软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*, 2023,34(8):3485−3506 [doi: 10.13328/j.cnki.jos.006865]

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

# GC-MCR: 有向图约束指导的并发缺陷检测方法*

李硕川[1], 王 赞[1], 马明旭[1], 陈 翔[2], 赵英全[1], 王海弛[1], 王昊宇[1]

[1](天津大学 智能与计算学部, 天津 300350)

[2](南通大学 信息科学技术学院, 江苏 南通 226019)

通信作者: 王赞, E-mail: wangzan@tju.edu.cn

**摘 要**: 约束求解应用到程序分析的多个领域, 在并发程序分析方面也得到了深入的应用. 并发程序随着多核处理器的快速发展而得到广泛使用, 然而并发缺陷对并发程序的安全性和可靠性造成了严重的影响, 因此, 针对并发缺陷的检测尤为重要. 并发程序线程运行的不确定性导致的线程交织爆炸问题, 给并发缺陷的检测带来了一定挑战. 已有并发缺陷检测算法通过约减无效线程交织, 以降低在并发程序状态空间内的探索开销. 比如, 最大因果模型算法把并发程序状态空间的探索问题转换成约束求解问题. 然而, 其在约束构建过程中会产生大量冗余和冲突的约束, 大幅度增加了约束求解的时间以及约束求解器的调用次数, 降低了并发程序状态空间的探索效率.针对上述问题, 提出了一种有向图约束指导的并发缺陷检测方法 GC-MCR (directed graph constraint-guided maximal causality reduction). 该方法旨在通过使用有向图对约束进行过滤和约减, 从而提高约束求解速度, 并进一步提高并发程序状态空间的探索效率. 实验结果表明: GC-MCR 方法构建的有向图可以有效优化约束的表达式,从而提高约束求解器的求解速度并减少求解器的调用次数. 与现有的 J-MCR 方法相比, GC-MCR 的并发程序缺陷检测效率可以取得显著提升, 且不会降低并发缺陷的检测能力, 在现有研究方法广泛使用的 38 组并发测试程序上的测试时间可以平均减少 34.01%.

**关键词**: 并发程序; 最大因果约减; 约束求解; 有向图; 冲突约束过滤

**中图法分类号**: TP311

中文引用格式: 李硕川, 王赞, 马明旭, 陈翔, 赵英全, 王海弛, 王昊宇. GC-MCR: 有向图约束指导的并发缺陷检测方法. 软件学报, 2023, 34(8): 3485–3506. http://www.jos.org.cn/1000-9825/6865.htm

英文引用格式: Li SC, Wang Z, Ma MX, Chen X, Zhao YQ, Wang HC, Wang HY. GC-MCR: Directed Graph Constraint-guided Concurrent Bug Detection Method. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3485–3506 (in Chinese). http://www.jos.org.cn/1000-9825/6865.htm

## GC-MCR: Directed Graph Constraint-guided Concurrent Bug Detection Method

LI Shuo-Chuan[1], WANG Zan[1], MA Ming-Xu[1], CHEN Xiang[2], ZHAO Ying-Quan[1], WANG Hai-Chi[1], WANG Hao-Yu[1]

[1](College of Intelligence and Computing, Tianjin University, Tianjin 300350, China)

[2](School of Information Science and Technology, Nantong University, Nantong 226019, China)

**Abstract**: Constraint solving has been applied to many domains of program analysis, and deeply applied in concurrent program analysis. Concurrent programs are specific domain software that has been widely used with the rapid development of multi-core processors. However, concurrent defects threaten the security and robustness of concurrent programs, thus it is of great importance to test concurrent programs. Due to the non-deterministic thread scheduling, one of the key challenges for concurrent program testing is how to reduce the thread interleaving space with exponential growth. The state-of-the-art approaches (i.e., J-MCR) tackle the challenge through constraint solving. However, it is found that there exist conflicts and redundancies inside constraints (i.e., the conflict of constraint clauses makes constraints unsatisfiable), solving those

* 基金项目: 国家自然科学基金(61872263); 天津市智能制造专项资金(20201180)
本文由“约束求解与定理证明”专题特约编辑蔡少伟研究员、陈振邦教授、王戟研究员、詹博华副研究员、赵永望教授推荐.
收稿时间: 2022-09-04; 修改时间: 2022-10-13; 采用时间: 2022-12-14; jos 在线出版时间: 2022-12-30

unsatisfiable constraints results in lower efficiency. Thus, a directed graph constraint-guided method called GC-MCR (directed graph constraint-guided maximal causality reduction) is proposed. By removing conflictive constraints and simplify redundant constraints, the times of constraint solving are reduced thereby further improving the efficiency. Comparing with state-of-the-art approach J-MCR, GC-MCR reduces the times of constraint solving by 19.36% on average and reduces the testing time on average by 34.01% on 38 concurrent programs.

**Key words**: concurrent program; maximum causality reduction; constraint solving; directed graph; conflict constraint filtering

目前, 约束求解作为符号执行的重要组成部分, 已在软件验证、程序分析等领域得到了广泛应用, 其在并发程序分析领域的贡献尤为突出. 并发程序以计算速度快且资源利用率高为特点, 在当今多核架构广泛应用的硬并发时代, 其应用越发普遍. 从微观角度看, 并发程序的线程调度(即某一时刻每个线程是否运行)是由 CPU 决定的, 而不是由用户决定的[1]. 因此, 在程序默认配置下, 线程的执行顺序具有一定的不确定性. 这种不确定性导致开发者难以预料程序的状态, 使得程序内部易出现并发缺陷, 可能会造成严重事故. 例如, 2012 年, Facebook 上市当天, 纳斯达克的交易程序出现并发缺陷(即由于大量的撤单导致程序一直重新运行并进入死循环, 出现了数据竞争问题), 导致 20 分钟无法开盘, 给 Facebook 造成了巨大损失. 故而在实际项目开发中, 需要对并发程序进行充分的测试, 以避免并发缺陷可能造成的不良影响. 因此, 针对并发软件的自动缺陷检测技术研究具有重大意义, 其研究成果有助于快速有效地检测出各类并发缺陷, 以保证并发软件的质量和可靠性.

近些年来, 针对并发程序的测试在软件工程领域得到了广泛的关注, 并发缺陷检测的相关技术逐渐成为研究热点. 目前, 国内外已经发表多篇与并发缺陷检测相关的综述论文[1−3], 现有研究从不同角度对并发缺陷检测进行了深入探索并取得了丰富的研究成果, 在此基础上, 也开发了一些成熟的并发检测工具. 目前常见的并发缺陷可以分为 4 种: 死锁[4]、数据竞争[5]、原子性违背[6]和顺序违背[7]. 为检测这 4 种并发缺陷, 研究人员提出了动态分析[8]、静态分析[9]以及动静结合分析[10]这 3 类方法. 其中,

- 动态分析通过动态执行被测程序, 旨在触发程序异常, 且触发的异常问题都是真实存在的;
- 静态分析基于缺陷模式的规则, 提取和对比分析代码中的模式以检测缺陷[11]. 符号执行作为静态分析的主要方式, 通过对符号化的程序路径约束进行约束求解, 探索程序状态空间;
- 动静结合分析则充分利用动态分析和静态分析的优点, 即动态分析可以提升静态分析可靠性, 静态分析可以降低动态分析的验证负载[2,3].

并发程序的测试和验证面临的主要挑战是线程交织(thread interleaving)空间爆炸问题, 即: 可能的线程交织数量随着线程数量和程序执行时间的增加而呈指数级增长, 使并发缺陷难以被检测. 解决线程交织爆炸的关键是识别冗余的线程交织[12]. 为解决这一问题, 研究人员提出了偏序约减方法(partial order reduction, POR)[13−15], 即: 通过检查程序各个状态和行为间的独立性, 以减小整体状态空间规模. 首先, POR 会将所有线程交织划分为不同的 Mazurkiewicz 轨迹[16], 即所有线程交织被分成的不同等价类; 然后, POR 从每个 Mazurkiewicz 轨迹中选取一个线程交织进行探索. 由于 Mazurkiewicz 轨迹基于 happens-before 关系[17], 且 happens-before 关系依赖于所有锁的释放和获取以及冲突的写事件和读事件, POR 在识别冗余线程的交织方面存在局限性, 即仅符合 happens-before 关系的线程交织中大部分线程交织存在冗余[18]. 针对这一问题, Huang 等人提出了最大因果约减算法(maximal causality reduction, MCR)[18]. MCR 基于 Mazurkiewicz 轨迹制定了新标准: 最大因果关系(maximal causality relation), 即根据读和写事件的值划分 Mazurkiewicz 轨迹中的最大可能等效线程交织集合. MCR 将探索并发程序状态空间的问题转换成约束求解问题, 稳定高效地检测并发缺陷. 然而, 异常庞大的并发程序状态空间会导致 MCR 为事件添加的约束极其严格且复杂, 复杂的约束构建易产生大量冗余甚至存在冲突的约束, 这些约束会导致约束求解耗时较长. 据实验统计, MCR 在约束求解上的耗时约占 MCR 运行总时间的 89.21%, 而冲突的约束在求解过程中甚至存在不可解的问题并消耗了大量时间, 大幅度降低并发缺陷的检测效率.

针对上述问题, 本文以最大因果约减算法中的约束为研究对象, 通过过滤冲突约束并约减冗余约束, 以提高并发缺陷检测的效率, 最终设计了一种有向图约束指导的并发缺陷检测方法 GC-MCR (directed graph

constraint-guided maximal causality reduction). GC-MCR 首先根据程序基本约束关系(即 happens-before 约束、锁互斥约束等)来构建有向图; 然后, 对基于最大因果约减算法[18]生成的读写约束进行解析与拆分, 即将读写约束构建成读写约束树, 根据程序基本约束关系的有向图, 获得读写约束树中每个节点的值, 该值表示以当前节点为根节点的子树是否与程序基本约束冲突; 最后, 根据读写约束树中每个节点的值, 判断是否过滤或约减约束. GC-MCR 通过对冲突约束进行过滤, 并约减冗余约束的表达式, 有效降低单次约束的求解时间甚至减少约束求解器的调用次数, 因此能够保证并发缺陷的检测能力, 并提高并发缺陷效率.

本文选择在已有研究工作中广泛使用的 38 组并发程序作为实验对象, 这 38 组并发程序包含的缺陷类型覆盖了死锁、数据竞争、原子性违背、顺序违背等. 实验结果表明: GC-MCR 方法无论是在约束求解器的调用次数以及单次调用时间方面, 还是从探索并发程序状态空间的效率方面, 都显著优于现有方法, 与 J-MCR 相比, 在总的调度数量没有损失的情况下, 即并发缺陷的检测能力没有任何损失, 程序的约束求解次数和总调用时间可平均减少 34.01%. 本文的主要贡献总结如下.

- 基于有向图设计并发约束的表示方法, 从有向图结构上识别出潜在的不可解并发约束, 优化冗余并发约束, 提高并发约束表示的准确性;
- 提出一种新的有向图约束指导的并发缺陷检测方法 GC-MCR. GC-MCR 基于本文提出的有向图表示方法构建完整的并发程序约束, 并对其进行过滤和约减, 可以有效减少约束求解成本, 从而提高探索并发程序状态空间的效率;
- 实现并开源了 GC-MCR (https://github.com/WingedVampires/GC-MCR.git), 并在 38 个并发测试程序上进行了验证. 实验结果表明: 相比于现有研究方法, GC-MCR 可以更快地找出并发程序中的缺陷, 其平均性能可提升 34.01%.

## 1 背景知识

本文的主要目标是通过有向图表示的方法构建完整的并发程序约束, 并对其进行过滤和约减, 减少约束求解成本, 提高探索并发程序状态空间的效率. 与本文方法相关的理论和方法包括: 作为 GC-MCR 的基础模型——最大因果模型(maximal causal model, MCM)[5,19]、在最大因果模型的基础上提出的最大因果约减算法[18]以及约束的构建与求解的工具 Z3[20].

### 1.1 最大因果模型

最大因果模型[5,19]根据给定的程序执行轨迹构建出最大且合理的因果模型, 其包含所有能生成原始轨迹的并发程序所生成的所有轨迹, 即: 假如并发程序 $P$ 能够生成原始轨迹, 则 $P$ 生成的所有轨迹都包含在最大因果模型中. 因此, 最大因果模型可以发现在庞大的状态空间中尚未被执行到的线程交织. 在 MCM 中, 事件(event)是线程在并发对象上执行的原子操作, 其中, 事件的属性包括: 访问一个并发对象的线程 *Thread*、对该并发对象进行的操作类型 *OP*、该事件访问的并发对象 *Target*、对并发对象执行的操作值 *Data*. 对于两个事件 $s_1$ 和 $s_2$, 当且仅当 $s_1$ 和 $s_2$ 所对应的全部属性都相等, 称事件 $s_1$ 和 $s_2$ 相等.

