竞争检测和重演方法是一个两阶段的数据竞争分析方法, 需要运行源程序 2 次. 第 1 阶段, 通过 CalFuzzer[32]或 RoadRunner[33]等程序运行跟踪平台捕获程序首次运行时产生的运行轨迹, 构造本次运行对应的程序 Petri 网模型, 并借助 Petri 网的展开分析程序中潜在的数据竞争错误, 获取各个潜在数据竞争对应的锁对象授权线程序列以及数据竞争的位置信息; 第 2 阶段, 通过 CalFuzzer 或其他程序主动调试平台, 在锁的授权操作或共享变量的读写操作前后, 执行特定的干预操作, 以此实现数据竞争的重演. 然而, 众所周知, 即使是同一个程序中同一个线程、同一个锁或者共享变量对象, 它们在两次不同的运行过程中可能会分配不同的 ID. 按照本节方法进行数据竞争重演的前提是要在同一对象的不同 ID 之间建立映射关系. 解决方案如下: (1) 就线程 ID 的对应而言, 两次运行中的第一个启动的线程必然是主线程, 由此可建立主线程之间的 ID 对应关系. 进而, 即使在两次不同的程序运行中, 主线程依次启动的子线程序列是一致的, 由此可以在这些子线程 ID 之间建立一一映射关系. 类似的道理, 根据各个子线程启动的线程序列又可建立其他线程 ID 之间的对应关系, 最终可

完成线程 ID 之间的一一对应; (2) 由于每个线程依次申请的锁对象序列是一致的, 据此可以在建立锁对象 ID 之间的对应关系; (3) 每个线程依次访问的共享变量序列也是一致的, 据此可以得到共享变量 ID 之间的一一映射关系.

5 原型系统与实验评估

5.1 原型系统

基于 Java 多线程程序主动调试的开源平台 CalFuzzer[32]研发了相应的数据竞争动态检测和重演工具. 基于该工具, 用户只需输入一个多线程程序, 工具便会调用 CalFuzzer 的程序事件流自动生成功能捕获程序的运行轨迹, 之后根据捕获的轨迹构建程序的 Petri 网模型, 再基于 Petri 网展开算法对程序 Petri 网模型中蕴含的潜在数据竞争进行检测, 对于每个潜在数据竞争生成相应的 Petri 网展开前缀; 最后, 根据各个潜在数据竞争对应的 Petri 网展开前缀, 生成锁对象的授权线程序列用作程序运行的调度方案, 进而, 调用 CalFuzzer 平台提供的 writeBefore、writeAfter、readBefore、readAfter、startBefore、startAfter、lockBefore、lockAfter、unlockAfter 等函数进行程序执行的调度和数据竞争的重演. 所开发工具的架构和运行界面分别见图 5、图 6 所示.

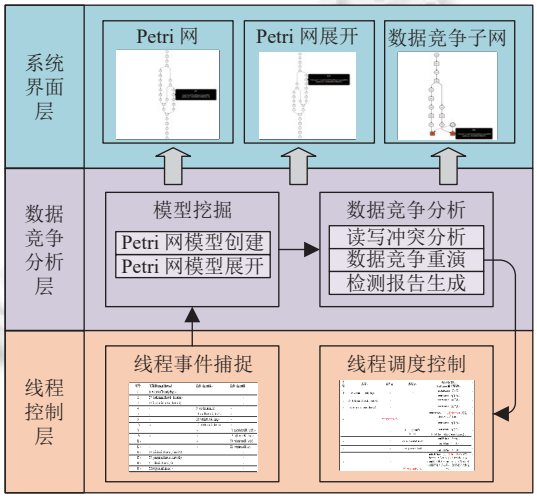
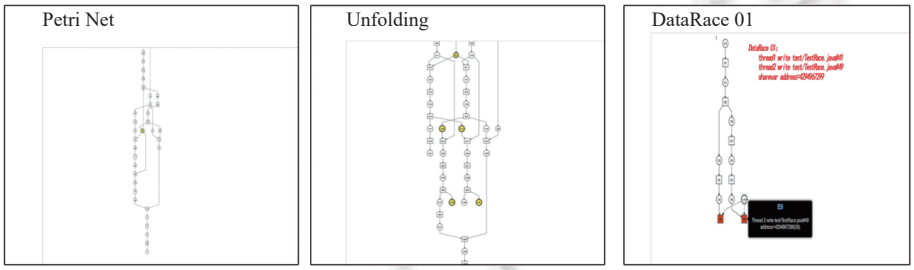