MCM 事件的类型和对应描述如下所示.

- *begin*($t$): 线程 $t$ 中的第 1 个事件;
- *end*($t$): 线程 $t$ 中最后一个事件;
- *read*($t$,$x$,$v$): 线程 $t$ 读到 $x$ 的值为 $v$;
- *write*($t$,$x$,$v$): 线程 $t$ 写入变量 $x$ 的值为 $v$;
- *lock*($t$,$l$): 线程 $t$ 获取锁 $l$;
- *unlock*($t$,$l$): 线程 $t$ 释放锁 $l$;
- *fork*($t$,$t'$): 线程 $t$ 创建新线程 $t'$;
- *join*($t$,$t'$): 阻塞线程 $t$ 直到 $t'$运行结束.

在 MCM 中, 程序的一条执行轨迹(execution trace)可以被抽象为一组事件的序列, 并且规定执行轨迹需要

满足顺序一致性, 即读写一致性、锁互斥和先发生于原则(happens-before). 其中, 读写一致性需要保证读(read)的值是执行轨迹在该读事件之前, 且距离其最近, 且与其访问同一个共享内存地址的写(write)的值. 锁互斥需要保证每个释放(release)锁事件之前都有同一个线程且同一个锁变量的获得(acquire)锁事件与其匹配, 且对每一个 Acquire-Release 锁对都不可与其他访问同一锁变量的锁对产生交织. happens-before 原则需要保证程序按照代码的顺序执行; 对 volatile 修饰的变量进行的写操作要先发生于对其进行的读操作; 对同一个锁的解锁操作要先发生于对其的加锁操作; 如果操作 $A$ 先发生于操作 $B$, 操作 $B$ 先发生于操作 $C$, 那么 $A$ 应该先发生于操作 $C$; 在线程 $A$ 中调用 $B$.*start*(·)之前的所有操作先发生于线程 $B$ 的所有操作; 在线程 $A$ 中调用 $B$.*join*(·)会先执行线程 $B$, 即线程 $B$ 中的所有操作先发生于在线程 $A$ 中的 *join* 后面的所有操作.

在 MCM 中, *feasible*($\tau$)是由能够产生执行轨迹 $\tau$的所有由程序的执行产生的顺序一致性轨迹构成最小的集合, *feasible*($\tau$)也是最大因果模型的最终结果. *feasible*($\tau$)需要满足前缀闭合性(prefix closedness)和局部确定性(local determinism). 前缀闭合性假设事件是独立、不可分割的, 且是根据程序的执行按顺序生成的, 因此, *feasible*($\tau$)必须是前缀闭合的. 局部确定性假设并发操作的执行是由同一线程中的之前发生的事件确定的, 并且可以在它们之后的任意时刻发生.

### 1.2 最大因果约减算法

最大因果约减算法通过构建并发程序的最大因果模型, 将探索程序状态空间问题转换为约束求解问题. 最大因果约减算法如图 1 所示, 先输入一个被测程序, 通过模型检查器(model checker)按照指定的线程交织得到程序的执行轨迹(trace). 在模型检查器中的调度器通过控制线程的调度探索指定线程交织. 然后, 通过最大因果引擎(maximal causality engine)生成新的种子线程交织, 并将其添加到被测程序的状态空间中, 使得该交织之后可以被模型检查器再次使用, 并通过运行获取新的种子交织. 对于一个线程交织 $s$, 通过前文介绍的最大因果模型, 可以获得一组唯一且规模最大的可行性交织集合, 记作 *MaxCausal*($s$). 且每个 MCR 程序运行产生的新的种子交织都是 *MaxCausal*($s$)中的可行交织.

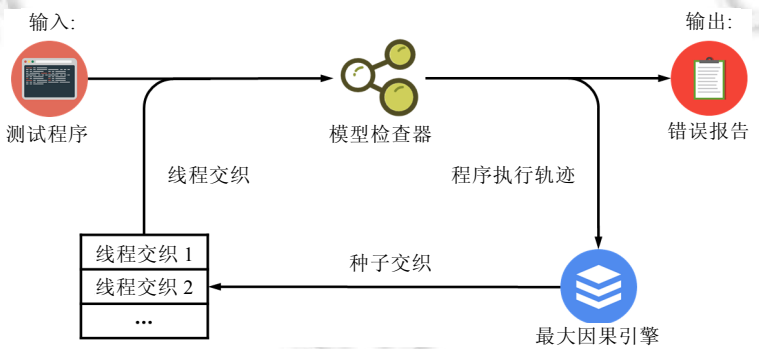



图 1 最大因果约减算法执行流程

在并发程序的运行过程中, 不同线程的交错执行会产生一系列线程交织. 而每个被触发的线程交织都会使程序到达某一种状态(state). 这里规定: 当程序执行之前产生新交织时, 如果从同一个变量上读到和之前不同的值从而使得程序进入了一个新的状态, 那么算法需要将这一状态加入到被测程序的状态空间中. 当被测程序的状态空间已满, 即所有的种子交织都已经被探索过, 且没有新的种子交织生成时, 即可以认为被测程序的状态空间已经被完全覆盖了. 这里, MCR 提出了一个新的约束 Race 约束, 即假设 *COP*($x$,$y$)表示一个全局变量在不同线程中的读写对, 那么对于程序中每一个存在的 *COP* 关系的全局变量, 需要对这些变量的每一个读事件 $R$ 进行约束, 即所有 happens-before 于 $R$ 的读事件读到与之前一样的值, 且 $R$ 应当读到一个与之前完全不一样的值. 由于每次程序读到新值都可能导致程序到达一个与之前不同的状态, 因此, 通过在程序的约束求解的过程中不断地读到新值, 可以获得一棵程序的状态树, 而这棵树就可以覆盖程序中所有可能的程序状态.

导致并发程序的状态空间庞大的原因有两方面: 一是由于线程的组合可以是多种多样的, 且其中大部分都是可行的以及线程的调度是不确定的; 另一方面, 并发程序的状态空间存在大量等价的线程交织, 除线程的调度顺序不同, 读取到的数据是一致的, 即这些等价的线程是不能够使程序到达一个新的不同的状态. 大量等价线程交织使其他算法执行时间相对较长. 而最大因果约减算法虽然也会重复地执行被测程序, 通过干预线程调度, 执行所有调度并检查当前调度的可行性, 产生新的种子交织. 然而, 该算法仅当读到新值时才会识别程序到达了一个新的状态, 并将当前线程调度加入到当前程序的状态空间中, 从而能够最大化约减状态空间.

### 1.3 约束的构建与求解

本文将最大因果约减算法中的约束分为程序基本约束和读写约束: 程序基本约束是程序中对 happens-before 原则的约束体现, 其中包括程序顺序约束、线程约束、锁约束等; 读写约束则是约束不同线程对同一共享变量按照指定读写要求进行读写, 其中, 读写要求是程序中所有线程的读事件中有且只有一个读事件 $R$ 读到新的值, 而发生在 $R$ 前的读事件需要读到旧的值. 如图 2 程序所示: $T1$ 和 $T2$ 是两个线程, 则图中程序基本约束是 $x1<x2$, $x3<x4$, $x4<x5$, $x5<x6$, 即同一线程中程序按照代码的顺序执行; 假如第 4 行 $x$ 应读到值 1, 则读写约束为 $x3<x1$, $x1<x4$, 即对 $x$ 的赋值为 1 的操作需要在第 4 行前执行, 且在第 3 行对 $x$ 赋值为 0 的操作后执行. 程序基本约束和读写约束是 and 关系, 即并列关系. 只有两种约束共同求解出的调度序列, 才是一个可执行的调度序列.

```
   T1              T2
1: x = 1        3: x = 0
2: y = 2        4: if (x > 0) {
                5:   y—
                6:   x = 7
                   }
```

图 2 约束构建示例程序

本文使用 Z3[20]对构建的约束文件进行求解, Z3 是由微软开发的开源约束求解器, 是用来解决一阶逻辑组合的判定性问题的一种通用求解器. 在给定部分约束条件的情况下, Z3 能够找到一组满足这些条件的解. 使用 Z3 进行求解需要编写 Z3 脚本, Z3 脚本包含一段特定的命令序列, 这一段命令序列将会被求解器使用特定算法解析通过, 最终求解器会返回公式命题的求解结果以及来自 Z3 的理论求解器的赋值. 通过使用 Z3 对 GC-MCR 方法得到的新约束进行求解, 然后根据求解结果获得新的调度序列, 探索并发程序状态空间.

## 2 GC-MCR 方法

### 2.1 研究动机

为了充分探索并发程序的状态空间, MCR[18]对已有的执行轨迹进行分析, 根据不同事件间的约束关系生成满足 MCM 和 Race 的约束, 并利用约束求解器进行求解. 最终, 根据求解结果对并发程序进行进一步的探索, 即: 若约束可解, 则按照求解得到的线程交织控制程序调度; 反之, 则表示该线程交织无法发生, 故跳过当前约束并求解其他约束. 尽管 MCR 可以通过较少的调度次数找到并发程序中潜在的并发缺陷, 然而, 已有的研究工作[21]以及本工作对 MCR 方法的重现与深入分析表明: MCR 在构建程序约束时粒度较粗, 生成大量的冗余或不可解约束, 造成约束求解次数过多, 额外消耗大量求解计算资源. 假如能够在求解前过滤不可解约束或约减冗余约束, 即可节省大量计算资源. 因此, 通过在求解前过滤不可解约束或约减冗余约束, 可节省大量求解成本, 从而提升并发缺陷的检测效率. 为此, 本文分别在 hashcode, numberaxis 和 dekker 这 3 个程序状态空间大小不同的测试程序上进行实验验证. 实验结果显示, MCR 在上述测试程序中分别进行约束求解 32, 242 和 492 次. 然而, 对这些约束进一步的分析发现: 分别有 21, 67 和 150 个约束可通过约束分析进行过滤而

无需使用约束求解器进行约束求解, 其余约束也能够进行不同程度的约减. 基于上述发现, 本工作对上述约束进行过滤和约减, 最终发现程序执行时间分别减少 51.78%、28.29%和 58.35%, 并发程序状态空间的探索效率显著提高. 本文后续在其他测试程序进行实验, 最终得到相同结论. 为阐述本文所提方法的基本原理, 后续方法介绍将以 dekker 程序为例.

实验发现: MCR 生成的约束中, 部分约束存在明显冲突, 且能够通过简单有效的方式处理这些冲突. 例如: 图 3 和图 4 分别为 dekker 程序在一次约束求解中出现的一个程序基本约束和两个读写约束, 其中, 每个 assert 语句由 assert 关键词和一个子约束构成, 每个子约束使用前缀表达式表示, 其中, 最内层的一组括号中的内容是一个约束单元, 每个约束单元可使用 and/or 进行组合并构成一个子约束. 例如: 图 4(b)中, (or (>$x$1 $x$9) (<$x$3 $x$6))是一个子约束, 其中, (>$x$1 $x$9)是一个约束单元, 实际意义为事件 $x$1 在事件 $x$9 之后发生; 图 3 中, (>$x$1 $x$2)既是一个约束单元, 又是一个子约束. 每个 assert 语句之间是 and 关系, 即当且仅当每个约束都为真, 才能表示整体约束是满足的.

```
(assert (> x1 x2))  (assert (< x5 x6))  (assert (< x7 x8))
(assert (> x3 x5))  (assert (> x6 x8))  (assert (< x4 x6))
(assert (> x3 x7))  (assert (< x6 x2))  (assert (> x9 x4))
(assert (< x4 x2))  (assert (> x3 x6))
```

图 3 程序基本约束示例

```
(assert (and (or (or (< x8 x7) (< x9 x4)) (and (> x1 x8)
(> x3 x5))) (and (or (> x1 x4) (> x1 x6)) (< x3 x6))))
```

(a) 冲突约束

```
(assert (or (> x1 x9) (< x3 x6)))
```

(b) 冗余约束

图 4 读写约束示例

- 冲突约束

图 4 中, 读写约束(a)中的 $x$8<$x$7, $x$4>$x$9, $x$3<$x$6 分别与图 3 中程序基本约束中的 $x$8>$x$7, $x$9>$x$4, $x$3>$x$6 冲突, 而其他约束与程序基本约束一致. 通过逻辑运算判断, 其最终结果为 false, 与程序基本约束冲突, 即满足程序基本约束的条件下读写约束不成立. 故定义这类与程序基本约束冲突的读写约束为冲突约束. 冲突约束即不可解约束. 并发测试程序 dekker 中存在 150 个冲突, 在使用约束求解器求解前, 识别并过滤这类冲突约束, 则能够节省大量计算资源.

- 冗余约束

进一步分析发现: 读写约束(b)中的 $x$3<$x$6 与程序基本约束中的 $x$3>$x$6 冲突; 而另一部分 $x$1>$x$9 与程序基本约束不冲突, 且在满足程序基本约束的条件下读写约束(b)成立. 故定义这类部分子约束与程序基本约束冲突, 且约束整体与基本约束不冲突的读写约束为冗余约束. 冗余约束能够根据程序基本约束进行约减, 图 4 中的冗余约束就可以被约减为(>$x$1 $x$9). 而对于约减后的约束, 约束求解器可以更快地进行求解, 节省程序在约束求解步骤中的求解计算资源. 而且对约束进行过滤与约减产生的时间开销远小于求解器求解原约束的时间开销.

基于上述分析, 本文提出了一种有向图约束指导的并发缺陷检测方法 GC-MCR. 首先, 把约束分为读写约束和程序基本约束. 然后, 先将程序基本约束构建成有向图, 再将读写约束构建成前缀树. 最后, 根据有向图判断读写约束中是否存在冲突约束和冗余约束: 若存在冲突约束, 则过滤掉当前约束; 若存在冗余约束, 则将其约减后, 再次调用约束求解器进行求解, 从而提高并发缺陷检测效率.

### 2.2 方法流程

#### 2.2.1 整体框架: 输入/输出及整体流程简介

GC-MCR 方法的输入是并发测试程序, 输出是程序的并发缺陷错误报告. 首先, 模型检查器执行并发测试程序, 获取程序的初始执行轨迹. 最大因果引擎分析处理执行轨迹, 生成满足 MCM 和 Race 约束的约束. 然后, GC-MCR 方法将约束分为读写约束和程序基本约束: 程序基本约束被分析并重构成有向图, 读写约束被

分析、拆分并重组成读写约束树. 随后, 约束过滤器根据有向图计算读写约束树中每个节点的值, 并根据读写约束树根节点的值判断: 是否跳过当前求解或约减当前约束. 假如进行约束求解, 求解得到的线程交织会存储到线程交织集合中. 最后, 模型检查器通过依次执行线程交织集合中的线程交织, 获取程序执行轨迹并检测程序是否存在并发缺陷: 假如检测到并发缺陷, 则输出错误报告; 反之, 则重复上述操作.

GC-MCR 方法的具体工作流程如图 5 所示.

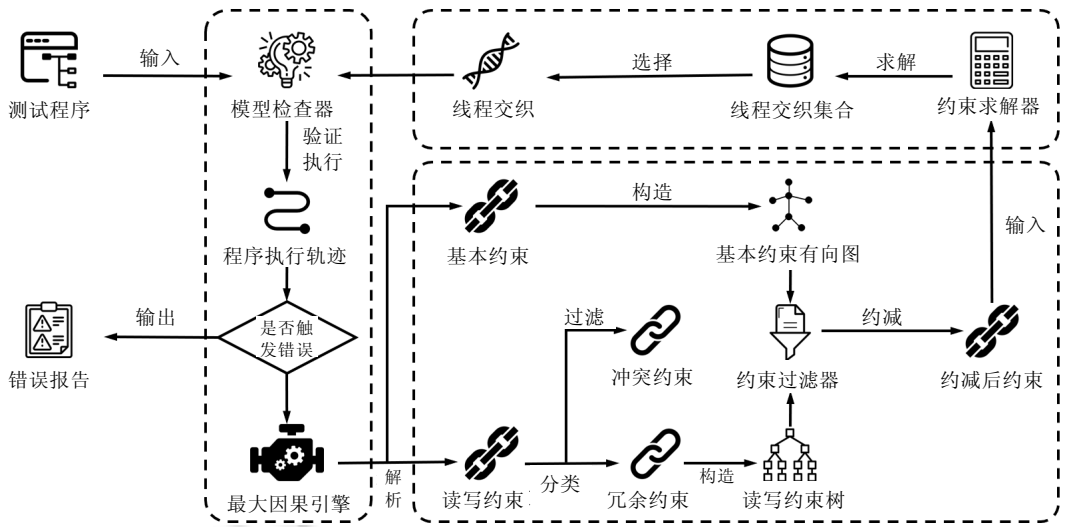



图 5 基于有向图构建约束的并发缺陷检测方法流程图

2.2.2 基于有向图的程序基本约束重构

图 3 中约束为 GC-MCR 生成的程序基本约束, 程序基本约束都满足 happens-before 原则. 根据顺序一致性模型的定义, 程序中的约束必须遵循 happens-before 原则, 因此不应该存在违背程序基本约束的约束. 图 6 是本方法生成的有向图, 表示程序中各个调度点在满足 happens-before 原则下的顺序关系.

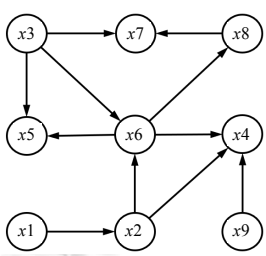



图 6 基本约束的有向图

由于程序基本约束中的顺序不等式存在传递性关系, 且不是所有调度点都存在这些顺序关系, 因此, 通过数轴上的大小表示难于归纳出所有调度点的顺序关系. 而为了清晰表明调度点的顺序关系, 本方法将这些顺序关系转化为有向图来表示. 具体实现时, 使用邻接表构建有向图, 并定义新的节点类, 其中包含当前节点对应的调度点和在有向图中直接指向该调度点的节点的集合. 在图 3 中, 若存在约束 $x1>x2$, 则在有向图中构建 $x1$ 指向 $x2$ 的边. 不断重复该过程, 直至最终构建出该有向图. 如算法 1 所示: 首先分析程序基本约束中每对调度点间的关系, 然后在图中找到对应节点, 最后将节点间指向关系存储在邻接表中. 通过上述步骤, 使用有向图表示程序基本约束关系, 仅通过遍历该有向图, 即可获得程序中线程调度点间的关系, 用于后续处理读写约束.

**算法 1**. 基于有向图的程序基本约束重构.

输入: $\Phi_{base}$: 程序基本约束;

输出: $G$: 由邻接表构成的有向图.

```
1  G←∅;
2  for φ_i in Φ_base do
3     option, target, next←analysisConstraint(φ_i);
4     targetNode, nextNode←findNode(target,next,G);
5     targetNode.largeNodes.add(nextNode);
6  end for
7  return G
```

2.2.3 读写约束的解析与拆分

读写约束的生成是 GC-MCR 的核心步骤, 该步骤确保所有共享变量的每一个读事件 $R$ 需要满足两个条件: 读事件 $R$ 要读到一个之前没有读到过的值; 所有发生在读事件 $R$ 之前的读事件要读到和原来一样的值. 这样保证只有一个读事件读到新值, 其余读事件仍读到原值. 例如: 对于变量 $x$, 若需要读取新值, 则让其中一个具有新的写值的写 $x$ 事件排在该读 $x$ 事件之前; 若需要保持读取到原值, 则应当保证新的 $x$ 写事件在该读事件之后. 因此, GC-MCR 能够通过控制读写事件, 改变调度点的顺序排布.

然而, 令一个读事件读到新值有多种限制和可能情况, 其中既包含保证新的 $x$ 的写入值的约束, 又包含保证其他读事件的值保持不变的约束, 因此读写约束颇为复杂. 如图 4 所示, 读写约束(a)是由多个 or 和 and 组合表示的约束. 现有方法将这类读写约束和先前生成的程序基本约束传入约束求解器中进行求解, 获得调度点的顺序关系后再解析成新的线程调度序列, 进行线程定向调度, 最终探索程序整个状态空间. 然而, 现有方法忽视了读写约束和程序基本约束之间的关系, 其中包含明显冲突的约束和冗余的约束, 冲突约束不需要调用约束求解器进行求解, 冗余约束可以约减后进行求解. 通过过滤冲突约束和约减冗余约束, 能够有效减少单次求解时间甚至减少约束求解器的调用次数, 显著提高并发缺陷检测效率.

因此, GC-MCR 对读写约束进行解析. 图 7 是根据图 4 中的读写约束(a)构造的前缀树, 其中: 非叶子节点皆是 and/or 的逻辑运算符, 叶子节点则是调度点的顺序关系. 后续用于遍历程序基本约束有向图进行冲突分析. 具体算法见算法 2.

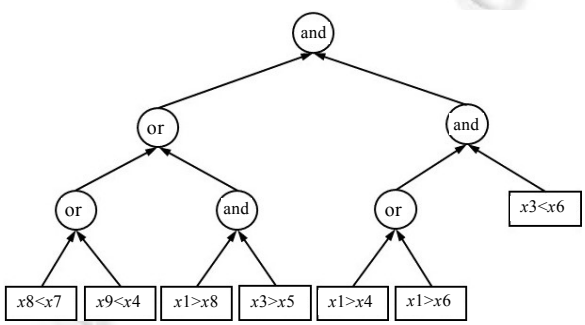



图 7 读写约束树

**算法 2**. 读写约束的解析与拆分.

输入: $\varphi_{rw}$: 程序的一条读写约束; $G$: 程序基本约束的有向图;

输出: $T$: 以前缀表达式表示的读写约束树, 树中的每个节点包含当前节点的布尔值以及从叶子节点到当前节点的约束表达式.

```
1  T←∅;
2  list←split(φ_rw);
3  for s in list do
```

```
4    if s∈{<,>} then
5        target, next←analysisConstraint(s);
6        isConflict←isConflictWithOrderMap(target,next,G);
7        expression←getConstraintExpression(s,target,next,isConflict);
8        T.add(Pair(isConflict,expression));
9    else
10       T.add(Pair(s,s));
11   end if
12 end for
13 return T;
```

例如, 其中一个叶子节点中 $x3>x9$, 查看在图 6 中是否存在 $x9>x3$, 即是否存在一条由 $x9$ 到 $x3$ 的路径(第 6 行): 若存在, 则设该叶子节点为 false, 表示该子表达式中存在冲突; 反之, 该叶子节点为 true, 表示该子表达式中不存在冲突. 然后, 对每个叶子节点执行该操作, 将全部叶子节点的结果替换为 true/false; 同时, 每个叶子节点还应存储当前节点的约束表达式(第 7 行、第 8 行), 后续用于约束表达式约减. 最终, 经过冲突分析后, 叶子节点应全部变为 true/false. 此时, 整棵树为一个由 and/or 和 true/false 组成的运算树. 最终, 只需将该运算树进行运算即可. 若结果为 false, 则表示该读写约束与程序基本约束冲突, 直接跳过此次对约束求解器的调用即可; 反之, 则约减当前约束. 被过滤约束没有调用约束求解器, 这部分时间被过滤算法优化, 从而提高并发缺陷检测效率.

#### 2.2.4 约束表达式约减

对冗余约束的约减, GC-MCR 将约减其中冲突的子表达式, 最后对约减后的约束表达式及程序基本约束的表达式调用约束求解器进行求解.

本文中, 读写约束树的每个节点既包含以当前节点为根节点的子树是否满足程序基本约束的值, 又包含子树的约束表达式, 称约束表达式前缀树中的叶子节点中的约束表达式为子表达式. 首先, 将子表达式与程序基本约束冲突的叶子节点的值置为 false, 其余叶子节点的值置为 true. 如图 8 所示, 赋值后的前缀树的叶子结点为 true/false, 在图中使用 T 表示 true, F 表示 false. 然后, 根据父节点的操作符及其两个子节点的值, 考虑以下 5 种情况.

a) 父节点为 and, 两个子节点皆为 true, 如图 9(a)所示: 将父节点的值置为 true, 并将父节点的约束表达式置为 and 关系的两个子节点的约束表达式;
b) 父节点为 or, 两个子节点皆为 true, 如图 9(b)所示: 将父节点的值置为 true, 并将父节点的约束表达式置为 or 关系的两个子节点的约束表达式;
c) 父节点为 or, 两个子节点皆为 false, 如图 9(c)所示: 将父结点的值置为 false, 并将父节点的约束表达式置为空;
d) 父节点为 and, 两个子节点中至少一个为 false, 如图 9(d)所示: 将父节点的值置为 false, 并将父节点的约束表达式置为空;
e) 父节点为 or, 两个子节点中仅一个为 true, 如图 9(e)所示: 将父结点的值置换为 true, 并将父节点的约束表达式置为值为 true 子节点的约束表达式.