图 5 数据竞争检测与重演原型系统架构



(a) Petri网 (b) 展开 (c) 一对数据竞争

图 6 原型系统运行界面截图

原型系统架构分为线程控制层、数据分析层、结果界面层 3 部分 (如图 5 所示). 最底层的线程控制层基于 CalFuzzer 内置 API 实现: 在线程事件捕捉模块中, 使用 SOOT 分析并修改程序的字节码, 在与并发相关的语句处插入回调函数, 在一次真实执行中捕捉多线程程序运行时的相关动作; 在线程调度控制模块中, 针对每个潜在的数据竞争, 根据上层数据竞争分析层得到的锁授权线程序列进行程序调度以重演数据竞争. 中间层为数据竞争分析层, 模型挖掘模块依据程序运行轨迹建立程序的 Petri 网模型, 并基于算法 1 对程序的 Petri 网模型进行展开; 数据

竞争分析模块根据程序 Petri 网模型的展开分析读写冲突、检测潜在数据竞争, 为每一个潜在数据竞争生成一个展开前缀, 并据此计算锁对象的授权线程序列用作数据竞争重演的程序调度方案; 最终, 结合重演结果生成数据竞争分析报告. 最上层为系统界面层, 向用户展示挖掘得到的程序 Petri 网模型、Petri 网模型的展开、所检测到的数据竞争对应的 Petri 网展开前缀, 同时给出数据竞争报告 (如图 6 所示).

图 6 给出的是所开发原型系统的部分界面截图, 图 6(a) 是根据程序运行轨迹挖掘和构建的多线程程序 Petri 网模型, 图 6(b) 为 Petri 网展开的界面, 图 6(c) 展示了一个潜在数据竞争对应的 Petri 网展开前缀和数据竞争检测结果, 图 6(c) 中的提示信息 Thread 1 write test/TestRace.java#41 address=4294967299(22) 表示代码 TestRace.java 中在 41 行 22 列线程 1 中地址为 4294967299 的共享变量 x 的写操作, 与代码 49 行 26 列线程 2 中该共享变量的写操作存在数据竞争.

5.2 实验评估与相关工作比较

就相关工作比较而言, 第 2 节结合程序 Program 1 从原理上说明了本文方法相比 VerifiedFT 等经典动态分析方法的优势, 即本文方法虽然是从单一运行轨迹挖掘得到程序的 Petri 网模型, 但是这个模型可以隐含多条潜在的程序运行轨迹, 从而可以减少数据竞争的漏报现象. 与此同时, 由于动态分析方法均依赖程序运行轨迹进行漏洞检测, 而完整的覆盖各种可能的程序执行路径难以获得, 丢失的路径中可能隐含着数据竞争, 故包括本文方法在内的数据竞争检测均可能存在一定漏报, 具体漏报率依赖于测试用例对程序执行路径的覆盖情况. 除此之外, 结合 CalFuzzer 软件平台[12]中公开的多个 Java 多线程程序实例进行了实验评估和对比, 实验对比结果见表 5.

表 5 数据竞争检测试验结果

程序实例	实际数据竞争数量	本文方法的检测结果	Calfuzzer集成方法HybridAnalysis+Racefuzzer的检测结果	本文方法运行时间 (s)	CalFuzzer集成方法HybridAnalysis+Racefuzzer的单次运行时间 (s)
Test1	3	无漏报和误报	无漏报和误报	4	2.2
Test2	0	无漏报和误报	无漏报和误报	3.1	1.1
Test3	3	无漏报和误报	有一定概率发生漏报 (单次运行概率大, 运行次数增加概率逐渐变小), 无误报	3.2	2.1
Test4	6	无漏报和误报	有一定概率发生漏报(单次运行概率大, 运行次数增加概率逐渐变小), 无误报	6	4.2
Test5	0	无漏报和误报	无漏报和误报	3.1	1.3
Test6	3	无漏报和误报	有一定概率发生漏报(单次运行概率大, 运行次数增加概率逐渐变小), 无误报	4	2.1
Test7	0	无漏报和误报	无漏报和误报	3	1.3
Test8	0	无漏报和误报	无漏报和误报	10	3
Test9	0	无漏报和误报	无漏报和误报	6.4	2.1
Test10	3	无漏报和误报	有一定概率发生漏报(单次运行概率大, 运行次数增加概率逐渐变小), 无误报	5.4	3
Test11	3	无漏报和误报	有一定概率发生漏报(单次运行概率大, 运行次数增加概率逐渐变小), 无误报	4	2.3
Test12	2	无漏报和误报	有一定概率发生漏报(单次运行概率大, 运行次数增加概率逐渐变小), 无误报	10	11
本文实例Program1	2	有一定概率发生一个漏报(运行次数增加概率逐渐变小). 仅以表1中的轨迹为依据发生1个漏报. 无误报	有一定概率发生两个或一个漏报(单次运行概率大, 运行次数增加概率逐渐变小). 仅以表1中的轨迹为输入发生2个漏报. 无误报	3.1	2.7