重复上述操作, 直至读写约束树中仅存在根节点. 此时, 根据根节点的值判断是否使用约束求解器进行约束求解: 若值为 false, 则不需要; 反之, 使用根节点的约束表达式, 即最简的读写约束表达式, 结合程序基本约束的表达式使用约束求解器进行求解, 在当前例子中, 最终根节点的值为 false, 因此可以直接跳过约束求解步骤. 执行 GC-MCR 方法的后续操作, 检测并发程序中的并发缺陷.

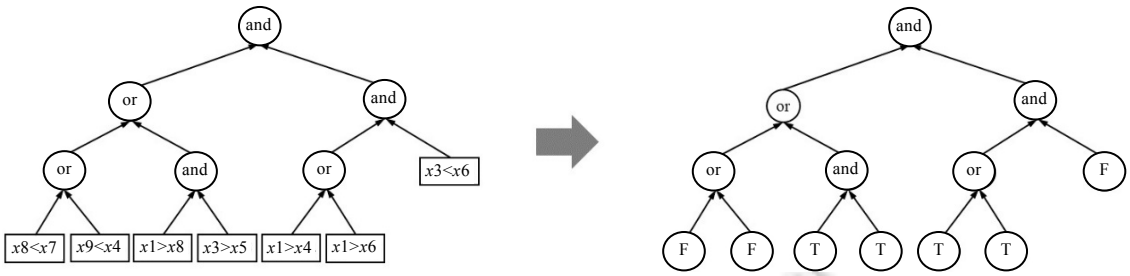
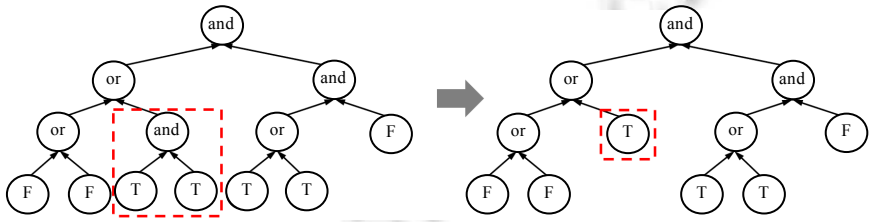
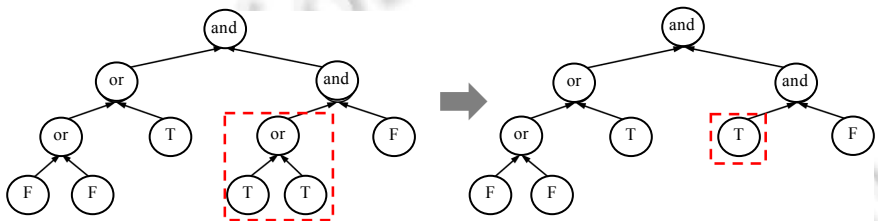


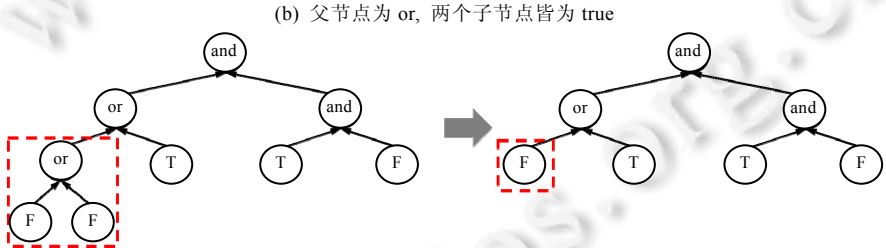

图 8　前缀树替换的示例



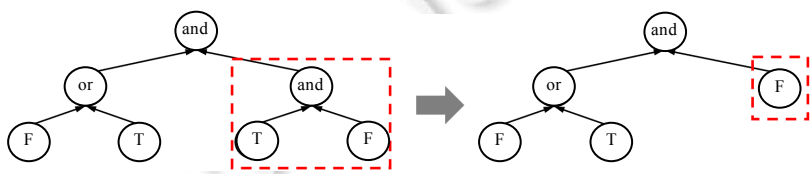

(a) 父节点为 and, 两个子节点皆为 true



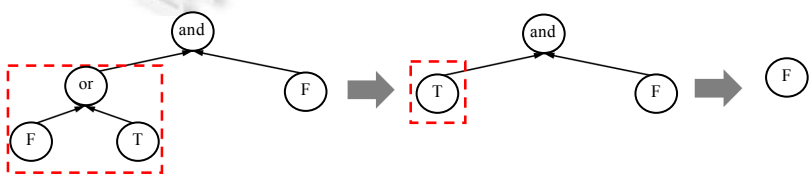

(b) 父节点为 or, 两个子节点皆为 true




(c) 父节点为 or, 两个子节点皆为 false




(d) 父节点为 and, 两个子节点中至少一个为 false




(e) 父节点为 or, 两个子节点中仅一个为 true

图 9　前缀树简化的示例

## 3 实验设计与结果分析

### 3.1 研究问题

为了验证 GC-MCR 方法的有效性, 本文从程序约束求解次数、程序运行时间和程序调度次数的角度设计了 3 个研究问题.

(1) RQ1: GC-MCR 方法是否有效减少程序约束求解的次数, 减少约束求解器的计算资源消耗?

由于调用约束求解器进行约束求解相比于程序调度更加耗时, 如果可以减少程序的约束求解次数, 降低约束求解器的计算资源消耗, 对并发缺陷检测有着重要意义. 现有方法[18]在约束进行约束求解前没有对约束进行任何处理, 使得许多可以提前被过滤的冲突约束被约束求解器求解, 增加约束求解的次数, 降低探索并发程序状态空间的效率. 因此, 本研究问题中, 本文希望分析与现有方法相比, GC-MCR 方法能否有效减少程序调用约束求解器进行约束求解的次数, 降低约束求解器的计算资源消耗.

(2) RQ2: GC-MCR 方法能否有效减少并发缺陷检测的时间, 提高探索并发程序状态空间的效率?

程序检测到并发缺陷所需时间是评估并发缺陷检测效率的一个重要指标. 通过统计程序约束求解的次数以及程序的运行时间, 还可以分析两者之间的关系, 即是否可以通过减少约束求解次数以降低并发缺陷检测所消耗的时间, 以及影响并发缺陷检测的因素. 因此, 在本研究问题中, 本文希望从多种角度来分析与现有方法相比, GC-MCR 方法能否有效减少并发缺陷检测的时间, 提高并发程序状态空间的探索效率.

(3) RQ3: GC-MCR 方法是否可以保证现有方法的并发缺陷检测能力?

现有的研究方法在并发缺陷检测上已取得一定的效果, 即可以使用最少的调度次数准确找到并发程序中的缺陷. 而并发程序的调度次数同时也是衡量并发缺陷检测能力的一个主要指标. 如果 GC-MCR 方法的调度次数与现有方法的调度次数相等甚至更少, 且发现相同缺陷, 则并发缺陷检测能力没有下降. 因此, 在本研究问题中, 本文希望验证与现有方法相比, GC-MCR 方法能否拥有相同甚至更优的并发缺陷检测能力.

### 3.2 实验配置

- 测试对象

本文通过从现有研究[18,22,23]广泛使用的并发数据集中收集一组包含 38 个含有并发缺陷的测试程序作为测试对象. 表 1 显示了这些类的详细信息, 其中,

- ✧ ID 列表示测试程序的实验序号;
- ✧ 测试程序的列表示测试程序的名称;
- ✧ 代码行数的列表示测试程序中的代码行数(使用 Statistic 工具统计);
- ✧ 线程数量的列表示测试程序中线程的数量;
- ✧ 变量数量的列表示测试程序中定义的变量数量, 其中包含来自它们父类的变量. 注意: 变量的统计不区分变量的访问权限(例如,公共或私有), 因为所有的实例变量都是正确共享的;
- ✧ 方法数量的列表示测试程序中公共方法的数量, 其中包括那些来自父类的方法;
- ✧ 最后, 并发缺陷类型的列表示测试程序中的并发缺陷的类型.

特别地, 为评估 GC-MCR 的有效性和可扩展性, 本文在真实世界程序 weblech 上对 GC-MCR 进行了进一步验证. weblech 是一个网站下载工具, 包含一个启动服务器、执行客户机请求和终止的测试驱动程序. 相比于其他测试程序, weblech 包含更多的程序事件以及更大程序状态探索空间.

- 对比方法

最大因果约减算法 Java 版本的实现, 将探索并发程序状态空间的问题转换成约束求解问题, 以检测并发缺陷. 本文提出的 GC-MCR 方法不仅探索并发程序状态空间的问题转换成约束求解问题, 还在此基础上提出了有向图表示方法, 构建完整的并发程序约束并对其进行过滤和约减, 降低方法在约束求解上的时间开销.

本文中, 所有相关实验的运行环境为具有一个四核 CPU Intel(R) Core(TM) CPU i7-7500U@2.70 GHz 2.90 GHz 和 8 GB 内存, 运行 Windows 10 企业版(64 位)的机器. 每次程序执行的超时设置为 90 min. 为了尽量减

少随机性因素的影响, 每个实验重复 10 次执行, 最后取平均值作为实验结果.

表 1 GC-MCR 测试对象集合描述

| ID | 测试程序 | 代码行数 | 线程数量 | 变量数量 | 方法数量 | 并发缺陷类型 |
|---|---|---|---|---|---|---|
| 1 | account | 377 | 10 | 1 | 19 | Atomicity |
| 2 | airline | 182 | 5 | 4 | 15 | DataRace |
| 3 | alarmclock | 372 | 9 | 19 | 21 | DataRace |
| 4 | allocation | 347 | 2 | 2 | 14 | Atomicity |
| 5 | array | 104 | 2 | 20 | 2 | DataRace |
| 6 | bakery | 126 | 2 | 6 | 5 | Atomicity |
| 7 | boundedbuffer | 159 | 3 | 1 | 2 | Deadlock |
| 8 | bubblesort | 160 | 4 | 4 | 7 | DataRace |
| 9 | checkfield | 65 | 2 | 2 | 4 | DataRace |
| 10 | clean | 145 | 12 | 10 | 9 | DataRace |
| 11 | consisitency | 59 | 3 | 2 | 2 | Atomicity |
| 12 | counter | 57 | 2 | 2 | 2 | DataRace |
| 13 | critical | 117 | 2 | 1 | 2 | Atomicity |
| 14 | cyclic | 72 | 4 | 1 | 3 | DataRace |
| 15 | datarace | 16 | 2 | 10 | 2 | DataRace |
| 16 | deadlock | 64 | 2 | 3 | 4 | Deadlock |
| 17 | dekker | 118 | 2 | 6 | 2 | Atomicity |
| 18 | dirkaccount | 167 | 2 | 7 | 10 | Deadlock |
| 19 | fileappender | 410 | 2 | 7 | 33 | Atomicity |
| 20 | hashcode | 2 461 | 2 | 1 | 3 | Atomicity |
| 21 | lamport | 160 | 2 | 9 | 2 | Atomicity |
| 22 | mergesort | 384 | 3 | 11 | 13 | DataRace |
| 23 | numberaxis | 1 637 | 2 | 43 | 110 | Atomicity |
| 24 | peruserpooldatasource | 682 | 2 | 35 | 65 | DataRace |
| 25 | peterson | 85 | 2 | 5 | 2 | Atomicity |
| 26 | pingpong | 294 | 5 | 9 | 13 | DataRace |
| 27 | pool | 243 | 2 | 4 | 2 | DataRace |
| 28 | rax | 95 | 2 | 2 | 2 | Deadlock |
| 29 | readerswriters | 608 | 3 | 12 | 18 | DataRace |
| 30 | replicatedcasestudies | 1 427 | 4 | 1 | 2 | Deadlock |
| 31 | rvexample | 75 | 2 | 3 | 2 | DataRace |
| 32 | sharedobject | 67 | 2 | 1 | 2 | DataRace |
| 33 | sharedpooldatasource | 516 | 2 | 30 | 51 | Atomicity |
| 34 | simple | 65 | 2 | 2 | 2 | Atomicity |
| 35 | store | 79 | 2 | 1 | 4 | Atomicity |
| 36 | transfer | 91 | 2 | 3 | 7 | Deadlock |
| 37 | waitnotify | 99 | 4 | 1 | 1 | DataRace |
| 38 | weblech | 35K | 3 | 49 | 90 | DataRace |

### 3.3 结果分析

(1) RQ1: GC-MCR 方法是否有效减少程序约束求解的次数, 减少约束求解器的计算资源消耗?

为了回答 RQ1, 本文系统地将 GC-MCR 方法应用于每个测试程序, 统计检测到第 1 个并发缺陷时所需要的约束求解次数、约束求解消耗时间, 检查两种方法之间是否存在差异. 为了与基准做对比, 本文选择将 GC-MCR 与 J-MCR 方法对比, 实验结果见表 2 和表 3. 其中, GC-MCR 约束求解次数和 J-MCR 约束求解次数分别表示使用 GC-MCR 方法和 J-MCR 方法找到测试程序中第一个并发缺陷所需的约束求解次数, 减少的约束求解次数表示相比于 J-MCR, 通过使用 GC-MCR 方法可以减少的约束求解次数; 减少的约束百分比表示减少的约束求解次数占 J-MCR 约束求解次数的百分比; GC-MCR 约束求解耗时和 J-MCR 约束求解耗时分别表示使用 GC-MCR 方法和 J-MCR 方法检测出测试程序第一个并发缺陷时, 约束求解器进行求解消耗的总时间; 减少的求解时间表示相比于 J-MCR, 通过使用 GC-MCR 方法可以减少的使用约束求解器进行求解消耗的总时间; 减少的求解时间百分表示减少的求解时间占 J-MCR 约束求解耗时的百分比; GC-MCR 约束过滤耗时表示使用 GC-MCR 方法找到测试程序中第一个并发缺陷所需的对约束进行过滤和约减消耗的总时间; 过滤求解时间百分比表示 GC-MCR 约束过滤耗时占 J-MCR 约束求解耗时的百分比.

表 2 GC-MCR 与 J-MCR 的约束求解次数对比

| ID | 测试程序 | GC-MCR 约束求解次数(次) | J-MCR 约束求解次数(次) | 减少的约束求解次数(次) | | 减少的约束百分比(%) | |
|---|---|---|---|---|---|---|---|
| 1 | account | 12 | 12 | – | 0 | – | 0.00 |
| 2 | airline | 2 303 | 2 530 | ↓ | 227 | ↑ | 8.97 |
| 3 | alarmclock | 24 526 | 26 799 | ↓ | 2273 | ↑ | 8.48 |
| 4 | allocation | 30 | 36 | ↓ | 6 | ↑ | 16.67 |
| 5 | array | 399 | 437 | ↓ | 38 | ↑ | 8.70 |
| 6 | bakery | 75 | 93 | ↓ | 18 | ↑ | 19.35 |
| 7 | boundedbuffer | 16 | 16 | – | 0 | – | 0.00 |
| 8 | bubblesort | 272 | 286 | ↓ | 14 | ↑ | 4.90 |
| 9 | checkfield | 4 | 14 | ↓ | 10 | ↑ | 71.43 |
| 10 | clean | 25 | 25 | – | 0 | – | 0.00 |
| 11 | consisitency | 8 | 12 | ↓ | 4 | ↑ | 33.33 |
| 12 | counter | 66 | 67 | ↓ | 1 | ↑ | 1.49 |
| 13 | critical | 2 | 3 | ↓ | 1 | ↑ | 33.33 |
| 14 | cyclic | 100 | 100 | – | 0 | – | 0.00 |
| 15 | datarace | 5 | 15 | ↓ | 10 | ↑ | 66.67 |
| 16 | deadlock | 1 | 1 | – | 0 | – | 0.00 |
| 17 | dekker | 342 | 492 | ↓ | 150 | ↑ | 30.49 |
| 18 | dirkaccount | 425 | 456 | ↓ | 31 | ↑ | 6.80 |
| 19 | fileappender | 172 | 175 | ↓ | 3 | ↑ | 1.71 |
| 20 | hashcode | 11 | 32 | ↓ | 21 | ↑ | 65.63 |
| 21 | lamport | 372 | 515 | ↓ | 143 | ↑ | 27.77 |
| 22 | mergesort | 317 | 546 | ↓ | 229 | ↑ | 41.94 |
| 23 | numberaxis | 175 | 242 | ↓ | 67 | ↑ | 27.69 |
| 24 | peruserpooldatasource | 7 | 8 | ↓ | 1 | ↑ | 12.50 |
| 25 | peterson | 28 | 37 | ↓ | 9 | ↑ | 24.32 |
| 26 | pingpong | 28 | 28 | – | 0 | – | 0.00 |
| 27 | pool | 9 | 9 | – | 0 | – | 0.00 |
| 28 | rax | 31 | 31 | – | 0 | – | 0.00 |
| 29 | readerswriters | 1 324 | 1 339 | ↓ | 15 | ↑ | 1.12 |
| 30 | replicatedcasestudies | 104 | 104 | – | 0 | – | 0.00 |
| 31 | rvexample | 76 | 92 | ↓ | 16 | ↑ | 17.39 |
| 32 | sharedobject | 3 | 8 | ↓ | 5 | ↑ | 62.50 |
| 33 | sharedpooldatasource | 4 | 5 | ↓ | 1 | ↑ | 20.00 |
| 34 | simple | 12 | 18 | ↓ | 6 | ↑ | 33.33 |
| 35 | store | 23 | 28 | ↓ | 5 | ↑ | 17.86 |
| 36 | transfer | 2 | 2 | – | 0 | – | 0.00 |
| 37 | waitnotify | 1 159 | 1 456 | ↓ | 297 | ↑ | 20.40 |
| 38 | weblech | 1 124 | 2 297 | ↓ | 1 173 | ↑ | 51.07 |

表 3 GC-MCR 与 J-MCR 的约束求解时间对比

| ID | 测试程序 | GC-MCR 约束求解耗时(ms) | J-MCR 约束求解耗时(ms) | 减少的求解时间(ms) | | 减少的求解时间百分比(%) | | GC-MCR 约束过滤耗时(ms) | 过滤求解时间百分比(%) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | account | 794 | 1 029 | ↓ | 235 | ↑ | 22.84 | 15 | 1.46 |
| 2 | airline | 304 448 | 360 345 | ↓ | 55 897 | ↑ | 15.51 | 246 | 0.07 |
| 3 | alarmclock | 3 658 195 | 3 880 957 | ↓ | 222 762 | ↑ | 5.74 | 3 206 | 0.08 |
| 4 | allocation | 2 278 | 2 646 | ↓ | 368 | ↑ | 13.91 | 60 | 2.27 |
| 5 | array | 56 899 | 98 283 | ↓ | 41 384 | ↑ | 42.11 | 272 | 0.28 |
| 6 | bakery | 8 860 | 21 655 | ↓ | 12 795 | ↑ | 59.09 | 95 | 0.44 |
| 7 | boundedbuffer | 1 011 | 1 373 | ↓ | 362 | ↑ | 26.37 | 38 | 2.77 |
| 8 | bubblesort | 48 876 | 62 142 | ↓ | 13 266 | ↑ | 21.35 | 443 | 0.71 |
| 9 | checkfield | 289 | 757 | ↓ | 468 | ↑ | 61.82 | 21 | 2.77 |
| 10 | clean | 7 236 | 11 049 | ↓ | 3 813 | ↑ | 34.51 | 64 | 0.58 |
| 11 | consisitency | 544 | 896 | ↓ | 352 | ↑ | 39.29 | 29 | 3.24 |
| 12 | counter | 28 457 | 33 786 | ↓ | 5 329 | ↑ | 15.77 | 82 | 0.24 |
| 13 | critical | 172 | 232 | ↓ | 60 | ↑ | 25.86 | 12 | 5.17 |
| 14 | cyclic | 23 573 | 27 273 | ↓ | 3 700 | ↑ | 13.57 | 70 | 0.26 |
| 15 | datarace | 353 | 1 158 | ↓ | 805 | ↑ | 69.52 | 31 | 2.68 |
| 16 | deadlock | 93 | 117 | ↓ | 24 | ↑ | 20.51 | 7 | 5.98 |
| 17 | dekker | 77 291 | 154 573 | ↓ | 77 282 | ↑ | 50.00 | 70 | 0.05 |
| 18 | dirkaccount | 101 272 | 216 110 | ↓ | 114 838 | ↑ | 53.14 | 1 098 | 0.51 |
| 19 | fileappender | 21 218 | 29 786 | ↓ | 8 568 | ↑ | 28.77 | 220 | 0.74 |

表 3 GC-MCR 与 J-MCR 的约束求解时间对比(续)

| ID | 测试程序 | GC-MCR 约束求解耗时(ms) | J-MCR 约束求解耗时(ms) | 减少的求解时间(ms) | 减少的求解时间百分比(%) | GC-MCR 约束过滤耗时(ms) | 过滤求解时间百分比(%) |
|---|---|---|---|---|---|---|---|
| 20 | hashcode | 970 | 2 374 | ↓ 1 404 | ↑ 59.14 | 36 | 1.52 |
| 21 | lamport | 22 329 | 38 688 | ↓ 16 359 | ↑ 42.28 | 86 | 0.22 |
| 22 | mergesort | 245 428 | 347 597 | ↓ 102 169 | ↑ 29.39 | 11 404 | 3.28 |
| 23 | numberaxis | 188 382 | 265 010 | ↓ 76 628 | ↑ 28.92 | 3 593 | 1.36 |
| 24 | peruserpooldatasource | 1 040 | 3 644 | ↓ 2 604 | ↑ 71.46 | 17 | 0.47 |
| 25 | peterson | 10 905 | 17 111 | ↓ 6 206 | ↑ 36.27 | 34 | 0.20 |
| 26 | pingpong | 8 918 | 13 409 | ↓ 4 491 | ↑ 33.49 | 28 | 0.21 |
| 27 | pool | 514 | 622 | ↓ 108 | ↑ 17.36 | 14 | 2.25 |
| 28 | rax | 11 302 | 16 625 | ↓ 5 323 | ↑ 32.02 | 58 | 0.35 |
| 29 | readerswriters | 220 750 | 322 420 | ↓ 101 670 | ↑ 31.53 | 2 823 | 0.88 |
| 30 | replicatedcasestudies | 16 219 | 29 711 | ↓ 13 492 | ↑ 45.41 | 133 | 0.45 |
| 31 | rvexample | 4 512 | 7 548 | ↓ 3 036 | ↑ 40.22 | 30 | 0.40 |
| 32 | sharedobject | 224 | 562 | ↓ 338 | ↑ 60.14 | 15 | 2.67 |
| 33 | sharedpooldatasource | 477 | 1 244 | ↓ 767 | ↑ 61.66 | 19 | 1.53 |
| 34 | simple | 689 | 1 580 | ↓ 891 | ↑ 56.39 | 22 | 1.39 |
| 35 | store | 2 165 | 3 037 | ↓ 872 | ↑ 28.71 | 46 | 1.51 |
| 36 | transfer | 153 | 194 | ↓ 41 | ↑ 21.13 | 16 | 8.25 |
| 37 | waitnotify | 108 400 | 179 769 | ↓ 71 369 | ↑ 39.70 | 142 | 0.08 |
| 38 | weblech | 207 126 | 406 561 | ↓ 199 435 | ↑ 49.05 | 497 | 0.12 |

根据表 2 和表 3 中的结果, 本文得到如下发现.

首先, 与 J-MCR 相比, GC-MCR 方法在大部分测试程序上可以进行更少次数的约束求解, 使用 GC- MCR 可以平均减少 20%左右的约束求解, 最高能够减少 71.43%的约束求解; 并且, 对于在 J-MCR 中约束求解次数少的测试程序, 其在使用 GC-MCR 后的过滤的约束求解次数占比越多.

其次, 由于 GC-MCR 方法减少了大部分测试程序的约束求解次数, 而且对少部分没有减少求解次数的测试程序进行了约束表达式约减, 所以 GC-MCR 的约束求解耗时相比 J-MCR 更低. 与 J-MCR 相比,使用 GC-MCR 可以平均减少 37%左右的用于约束求解的计算资源, 最多能够减少 71.46%的用于约束求解的计算资源.

最后, GC-MCR 方法对约束进行过滤和约减所需的时间远少于使用 GC-MCR 减少的约束求解时间, 其最高仅占 J-MCR 约束求解耗时的 8%. 因此可以得出: 由于基于有向图的约束约减和约束过滤时间开销远小于求解器求解带有冲突约束的时间开销, 通过对约束表达式进行提前过滤并约减, 确实能够有效减少程序调用约束求解器进行约束求解的次数, 降低程序在约束求解部分的耗时, 减少约束求解器的计算资源消耗.