对于程序实例 Test1, 由于程序中不存在锁对象或锁的竞争, 任意一条程序运行轨迹中均能呈现出程序中存在的数据竞争漏洞, 故本文方法和 CalFuzzer 中集成的基于 HybridAnalysis 与 RaceFuzzer 的数据检测方法均能完整、准确的检测出程序的数据竞争缺陷; 对于程序实例 Test3–6、Test10–12, 程序中存在锁对象的竞争等复杂的并发

操作与程序行为, 有的运行轨迹能呈现出程序中的数据竞争缺陷, 而有的运行轨迹则无法直接呈现. 此时, CalFuzzer 中集成的方法通常需要多次运行、分析多条执行轨迹方能完整的检测出程序中存在的数据竞争 (实验表明, 单次运行通常会出现漏报, 运行 3 次以上通常能检测出所有的数据竞争), 而本文所提方法只需基于单条运行轨迹就可准确作出检测, 没有任何漏报现象; 其余程序实例本身不存在数据竞争缺陷, 所以两类方法均无漏报. 此外, 由于本文方法和 CalFuzzer 集成的方法均进行了数据竞争重演, 故所有的程序实例均没有误报.

从时间性能来看, 本文所提数据竞争检测方法的时间消耗大部分情况下高于 CalFuzzer 中集成的方法, 原因在于本文挖掘到的 Petri 网模型可能蕴含有多条运行轨迹, 对该 Petri 网进行展开时通常会消耗更多的时间. 而且, 程序中存在的线程数量和并发行为越多, 本文方法消耗的时间越多. 为进行时间测试, 设计了不同规模的多个多线程程序实例进行了性能测试. 具体而言, 每个测试程序设置两个共享变量进行自增操作, 设置两把锁分别进行获取和释放, 然后不断增加线程规模, 测试结果见表 6. 由测试结果可见, 当线程数目较少时, 我们的方法具有比较好的性能表现, 但当线程数量增加到 10 个以上时, 运行时间会以指数形式增加, 这主要是由于 Petri 网展开规模的快速增长. 不过, 这种时间上的损失带来的好处是漏报现象的减少.

表 6 Petri 网展开的性能测试结果

线程数	库所数	变迁数	条件数	事件数	截断事件数	运行时间 (ms)
2	16	14	20	15	0	6
4	42	38	142	113	5	101
6	68	62	821	654	17	250
8	94	86	86	3490	321	2293
10	120	110	22176	17557	1793	13988
12	146	134	107556	85015	9217	216820

在对大规模并发程序进行数据竞争检测时, 为提高效率, 一个可能的方法是先基于 lockset 等效率高但是准确性不足的方法定位潜在的数据竞争, 然后, 以每组潜在的数据竞争为对象, 有针对性的进行程序 Petri 网模型的展开和数据竞争检测. 实际上, 文献 [26] 提出的 Petri 网反向展开算法可用于解决此问题. 不过, 文献 [26] 将 Petri 网的反向展开用于程序的静态分析, 它根据程序源码构造程序的 Petri 网模型、进而借助反向展开检测数据竞争. 但是, 当中需要用户人为给出一组可能存在竞争的共享变量读写操作, 之后进行针对性的检测. 而且, 静态分析方法在大规模程序分析时同样存在效率问题, 误报率较高. 后续工作中, 作者将尝试将本文方法与文献 [26] 中的方法进行结合, 在提高效率的同时降低误报率.

最后, 传统的数据竞争检测方法只会指出存在数据竞争的语句位置, 而本文方法所得数据竞争锁伴随的 Petri 网展开前缀可以直观展现这种缺陷的发生场景. 而且, 根据这种 Petri 网展开前缀可以有效地进行思索的重演, 以此保证潜在数据竞争的真实性. 不过, 本文方法也存在很多不足, 比如, 本文未考虑 wait、notify 等对数据竞争检测可能带来影响的其他同步原语.

6 总结与分析

本文提出了一种基于 Petri 网展开的多线程程序数据竞争检测和重演方法, 其主要的检测步骤和特色如下: 首先, 通过对程序某次运行轨迹的分析构建程序的 Petri 网模型, 该模型虽由单一运行轨迹挖掘得到, 但可能隐含着多个不同的程序可执行操作序列, 从而可降低传统数据竞争动态检测方法的漏报率; 之后, 将程序的数据竞争检测问题转换为 Petri 网模型中共享变量读写/写写操作的并发关系识别问题, 并借助 Petri 网展开技术完成了程序中潜在数据竞争的检测; 最后, 基于各个潜在数据竞争伴随的 Petri 网展开前缀给出了潜在数据竞争的触发路径, 并以此为基础, 借助 CalFuzzer 平台实现了数据竞争的重演, 保证了数据竞争检测真实性. 此外, 相比数据竞争的静态检测方法而言, 本文仅分析单一运行轨迹对应的 Petri 网模型, 这有效缩减了问题的复杂度, 而且能保证数据竞争检测的真实性. 相比传统的数据竞争动态检测方法而言, 本文从单一运行轨迹挖掘得到的 Petri 网模型可以蕴含多条

不同的程序运行轨迹, 可以减少数据竞争的漏报现象.

不过, 目前所提方法仍然存在诸多不足. 比如, 本文未考虑 wait、notify 等对数据竞争检测可能带来影响的其他同步原语, 对大规模程序进行分析时的性能有待优化等. 此外, 在进行数据竞争重演的过程中, 程序在如下两种情况下可能出现死锁: 其一, 被检测程序本身存在死锁, 在重演过程中该死锁被触发了; 其二, 被检测程序本身不存在死锁, 但是拟重演的潜在数据竞争是一个误报, 此时本文给出的程序调度和重演算法可能会导致死锁. 上述两种情况均会导致数据竞争重演的失败, 使得算法无法判定潜在数据竞争的真伪. 针对这一问题, 可以在重演的过程中借助死锁的动态检测方法[35,36]进行死锁分析, 一旦检测到死锁的发生, 则对当时的锁图和相关的程序运行状态进行分析, 分析死锁产生的根源, 据此开展后续的程序漏洞修复或者开展进一步的数据竞争分析工作, 相关研究将在后续工作中展开.

References:

[1] Su XH, Yu Z, Wang TT, Ma PJ. A survey on exposing, detecting and avoiding concurrency bugs. Chinese Journal of Computers, 2015, 38(11): 2215–2233 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2015.02215]

[2] Guan SP, Wang L. Uncertainty analysis of clouded control system with its controller design. Acta Automatica Sinica, 2022, 48(11): 2677–2687 (in Chinese with English abstract). [doi: 10.16383/j.aas.c190529]

[3] Wang CL, Tao YG, Yang P, Liu ZJ, Zhou Y. Parallel task assignment optimization algorithm and parallel control for cloud control systems. Acta Automatica Sinica, 2017, 43(11): 1973–1983 (in Chinese with English abstract). [doi: 10.16383/j.aas.2017.c160504]

[4] Netaer RHB, Miller BP. What are race conditions? Some issues and formalizations. ACM Letters on Programming Languages and Systems, 1992, 1(1): 74–88. [doi: 10.1145/130616.130623]

[5] Yu Z, Yang Z, Su XH, Wang TT. A survey on methods of data race detection and verification on multithreaded program. Intelligent Computer and Applications, 2017, 7(3): 123–126, 129 (in Chinese with English abstract). [doi: 10.3969/j.issn.2095-2163.2017.03.034]

[6] Engler D, Ashcraft K. RacerX: Effective, static detection of race conditions and deadlocks. In: Proc. of the 19th ACM Symp. on Operating Systems Principles. Bolton Landing: ACM, 2003. 237–252. [doi: 10.1145/945445.945468]

[7] Voung JW, Jhala R, Lerner S. RELAY: Static race detection on millions of lines of code. In: Proc. of the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. Dubrovnik: ACM, 2007. 205–214. [doi: 10.1145/1287624.1287654]

[8] Pratikakis P, Foster JS, Hicks M. LOCKSMITH: Context-sensitive correlation analysis for race detection. ACM SIGPLAN Notices, 2006, 41(6): 320–331. [doi: 10.1145/1133255.1134019]

[9] Savage S, Burrows M, Nelson G, Sobalvarro PG, Anderson TE. Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. on Computer Systems, 1997, 15(4): 391–411. [doi: 10.1145/265924.265927]

[10] Von Praun C, Gross TR. Object race detection. ACM SIGPLAN Notices, 2001, 36(11): 70–82. [doi: 10.1145/504311.504288]

[11] Elmas T, Qadeer S, Tasiran S. Goldilocks: Efficiently computing the happens-before relation using locksets. In: Proc. of the 1st Combined Int'l Workshops on Formal Approaches to Software Testing and Runtime Verification. Seattle: Springer, 2006. 193–208. [doi: 10.1007/11940197_13]

[12] Netzer RHB. Race condition detection for debugging shared-memory parallel programs [Ph.D. Thesis]. Madison: University of Wisconsin-Madison, 1991.