通过进一步分析发现: 在 J-MCR 算法中约束求解次数多的测试程序, 其按特点被分为两种, 一是线程多的测试程序, 二是线程相对较少但每个线程中对同一变量的读写事件很多的测试程序. 前者由于其线程较多, 线程的交织数量会很多, 因此需要进行大量的约束求解. 然而在使用了 GC-MCR 方法后, GC-MCR 会在固定程间的执行先后顺序后去判断读写约束是否冲突, 固定线程执行顺序可以更加充分地判断冲突, 过滤掉更多约束,使 GC-MCR 检测并发缺陷的效果更好. 后者由于线程数量较少, 对同一变量的读写事件很多, 因此即使固定了线程的执行顺序, 很多读写事件的顺序组合依旧成立, 故使用 GC-MCR 方法不一定过滤掉很多约束, 因此导致过滤约束的效果很不稳定. 未来可以进行更深一步的研究去解决这个问题.

(2) RQ2: GC-MCR 方法能否有效减少并发缺陷检测的时间, 提高探索并发程序状态空间的效率?

为了回答 RQ2, 本文系统地将 GC-MCR 方法应用于每个测试程序, 统计检测到第一个并发缺陷时所需要的时间, 检查两种方法之间是否存在差异. 本文选择将 GC-MCR 与 J-MCR 方法进行了对比, 实验结果见表 4. 其中, GC-MCR 平均执行时间和 J-MCR 平均执行时间分别表示使用 GC-MCR 方法和 J-MCR 方法找到测试程序中第 1 个并发缺陷所需的平均时间; 程序执行相差时间表示被测试程序使用 J-MCR 和使用 GC-MCR 所需的时间的差值; 性能提升表示程序执行相差时间占 J-MCR 平均执行时间的百分比.

表 4 GC-MCR 与 J-MCR 的平均执行时间对比

| ID | 测试程序 | GC-MCR 平均执行时间(ms) | J-MCR 平均执行时间(ms) | 程序执行相差时间(ms) | 性能提升(%) |
|---|---|---|---|---|---|
| 1 | account | 929 | 1 170 | ↓ 241 | ↑ 20.60 |
| 2 | airline | 308 183 | 363 946 | ↓ 55 763 | ↑ 15.32 |
| 3 | alarmclock | 3 751 189 | 3 976 459 | ↓ 225 270 | ↑ 5.67 |
| 4 | allocation | 2 445 | 2 813 | ↓ 368 | ↑ 13.08 |
| 5 | array | 57 881 | 98 859 | ↓ 40 978 | ↑ 41.45 |
| 6 | bakery | 9 104 | 21 856 | ↓ 12 752 | ↑ 58.35 |
| 7 | boundedbuffer | 1 143 | 1 509 | ↓ 366 | ↑ 24.25 |
| 8 | bubblesort | 49 650 | 62 486 | ↓ 12 836 | ↑ 20.54 |
| 9 | checkfield | 400 | 811 | ↓ 411 | ↑ 50.68 |
| 10 | clean | 7 380 | 11 248 | ↓ 3 868 | ↑ 34.39 |
| 11 | consisitency | 699 | 982 | ↓ 283 | ↑ 28.82 |
| 12 | counter | 28 581 | 33 977 | ↓ 5 396 | ↑ 15.88 |
| 13 | critical | 271 | 320 | ↓ 49 | ↑ 15.31 |
| 14 | cyclic | 23 361 | 27 442 | ↓ 4 081 | ↑ 14.87 |
| 15 | datarace | 595 | 1 641 | ↓ 1 046 | ↑ 63.74 |
| 16 | deadlock | 172 | 243 | ↓ 71 | ↑ 29.22 |
| 17 | dekker | 78 075 | 155 317 | ↓ 77 242 | ↑ 49.73 |
| 18 | dirkaccount | 102 830 | 218 181 | ↓ 115 351 | ↑ 52.87 |
| 19 | fileappender | 21 954 | 30 468 | ↓ 8 514 | ↑ 27.94 |
| 20 | hashcode | 1 233 | 2 557 | ↓ 1 324 | ↑ 51.78 |
| 21 | lamport | 22 791 | 39 056 | ↓ 16 265 | ↑ 41.65 |
| 22 | mergesort | 284 283 | 380 757 | ↓ 96 474 | ↑ 25.34 |
| 23 | numberaxis | 194 446 | 271 151 | ↓ 76 705 | ↑ 28.29 |
| 24 | peruserpooldatasource | 1 560 | 3 980 | ↓ 2 420 | ↑ 60.80 |
| 25 | peterson | 11 271 | 17 240 | ↓ 5 969 | ↑ 34.62 |
| 26 | pingpong | 9 048 | 13 640 | ↓ 4 592 | ↑ 33.67 |
| 27 | pool | 720 | 850 | ↓ 130 | ↑ 15.29 |
| 28 | rax | 11 426 | 16 830 | ↓ 5 404 | ↑ 32.11 |
| 29 | readerswriters | 224 668 | 327 420 | ↓ 102 752 | ↑ 31.38 |
| 30 | replicatedcasestudies | 16 667 | 30 071 | ↓ 13 404 | ↑ 44.57 |
| 31 | rvexample | 4 840 | 7 724 | ↓ 2 884 | ↑ 37.34 |
| 32 | sharedobject | 330 | 650 | ↓ 320 | ↑ 49.23 |
| 33 | sharedpooldatasource | 920 | 1 590 | ↓ 670 | ↑ 42.14 |
| 34 | simple | 821 | 1 669 | ↓ 848 | ↑ 50.81 |
| 35 | store | 2 310 | 3 129 | ↓ 819 | ↑ 26.17 |
| 36 | transfer | 270 | 330 | ↓ 60 | ↑ 18.18 |
| 37 | waitnotify | 110 780 | 182 196 | ↓ 71 416 | ↑ 39.20 |
| 38 | weblech | 9 296 869 | 1 753 2083 | ↓ 8 235 214 | ↑ 46.97 |

根据表 4 中的内容, 本文观察到以下结果. 首先, 相比于 J-MCR 方法, 使用 GC-MCR 方法过滤并简化约束后可以有效减少并发缺陷检测的时间, 总调用时间平均减少 34.01%, 即程序的总体性能平均提升 34.01%. 其次, 再根据表 2 可以发现, 过滤约束百分与性能提升的百分比不成正比. 再次, 对于那些 J-MCR 平均执行时间越短的测试程序, 即使用 J-MCR 时约束求解次数越多的测试程序, 使用 GC-MCR 方法过滤并约减约束后其性能提升越高. 最后, 对于那些没有约束被过滤的测试程序, 使用 GC-MCR 方法约减其约束同样可以减少程序执行时间, 提升性能.

特别地, 在真实世界应用程序 weblech 中, 本文在 RQ1 和 RQ2 得到的结论依旧有效, 且进一步验证了 GC-MCR 方法的有效性, 表明 GC-MCR 能够检测状态空间庞大的并发程序的并发缺陷, 并可应用于真实世界应用软件.

GC-MCR 平均执行时间由过滤和约减约束耗时、约束求解耗时和程序调度耗时这 3 个部分组成. 进一步分析发现: 根据表 3, 过滤和约减约束的时间开销远小于求解器求解带有冲突约束的时间开销, 故一旦存在可过滤的约束或可约减的约束, 即可减少约束求解部分的时间开销(约束求解部分包括约束过滤和约减). 继而, 通过 GC-MCR 总体的并发缺陷检测时间降低, 发现约束求解部分减少的时间开销远多于程序调度的耗时开销的变化量, 即缺陷检测时间减少主要来自约束求解部分减少的时间开销与过滤和程序调度的耗时开销变化之

差. 该结果出现的原因仍是程序约束求解的时间开销远大于程序调度的时间开销, GC-MCR 仅对约束进行过滤和约减, 未改变程序的调度, 致使两者的变化量存在与之前相同的数量关系.

(3) RQ3: GC-MCR 方法是否可以保证现有方法的并发缺陷检测能力?

为了回答 RQ3, 本文系统地将 GC-MCR 方法应用于每个测试程序, 统计检测到第一个并发缺陷的类型, 检查两种方法之间是否存在差异. 实验结果见表 5, 其中, GC-MCR 调度次数和 J-MCR 调度次数分别表示使用 GC-MCR 方法和 J-MCR 方法找到测试程序中的并发缺陷所需的调度次数; GC-MCR 的并发缺陷和 J-MCR 的并发缺陷分别表示使用 GC-MCR 方法和 J-MCR 方法找到的测试程序中的并发缺陷.

表 5 GC-MCR 与 J-MCR 的有效调度次数对比

| ID | 测试程序 | 调度次数 | | 并发缺陷 | |
|---|---|---|---|---|---|
| | | GC-MCR | J-MCR | GC-MCR | J-MCR |
| 1 | account | 3 | 3 | Atomicity | Atomicity |
| 2 | airline | 434 | 445 | DataRace | DataRace |
| 3 | alarmclock | 888 | 915 | DataRace | DataRace |
| 4 | allocation | 4 | 4 | Atomicity | Atomicity |
| 5 | array | 22 | 22 | DataRace | DataRace |
| 6 | bakery | 15 | 15 | Atomicity | Atomicity |
| 7 | boundedbuffer | 1 | 1 | Deadlock | Deadlock |
| 8 | bubblesort | 10 | 10 | DataRace | DataRace |
| 9 | checkfield | 5 | 5 | DataRace | DataRace |
| 10 | clean | 2 | 2 | DataRace | DataRace |
| 11 | consisitency | 4 | 4 | Atomicity | Atomicity |
| 12 | counter | 2 | 2 | DataRace | DataRace |
| 13 | critical | 2 | 2 | Atomicity | Atomicity |
| 14 | cyclic | 8 | 8 | DataRace | DataRace |
| 15 | datarace | 3 | 3 | DataRace | DataRace |
| 16 | deadlock | 2 | 2 | Deadlock | Deadlock |
| 17 | dekker | 145 | 145 | Atomicity | Atomicity |
| 18 | dirkaccount | 2 | 2 | Deadlock | Deadlock |
| 19 | fileappender | 3 | 3 | Atomicity | Atomicity |
| 20 | hashcode | 5 | 5 | Atomicity | Atomicity |
| 21 | lamport | 75 | 75 | Atomicity | Atomicity |
| 22 | mergesort | 2 | 2 | DataRace | DataRace |
| 23 | numberaxis | 17 | 17 | Atomicity | Atomicity |
| 24 | peruserpooldatasource | 3 | 3 | DataRace | DataRace |
| 25 | peterson | 5 | 5 | Atomicity | Atomicity |
| 26 | pingpong | 4 | 4 | DataRace | DataRace |
| 27 | pool | 2 | 2 | DataRace | DataRace |
| 28 | rax | 2 | 2 | Deadlock | Deadlock |
| 29 | readerswriters | 10 | 10 | DataRace | DataRace |
| 30 | replicatedcasestudies | 1 | 1 | Deadlock | Deadlock |
| 31 | rvexample | 21 | 21 | DataRace | DataRace |
| 32 | sharedobject | 3 | 3 | DataRace | DataRace |
| 33 | sharedpooldatasource | 1 | 1 | Atomicity | Atomicity |
| 34 | simple | 6 | 6 | Atomicity | Atomicity |
| 35 | store | 2 | 2 | Atomicity | Atomicity |
| 36 | transfer | 2 | 2 | Deadlock | Deadlock |
| 37 | waitnotify | 560 | 560 | DataRace | DataRace |
| 38 | weblech | 1 096 | 2 024 | DataRace | DataRace |

根据表 5 中的内容, 本文观察到以下结果.

首先, GC-MCR 能够正确检测出测试程序中的并发缺陷. GC-MCR 和 J-MCR 检测到的并发缺陷一致, 且在大部分测试程序上以相同的线程调度触发了并发错误, 即 GC-MCR 方法与 J-MCR 相比, 并发缺陷的检测能力没有任何损失. 当程序状态空间较小时, 随机性对程序并发空间的探索影响较小, 即与 J-MCR 相比, GC-MCR 可以通过对并发程序执行相同次数检测出并发缺陷.

其次, GC-MCR 在 airline, alarmclock 以及 weblech 这 3 个测试程序上, 相比于 J-MCR 可以通过更少的线程调度触发并发错误, 表明随着程序状态空间庞大时, 随机性对程序并发空间的探索影响增大, 冗余的约束导致现有方法探索冗余的线程调度, 从而降低了检测效率. 而 GC-MCR 可以通过比 J-MCR 更少次数的程序执行检测出并发缺陷, 有效过滤冲突以及约减冗余约束. 即: 通过 GC-MCR 过滤掉的冲突约束都是无效约束,

对于冗余约束地约减也是等效且合理的. 因此, 约束过滤和约束约减后, 测试框架的完备性没有降低, 没有漏掉任何线程交织序列, 说明 GC-MCR 方法在提升了性能的同时, 没有漏掉任何测试线程序列, 既提升了并发约束检测的效率, 又保证了测试框架的完备性.