[13] Perkovic D, Keleher PJ. Online data-race detection via coherency guarantees. ACM SIGOPS Operating Systems Review, 1996, 30(SI): 47–57. [doi: 10.1145/248155.238760]

[14] ItzkovitzA, Schuster A, Zeev-Ben-Mordehai O. Toward integration of data race detection in DSM systems. Journal of Parallel & Distributed Computing, 1999, 59(2): 180–203. [doi: 10.1006/jpdc.1999.1574]

[15] Flanagan C, Freund SN. FastTrack: Efficient and precise dynamic race detection. ACM SIGPLAN Notices, 2009, 44(6): 121–133. [doi: 10.1145/1543135.1542490]

[16] Ha OK, Jun YK. An efficient algorithm for on-the-fly data race detection using an epoch-based technique. Scientific Programming, 2015, 2015: 13. [doi: 10.1155/2015/205827]

[17] Wilcox JR, Flanagan C, Freund SN. $V_{ERIFIED}$FT: A verified, high-performance precise dynamic race detector. ACM SIGPLAN Notices, 2018, 53(1): 354–367. [doi: 10.1145/3200691.3178514]

[18] 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. [doi: 10.1002/cpe.1064]

[19] Xie XW, Xue JL, Zhang J. $A_{CCULOCK}$: Accurate and efficient detection of data races. Software: Practice and Experience, 2013, 43(5): 543–576. [doi: 10.1002/spe.2121]

[20] Yu MS, Bae DH. Simplelock^{+}: Fast and accurate hybrid data race detection. The Computer Journal, 2016, 59(6): 793–809. [doi: 10.1093/comjnl/bxu119]

[21] Sen K. Race directed random testing of concurrent programs. In: Proc. of the 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Tucson: ACM, 2008. 11–21. [doi: 10.1145/1375581.1375584]

[22] Wu ZD, Lu K, Wang XP, Zhou X. Collaborative technique for concurrency bug detection. Int'l Journal of Parallel Programming, 2015, 43(2): 260–285. [doi: 10.1007/s10766-014-0304-y]

[23] Lu K, Wu ZD, Wang ZP, Chen C, Zhou X. RaceChecker: Efficient identification of harmful data races. In: Proc. of the 23rd Euromicro Int'l Conf. on Parallel, Distributed, and Network-based Processing. Turku: IEEE, 2015. 78–85. [doi: 10.1109/PDP.2015.19]

[24] McMillan KL. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: von Bochmann G, Probst, eds. Computer Aided Verification. Berlin, Heidelberg: Springer, 1992. 164–177. [doi: 10.1007/3-540-56496-9_14]

[25] Yuan CY. Petri Net Application. Beijing: Science Press, 2013 (in Chinese).

[26] Hao ZY, Lu FM. Reverse unfolding of Petri nets and its application in program data race detection. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1612–1630 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6240.htm [doi: 10.13328/j.cnki.jos.006240]

[27] Pang SC, Jiang CJ, Sun P, Zhou CH. Property analysis of shared composition Petri nets. Acta Automatica Sinica, 2004, 30(6): 944–948 (in Chinese with English abstract). [doi: 10.16383/j.aas.2004.06.019]

[28] You D, Wang SG, Zhou MC. Siphon controllability definitions and issues. Acta Automatica Sinica, 2014, 40(12): 2687–2696 (in Chinese with English abstract). [doi: 10.3724/SP.J.1004.2014.02687]

[29] Reising W. Petri Nets—An Introduction. Springer Science & Business Media, 2012.

[30] Esparza J, Römer S, Vogler W. An improvement of McMillan' unfolding algorithm. In: Proc. of the 2nd Int'l Workshop on Tools and Algorithms for the Construction and Analysis of Systems. Passau: Springer, 1996. 87–106. [doi: 10.1007/3-540-61042-1_40]

[31] Schimmel J, Molitorisz K, Jannesari A, Tichy WF. Combining unit tests for data race detection. In: Proc. of the 10th IEEE/ACM Int'l Workshop on Automation of Software Test. Florence: IEEE, 2015. 43–47. [doi: 10.1109/AST.2015.16]

[32] Joshi P, Naik M, Park CS, Sen K. CALFUZZER: An extensible active testing framework for concurrent programs. In: Proc. of the 21st Int'l Conf. on Computer Aided Verification. Grenoble: Springer, 2009. 675–681. [doi: 10.1007/978-3-642-02658-4_54]

[33] Flanagan C, Freund SN. The RoadRunner dynamic analysis framework for concurrent programs. In: Proc. of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. Toronto: ACM, 2010. 1–8. [doi: 10.1145/1806672.1806674]

[34] Valléerai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot: A Java bytecode optimization framework. In: Proc. of the 1st CASCON Decade High Impact Papers. Toronto: IBM Corp., 2010. 214–224. [doi: 10.1145/1925805.1925818]

[35] Lu FM, Zheng JJ, Bao YX, Zeng QT, Duan H, Wang XY. Deadlock detection of multithreaded programs based on lock-augmented segmentation graph. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1682–1700 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6244.htm [doi: 10.13328/j.cnki.jos.006244]

[36] Lu FM, Cui MH, Bao YX, Zeng QT, Duan H. Deadlock detection method based on petri net mining of program trajectory. Computer Integrated Manufacturing Systems, 2021, 27(9): 2611–2624 (in Chinese with English abstract). [doi: 10.13196/j.cims.2021.09.014]

附中文参考文献:

[1] 苏小红, 禹振, 王甜甜, 马培军. 并发缺陷暴露、检测与规避研究综述. 计算机学报, 2015, 38(11): 2215–2233. [doi: 10.11897/SP.J.1016.2015.02215]

[2] 关守平, 王梁. 云控制系统不确定性分析与控制器设计方法. 自动化学报, 2022, 48(11): 2677–2687. [doi: 10.16383/j.aas.c190529]

[3] 王彩璐, 陶跃钢, 杨鹏, 刘作军, 周颖. 云控制系统并行任务分配优化算法与并联控制. 自动化学报, 2017, 43(11): 1973–1983. [doi: 10.16383/j.aas.2017.c160504]

[5] 禹振, 杨振, 苏小红, 王甜甜. 多线程程序数据竞争检测和验证方法研究综述. 智能计算机与应用, 2017, 7(3): 123–126, 129. [doi: 10.3969/j.issn.2095-2163.2017.03.034]

[25] 袁崇义. Petri网应用. 北京: 科学出版社, 2013.

[26] 郝宗寅, 鲁法明. Petri网的反向展开及其在程序数据竞争检测的应用. 软件学报, 2021, 32(6): 1612–1630. http://www.jos.org.cn/1000-9825/6240.htm [doi: 10.13328/j.cnki.jos.006240]

[27] 庞善臣, 蒋昌俊, 孙萍, 周长红. 共享合成Petri网的性质分析. 自动化学报, 2004, 30(6): 944–948. [doi: 10.16383/j.aas.2004.06.019]
[28] 尤丹, 王寿光, 周孟初. 信标可控性定义及问题. 自动化学报, 2014, 40(12): 2687–2696. [doi: 10.3724/SP.J.1004.2014.02687]
[35] 鲁法明, 郑佳静, 包云霞, 曾庆田, 段华, 王晓宇. 基于锁增广分段图的多线程程序死锁检测. 软件学报, 2021, 32(6): 1682–1700. http://www.jos.org.cn/1000-9825/6244.htm [doi: 10.13328/j.cnki.jos.006244]
[36] 鲁法明, 崔明浩, 包云霞, 曾庆田, 段华. 基于程序运行轨迹Petri网模型挖掘的死锁检测方法. 计算机集成制造系统, 2021, 27(9): 2611–2624. [doi: 10.13196/j.cims.2021.09.014]

鲁法明(1981－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为 Petri 网, 并行程序验证, 过程挖掘.

包云霞(1979－), 女, 副教授, 主要研究领域为 Petri 网, 并行程序分析与验证.

黄莹(1998－), 女, 硕士生, CCF 学生会员, 主要研究领域为并行程序验证, 机器学习.

唐梦凡(1998－), 女, 硕士生, 主要研究领域为并行程序验证.

曾庆田(1976－), 男, 博士, 教授, CCF 高级会员, 主要研究领域为 Petri 网, 人工智能.