通过进一步分析发现, GC-MCR 方法使用有向图对约束进行过滤以及约减. 对于约束求解步骤中的约束部分, GC-MCR 要么选择过滤当前约束, 要么选择对当前约束进行约减, 得到等价的约束表达式, 其中没有任何操作对约束的求解结果进行修改, 故尽管程序执行存在随机性, GC-MCR 仍可以保证调度方案与 J-MCR 一致或更少, 故 GC-MCR 保证了其完备性, GC-MCR 方法可以保证现有方法的并发缺陷检测能力.

## 4 讨 论

本文提出了基于有向图的并发约束表示方法, 并基于此设计并实现构建约束的并发缺陷检测方法 GC-MCR. 现有的研究方法基于最大因果模型构建并发程序约束, 然而在并发程序的表示上, 尚没有系统的研究. 本文提出的基于有向图的表示方法从结构上优化了并发程序约束的表示: 一方面, 借助有向图表示的优势, 在构建过程中即可识别出不可解的约束, 减少了约束求解的成本; 另一方面, 精减了构建的并发程序约束, 对于相同的线程交织, GC-MCR 可以构建出更准确更精减的并发程序约束, 继而提升了约束求解速度. 因此, 在有限的测试资源下, 约束求解的成本降低, 约束求解部分节省的资源用于探索并发程序状态空间, 故而并发缺陷的检测效率提升. 实验结果也表明: 相比于现有方法, GC-MCR 可以更快地找出并发程序中的缺陷. 然而, 通过分析实验结果发现: GC-MCR 在检测效率上的提升并不稳定, 在平均执行时间的提升从 5%–64%不等; 同时, 在调度次数上的优势并不明显. 接下来, 本节将从程序状态空间、因果模型及并发程序这 3 个角度进行讨论.

(1) GC-MCR 性能提升受并发程序状态空间影响. 并发程序的状态空间随着并发程序内的线程数量及读写操作数量的增长而呈指数级增长; 同时, 随着并发程序状态空间的增长, 找到并发缺陷前的约束求解次数及程序调度次数也随之增长. 从实验结果提升的百分比中可以发现: 受程序状态空间的影响, GC-MCR 在约束求解次数及平均时间上的提升并不稳定. 一方面, 基于现有的最大因果模型, 程序的线程数量读写操作越多, 其构建的有效并发约束越多. 从表 2 的实验结果中可以看出, 减少的约束求解次数随着程序状态空间的增长而增长, 然而约束百分比的提升却因为其庞大的有效并发约束空间的增长而减少. 另一方面, 本文基于有向图的构建约束方法相比于现有方法, 需要对约束进行重新表示并分析. 因此, 随着程序状态空间的增长, 有向图表示方法的成本也随之增长. 从表 4 的结果中可以看出: 由于约束求解过于耗时, 有向图表示方法的成本依然低于现有方法约束求解的成本;

(2) 调度次数受最大因果模型的限制. 本文提出基于有向图对并发程序进行表示, 并基于此提出并发程序约束的构建方法. 已有研究表明, 最大因果模型可以最大化的减少探索冗余的并发程序状态空间. 因此, 本文约束构建的核心模型同样基于最大因果模型. 从表 5 的实验结果可以看出: GC-MCR 的有效调度序列与现有方法相比优势并不明显, 其采用相同的最大因果模型为主要原因. GC-MCR 的优势在于更有效地生成符合最大因果模型的并发约束, 减少约束求解的成本, 继而更快地找到并发缺陷;

(3) 各项评测数据与并发测试程序的关系. 本文用于评测 GC-MCR 和 J-MCR 并发缺陷检测能力的评测指标分别为约束求解次数、平均执行时间、约束过滤耗时、调度次数. 由于本文基于最大因果模型, 约束求解次数、约束过滤耗时和调度次数都受限制于最大因果模型, 因此三者都与并发测试程序中的线程数量和不同线程中对变量的读写次数相关. 而平均执行时间不但与约束求解次数、约束过滤耗时和调度次数相关联, 而且硬件的配置及执行速度也会影响到平均执行时间. 由于平均执行时间和调度次数能够直观且全面地评价方法的并发缺陷检测能力, 本文将调度次数和平均执行时间作为评测的评测指标.

### 4.1 有效性影响因素分析

本节主要对可能会影响到本文实验结果有效性的因素进行分析, 具体分析如下.

(1) 内部有效性分析. 考虑本文方法的实现较复杂, 因此可能影响到本文实验结果有效性的内部因素主要为本文方法的代码实现是否正确. 为了减少代码实现的误差给本文实验结果带来的影响, 本文使用到的相关技术均使用成熟的第三方框架, 例如 ASM[24], Z3[20]等. 同时, 本文参考了已有研究成果的代码实现, 例如 MCR[18], CovCon[25]等, 且编码过程采用 peer review 的方式, 最大程度上保证了代码实现的正确性;

(2) 外部有效性分析. 影响本文实验结果的结论是否具有一般性的因素主要包含 3 个因素.

- 第 1 个因素是本文使用的数据集是否具有代表性. 为确保本文研究结论的有效性, 本文采用现有并发缺陷检测研究中常用的 27 个并发数据集, 该数据集涵盖了并发缺陷常见的错误类型;
- 另一方面, 由于本文依赖的约束求解器 Z3 是 CPU 密集型工具, 因此, 第 2 个影响因素为实验平台是否一致. 为了减少实验平台对本文实验结果带来的影响, 本文所有实验均在同一实验平台上进行; 同时, 所有实验均运行多次, 以消除可能的随机性对实验结果造成的影响;
- 第 3 个影响因素为本文使用的对比方法是否具有代表性. 本文选择在 Java 程序上验证 GC-MCR 的有效性. 相比之下, 其他并发测试方法, 例如基于饱和覆盖的主动测试方法 Maple[26], IDAT[27]以及基于 CBMC 实现的多个并发测试方法[28−30]均是在 C 语言上对其方法的有效性进行了验证. SeqCheck[23]通过对程序分支信息进行建模, 通过预测事件序列的可行性来检测 Java 程序中的并发缺陷, 然而该方法并未公开源代码. 考虑到上述工作在 Java 程序上进行复现的成本较高, 且无法保障复现程序的正确性, 同时, 在最新研究工作的 J-MCR 已经验证了其方法的先进性, 具有一定的代表性, 故本工作主要与 J-MCR 进行对比;

(3) 结论有效性分析. 影响本文结论有效性的因素主要为本文在实验分析过程中使用的评价指标是否合理. 针对此问题, 本文采用现有并发缺陷检测研究中两个常用的评判并发缺陷检测能力的评价指标——检测到第一个并发缺陷所用的时间及检测到第一个并发缺陷时并发程序的调度次数. 同时, 为了衡量 GC-MCR 是否能有效地减少约束求解的次数, 本文引入约束求解次数评价指标, 进一步确保本文结论的有效性.

## 5 相关工作

### 5.1 并发程序覆盖度量

并发程序覆盖标准可用于量化测试套件的丰富程度(例如, 程序是否经过充分测试)或提供有关测试用例生成的实用指南(例如, 作为程序模糊引擎中使用的目标函数). 例如, HaPSet[31]是收集程序中的排序约束, 并通过分析约束来指导程序的测试. CovCon[25]方法是通过分析记录的执行来提取常见执行的方法对, 生成有可能覆盖那些未覆盖方法对的测试用例. TSA[32]旨在通过生成线程调度来覆盖未覆盖的覆盖需求(coverage requirement), 从而实现对于并发程序的高同步覆盖. ConSuite[33]方法是通过静态分析线程交织集并且检查程序执行记录以检查特定线程交织是否被覆盖, 然后使用遗传算法来生成可以覆盖更多线程交织的测试. AutoConTest[34]方法则是考虑调用上下文信息, 动态以及迭代地计算出覆盖需求, 根据顺序覆盖生成顺序测试, 将顺序测试组装成并发测试.

Yang 等人[35]提出了一种基于全双路径覆盖的定义使用对覆盖(definition-use coverage). Krena 等人[36]提出了一种基于现有动态或静态分析方法(例如 Eraser[37]和 GoldiLocks[38])为测试并发软件推导出新覆盖率指标的方法. 他们扩展了多个现有的并发测试覆盖率指标, 例如 ConcurPairs、定义使用对覆盖和同步对覆盖(synchronisation pair coverage). MAP-Coverage[22]方法使用内存访问模式(memory access pattern)抽象测试执行并测量覆盖了哪些内存访问模式, 然后生成可以覆盖更多内存访问模式的测试用例. MAP-Coverage 比线程交

织覆盖更抽象, 并且比方法对覆盖更与 bug 相关.

### 5.2 模型检测和约束求解

现有并发缺陷检测研究中, 模型检测技术[39−41]常被用于穷举探索线程调度空间, 以此来检测程序中潜在的并发缺陷. 例如, CHESS[42]以上下文有界的方式动态探索目标程序的不同线程调度, Shacham 等人[40]基于模型检查器为 lockset 算法[43−45]报告的竞争构建测试用例. 然而, 面对程序路径和调度空间的指数级增长, 模型检查技术难以扩展到大型多线程程序. 故而, Huang 等人提出 MCR[18]方法, 将探索并发程序状态空间的问题转换成约束求解问题, 从而减少对并发程序冗余状态空间的探索, 提升并发缺陷的检测效率. 同样, STORM 方法[46]将约束求解与验证条件生成结合, 通过构建 BoogiePL 程序的全局存储的映射之间的约束, 从而检测 Windows 设备驱动程序中潜在的缺陷.

约束求解主要依赖 SMT 求解器或者 SAT 求解器, 目前, 致力于提升求解器能力的工作[47−49]层出不穷, 但是约束难解问题仍旧是亟待解决的问题. 为此, Wang 等人[47]提出了通过深度强化学习来利用神经网络的决策能力加速约束求解. 与上述工作不同, 本文通过提出有向图表示方法, 将并发程序约束中的程序基本约束构建有向图, 并根据有向图对并发程序约束进行过滤和约减, 提高约束求解过程的速度, 降低约束求解的计算资源消耗.

### 5.3 并发程序缺陷检测

并发程序缺陷检测通常研究 3 个问题, 即如何提高检测效率、如何提高检测的有效性以及如何减少误报. 并发缺陷检测的经典方法有 happens-before 分析方法[50,51]和 lockset 算法[43−45], 它们被广泛用于检测并发程序的错误. 例如: RaceChecker[50]是一个数据竞争检测器, 它在报告要验证的潜在竞争之前, 使用 happens-before 关系来修剪不可行的竞争; Eraser[36]提出了一种 lockset 算法, 通过监视每个共享内存引用和锁定行为来检测基于锁的并发程序中的错误. lockset 的改进方法[43−45,52]都对 lockset 算法进行了改进, 以减少开销和误报. lockset 与 happens-before 的组合算法[38,53−56]将 lockset 算法与 happens-before 分析方法相结合 .然而, 由于静态分析方法的局限性, 与动态并发缺陷检测方法相比, lockset 算法和 happens-before 分析方法经常会出现误报.

误报意味着与缺陷无关的线程交互被视为错误. 静态检测技术不执行程序, 而是通过分析源代码来检测缺陷. 因此, 检测器无法正确确定 happens-before 信息, 这可能会导致误报. MCR[18]方法将探索并发程序状态空间的问题转换成约束求解问题, 根据每次求解结果控制程序的调度, 能够使用最少的程序执行次数探索整个程序的状态空间. ConRacer[57]方法使用控制流分析来构建调用图, 使用上下文敏感的别名分析查找别名对象, 最后使用 happens-before 分析以减少误报和漏报. Maple[26]提出了一种基于饱和覆盖的主动测试方法. Yue 等人[27]基于 Maple 提出了一种多线程程序的测试用例多样性度量. Alglave 等人[29]提出了一种整数差分逻辑编码, 借助有界模型检查, 从而对部署的并发系统代码进行验证. He 等人[28]提出了一种应用于 SC 的多线程程序验证的排序一致性理论, 实现了增量式一致性检验、最小冲突子句生成和专业化理论传播. Yin 等人[30]根据事件顺序图(event order graph, EOG)设计了基于 EOG 的反例验证和优化生成算法, 能够获得较小但有效的优化约束. 基于此, 提出了一种基于调度约束的多线程 C 程序验证抽象细化方法. SeqCheck[23]方法对程序分支进行建模, 并对事件序列的可行性进行预测. 其首先计算事件集以确定序列的可行性, 然后通过构造图以重新排序集合中的事件, 最后计算图中顺序的闭包.

本文工作与以上工作都有一定的区别, 本文不仅将探索并发程序状态空间的问题转换成约束求解问题, 还在此基础上提出了有向图表示方法, 构建完整的并发程序约束并对其进行过滤和约减, 减少了约束求解成本, 继而提高探索并发程序状态空间的效率.

## 6 总结与展望

本文基于有向图构建约束, 加速最大因果约减算法的约束求解, 在调用约束求解器前对约束进行操作, 将读写约束进行拆分与解析, 并将其与程序基本约束进行冲突检测, 过滤冲突约束并约减冗余约束, 减少单

次求解耗时, 甚至减少约束求解器的调用, 减少约束求解器的计算资源消耗, 继而提高探索并发程序中所有可能状态的效率. 本文在 38 组并发测试程序上对 GC-MCR 方法进行了实证研究. 实验表明: GC-MCR 方法在保证了调度的完备性和 MCR 方法的并发缺陷检测能力的情况下, 减少了单次约束求解的耗时和对约束求解器的调用, 降低了约束求解器的计算资源消耗, 提高了探索被测程序状态空间的效率, 显著优于现有的方法.

在未来的研究工作中, 可以添加一些可行性松弛, 通过添加分支事件的可行性松弛相关算法, 松弛分支事件的具体的可行性约束要求, 在不影响准确性的情况下, 进一步提高并发缺陷检测的效率.

**References**:


[1] Jiang YY, Xu C, Ma XX, *et al*. Approaches to obtaining shared memory dependences for dynamic analysis of concurrent programs: A survey. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 747−763 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5193.htm [doi: 10.13328/j.cnki.jos.005193]

[2] Bo LL, Jiang SJ, Zhang UM, *et al*. Research progress on techniques for concurrency bug detection. Computer Science, 2019, 46(5): 13−20 (in Chinese with English abstract).

[3] Su XH, Yu Z, Wang TT, *et al*. A survey on exposing, detectingand avoiding concurrency bugs. Chinese Journal of Computers, 2015, 38(11): 2215−2233 (in Chinese with English abstract).

[4] Putchala MK, Bryant AR. Synchron-ITS: An interactive tutoring system to teach process synchronization and shared memory concepts in an operating systems course. In: Proc. of the 2016 Int'l Conf. on Collaboration Technologies and Systems (CTS 2016). 2016. 180−187.

[5] Huang J, Meredith PO, Rosu G. Maximal sound predictive race detection with control flow abstraction. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2014). 2014. 337−348.

[6] Park S, Vuduc RW, Harrold MJ. Falcon: Fault localization in concurrent programs. In: Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering (ICSE 2010). 2010. 245−254.

[7] Lu GZ, Xu L, Yang YB, Xu BW. Predictive analysis for race detection in software-defined networks. Science China Information Sciences, 2019, 62(6): 062101:1−062101:20.

[8] Cai Y, Lu Q. Dynamic testing for deadlocks via constraints. IEEE Trans. on Software Engineering, 2016, 42(9): 825−842.

[9] Engler DR, Ashcraft K. RacerX: Effective, static detection of race conditions and deadlocks. In: Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP 2003). 2003. 237−252.

[10] Letko Z, Vojnar T, Krena B. AtomRace: Data race and atomicity violation detector and healer. In: Proc. of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2008), Held in Conjunction with the ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA 2008). 2008.

[11] Huang J, Zhang C, Dolby J. CLAP: Recording local executions to reproduce concurrency failures. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2013). 2013. 141−152.

[12] Lu S, Tucek J, Qin F, Zhou YY. AVIO: Detecting atomicity violations via access-interleaving invariants. IEEE Micro, 2007, 27(1): 26−35.

[13] Clarke EM, Grumberg O, Minea M, Peled D. State space reduction using partial order techniques. Int'l Journal on Software Tools for Technology Transfer, 1999, 2(3): 279−287.

[14] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. In: Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2005). 2005. 110−121.

[15] Godefroid P. Partial-order Methods for the Verification of Concurrent Systems—An Approach to the State-explosion Problem. Springer, 1996.

[16] Mazurkiewicz A. Trace Theory. Petri Nets: Applications and Relationships to Other Models of Concurrency. Berlin, Heidelberg, 1987. 278−324.

[17] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7): 558−565.

[18] Huang J. Stateless model checking concurrent programs with maximal causality reduction. In: Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2015. 165−174.

[19] Serbanuta TF, Chen F, Rosu G. Maximal causal models for sequentially consistent systems. In: Proc. of the 3rd Int'l Conf. on Runtime Verification (RV 2012). Revised Selected Papers, 2012. 136−150.

[20] de Moura LM, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), Held as Part of the Joint European Conf. on Theory and Practice of Software (ETAPS 2008). 2008. 337−340.

[21] Zhao YQ, Wang Z, Liu S, *et al*. Achieving high MAP-coverage through pattern constraint reduction. IEEE Trans. on Software Engineering, 2022. [doi: 10.1109/TSE.2022.3144480]

[22] Wang Z, Zhao YQ, Liu S, *et al*. MAP-coverage: A novel coverage criterion for testing thread-safe classes. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2019). 2019. 722−734.

[23] Cai Y, Yun H, Wang JQ, *et al*. Sound and efficient concurrency bug prediction. In: Proc. of the 29th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE 2021). 2021. 255−267.

[24] ASM bytecode analysis framework. http://asm.ow2.org/

[25] Choudhary A, Lu S, Pradel M. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In: Proc. of the 39th Int'l Conf. on Software Engineering (ICSE 2017). 2017. 266−277.

[26] Yu J, Narayanasamy S, Pereira C, *et al*. Maple: A coverage-driven testing tool for multithreaded programs. In: Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications. 2012. 485−502.

[27] Yue H, Wu P, Chen TY, *et al*. Input-driven active testing of multi-threaded programs. In: Proc. of the 2015 Asia-Pacific Software Engineering Conf. (APSEC). 2015. 246−253.

[28] He F, Sun ZH, Fan HY. Satisfiability modulo ordering consistency theory for multi-threaded program verification. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation (PLDI 2021). 2021. 1264−1279. [doi: 10.1145/3453483.3454108]

[29] Alglave J, Kroening D, Tautschnig M. Partial orders for efficient bounded model checking of concurrent software. In: Proc. of the 25th Int'l Conf. on Computer Aided Verification (CAV 2013). 2013. 141−157.

[30] Yin L, Dong W, Liu WW, *et al*. On scheduling constraint abstraction for multi-threaded program verification. IEEE Trans. on Software Engineering, 2020, 46(5): 549−565.

[31] Wang C, Said M, Gupta A. Coverage guided systematic concurrency testing. In: Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE 2011). 2011. 221−230.

[32] Hong S, Ahn J, Park S, *et al*. Testing concurrent programs to achieve high synchronization coverage. In: Proc. of the Int'l Symp. on Software Testing and Analysis (ISSTA 2012). 2012. 210−220.

[33] Steenbuck S, Fraser G. Generating unit tests for concurrent classes. In: Proc. of the 6th IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST 2013). 2013. 144−153. [doi: 10.1109/ICST.2013.33]

[34] Terragni V, Cheung SC. Coverage-driven test code generation for concurrent classes. In: Proc. of the 38th Int'l Conf. on Software Engineering (ICSE 2016). 2016. 1121−1132. [doi: 10.1145/2884781.2884876]

[35] Yang CD, Souter AL, Pollock LL. All-du-path coverage for parallel programs. In: Proc. of the ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA'98). 1998. 153−162.

[36] Krena B, Letko Z, Vojnar T. Coverage metrics for saturation-based and search-based testing of concurrent software. In: Proc. of the 2nd Int'l Conf. on Runtime Verification (RV 2011). LNCS 7186, 2012. 177−192.

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

[38] Choi JD, Lee K, Loginov A, *et al*. Efficient and precise datarace detection for multithreaded object-oriented programs. In: Proc. of the 2002 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). 2002. 258−269. [doi: 10.1145/512529.512560]

[39] Musuvathi M, Qadeer S, Ball T, *et al*. Finding and reproducing heisenbugs in concurrent programs. In: Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2008). 2008. 267−280.

[40] Shacham O, Sagiv M, Schuster A. Scaling model checking of dataraces using dynamic informatio. Journal of Parallel and Distributed Computing, 2007, 67(5): 536−550.

[41] Khurshid S, Pasareanu CS, Visser W. Test input generation with Java PathFinder: Then and now (invited talk abstract). In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA 2018). 2018. 1−2.

[42] Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. In: Proc. of the ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation. 2007. 446−455.

[43] Engler D, Ashcraft K. RacerX: Effective, static detection of race conditions and deadlocks. In: Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP 2003). 2003. 237−252.

[44] Nishiyama H. Detecting data races using dynamic escape analysis based on read barrier. In: Proc. of the 3rd Virtual Machine Research and Technology Symp. 2004. 127−138.

[45] Elmas T, Qadeer S, Tasiran S. Precise race detection and efficient model checking using locksets. Technical Report, MSR-TR-2005-118, Microsoft Research Microsoft Corporation, 2005.

[46] Lahiri SK, Qadeer S, Rakamaric Z. Static and precise detection of concurrency errors in systems code using SMT solvers. In: Proc. of the 21st Int'l Conf. on Computer Aided Verification (CAV 2009). 2009. 509−524.

[47] Wang F, Rompf T. From gameplay to symbolic reasoning: Learning SAT solver heuristics in the style of Alpha (Go) zero. arXiv: 1802.05340v1, 2018.

[48] Bello I, Pham H, Le QV, *et al*. Neural combinatorial optimization with reinforcement learning. In: Proc. of the 5th Int'l Conf. on Learning Representations (ICLR 2017). 2017.

[49] Xu L, Hoos H, Leyton-Brown K. Predicting satisfiability at the phase transition. In: Proc. of the 26th AAAI Conf. on Artificial Intelligence. 2012.
[50] Lu K, Wu Z, Wang XP, *et al*. RaceChecker: Efficient identification of harmful data races. In: Proc. of the 23rd Euromicro Int'l Conf. on Parallel, Distributed, and Network-based Processing (PDP 2015). 2015. 78−85.
[51] Perkovic D, Keleher PJ. Online data-race detection via coherency guarantees. In: Proc. of the 2nd USENIX Symp. on Operating Systems Design and Implementation (OSDI). 1996. 47−57.
[52] von Praun C, Gross TR. Object race detection. In: Proc. of the 2001 ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages and Applications (OOPSLA 2001). 2001. 70−82.
[53] Yu Y, Rodeheffer T, Chen W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In: Proc. of the 20th ACM Symp. on Operating Systems Principles (SOSP 2005). 2005. 221−234.
[54] von Praun C, Gross TR. Static conflict analysis for multi-threaded object-oriented programs. In: Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation. 2003. 115−128.
[55] Elmas T, Qadeer S, Tasiran S. Goldilocks: A race and transaction-aware Java runtime. In: Proc. of the ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation. 2007. 245−255.
[56] 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.
[57] Zhang Y, Liu H, Qiao L. Context-sensitive data race detection for concurrent programs. IEEE Access, 2021, 9: 20861−20867.

## 附中文参考文献:

[1] 蒋炎岩, 许畅, 马晓星, 等. 获取访存依赖: 并发程序动态分析基础技术综述. 软件学报, 2017, 28(4): 745−763. http://www.jos.org.cn/1000-9825/5193.htm [doi: 10.13328/j.cnki.jos.005193]
[2] 薄莉莉, 姜淑娟, 张艳梅, 等. 并发缺陷检测技术研究进展. 计算机科学, 2019, 46(5): 13−20.
[3] 苏小红, 禹振, 王甜甜, 等. 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.

**李硕川**(1999－), 男, 硕士生, CCF 学生会员, 主要研究领域为并发缺陷检测, 系统软件分析.

**王赞**(1979－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为软件测试, 机器学习.

**马明旭**(1999－), 男, 学士, 主要研究领域为并发测试.

**陈翔**(1980－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为智能软件工程, 软件仓库挖掘, 软件测试与维护.

**赵英全**(1994－), 男, 博士生, 主要研究领域为编译器测试, JVM 测试, 并发测试.

**王海弛**(1996－), 男, 博士生, 主要研究领域为系统软件分析.

**王昊宇**(1999－), 男, 硕士生, CCF 学生会员, 主要研究领域为并发缺陷检测, 编译器测试.